## Contents

1 Introduction .................................................. 3
   1.1 Aim of the course ........................................... 3
   1.2 Programming paradigms ..................................... 4
   1.3 The object-oriented paradigm .............................. 5

2 Objects and Classes ........................................... 7
   2.1 Classes ..................................................... 8
   2.2 Equality and copy ......................................... 8
   2.3 Recursion among classes, objects versus pointers ....... 10
   2.4 this ....................................................... 10
   2.5 Class variables and methods .............................. 11
   2.6 An algorithm based on objects ............................ 11

3 Inheritance .................................................... 12
   3.1 Heir classes ............................................... 12
   3.2 Overriding ............................................... 13
   3.3 Inheritance vs “programming by cases” .................. 14
   3.4 Wrapper classes .......................................... 19
   3.5 Interfaces and abstract classes .......................... 20
   3.6 super ...................................................... 21
   3.7 Static versus dynamic binding, hiding versus overriding .. 22

4 Formal definition of a toy Java-like language ............... 23
   4.1 Syntax .................................................... 23
   4.2 Execution model .......................................... 24
   4.3 Type system .............................................. 28
   4.4 Soundness of the type system ............................ 32
   4.5 Adding inheritance ....................................... 33

5 Typing issues .................................................. 35
   5.1 Checked exceptions ....................................... 35
   5.2 Casts and run-time type checks ........................... 39
   5.3 Type constraints in overriding ............................ 40
   5.4 Overloading ............................................... 44
   5.5 Languages where classes are distinct from types ........ 45
   5.6 MyType .................................................... 46
1 Introduction

1.1 Aim of the course

The aim of this course is to illustrate in a rigorous way (formally or semi-formally):

- basic concepts
- different design choices (w.r.t. expressive power, desirable properties)
- open issues

in object-oriented languages.

Some open issues in object-oriented languages:

- class-based versus object-based, inheritance versus delegation
- structural types versus name types
- static binding versus dynamic binding
- multiple inheritance
- separate compilation and dynamic linking
- support for parameterization/polymorphism.

An example of type problem (we use a Java syntax but assume rules from the Eiffel type system):

class Rectangle {
    int length, width;
    boolean equals (Rectangle r) {
        return length == r.length && width == r.width;
    }
}

class Cuboid extends Rectangle {
    int height;
    boolean equals (Cuboid c) {
        return length == c.length &&
            width == c.width && height == c.height;
    }
}

In Eiffel the definition of the method equals in Cuboid is considered to override that in Rectangle, since in overriding it is allowed to assign more specific types to the parameters. However, with this rule (which is called covariant rule), the type system is not sound, as shown by the following code:

Rectangle r = new Rectangle();
Rectangle r1 = new Cuboid();
System.out.println(r1.equals(r)); // statically well-typed

At run-time the version in Cuboid is invocated, hence we get a run-time error. This problem was already known in 1985 [3], but Eiffel was designed and implemented with the “covariant” rule (1988). Then, the problem was discovered and published [6], but at this stage it was difficult to treat the problem preserving backwards compatibility. We will discuss in deep this problem and possible solutions in Sec.5.3.

This example clearly shows how formal descriptions can be useful and important in language design.
1.2 Programming paradigms

Computational paradigms  It is not easy to define in a precise way the notion of programming paradigm. Informally, we can say that there are “families” of languages which basically follow the same execution model. In the following table there is an attempt at a brief description of the main programming paradigms.

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Underlying model</th>
<th>Running a program is …</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imperative</td>
<td>Store = abstraction of memory</td>
<td>Executing a command</td>
<td>Pascal, C, Ada</td>
</tr>
<tr>
<td>Functional</td>
<td>Functions (abstraction, application)</td>
<td>Evaluating an expression</td>
<td>Lisp, ML</td>
</tr>
<tr>
<td>Logic</td>
<td>Predicates and formulas</td>
<td>Finding solutions for a goal</td>
<td>Prolog</td>
</tr>
<tr>
<td>Object Based</td>
<td>Object’s universe</td>
<td>Sending a message to an object</td>
<td>Smalltalk, Eiffel, Java</td>
</tr>
</tbody>
</table>

In the imperative paradigm, the effect of a program is that of performing, from an initial store, a sequence of transformations by executing commands, until a final store is reached.

Formally, a store can be modeled by a function from locations (memory addresses) into values

\[
    \text{Store} = \text{Loc} \rightarrow \text{Val}
\]

hence a program can be interpreted as a function

\[
    p: \text{Store} \rightarrow \text{Store}
\]

In the functional paradigm, instead, a program is just a collection of function definitions, and a program can be interpreted as a value, obtained by an initial function call.

In the logic paradigm, a program is a collection of predicate definitions, given by means of clauses which assert that we can deduce the truth of some facts from the truth of some others. Executing a program means to find solutions (values for variables) which make true some formula (called goal).

The object-oriented paradigm, as we will discuss in the following, is characterized by different orthogonal features. From the computational point of view, that is, for what concerns the execution model (we will call it, more appropriately, object-based paradigm in this respect), the basic idea is similar to the imperative paradigm. Indeed, also in this case the program execution performs, from an initial state, a sequence of transformations, until a final state is reached. However in this case the evolving system, rather than to an abstraction of the computer memory, corresponds to a “universe of objects” and the evolution starts by sending an initial message to one of these objects. This object can, in reaction, send messages to other objects, and so on. An analogous situation happens when we use a graphical interface with windows, menus, buttons etc.; an action on some object leads to a reaction which may involve other objects.

Organizational paradigms  The paradigms we have discussed until now are computational paradigms, that is, concerning the execution model of programs.

We can also classify languages w.r.t. organizational paradigms, that is, concerning, rather than how single algorithms are written, how a complex program can be structured.

We list below some organizational paradigms.

Procedural  Modules correspond to single algorithmic units (procedures).

Abstract Data Type (ADT from now on)  Modules correspond to data types, that is, units which define some types of values and the operations for manipulating them.

Module-based  This is a more general paradigm where modules are collections of heterogeneous components, e.g., types, procedures, exceptions, variables, etc.

Generic Programming  Modules can be parameterized (for instance, a module \( \text{LIST}(T) \) which implements lists of a generic type \( T \)).

Object-Oriented  Modules are classes, and new modules can be constructed by partially modifying existing modules via the inheritance mechanism (see Sec.3).

Each language may have its computational and organizational paradigms, and also different paradigms may coexist in a language. For instance Smalltalk has a pure object-oriented paradigm; ML is a functional language which supports an ADT and Generic Programming organizational paradigm; C++ is a mixed-paradigm language (both imperative and object-oriented).
1.3 The object-oriented paradigm

A tentative definition It is difficult to give a precise definition of “object-oriented paradigm” (the term is often abused). We will individuate three main aspects:

Computational paradigm based on objects In this computational paradigm, elementary units of calculation are not functions or procedures (single algorithms) but “objects”, that is, groups of inter-related algorithms.

This computational paradigm based on objects is completely orthogonal to the inheritance notion, and even to the notion of state, and can be applied even to purely functional objects. However, in real languages objects are usually equipped with a state which can be modified, that is, a paradigm based on imperative objects is adopted. This corresponds to a view of an object which resembles objects from the real world, whose state may change, and which offer to the outside an interface of possible operations for being manipulated.

A language whose computational model is based on objects is said object-based, and does not need the notions of class and inheritance. However, the object-based languages without classes did not have much success in practice; on the contrary, they have been deeply analyzed as computational model underlying object-oriented languages [1].

Organizational paradigm In this sense, the object-oriented paradigm can be seen as a methodology for organizing software, which considers as basic software modules classes (that is, schemata for generating objects in the sense above), and where new modules can be defined from existing ones by an extension operation called inheritance which also allows to modify (redefine) old components. Assume, for instance, that we have a software module implementing integer lists and that we want to write a module which implements integer sets (that is, lists without repetitions), hence we want to modify, for instance, the operation which adds an element in such the way that no duplicate elements are added. In a traditional language we must “copy” the code of the existing module, modifying some parts. The object-oriented approach offers a linguistic support for obtaining the same result “for free”.

Overloading with dynamic resolution This aspect is tightly coupled with the previous (inheritance).

We say that a language supports overloading if it is possible to use the same name to denote different semantic entities, tipically of different types, e.g., to use the same name add to denote an operation which adds elements, say, to a set, to a list, to a table, and so on.

Note the difference between an overloaded function (there is a different definition for each type) and a polymorphic function as, e.g., in ML (there is only one definition which can be applied to arguments of different types).

In programming languages, overloading is very useful since there exist many situations in which it is natural to use the same name for operations on different types which are intuitively analogous. Hence, many programming languages allow forms of overloading: however, this leads to the problem that in a function call overloading resolution is needed, that is, to define rules for deciding, depending on the context, which definition has to be selected.

In traditional programming languages overloading resolution is static, that is, it takes place before the program is run. This form of overloading can be seen as just a useful abbreviation which allows the programmer to use only one name and not to be forced to invent a different one for each possible type of context. The algorithm of overloading resolution takes in input the code with overloaded symbols and produces in output code with no overloading, where each symbol has been annotated with the type information which is necessary for solving ambiguity.

In object-oriented languages, on the contrary, the mechanism of inheritance with redefinition illustrated above is coupled with dynamic overloading resolution; for some names (those of methods) the resolution takes place at run-time. This form of dynamic overloading allows to have functions whose result depends on the argument types. Formal foundations for this kind of functions have been provided in [4].

Motivations The main motivations to the introduction of the object-oriented approach have been historically the following.

Enhance software reuse Inheritance has been introduced as the solution (even though it has many limitations which we will discuss later on) to the demand for mechanisms for allowing reuse and easy modification of existing software, which was discovered to be more important, in a sense, than writing new software. Indeed, it was discovered (also by statistic analysys) that only a very small part of software which was developed was actually “new”.

Influence of the ADT paradigm The object-oriented approach has taken, even though in a different context, many principles of the ADT approach: organization of software in modules corresponding to data types and not single algorithms, distinction between interface and implementation of a module, incapsulation (hiding).

“Real world metaphore” The computational paradigm based on objects has been proposed as a paradigm wich resembles “the real world”. The object-oriented approach, indeed, has been developed, with the Smalltalk language, simultaneously to the first operating systems with iconic interface (Macintosh). Indeed, these interfaces are based on the same paradigm of “objects which answer to messages” (in this case, user actions).
Historical notes

- The recognized ancestor of the object-oriented languages is Simula 67 (Dahl and Nygaard, University of Oslo, end sixties): a language for simulations which has introduced the class notion.

- The first object-oriented language has been Smalltalk (the first ideas are due to Alan Key, and, as said above, have been developed at the same time of the new generation of operating systems; the first stable version of the language is due to Goldberg and Robson, Xerox PARC, in 1980): from Smalltalk comes the now currently used terminology. Smalltalk is an interpreted language and has no static type-checking.

- Other two significant languages are Eiffel (Meyer 1985), designed with the aim of bring the concepts of object-oriented programming in a software engineering context, and C++, an object-oriented extension of C which, however, follows a mixed paradigm, keeping some imperative features as pointers.

- Today a large variety of object-oriented languages exists. We mention some of them.
  - Object-oriented versions of existing languages, like extensions of Lisp (Loops, Flavors, Ceyx), leading in 1993 to CLOS (Common Lisp Object System); extensions of C (the already mentioned C++, by B. Stroustrup, AT&T, and Objective C by B. Cox); Ada 95: Borland Pascal.
  - Other languages: Oberon (by N. Wirth, a descendant of Modula-2); Modula-3 (DEC); Trellis (DEC), which combines genericity and multiple inheritance; Sather (a subset of Eiffel); Beta (a direct descendant of Simula); Self (object-based, supporting the delegation notion).
  - In addition to programming languages, there exist many other important applications of the object-oriented approach, for instance in databases and specification and design methodologies.

Wegner’s classification  An object-oriented language is (following P. Wegner’s classification [9], which we will adopt), a language which supports the three notions of object, class and inheritance.

Hence, we can consider three categories of languages which “approximate” the notion of object-oriented language:

- **object-based** language which supports the object notion
- **class-based** language which supports object and class notions
- **object-oriented** language which supports the notions of object, class and inheritance.

Object-based paradigm in traditional languages  Some “traditional” languages (for instance Ada and Modula-2) actually support the object notion, since it is possible to write software modules which correspond to “objects” that is, abstract data types with an internal state.

An abstract data type (ADT) defines a set of types and a family of operations for manipulating values of these types; however, implementation of both types and operations is hidden to external users.

An object defines, analogously, a family of operations; however, differently from an ADT, it also has an internal state. The following example shows, in an Ada-like syntax, a module (called *package* in Ada) which implements a single stack object with the usual operations.

```ada
package STACK --note: models ONE stack
interface
  constant max = 100;
  function Top return INTEGER;
  procedure Pop;
  procedure Push (z : INTEGER);
implementation
  ... private definitions of types and variables (e.g. an array of 100 integers)
  ... bodies of functions and procedures declared in the interface, + possibly others
```

A module like above can be used as follows:

```ada
use STACK;
...
STACK.Push(2); -- the message Push(2) is sent to the object STACK
...
```
The command `STACK.Push(2)` can be “read”, following the object-oriented terminology, as follows: the message `Push(2)` is sent to the object `STACK`.

Note that, differently from what happens in the ADT paradigm, the effect of calls depend on the internal state of the object. That is, instead of functions

\[ op: A_1 \times \ldots \times A_n \to A \]

we have functions with side-effect, callede methods

\[ op: (OState \times)A_1 \times \ldots \times A_n \to [A](\times OState) \]

Brackets are used to denote that the state argument and result is not explicitly indicated in the type of methods. Square brackets, indeed, denote optionality, since a method can either return a result or just have an effect on the state (the notions of function and procedure are unified).

Hence, in the ADT paradigm, the functions which constitute a module interface always give the same result on the same argument, whereas in the object-based paradigm a method invocation will give different results depending on the internal state of the object on which the method is invoked. Note the analogy with the objects of the real world.

In summary, the object notion is characterized by

- an operational interface; in the object-oriented terminolgy, “the set of messages the object can understand”;
- a hidden implentation of the operations;
- an internal state, usually hidden, which influences the behaviour of the object, that is, answers to messages (different invocations of the same operation may produce different answers).

**Class-based paradigm in traditional languages**  
In many situations it can be useful to have different “instances” of the same object schema; referring to the preceding example, to have many stacks. This can be achieved in a class-based language, that is, a language where it is possible to write software modules which correspond to classes, that is, object schemate which can be instantiated.

For instance, in Ada a class can be simulated with a generic package, (that is, a parametric module), with no parameters, as shown below.

```ada
generic package STACK -- models a stack schema  
... as above
```

A parametric module as above can be used as follows:

```java
use STACK;
package STACK1 = new STACK;
package STACK2 = new STACK;
...
STACK1.Push(2); -- the message Push(2) is sent to the object STACK1
STACK2.Push(2); -- the message Push(2) is sent to the object STACK2
...
```

Hence, Ada can be considered a class-based language. However, Ada objects are not “first class values”, that is, it is not possible to manipulate them as other values. Indeed, it is not possible to assign objects to variables, pass objects as parameters, and so on. Moreover, it is not possible to create class instances dynamically, but objects are static entities (there exist one object for each package declaration which is an instantiation of the generic package).

These features (objects = first class values, dynamic creation) are present instead in the class notion of the object-oriented languages, which we will present in the next section.

## 2 Objects and Classes

In this section we will present, using Java syntax, the ingredients of a class-based language: class definitions, fields and methods, object creation, object sharing, object identity. We will also compare the object notion with the pointer notion. See also [2] Sec.2.1, [1] Chapter 1, Sec.2.1.
2.1 Classes

This is an example of class declaration.

```java
class Rectangle {
    int length, width;
    void setLength (int l) {length = l;}
    void setWidth (int w) {width = w;}
    int area () {return width * length;}
    void print ()
        {System.out.println("I am a rectangle: = " + length + ", " + width);} 
}
```

As shown by the example, a class defines an object’s schema. The components of a class definition are of two main kinds: the variables which compose the state, called either attributes or fields or instance variables depending on the language, and the operations, called methods. In the example, the fields are `length` and `width`, and the methods are `setLength`, `setWidth`, `area`, and `print`.

The class `Rectangle` introduces both the object type `Rectangle` and a schema (template) for creating objects of type `Rectangle`, called instances of the class.

Note that the fact that a class is also used as a type happens in Java and in most widely used object-oriented languages. However, we will see in Sec.5.5 that it is possible to design object-oriented languages where the notion of class and type are independent. This is an example of how to use a class declaration.

```java
Rectangle r=new Rectangle();
r.setLength(2);r.setWidth(3);r.print();
```

The execution of this fragment will produce:

I am a rectangle: = 2, 3

In the first line a variable of type `Rectangle` is declared, which is initialized to a new instance of the class. Each object of class `Rectangle` has its own state (fields `length` and `width`). The terminology `instance variables` indeed refers to the fact that each instance of the class has its own copy of these variables.

Usually, the object’s state should be not visible to the outside, but should be inspected and modified only by calling methods. In some object-oriented languages (e.g., Smalltalk) attributes are always hidden, while in other languages the choice is left to the programmer (e.g., in Java by means of the `private` modifier).

Each object has an operational interface, which describes possible interactions with the outside world (names and parameter/return types of methods).

The “dot notation” `r.setLength(2)` suggests that the invoked method “belongs” to the receiver object `r`. Note indeed that methods correspond to functions/procedures in traditional languages, but with an argument (the first) which plays a special role, since the definition of the method to be invoked can be found in the class of this object.

All objects of the same class have the same operational interface and the same behaviour, that is, the same implementation of methods.

2.2 Equality and copy

Each object has its own identity which is preserved during execution.

```java
Rectangle r1=new Rectangle();
Rectangle r2=new Rectangle();
Rectangle r3=r1;
```

After the execution of this code fragment, there exist two objects of class `Rectangle`; let us call them `r`, `r'`. The variables `r1` and `r3` denote the rectangle `r`, while the variable `r2` denotes the rectangle `r'`. These two rectangles have the same state, but different identities (in Java notation, `r1 != r2` and `r1 == r3` hold).

Informally, the execution model of a program in an object-oriented language is an evolving universe of objects. At each stage in the execution there exist a finite number of objects, each of them having an identity and an internal state. The universe may change in three ways:

- creating new objects
- modifying the state of existing objects
deleting existing objects. However, this last operation is usually done in object-oriented languages by an automated garbage collector.

In object-oriented languages we can define two different notions of equality. The former corresponds to equality of identities and is usually predefined in the language (for instance, in Java it is denoted by ==). Two expressions are equal in this sense if and only if they denote the same object, as r1 and r3 in the example; at the implementation level, indeed, they return the same address. As a matter of fact, implementation of object-oriented languages typically uses an object table, where each object has a unique object identifier (oid). When a new object is created a new row is allocated in the table.

The latter notion of equality, on the contrary, corresponds to some desired relation between the object states, and must be defined and implemented by the user.

For instance, in the class Rectangle we can add an equality test.

```java
class Rectangle{
    ... 
    boolean equals (Rectangle r) {
        return length == r.length && width == r.width;
    }
    ...
}
```

Among the various equalities one can define on object states, two are of special interest:

1. two objects are equal if their corresponding fields are identical, that is, denote the same value of a primitive type or object,
2. two objects are equal if their corresponding fields denote either the same value (if they are of a primitive type) or objects which are (recursively) equal in the same sense.

Note that the two definitions coincide in the case in which all attributes are of primitive types, as it is in the Rectangle case. Note also, in the above example, that the fact that in an object-oriented language methods have a privileged argument, the first (the receiver of the message), forces to write in an “asymmetric” way even operations with two arguments which are intuitively symmetric, as equality (methods corresponding to these operations are sometimes called binary methods). This aspect of the object-oriented languages is sometimes criticized; in some languages, as Java, it is possible to define operations without a privileged argument by using class methods, see Sec.2.5. We will analyze in detail type problems related to binary methods in Sec.5.3.

The assignment to variables of an object type has an effect which is analogous to that of the assignment to variables of a pointer type in an imperative language. After the assignment

```
Rectangle r3=r1;
```

r3 and r1 denote the same object. If, on the contrary, we want to assign to a variable a copy of an existing object, we need to use a user-defined method.

```
Rectangle r3= r1.copy();
```

After this assignment, r3 and r1 do not denote the same object (r3 == r1 does not hold), but r1.equals(r3) holds.

A possible definition of the method copy is the following:

```java
Rectangle copy () {
    Rectangle r = new Rectangle ();
    r.length = length;
    r.width = width;
    return r;
}
```

(A more compact definition can be written in Java using a constructor.)

Object-oriented languages usually provide a predefined copy operation. In Java, for instance, Object clone() is a method of the class Object (the root of the class hierarchy, see Sec.3.2) which performs a shallow copy of the object on which it is invoked; that is, returns a new object whose fields are exactly the same of the receiver. In other words, a shallow copy is equal in sense (1) to the receiver.

A deep copy of an object, instead, is a new object whose fields are (recursively) deep copies of the corresponding fields of the receiver, that is, a deep copy is equal in sense (2) to the receiver. Of course deep copy and shallow copy coincide in the case in which all fields are of primitive types, as it is for the class Rectangle. Deep copy, if desired, must be implemented by the programmer, for instance in Java redefining the method clone.¹

In summary, the class notion in object-oriented languages can be characterized as follows.

¹In this case, however, the programmer must take into account that the method clone returns an Object, hence use cast to get an object of the desired type.
A class defines a schema for creating objects (the instances of the class).

Objects of the same class share both the operational interface and the implementation.

Objects are “first class values”, that is, can be manipulated exactly as primitive values are.

Instances of a class are dynamically created.

2.3 Recursion among classes, objects versus pointers

We say that a class $A$ is client or uses a class $B$ if $B$ is used inside $A$ as either type or an object schema (that is, for creating instances).

The clientship relation can be either directly or mutually recursive. This typically happens in classes which implement recursive data structures as lists or trees. Note that in an object-oriented language recursion may take place in three different situations:

- recursion among classes;
- (in case of recursion among classes) recursion among objects; for instance, if classes are $A$ e $B$ mutually recursive in the sense that an object of class $A$ has a subcomponent of type $B$ and conversely, then an object $o$ of class $A$ may have a subcomponent of type $B$ which has as subcomponent, in turn, $o$;
- recursion among methods.

The fact that objects may have subcomponents which are in turn objects may lead to sharing of subcomponents, hence aliasing. In this sense the object paradigm is analogous to the imperative paradigm with pointers. However, there are some differences which make the object paradigm cleaner.

- In an object-oriented language it is possible to directly express recursive data type definitions by recursive classes.
- In an imperative language we must declare a pointer type $P$ to a type $T$, while in an object-oriented language we only declare an object type. As a consequence, in an imperative language it is possible, given an expression of type $P$, to access the corresponding value of type $T$ by a “dereferencing” operator (denoted, e.g., by * in C). For instance, given two variables $p, q$ of a pointer type, it is possible to perform an assignment $*p = *q$. In an object-oriented language, instead, the state of objects can only be accessed indirectly via method calls (or selection of visible fields, if any).
- In an object-oriented language the variables of object types are usually automatically initialized to the value “undefined object” (null in Java), or initialization is required by the compiler. Moreover, the dynamic deallocation is performed by an automated garbage collector. These two features avoid a number of programming errors. As discussed above, it is still left to the programmer the definition of copy methods.

2.4 this

All object-oriented languages provide a keyword (this in Java and C++, self in Smalltalk, current in Eiffel, etc.) used as a constant identifier which denotes, during the execution of a method’s body, the object on which the method has been invoked. In Java it is also possible to refer to this implicitly; for instance this method definition

```java
boolean equals (Rectangle r) {
    return length == r.length && width == r.width;
}
```

is equivalent to the following

```java
boolean equals (Rectangle r) {
    return this.length == r.length && this.width == r.width;
}
```

In some situations, however, it is necessary to explicitly use this. A first case is when there are variables which hide instance variables.

```java
class Rectangle {
    int length;
    ...
    void setLength (int length) {
        this.length = length;
    }
}
```
A more significant situation is when we need to pass the current object as argument or return as result of a method call. For instance, assume that we want to add in the class Rectangle a method which compares the current rectangle with another and returns that which has greater area:

```java
Rectangle greaterArea(Rectangle r) {
    if (area() > r.area()) return this;
    else return r;
}
```

### 2.5 Class variables and methods

In many object-oriented languages (including Smalltak and Java) it is possible to define class variables and methods. In Java these fields and methods are also called static because the binding for them is static (see Sec.3.7).

Assume, for instance, that we want to add in the class Rectangle a field which has as value the number of existing instances of the class. Clearly, this is not a property of a single object of the class, but rather of the class itself.

```java
class Rectangle{
    static rectNum = 0;
    static void printNofRects(){
        System.out.println("Alive rectangles: "+ rectNum);
    }
    Rectangle(int l, int w) {
        length = l;
        width = w;
        rectNum++;
    }
}
```

The class field rectNum is incremented each time a new rectangle is created (invoking the constructor). Differently from what happens for instance variables, there exists only one copy of class variables, which is shared among all the existing instances of the class. In the example, there exists only one copy of rectNum. Analogously, a class method is not invoked on a single object, but on the class itself. This is expressed by the syntax of the call.

```java
Rectangle.printNofRects(); /* to be preferred */
r.printNofRects(); /* allowed but to be avoided */
```

In the body of a class method, obviously, it is not possible to refer to instance variables, but only to class variables, and it is not possible to use either explicitly or implicitly this. In the body of an instance method, on the contrary, it is possible to refer to both instance and class variables and methods.

Class variables are allocated and initialized when the system loads the class. Class methods can also be used for simulate usual data types, defining classes without state which will never be instantiated, but only used as collections of operations. For instance, the class Math in package java.lang defines the usual mathematical operations as class methods.

### 2.6 An algorithm based on objects

We show now an example of how a classical algorithm can be expressed in an unusual style using the computational object paradigm. The problem is that of finding all prime numbers less than a given number, and the algorithm is that known as “Erathostenes sieve”, which consists in canceling from a table which contains all the numbers all those which are multiple of 2, then multiples of 3, then multiples of next non canceled number, that is, 5, and so on. When there are no longer numbers to test, remaining numbers are prime. In the version based on objects, the algorithm is implemented using many “sieve” objects, each of them representing a prime number. Each sieve receives some numbers, and the task of the sieve which represents prime number p is to filter these numbers only keeping those which are non multiple of p. The situation is depicted in the figure.

```
myPrime 2
  next
myPrime 3
  next
myPrime 5
  next
  ...
myPrime p
```

Sieve objects are dynamically generated; indeed, if a number can pass throughout all existing sieves, then we can conclude that it is prime, and we add at the end of the sequence a new sieve corresponding to this number. An object of class PNG (for Prime Numbers Generator) starts the algorithm generating the numbers to be tested and creating the first sieve.
class Sieve {
    int myPrime;
    Sieve next;
    static Stack myStack = new Stack();

    void filter (int i){
        if (myPrime == 0) {
            myPrime = i;
            next = new Sieve();
            myStack.push(i);
        }
        else if (i%myPrime!=0) next.filter(i);
    }
}

class PNG{
    public static void main (String argv[]){
        Sieve first = new Sieve();
        for (int i=2; i < 100 ; i++)
            first.filter(i);
        Sieve.myStack.print();
    }
}

Note that the use of a class variable myStack, whose class Stack we assume to be implemented elsewhere, allows to model in a natural way the fact that the stack which collects prime numbers is the same for all sieves. Note also that the way algorithm is implemented makes actually possible for sieves to work “in parallel” (this could be done in Java by using threads).

3 Inheritance

In this section we illustrate the inheritance notion in object-oriented languages, motivating its importance for software incremental development and reuse. See also [2] Sec.2.2, [1] Sec.2.2, 2.3, 2.4.

3.1 Heir classes

A class definition as the following

class Cuboid extends Rectangle {
    private int height;
    void setHeight (int h) { height = h;}
    int volume () { return area () * height;}
}

defines the class Cuboid as heir (subclass) of the Rectangle class, which is called parent class (superclass). We also say that Cuboid inherits from Rectangle. The definition of Cuboid is obtained by extending the definition of Rectangle with a new field height and two new methods setHeight and volume, as it would happen by copying the definitions of the Rectangle components in the new class. Intuitively, we want to define a cuboid as a specialization of a rectangle, that is, an object which has all the features a rectangle has plus others.

Class Cuboid reuses the code of Rectangle and extends it. At the implementation level, methods of Rectangle are not duplicated in Cuboid, but are retrieved from the parent class when a method is invoked. Note that this also holds in absence of the source code for Rectangle; referring to Java, it is enough that the Rectangle.class file is available.

Note that a new method, e.g., volume, can use in its body both old (area) and new (height) components. The two classes can be used as illustrated by the following example.

class RectCuboidTest {
    public static void main(String argv[]) {
        Rectangle r = new Rectangle();
        ...
        Cuboid c = new Cuboid();
        c.setLength(3); c.setWidth(3); c.setHeight(3);
        r = c;
        ...
    }
};
The assignment \( r = c \) illustrates an important distinguishing feature that in object-oriented language is coupled with inheritance: an expression of an heir type (subtype) can appear in a context of the parent type (supertype). For instance, an expression of type `Cuboid` can be assigned to a variable of type `Rectangle`. Intuitively, indeed, this assignment is correct since `Cuboid`, being a heir of `Rectangle`, has all its methods, hence a cuboid can understand (at least) all messages a rectangle can understand; an assignment in the converse direction \( c = r \), instead, could cause a run-time error, for instance in the case of a subsequent invocation \( c\text{.volume()} \).

Note that, since we are considering until now languages where class names also play the role of types, the subtyping relation is determined by the inheritance hierarchy. We will see in Sec.5.5 that is to possible to design object-oriented languages where the notion of class and type (hence inheritance and subtyping) are independent.

Note, moreover, that after the assignment \( r = c \) we have, in a sense, “lost some information”, more precisely we have lost the type information that \( r \) denotes an object of class `Cuboid`. A further invocation \( r\text{.volume()} \) would be, indeed, not statically well-formed, even though sensible at run-time.

Hence, when we use an object of a subtype \( H \) in a context of a supertype \( P \), this object can be manipulated only via the (less rich) interface offered by \( P \).

The fact that an expression of a subtype can be used also in contexts of a supertype is usually expressed by the following typing rule said subsumption rule

\[
\vdash e : T \quad \vdash T \leq T' \quad \vdash e : T'
\]

where \( \vdash e : T \) means that the expression \( e \) has type \( T \) and \( \vdash T \leq T' \) means that \( T \) is a subtype of \( T' \).

However, we will see in Sec.4.5 that subsumption rule in the general form above does not hold in Java.

Note that the subtype notion and subsumption rule are not peculiar to object-oriented languages; for instance in Pascal an interval type, e.g., `type digit = 0 .. 9` is a subtype of the corresponding scalar type `integer`, hence an expression of type `digit` may be used everywhere an `integer` is expected.

### 3.2 Overriding

In the preceding example the class `Cuboid` was defined by simple extension of `Rectangle`, that is, inheriting their methods without modifying the definitions. However, in general a heir class can redefine parent’s methods, as illustrated by the example below.

```java
class Rectangle {
    ...
    void init () { length = 1; width = 1; }
    void print () {
        System.out.println("I am a rectangle: = " + length + ", " + width);
    }
}

class Cuboid extends Rectangle {
    ...
    void init () { length = 1; width = 1;height = 1; }
    void print () {
        System.out.println("I am a cuboid: " + length + ", " + width + ", " + height);
    }
}
```

We say that methods `init` and `print` are overridden (redefined). In this case obviously at the implementation level the two classes will not share the code for these methods.

In case of redefinition there is the problem that, given a method call, we have to decide which is the version which must be invoked. Usually in object-oriented languages the so-called late binding (dynamic binding, dynamic method-dispatch), is supported, that is, the version to be invoked is determined by the dynamic type of the receiver, that is, the class of the object currently denoted by the receiver expression, while the static type is the type which can be assigned to the expression by checking the code at compile-time.

Hence the decision on the method version to be selected cannot be taken at compile-time. Consider the following example:

```java
Rectangle r = new Rectangle(); ...
r.print(); //print in Rectangle is invoked
Cuboid c = new Cuboid(); ...
r = c;
r.print(); //print in Cuboid is invoked
```
In the last line \( r \) has static type `Rectangle` (the type assigned to the variable in the declaration), dynamic type `Cuboid`. Verification of static well-formedness of code is made at compile-time; in particular, the call `r.print()` is considered well-formed since class `Rectangle` has a method `print()`. However, at run-time the method which is invoked is that in `Cuboid`. If the method version to be invoked was determined at compile-time (static binding), then in the example the method in `Rectangle` would be selected.

The static type of an expression is determined, as already said, at compile-time, hence does not change during execution, while dynamic type may change; due to typing rules, however, the dynamic type always must be a subtype of the static type. At the implementation level, the version of the method to be executed is determined as follows: given a call \( o.m(...) \), first of all the dynamic type of \( o \) is determined, say `C`. If `C` contains a definition for `m`, then the method look-up is terminated; otherwise, the method is searched in the parent class of `C`, and so on, until the class `Object` (the root of the inheritance hierarchy) is reached.

Note that the inheritance hierarchy, indeed, turns out to be a tree in the case that we have considered of single inheritance (each class has exactly one parent, except `Object` which has no parent class).

This is depicted in the following figure.

Method look-up as described above can fail, that is, reach `Object` without finding a definition of the method which is searched. For instance, this can happen in an untyped language as Smalltalk\(^2\), where it is possible to assign to a variable an arbitrary object, hence, given a call to a method \( m \) where the receiver is denoted by a variable \( x \), there is no way of foreseeing whether, at execution time, \( x \) will denote an object which can understand the message \( m \) (that is, an instance of a class which has \( m \) among its either directly declared or inherited methods). If the method look-up fails, then an error is raised called “message not understood”.

In a typed language, typing rules should prevent this situation; if this is the case, then we say that the type system is *sound*.

In Java the type system is sound; however, some typing errors are only detected at run-time, as in down-casting (see Sec.5.2).

### 3.3 Inheritance vs “programming by cases”

We illustrate now by an example how inheritance, coupled with dynamic binding, allows, in the implementation of “union” types, to drastically change organization of code and to achieve better modularity and extensibility.

By “union” type we mean a type whose set of values is the disjoint union of some sets. Assume, for instance, that we want to implement a data type “geometric figure” and a figure can be either a rectangle, or a circle, or a triangle.

The standard way to implement a union type in a language which is not object-oriented is by ad-hoc types, as variant record types in Pascal, unions in C, union types in ML. For instance in Pascal:

```pascal
type FigureKind = (Rectangle, Circle, Triangle);

type Figure = record
  case kind : FigureKind of
    Rectangle : length, width : real;
    Circle : radius : real;
    Triangle : basis, height : real;
end
```

An operation which manipulates figures, for instance a function which calculates area, is then typically implemented by a “by cases” definition, using an ad-hoc construct (case in Pascal, switch in C, pattern-matching in ML).

```pascal
function area (f: Figure) : real =
  case f.kind of
    Rectangle : area:= f.length * f.width;
    Circle : area:= f.radius * f.radius * Pi;
    Triangle : area:= f.basis * f.height /2
```

\(^2\)Note: untyped means that language expressions are untyped, but objects still are instances of a unique class.
In an object-oriented language, instead, the standard implementation of a union type consists in defining a parent class (abstract if possible in the language, see Sec.3.5), corresponding to the union type, with a subclass for each existing case. All the operations of the data type which are different in the various cases will be implemented with methods which are not defined (abstract in Java) in the parent class and defined in the appropriate way in the heir class. Obviously, operations whose definition is the same in all or most cases will be implemented in the parent class and redefined only in those subclasses where the behaviour is different.

In our example:

```java
abstract class Figure {
    ...
    abstract double area ();
}

class Rectangle extends Figure {
    private double length;
    private double width;
    double area () {
        return length * width;
    }
}

class Circle extends Figure {
    static final double Pi = 3.1416;
    private double radius;
    double area () {
        return radius * radius * Pi;
    }
}

class Triangle extends Figure {
    private double basis;
    private double height;
    double area () {
        return basis * height /2;
    }
}
```

In this way, a client class can use the abstract class `Figure` as a type, and assign to variables of this type objects of the desired subtype. A call of the method `area ()` will cause the execution of the “right” version each time automatically, thanks to dynamic binding.

```java
class FigureTest {
    public static void main(String argv[]) {
        Figure[] r = new Figure[10];
        ...
        double areaTot = 0.0;
        for (int i = 0;i < r.length;i++)
            areaTot += r[i].area();
    ...
    }
}
```

Note that it is possible to manipulate an array of heterogeneous figures. An external user should not care about the version of the method which is executed at each iteration, and can even ignore which kinds of figures actually exist.

The most important difference between the two approaches, however, is related to the way in which an extension of the original type is handled. Assume, for instance, that later in the development we want to add a new category of figures, say, ellipses. In the traditional approach it is clear that it is necessary to change the definitions of the types `FigureKind` and `Figure` and the code of `area`, and analogously that of all the operations which were defined by cases.

In the object-oriented approach, instead, it is enough to define a new heir class `Ellipsis` of `Figure`, with its own definition of the method `area ()`: it is not necessary to modify the existing code. It is clear the enormous advantage of this solution. This possibility the object-oriented approach offers to extend functionalities of the code without need of modifying already written code is also called open-closed code principle. Indeed, it is possible to write software modules which are closed in the sense that they are complete and can be used, but open in the sense that it is always possible to add new functionalities (and without even having the source code available).
In [7] an interesting schema is proposed which illustrates the difference between the “by cases” approach (called ADT by Cook, in the sense that it is the approach taken in languages which follow the ADT paradigm, see Sec.1.2), and the object-oriented approach.

The starting remark is that often in a data type operations can be classified into two categories, constructors and observations. For instance, in the case of lists, constructors are empty and cons, observations are isEmpty, head, tail. The data type can, then, be described by a table which indicates for each constructor the corresponding results of observations.

For instance, for lists the table will be as follows.

<table>
<thead>
<tr>
<th>observations</th>
<th>constructor of l</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>true</td>
</tr>
<tr>
<td>cons(e, l')</td>
<td>false</td>
</tr>
<tr>
<td>head(l)</td>
<td>undefined</td>
</tr>
<tr>
<td>tail(l)</td>
<td>undefined</td>
</tr>
</tbody>
</table>

In case of observations with many arguments the table will have more than 2 dimensions, for instance this is the table corresponding to the observation eq: list list → bool:

<table>
<thead>
<tr>
<th>constructor of l1</th>
<th>constructor of l2</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>true</td>
</tr>
<tr>
<td>cons(e1, l1')</td>
<td>false</td>
</tr>
<tr>
<td>cons(e2, l2')</td>
<td>false</td>
</tr>
<tr>
<td>l1 = e1 ∧ eq(l1', l2')</td>
<td></td>
</tr>
</tbody>
</table>

The ADT approach corresponds, then, to decompose the table “by rows”; indeed a separate function is defined for each row. The object-oriented approach corresponds, on the contrary, to decompose “by columns”, since a heir class is defined for each constructor.

For instance, here below there is an implementation of lists in Java which corresponds to the first table (we have omitted for simplicity handling of error situations to be performed in Java by exceptions, see Sec.5.1.).

```java
abstract class List {
    abstract boolean isEmpty ();
    abstract int head ();
    abstract List tail ();
}

class NonEmptyList extends List {
    private int head; 
    private List tail;
    NonEmptyList (int head, List tail) {
        this.head = head; this.tail = tail;
    }
    boolean isEmpty () { return false; }
    int head () { return head; } 
    List tail () { return tail; }
}

class EmptyList extends List {
    boolean isEmpty () { return true; }
    int head () { ... error }
    List tail () { ... error }
}
```

In both cases encapsulation can be enforced, in the sense that data can be constructed only via constructors and manipulated only via observations.

Let us discuss now what happens in both approaches if we add new operations.

Adding constructors Assume to add a new constructor, say interval: int int → list which constructs the lists corresponding to the intervals \([n, m]\) of \(\mathbb{Z}\) \((n \leq m)\), so that we can optimize the representation of these special lists (indeed it is enough to store \(n\) and \(m\)).

The table must be extended as follows.

---

3 Not to be confused with Java constructors.
The object-oriented approach turns out to be much more convenient, as already noticed; indeed in the ADT paradigm it is necessary to change the definition of the type and to change (adding one case) the definitions of all the observations. In the object-oriented approach, indeed, it is enough to add a new class corresponding to the new constructor.

For instance in Java we can simply add the following new class.

```java
class IntervalList extends List {
    private int a, b;

    public IntervalList(int a, int b) {
        if (b < a) ... error
        else {
            this.a = a; this.b = b;
        }
    }

    public boolean isEmpty() { return false; }
    public int head() { return a; }
    public List tail() {
        if (a < b) return new IntervalList(a + 1, b);
        else return new EmptyList();
    }
}
```

Again, we have omitted handling of errors, to be performed in Java by exceptions, see Sec.5.1.

**Adding observations** Assume now to add an observation $\text{length} : \text{list} \rightarrow \text{int}$ which returns the length of the list.

```plaintext
<table>
<thead>
<tr>
<th>observations</th>
<th>constructor of l</th>
<th>interval(n, m)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isEmpty(l)</code></td>
<td><code>true</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>head(l)</code></td>
<td><code>undefined</code></td>
<td><code>e</code></td>
</tr>
<tr>
<td><code>tail(l)</code></td>
<td><code>undefined</code></td>
<td><code>l'</code></td>
</tr>
<tr>
<td><code>length(l)</code></td>
<td><code>0</code></td>
<td><code>length(l') + 1</code></td>
</tr>
</tbody>
</table>

First of all, in both approaches, the new observation can be defined either as a new primitive operation (that is, by cases, as shown above), or as a derived operation, that is, in terms of the previously defined observations, e.g.,

\[
\text{length}(l) = \text{cond}(\text{isEmpty}(l), 0; \text{length}(\text{tail}(l)) + 1).
\]

A definition as new primitive operation often offers an advantage in terms of optimization; in the example, in the case of interval lists.

In the ADT paradigm adding a new operation is simple, and consists in adding a new function either defined by cases or derived.

In the object-oriented paradigm there are two different possibilities:

- add the definition as derived operation in the parent class, as shown below:

  ```java
  abstract class List {
      ...
      int length() {
        if (isEmpty()) return 0;
        else return tail().length() + 1;
      }
  }
  ```

  (only one class is modified, but no optimization is possible);
```
• add a definition as primitive operation in each heir class, as shown below

    abstract class List {
        ...
        abstract int length();
    }

    class NonEmptyList extends List {
        ...
        int length() { return 1 + tail.length(); }
    }

    class EmptyList extends List {
        ...
        int length() { return 0; }
    }

    class Interval extends List {
        ...
        int length { return m-n+1; }
    }

    (all classes are modified, but optimization is possible).

Also an intermediate solution is possible (actually the best), that is, to give the derived definition in the abstract class and redefine it only in those heir classes where redefinition gives an advantage in terms of efficiency (as for the interval lists).

**Exercise 1** Consider the following Java implementation of binary trees with integer labels (we omit for simplicity handling of error situations by exceptions).

    abstract class BTree {
        abstract boolean isEmpty ();
        abstract int label ();
        abstract BTree left ();
        abstract BTree right ();
    }

    class Node extends BTree {
        private int label;
        private BTree left, right;
        Node (int label, BTree left, BTree right) {
            this.label = label; this.left = left; this.right = right;
        }
        boolean isEmpty () { return false;}
        int label () { return label; }
        BTree left () { return left;}
        BTree right () { return right;}
    }

    class Empty extends BTree {
        boolean isEmpty () { return true;}
        int label () { /*/error */}
        BTree left () { /*/error}
        BTree right () { /*/error}
    }

    Be careful: do not confuse an empty tree (modeled by an instance of the class Empty) with null.

1. add in BTree a non-abstract method which returns the height of a tree (length of a maximal path, 0 if empty);
2. add in BTree an abstract method which returns the number of nodes of a tree, and implement the method in the subclasses;
3. add a heir class Leaf whose instances represent leaves;
4. define a method (be careful: where and of which kind?) which given two integers $n$ and $m$ returns a binary tree (preferably with minimal height) having as (distinct) labels numbers from $n$ to $m$ (hence empty if $n > m$).
3.4 Wrapper classes

As further example illustrating the difference between the ADT and the object-oriented approach, we show two examples of class whose instances simulate the behaviour of values of primitive types (these classes are sometimes said "wrappers"). As first example we define an interface (see Sec.3.5) $\mathbf{Bool}$ whose objects answer to the messages $\text{not}$, $\text{and}$ and $\text{or}$.

```java
interface Bool {
    Bool not ();
    Bool and (Bool b)
    Bool or (Bool b)
}
```

class True implements Bool {
    Bool not () {return new False();}
    Bool and (Bool b) {return b;}
    Bool or (Bool b) {return this;}
}

class False implements Bool {
    Bool not () { return new True(); }
    Bool and (Bool b) { return this;}
    Bool or (Bool b) { return b;}
}

Note that, while boolean values are only two, it is possible to create many instances of $\mathbf{True}$ and $\mathbf{False}$.

Analogously, we can define an abstract class $\mathbf{Nat}$ whose instances behave as natural numbers. Natural numbers can be inductively defined as follows:

- zero is a natural number,
- the successor of a natural number is a natural number.

Applying the standard methodology for implementing union types described above, we get the following definition.

```java
abstract class Nat {
    abstract int val ();
    abstract Nat add (Nat n);
    abstract Nat mult (Nat n);
    abstract boolean isZero ();
    abstract boolean equals (Nat n);
    ...
}
```

class Zero extends Nat {
    int val () { return 0;}
    Nat add (Nat n) { return n;}
    Nat mult (Nat n) { return this;}
    boolean isZero () { return true; }
    boolean equals (Nat n) { return n.isZero();}
}

class Succ extends Nat {
    private Nat pred;
    Succ (Nat n) { pred = n;}
    int val () { return pred.val() +1;}
    Nat add (Nat n) return new Succ(pred.add(n));
    Nat mult (Nat n) { return pred.mult(n).add(n);}
    boolean isZero () { return false; }
    boolean equals (Nat n) {
        if (n.isZero()) { return false; }
        else { return pred.equals(((Succ) n).pred);}
    }
}

Note in the definition of $\text{equals}$ that we need a down cast for accessing the field $\text{pred}$ of $n$.

Note also that in the version proposed the sum, product, value and equality operations are all defined as primitives, that is, by cases in heir classes (see discussion in Sec.3.3).
Exercise 2 Write a version in which all these are defined as derived operations in the abstract class by only using as primitive operations (hence `abstract` in `Nat`): `bool isZero()` and another operation `Nat pred()` which returns the predecessor.

3.5 Interfaces and abstract classes

As already mentioned in Sec.2.1, we are considering until now languages where a class plays the double role of a type and a schema for instantiating objects.

Actually, these two notions are in principle independent: the type of an object corresponds to its operational interface (name and types of components which can be used by a client), while the class provides a particular implementation of this interface. Hence, many classes could implement the same object type. We will discuss more in detail languages which support this distinction in Sec.5.5.

However, also languages where classes (schemata) are types as Java partially support this point of view, in the sense that it is also possible to declare “pure types”. An example are Java interfaces, as that shown below, where it is only possible to declare `abstract` (that is, with no body) methods (and constants).

```java
interface Stack {
    boolean isEmpty ();
    int pop();
    void push (int e);
    void print ();
}
```

It is possible to declare many classes which `implement` the same interface (conversely, a class may implement many interfaces).

```java
class StackByArray implements Stack { ... };
class StackByList implements Stack{ ... };
```

Obviously, a class which implements an interface must have all methods declared in the interface. An interface cannot be used as a schema (that is, for creating instances), but using an interface as a type allows to write code which is independent from the implementation, and also to mix different implementations, as shown below.

```java
class StackTest{
    public static void main (String argv[]) {
        Stack s = new StackByList ();
        for (int i=1;i<20;i++) s.push(i);
        for (int i=1;i<5;i++) s.pop();
        s = new StackByArray () ;
        for (int i=1;i<20;i++) s.push(i);
        for (int i=1;i<5;i++) s.pop();
    }
}
```

Note also that the interface notion allows a limited form of `multiple inheritance` (that is, the situation when a class can have many parents), hence from a methodological point of view is useful when we want to describe objects which in a sense “combine” features of two different kinds (as in the example below). Indeed in this case conflict problems which are typical of multiple inheritance do not arise.

```java
interface Figure {
    void Move (double dx, double dy);
    void Draw ();
    void UnDraw();
}
interface ColoredObject {
    void ChangeColor (color c);
}
class ColoredFigure implements Figure, ColoredObject { ... }
```

Java (and analogously many other object-oriented languages) also provides an `abstract class` notion which is somehow intermediate between that of interface and of (non-abstract) class. In an abstract class it is possible to declare `some` methods as abstract (other terms are `deferred` in Eiffel, `pure virtual` in C++).

From the methodological point of view, abstract classes are mainly useful when we want to define derived operations on top of a kernel of primitive operations, that is, independently from the implementation of these primitives.

Consider, for instance, the example on lists in Sec.3.3.

Exercise 3 Add in this example operations in the abstract class for equality, appending a list to another, reversing a list.
In Java it is not allowed to create instances of abstract classes (in this way it is guaranteed that a body is always found in method look-up at run-time), while in other languages, e.g., Smalltalk, this is permitted (hence the selection of an abstract method in a method call leads to a run-time error).

Note, however, that, differently from interfaces which are only part of the subtyping hierarchy, abstract classes are part of the inheritance hierarchy, hence they are taken into account in method look-up (in Java it is even allowed to redefine a method as abstract) and in execution of constructors (which can be declared).

### 3.6 super

Many object-oriented languages including Smalltalk and Java allow the possibility of referring, in a method body, to the methods of the parent class by using the `super` keyword. When `super` is used as receiver in a method invocation, the effect is the same as in call with receiver `this`, except that the method look-up starts in this case from the parent class. If no method body is found, then the search goes up in the inheritance hierarchy as in the normal case.

In this way it is possible to “recover” the version of the method in the parent class even in the case when the method has been redefined.

From a methodological point of view, `super` is typically used in case of “conservative” redefinition, that is, when the body of the method in the heir class performs all the actions the parent’s version did, plus possibly others, as in the example below.

```java
class Cuboid extends Rectangle {
    ...
    boolean equals (Cuboid c) {
        return super.equals(c) && height == c.height;
    }

    void init () {
        super.init ();
        height = 1;
    }
}
```

In order to precisely describe the semantics of `super`, the following points need to be clarified:

- binding for `super` is static, that is, in a call `super.m(...)`, method look-up starts from the superclass of that containing the currently executed method body rather than the superclass of the class which is the dynamic type of the receiver;

- this static binding only holds for the `super` call and does not affect subsequent calls in the method which has been executed as effect of the `super` call.

The following example illustrates how `this` and `super` work.

```java
class One {
    int test () { return 1;}
    int result() { return this.test(); }
}

class Two extends One {
    int test () { return 2;}
    int superTest () { return super.test(); }
    int superResult () { return super.result();}
}

class Three extends Two {
    int test () { return 3;}
    int result () { return 3;}
}

class Four extends Three {
    int test () { return 4;}
}

public class SuperTest {
    public static void main(String[] argv) {
        One o1 = new One();
    }
}
```
Two o2 = new Two();
Two o3 = new Three();
Two o4 = new Four();
System.out.println(o1.test()); //1
System.out.println(o1.result()); //1
System.out.println(o2.test()); //2
System.out.println(o2.result()); //2
System.out.println(o2.superTest()); //1
System.out.println(o2.superResult()); //2
System.out.println(o3.test()); //3
System.out.println(o3.result()); //3
System.out.println(o3.superTest()); //1
System.out.println(o3.superResult()); //3
System.out.println(o4.test()); //4
System.out.println(o4.result()); //3
System.out.println(o4.superTest()); //1
System.out.println(o4.superResult()); //4

3.7 Static versus dynamic binding, hiding versus overriding

For fields and class methods binding is static. This means that, in case the heir class declares a field with the same name or static method with the same name and parameter types of one in the parent class, the parent’s field or static method is hidden in the heir. In other words, referring for instance to fields, objects of the heir class actually have two fields, and in a field selection e.f the field which is selected depends on the static type of expression e.
This is illustrated by the following example.

class One {
    String str = "One";
    void show () {
        System.out.println(str);
    }
}
class Two extends One {
    int str = 2;
    void show () {
        System.out.println(str);
    }
    void superShow () {
        System.out.println(super.str);
        super.show();
    }
}

public class HidingTest {
    public static void main (String[] args) {
        Two two = new Two();
        One one = two;
        one.show(); //2
two.show(); //2
        System.out.println(one.str); //One
        System.out.println(two.str); //2
        System.out.println(((One)two).str); //One
        two.superShow(); //One, One
    }
}

Note that the fact that binding for fields is static, whereas binding for (instance) methods is dynamic can be confusing, since accessing a field directly or via a method call we can get two different results, as shown by the example above. However field hiding was included, e.g., in Java design in order to allow adding a new field in a parent class without breaking well-formedness of the heirs. Note, finally, that adopting the “purist” solution which consists in always declare fields private and defining accessors methods the problem is avoided.
Exercise 4  Consider the following Java classes:

class Parent {
    int k = 1;
    int m () { return 1;}
    int g () { return m(); }
    int km () { return k; }
}

class Heir1 extends Parent {
    int f () { return m();}
    int m () { return 2;}
}

class Heir2 extends Parent {
    int k = 2;
    int f () { return super.g(); }  
    int m () { return 3;}
    int km () { return k; }
}

class Main {
    public static void main(String[] argv) {
        Parent p; Heir1 h1; Heir2 h2;
        p = new Heir1(); h2 = new Heir2();
        ...  
        }
    }

Write, for each of the cases, what happens at compile time and what happens, in case of successful compilation, at run-time, explaining why, if we replace dots with the code below.

1. System.out.println(p.f());
2. System.out.println(p.g());
3. h1 = p; System.out.println(p.m());
4. p = h2; System.out.println(p.m());
5. System.out.println(h2.f());
6. p = h2; System.out.println(p.k);
7. p = h2; System.out.println(p.km());

4  Formal definition of a toy Java-like language

In this section we introduce a toy language MINJAVA which corresponds to a small Java subset with the essential features of an object-oriented imperative language. We provide a model of execution (operational semantics) of MINJAVA programs, a formal definition of typing rules, and we show that the type system is safe in the sense that well-typed programs never crash at run-time. For making presentation easier, we first consider a class-based version (only classes, but no inheritance) and then we add inheritance in Sec.4.5.

4.1 Syntax

The syntax of MINJAVA is given in Fig.1. A MINJAVA class declares a collection of fields and (instance) methods. Types of the language are either class names or primitive types as boolean and int (there is no void type). There is no distinction between statements and expressions, so the body of a method is just an expression. There is no explicit return keyword as in Java, but you may think of the last expression in the sequence which constitutes a method body as if it was the return expression.

Expressions are field selection, assignment to a field, method call, creation of a new object (there are no user-defined constructors, so only the constructor with no parameters exists and we omit brackets for brevity), variable (we assume that there is a distinguished variable this denoting the current object which cannot be declared as either parameter or local variable), and finally a sequence of two expressions with possibly initialization of a local variable to the result of the first expression (square brackets
\[
\begin{align*}
P & ::= \text{class}_1 \ldots \text{class}_n \\
\text{class} & ::= \text{class} \ C \ \{ \text{fields meths} \} \\
\text{fields} & ::= \text{field}_1 \ldots \text{field}_n \\
\text{field} & ::= \ T f_1 \\
\text{meths} & ::= \ \text{meth}_1 \ldots \text{meth}_n \\
\text{meth} & ::= \ T \ m (T_1 x_1, \ldots, T_n x_n) \ \{ e_f \} \\
T & ::= \ C \ | \ \text{boolean} \ | \ \text{int} \ | \ \ldots \\
e & ::= \ e_f \ | \ e_1 \ f = e_2 \ | \ e.m(e_1, \ldots, e_n) \\
& \quad | \ \text{new} \ C \ | \ x \ | \ [T x =] e_1 \ ; \ e_2 \\
& \quad \quad | \ \text{true} \ | \ \text{false} \ | \ N \ | \ \ldots
\end{align*}
\]

Figure 1: Syntax of MINIJAVA

denotes optionality). Note that variables in MINIJAVA are, in fact, constant identifiers, since we do not allow assignments to variables but only to fields (this allows a simpler formal execution model, while not affecting generality). Moreover we assume that there are expressions corresponding to operations on primitive types, e.g., boolean constants and integer literals.

As mentioned above, MINIJAVA is designed in order to be as simple as possible but to keep all essential features of an imperative class-based language; we could then add void type and statements, control structures (e.g. if statement), operations on primitive types, and so on.

Here is an example illustrating how the Java class Rectangle used in previous examples could be written in MINIJAVA syntax (assuming to have the product operation on integers).

class Rectangle {
    int length;
    int width;
    int setLength (int l) { this.length = l;}
    int setWidth (int w) { this.width = w;}
    int area () { this.width * this.length;}
}

Here is another example illustrating how mutually recursive objects can be constructed in MINIJAVA.

class A {
    int f1;
    B f2;
    B m (B b) { b.f2 = this; this.f2 = b; b; }
}
class B {
    int f1;
    A f2;
}
class Test {
    A test () {
        A a = new A;
        B b = new B;
        a.m(b);
        a;
    }
}

4.2 Execution model

We premit some notations used in the sequel.

Notations We denote by \( A \rightarrow_{fn} B \) the set of partial functions \( f \) from \( A \) into \( B \) which are defined on a finite subset of \( A \), denoted by \( \text{Def}(f) \); \( f[g] \) denotes the function which is equal to \( g \) whenever \( g \) is defined (that is, on elements in \( \text{Def}(g) \)), to \( f \) elsewhere; we write \( a_1 : b_1, \ldots, a_n : b_n \) or \( a_i : b_{i_1}^{i_1 \ldots i_n} \) to denote the function which gives \( b_i \) on \( a_i \), is undefined elsewhere.
Formal model of the object’s universe  The universe of objects we have informally described in previous sections can be modelled by a function from object identifiers into object states. That is, the set of stores is defined as follows:

\[
\text{Store} = \text{OID} \rightarrow_{fn} \text{OState}
\]

where \( \text{OID} \) is an arbitrary set, whose nature does not matter, of object identifiers. We assume that each object identifier belongs to a class, that there are infinite object identifiers for each class, and we will write \( \ell^C \) to denote an object identifier of class \( C \). Object states are finite mappings from field names (elements of \( F \)) into values, which are either object identifiers or the “null object” value or constants of primitive types (elements of \( \mathbb{Z}, \mathbb{B}, \ldots \)).

\[
\text{OState} = F \rightarrow \text{Val}
\]

\[
\text{Val} = \text{OID} \cup \{\text{null}\} \cup \mathbb{Z} \cup \mathbb{B} \cup \ldots
\]

Execution is performed w.r.t. a local state where variables denote either existing objects in the store or primitive values, that is, a local state is a function from variables into values.

\[
\text{LocalState} = \text{Var} \rightarrow_{fn} \text{Val}
\]

Exercise 5  Express in this model, given two variables \( x \) and \( y \), and a local state \( \rho \):

- object identity, and equality notions 1 and 2 between \( x \) and \( y \)
- effect of assignment \( x = y \) (not allowed in MINIJAVA except than in initialization of \( x \)), shallow copy, deep copy.

Note the difference with the model of pointers in imperative language. Here a store (abstraction of the computer’s memory) is a function from a set of locations (addresses) into values.

\[
\text{Store} = \text{Loc} \rightarrow_{fn} \text{Val}
\]

and a value stored in a location can be in turn a location.

\[
\text{Val} = \text{Loc} \cup \mathbb{Z} \cup \mathbb{B} \cup \ldots
\]

Hence, a variable \( x \) of type pointer to, say, integers, will denote a location, say \( l \), whose associated value in the store is another location, say \( l^0 \), whose associated value is an integer.

Reduction rules  We will formally model a step in the execution of a program \( P \) by a relation

\[
\sim_P \subseteq \text{Conf} \times \text{Conf}
\]

where \( \text{Conf} \) is a set of configurations, which are pairs consisting of an expression to be executed and a current store, that is, \( \text{Conf} = \text{Exp} \times \text{Store} \).

We assume that the execution starts from an expression which does not contain variables (corresponding to a main method in Java), and an empty store. The index \( P \) is necessary since during execution some information is needed on which are the fields and the method bodies of classes in the program (see rules in Fig.2).

Elements in \( \text{Exp} \) are run-time expressions, that is, expressions of MINIJAVA extended for representing intermediate steps in computation, as shown below:

\[
e ::= \ e + e | e - e | e \cdot e | e / e | e \% e | \text{new } C \mid x | \text{true} | \text{false} | \text{null} | e \rightarrow e | e \ll e | e \gg e | \text{nullExc}
\]

We denote by \( \text{nullExc} \) the error which is obtained at run-time when there is an attempt at accessing fields or methods on \( \text{null} \).

We give now, in Fig.2, the reduction rules which model execution steps. Note that we do not explicitly use local states but simply replace variables by their values.

We assume that there are no duplicate class names in a program, and no duplicate field and method names in a class (hence no overloading), and denote by

- \( \text{Fields}_P(C) \) the fields of class \( C \) in program \( P \), if any, undefined otherwise,
- \( \text{Body}_P(C, m) \) the pair \( (x_1 \ldots x_n, e) \) where \( x_1 \ldots x_n \) are the parameters and \( e \) the body of the method with name \( m \) of class \( C \) in program \( P \), if any, undefined otherwise.
In the rule for object creation, we assume that for each type there is a default value to which a field of this type is initialized. In

We show now a full example of evaluation of expression.

Consider the following MiniJava class:

```java
class A {
    int f;
}
class B {
}
```

Note that, for each kind of expression, there is one rule which corresponds to “normal” execution, possibly one rule which corresponds to run-time error (attempt at accessing a field or method on null), and some rules, marked as “propagation” rules in the figure, which express the fact that some subcomponent of the expression still needs to be evaluated. Note that evaluation takes place from left to right. We have omitted obvious rules for propagation of the `nullExc` error (Exercise: write these rules).

In the rule for object creation, we assume that for each type there is a default value to which a field of this type is initialized. In particular we assume that this initial value is null for object types, and, e.g., 0 for integers and `false` for booleans, as in Java.

We show now a full example of evaluation of expression.

Consider the following MiniJava class:

```java
class A {
    int f;
}
class B {
}
```
We show the evaluation of the following expression, starting from an empty store, in the program context consisting of the two classes:

\[
\text{B} \ b = \text{new} \ \text{B.m}(3, \text{new} \ \text{A}); \ b.\text{f2.f}=5;
\]

**Step 1:**

\[
\text{(new B,} \emptyset) \leadsto_P (\langle 3, \sigma \rangle).
\]

\[
\text{(new B.m}(3, \text{new} \ \text{A}), \emptyset) \leadsto_P (\langle 3, \text{new} \ \text{A}, \sigma \rangle).
\]

\[
(B \ b = \text{new} \ \text{B.m}(3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \emptyset) \leadsto_P (B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma)
\]

where \(\sigma = (f_1: 0, f_2: \text{null})\)

**Step 2:**

\[
\text{(new A,} \sigma) \leadsto_P (\langle 3, \sigma_1 \rangle).
\]

\[
\langle 3, \text{new} \ \text{A}, \sigma \rangle \leadsto_P (\langle 3, \text{new} \ \text{A}, \sigma_1 \rangle).
\]

\[
(B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma_1) \leadsto_P (B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma_1).
\]

where \(\sigma_1 = (f_1: 0, f_2: \text{null}), \sigma^A = (f: 0)\)

**Step 3:**

\[
\langle 3, \text{new} \ \text{A}, \sigma_1 \rangle \leadsto_P (\langle 3, \sigma_2 \rangle).
\]

\[
\langle 3, \text{new} \ \text{A}, \sigma_1 \rangle \leadsto_P (\langle 3, \sigma_2 \rangle).
\]

\[
(B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma_1) \leadsto_P (B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma_2).
\]

where \(\sigma_2 = (f_1: 3, f_2: \text{null}), \sigma^A = (f: 0)\)

**Step 4:**

\[
\text{(3; 3, \text{new} \ \text{A}, \sigma_2) \leadsto_P (\langle 3, \sigma_2 \rangle)}
\]

\[
\text{(3; 3, \text{new} \ \text{A}, \sigma_2) \leadsto_P (\langle 3, \sigma_2 \rangle)}
\]

\[
(B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma_2) \leadsto_P (B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma_2).
\]

**Step 5:**

\[
\text{(3; 3, \text{new} \ \text{A}, \sigma_2) \leadsto_P (\langle 3, \sigma_2 \rangle)}
\]

\[
\text{(3; 3, \text{new} \ \text{A}, \sigma_2) \leadsto_P (\langle 3, \sigma_2 \rangle)}
\]

\[
(B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma_2) \leadsto_P (B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma_2).
\]

**Step 6:**

\[
\text{(3; 3, \text{new} \ \text{A}, \sigma_2) \leadsto_P (\langle 3, \sigma_2 \rangle)}
\]

\[
\text{(3; 3, \text{new} \ \text{A}, \sigma_2) \leadsto_P (\langle 3, \sigma_2 \rangle)}
\]

\[
(B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma_2) \leadsto_P (B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma_2).
\]

where \(\sigma_3 = (f_1: 3, f_2: \text{null}), \sigma^A = (f: 0)\)

**Step 7:**

\[
\text{(3; 3, \text{new} \ \text{A}, \sigma_2) \leadsto_P (\langle 3, \sigma_2 \rangle)}
\]

\[
\text{(3; 3, \text{new} \ \text{A}, \sigma_2) \leadsto_P (\langle 3, \sigma_2 \rangle)}
\]

\[
(B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma_2) \leadsto_P (B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma_2).
\]

**Step 8:**

\[
\text{(3; 3, \text{new} \ \text{A}, \sigma_2) \leadsto_P (\langle 3, \sigma_2 \rangle)}
\]

\[
\text{(3; 3, \text{new} \ \text{A}, \sigma_2) \leadsto_P (\langle 3, \sigma_2 \rangle)}
\]

\[
(B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma_2) \leadsto_P (B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma_2).
\]

**Step 9:**

\[
\text{(3; 3, \text{new} \ \text{A}, \sigma_2) \leadsto_P (\langle 3, \sigma_2 \rangle)}
\]

\[
\text{(3; 3, \text{new} \ \text{A}, \sigma_2) \leadsto_P (\langle 3, \sigma_2 \rangle)}
\]

\[
(B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma_2) \leadsto_P (B \ b = (3, \text{new} \ \text{A}); \ b.\text{f2.f}=5, \sigma_2).
\]
Step 10:

\[
\frac{\langle h, f \cdot e, \sigma_3 \rangle \triangleright_{p} \langle h, \sigma_3 \rangle}{\langle h, e = 5, \sigma_3 \rangle \triangleright_{p} \langle h, \sigma_3 \rangle}
\]

(Propagation)

\[
\frac{\langle h, e \cdot 5, \sigma_3 \rangle \triangleright_{p} \langle 5, \sigma_4 \rangle}{\sigma_4 = \langle e \cdot (e \cdot 3), e \cdot (e \cdot 5) \rangle}
\]

(Field assignment)

Starting from an initial configuration, by repeatedly applying reduction rules (note that for each reduction step a proof tree involving reduction steps of subcomponents is needed in general, as shown in the example), one of the following situations can occur:

- Normal termination. This means that, after a finite number of steps, we obtain a configuration whose first element is a value, as in the example above.

**Exercise 6** Evaluate, e.g., the body of method `test` in the example in Sec. 4.1.

- Abnormal termination. This means that, after a finite number of steps, we obtain a configuration whose first element is an error (here only `nullExc`).

**Exercise 7** Write an initial configuration for the program of the previous example which leads to abnormal termination, and show reduction steps.

- Non-terminating execution. This can take place, of course, in presence of recursive method definitions.

**Exercise 8** Write a program with a recursive method (for instance, adding a method to the program of the example) and an initial configuration which leads to non-termination, and show some reduction steps.

- “Stuck” execution. This means that, after a finite number of steps, we obtain a configuration whose first element is neither a value nor a run-time error but no derivation rules can be applied. This situation formally models a crash in the execution, and should be prevented by the type system, in the sense that for well-typed programs and starting expressions we will never obtain stuck execution (as we will see in the following sections).

**Exercise 9** Write an initial configuration for the program of the previous example which gets stuck, and show reduction steps.

**Exercise 10** Assume we extend MINIJAVA with a conditional operator, with the following syntax

\[
e ::= \text{if}(e)\{e_1\}\text{else}\{e_2\}
\]

and the standard meaning.

Add reduction rules for the conditional operator (you should add three rules, one for propagation). What should happen to expressions such as `if (1) {2} else {3}`?

### 4.3 Type system

In programming languages, programs as defined by the grammar (e.g., as MINIJAVA in Fig. 1) can be ill-formed (e.g., identifiers may be used but not declared, the body of a method can return a value which is not of the expected type, and so on). Hence, it is necessary to formally define when a program is well-formed. This formal definition will then be the basis for the implementation of a type-checker for the language.

We will denote the fact that a program is well-formed by the following judgment:

\[
\vdash P \diamondsuit
\]

Of course, in order to decide whether or not a program is well-formed, we must also (inductively) define when subcomponents are well-formed. However, note that for well-formedness of subcomponents it makes no sense to simply say yes or not, since well-formedness *does* depend on the context where they are inserted. For instance, an expression containing a variable will be well-formed if inserted in a context where this variable has been declared.
In type systems for programming languages, the type information about the context is modeled by a so-called *type environment*. In the type system for MiniJava, the type environment will consist of two components $\Gamma$ and $\Pi$, the former, which we will call *class type environment*, containing type information about classes declared in the program, the latter, which we will call *local type environment*, containing type information about the variables (including parameters and this) declared in a method body.

The type information related to a class will be represented by a *class type*, that is, a pair $CT = (FST, MST)$ consisting of a *fields type* and a *methods type*. A fields type $FST$ is a mapping from field names into types, that is, $FST = f_i : T_i^{i \in 1..n}$, hence it corresponds to a sequence of field declarations with no duplicated field names. A methods type $MST$, analogously, is a mapping from method names into *method types* which are pairs consisting of the return and the parameter types, that is, $MST = m_i : (T_i, T_{i-1}^{i \in 1..n})$, hence it corresponds to a sequence of method headers (forgetting parameters) with no duplicated method names. (We denote by $T$ sequences of types.)

Altogether, a class type roughly corresponds to a class deprived of method bodies, hence only keeping the type information.

We introduce the following kinds of judgment expressing well-formedness of subcomponents of a program in a given context.

- $\Gamma \vdash C : \text{class}$ means that $C$ is a class in the context of class type environment $\Gamma$.
- $\Gamma \vdash T : \text{type}$ means that $T$ is a type in the context of class type environment $\Gamma$, that is, either $\Gamma \vdash C : \text{class}$ holds, or $T$ is a primitive type.
- $\Gamma \vdash \text{fields} : FST$ means that fields declaration *fields* is well-formed and has fields type $FST$ in the context of class type environment $\Gamma$.
- $\langle \Gamma, C \rangle \vdash \text{meths} : MST$ means that methods declaration *meths* is well-formed and has methods type $MST$ in the context of class type environment $\Gamma$ and class $C$ (here and below the current class is needed as type for this).
- $\langle \Gamma, C \rangle \vdash \text{meth} : MT$ means that method *meth* is well-formed and has method type $MT$ in the context of class type environment $\Gamma$ and class $C$.
- $\langle \Gamma, \Pi \rangle \vdash e : T$ means that expression $e$ has type $T$ in the context of class type environment $\Gamma$ and local type environment $\Pi$.

In Fig.3 we give the typing rules which define the well-formedness of programs and their subcomponents. We denote by $\text{Name(class)}$, where *class* is a class declaration, the name of the declared class. We write $\Gamma \vdash C : CT$ if $\Gamma(C) = CT$, and analogously $\Pi \vdash x : T$ if $\Pi(x) = T$. 

29
### Figure 3: Typing rules for MINIJAVA

<table>
<thead>
<tr>
<th>Type</th>
<th>$\Gamma \vdash C : -$</th>
<th>$\Gamma \vdash T : C$</th>
<th>$\Gamma \vdash T : M$</th>
<th>$\Gamma \vdash boolean : C$</th>
<th>$\Gamma \vdash int : C$</th>
</tr>
</thead>
</table>

**Program**

\[
C_i : CT_i^{\leq 1..n} \vdash \text{class } C_i : CT_i \quad \forall i \in 1..n
\]

\[
\vdash \text{class } C_i \ldots \text{class } C_n : C
\]

\[
\text{Name}(\text{class } i) = C_i \quad \forall i \in 1..n
\]

\[
i \neq j \Rightarrow C_i \neq C_j \quad \forall i, j \in 1..n
\]

**Class**

\[
\Gamma \vdash \text{fields } : FST \quad \langle \Gamma, C \rangle \vdash \text{meths } : MST
\]

\[
\Gamma \vdash \text{class } C \{ \text{fields } \} : \langle FST, MST \rangle
\]

**Fields declaration**

\[
\Gamma \vdash T_i : C \quad \forall i \in 1..n
\]

\[
\Gamma \vdash f_i : T_i \quad i \neq j \Rightarrow f_i \neq f_j \quad \forall i, j \in 1..n
\]

**Methods declaration**

\[
\langle \Gamma, C \rangle \vdash T_i n_i (\text{params}_i) \{ e_i ; \} : MT_i \quad \forall i \in 1..n
\]

\[
\langle \Gamma, C \rangle \vdash T_1 f_1 ; \ldots T_n f_n ; : \{ f_i : T_i^{\leq 1..n} \} \quad i \neq j \Rightarrow f_i \neq f_j \quad \forall i, j \in 1..n
\]

**Method**

\[
\langle \Gamma, C \rangle \vdash T T_1 \ldots T_n : \{ e_1 ; \ldots e_n ; \} : T
\]

\[
\langle \Gamma, C \rangle \vdash T \text{m}(T_1 x_1, \ldots, T_n x_n) \{ e ; \} : \{ T_1 \ldots T_n \}
\]

\[
i \neq j \Rightarrow x_i \neq x_j \quad \forall i, j \in 1..n
\]

**Field selection**

\[
\langle \Gamma, C \rangle \vdash e : C \quad \Gamma \vdash e : f : T
\]

\[
f_i = f, T_i = T
\]

**Field assignment**

\[
\langle \Gamma, C \rangle \vdash e_1 : C \quad \Gamma \vdash e_1 : f_i : T_i^{\leq 1..n} -
\]

\[
\langle \Gamma, C \rangle \vdash e_2 : T
\]

\[
f_i = f, T_i = T
\]

**Method call**

\[
\langle \Gamma, C \rangle \vdash e : C \quad \Gamma \vdash e : (\ldots, m_j : MT_j^{\leq 1..m})
\]

\[
\langle \Gamma, C \rangle \vdash e_i : T_i \quad i \in 1..n
\]

\[
\langle \Gamma, C \rangle \vdash e_i : m(e_1, \ldots, e_n) : T
\]

\[
m_j = m, MT_j = \{ T, T_1 \ldots T_n \}
\]

**Object creation**

\[
\langle \Gamma, C \rangle \vdash new C : C
\]

**Parameter/local variable**

\[
\Pi \vdash x : T
\]

\[
\Gamma \vdash T : C
\]

\[
x \neq \text{this}
\]

\[
\text{This}
\]

\[
\Pi \vdash \text{this } : C
\]

\[
\Gamma \vdash C : C
\]

**Sequence with optional local declaration**

\[
\langle \Gamma, C \rangle \vdash e_1 : T_1 \quad \langle \Gamma, C[x := t_1] \rangle \vdash e_2 : T_2
\]

\[
\langle \Gamma, C \rangle \vdash e_1 : T_1 \quad \langle \Gamma, C \rangle \vdash e_2 : T_2
\]

\[
\langle \Gamma, C \rangle \vdash e_1 : T_1 \quad \langle \Gamma, C \rangle \vdash e_2 : T_2
\]

\[
\langle \Gamma, C \rangle \vdash x = e_1 ; e_2 : T_2
\]

\[
\langle \Gamma, C \rangle \vdash e_1 ; e_2 : T_2
\]

**Constant**

\[
\langle \Gamma, C \rangle \vdash \text{true } : \text{boolean}
\]

\[
\langle \Gamma, C \rangle \vdash \text{false } : \text{boolean}
\]

\[
\langle \Gamma, C \rangle \vdash \text{N } : \text{int}
\]
We show now some examples of application of typing rules. Consider the following program $P$.

class A {
    int f;
    int m (B b, int x) { b.f=x; }
}
class B {
    int f;
    int m (A a, int x) { a.m(this,x); }
}

Let us denote by $\Gamma_P$ the class type environment which can be extracted from $P$, that is

$$
\Gamma_P = 
\begin{align*}
A &:\{ f: \text{int}, m: \{ \text{int,B int} \} \}, \\
B &:\{ f: \text{int}, m: \{ \text{int,A int} \} \}.
\end{align*}
$$

Let us denote by $\text{class}_A$, $\text{class}_B$, the two class declarations in $P$, and by $\text{meth}_A$, $\text{meth}_B$ the two method declarations in $A$ and $B$, respectively. The proof that $P$ is well-formed is shown below.

$$
\begin{array}{c}
\Gamma_P \vdash \text{int} o_{\text{type}} \quad (\text{type}) \\
\Gamma_P \vdash f : f : \text{int} \quad (\text{Fields dec}) \\
\Gamma_P \vdash \text{class}_A : \{ f : \text{int}, m : \{ \text{int,B int} \} \} \quad (\text{Class}) \\
\Gamma_P \vdash \text{class}_B : \{ f : \text{int}, m : \{ \text{int,A int} \} \} \quad (\text{Class}) \\
\end{array}
\quad \Rightarrow \quad P \circ
$$

The proofs of well-formedness for the two method declarations are below: we denote by $\Pi_A$ the local type environment $b : B, x : \text{int}, \text{this} : A$, and by $\Pi_B$ the local type environment $a : A, x : \text{int}, \text{this} : B$.

$$
\begin{array}{c}
\Gamma_P \vdash B : - \quad (\text{type}) \\
\Gamma_P \vdash B o_{\text{class}} \quad (\text{type}) \\
\Gamma_P \vdash B o_{\text{type}} \quad (\text{Type}) \\
\Gamma_P \vdash B : f : \text{int} \quad (\text{Fields dec}) \\
\Gamma_P \vdash B : (f : \text{int}, -) \quad (\text{Class}) \\
\Gamma_P \vdash B : f : \text{int} \quad (\text{Fields dec}) \\
\Gamma_P \vdash B : (f : \text{int}, -) \quad (\text{Class}) \\
\end{array}
\quad \Rightarrow \quad \Gamma_P \vdash \text{meth}_A : m : \{ \text{int,B int} \} \quad (\text{Method})
$$

$$
\begin{array}{c}
\Gamma_P \vdash a : A \\
\Gamma_P \vdash A : \{ m : \{ \text{int,A int} \} \} \\
\Gamma_P \vdash \text{this} : A \\
\Gamma_P \vdash x : \text{int} \\
\Gamma_P \vdash B : f : \text{int} \quad (\text{Field assignment}) \\
\Gamma_P \vdash b, f=x; : \text{int} \\
\Gamma_P \vdash B : (f : \text{int}, -) \quad (\text{Class}) \\
\Gamma_P \vdash \text{meth}_B : m : \{ \text{int,A int} \} \quad (\text{Method})
\end{array}
$$

We prove now that the following expression $e$

$$
\text{new } B.m(\text{new } A, 7)
$$

is well-formed in the class type environment $\Gamma_P$ and the empty local environment.

$$
\begin{array}{c}
\Gamma_P \vdash B o_{\text{class}} \quad (\text{Object creation}) \\
\Gamma_P \vdash B : m : \{ \text{int,A int} \} \\
\Gamma_P \vdash A o_{\text{class}} \quad (\text{Object creation}) \\
\Gamma_P \vdash \text{this} : A \\
\Gamma_P \vdash \text{m} : \{ \text{int,A int} \} \\
\end{array}
\quad \Rightarrow \quad \Gamma_P \vdash \text{new } B.m(\text{new } A, 7) : \text{int}
$$

Exercise 11

- Evaluate $e$ starting from the empty store.
- Assume we extend MINIJAVA with the identity operator as in Java, with the following syntax

$$
e ::= e_1 == e_2
$$

and the standard meaning.

Add reduction and typing rules for the operator.

- Assume we extend MINIJAVA with static methods as in Java, with the standard meaning. That is, we modify the syntax as follows.

$$
meth ::= [\text{static}] T m(T_1\ldots T_n x_n) \{ e; \} \\
e ::= \ldots | C.m(e_1,\ldots,e_n)
$$

Show which extensions are needed in reduction and typing rules. More precisely:
1. add reduction rules for static method call;
2. describe how the definition of method types should be changed;
3. add a typing rule for static method declaration;
4. add a typing rule for static method call.

4.4 Soundness of the type system

We now express formally the property that the type system is sound. In its general formulation, this property states that execution of well-typed expressions never gets stuck, and is usually proved as consequence of two different properties: subject reduction and progress. The former states that by applying a reduction step to a well-typed expression we still get a well-typed expression (or an error). The latter states that a well-typed expression which is not a value (or an error) can always be reduced further.

Note the difference between run-time errors, that is, errors which we know may happen during execution, but we cannot or are not interested in preventing by the type system, as nullExc in MINIJAVA, and stuck execution, which models an unexpected crash, and should be prevented by the type system.

In MINIJAVA, since it is an imperative language, reduction is defined on configurations which are pairs consisting of a (run-time) expression and a store. As a consequence, we have to first extend the typing rules to configurations, as shown in Fig.4.

\[\begin{align*}
\frac{\Gamma \vdash \sigma \circ \langle \Gamma, \emptyset \rangle \vdash e : T}{\Gamma \vdash \langle e, \sigma \rangle : T} & \quad \frac{\Gamma \vdash C \circ_{\text{class}}}{\langle \Gamma, \Pi \rangle \vdash i^C : C} & \quad \frac{\Gamma \vdash C \circ_{\text{class}}}{\langle \Gamma, \Pi \rangle \vdash \text{null} : C} \\
\sigma(i^C) = h_i : \nu_i^{1..n} \Rightarrow \Gamma \vdash C : \langle f_i : T_i^{1..n}, \_ \rangle, \forall i \in 1..n \quad (\Gamma, \emptyset) \vdash \nu_i : T_i & \quad \Gamma \vdash \sigma \circ
\end{align*}\]

Figure 4: Typing rules for MINIJAVA configurations

The first rule states that a configuration is well-typed in the context of class type environment $\Gamma$ when it consists of a well-formed store and a well-typed (run-time) expression.

The following two rules extend typing to the two kinds of expressions added in run-time expressions, that is, object identifier and null. Note that null has any object type.

The last rule states that a store is well-formed in the context of class type environment $\Gamma$ when each existing object belongs to an existing class whose fields match in name and type the fields of the object state.

Let us denote by $\Gamma_P$ the class type environment extracted from $P$ by, roughly, forgetting method bodies.

The subject reduction and progress property for MINIJAVA are expressed by the two theorems below.

**Theorem 1** Given a program $P$, and a configuration $\langle e, \sigma \rangle$,

if $\vdash P \circ \sigma, \Gamma_P \vdash \langle e, \sigma \rangle : T, \langle e, \sigma \rangle \rightsquigarrow_P \langle e', \sigma' \rangle$

then either $\Gamma_P \vdash \langle e', \sigma' \rangle : T$, or $e' = \text{nullExc}$.

The proof is by induction on the structure of (run-time) expressions, that is, we prove that the property holds for an expression assuming that it holds for all subcomponents.

We show as an example how the proof for the case of field selection works.

Assume that $\vdash P \circ \sigma, \Gamma_P \vdash \langle e, \sigma \rangle : T, \langle e, \sigma \rangle \rightsquigarrow_P \langle e', \sigma' \rangle$.

The reduction step has been obtained by applying one of the three rules for field selection in Fig.2. Let us consider the three cases.

- If we applied first rule, then $e = i^C$; moreover the type $T$ for $i^C f$ has been deduced by applying typing rule for field selection in Fig.3, hence $\langle \Gamma_P, \emptyset \rangle \vdash i^C : C$ and $\Gamma_P \vdash C : \langle \_ \_ f_i : T_i^{1..n} \rangle$ hold, with $f = f_i, T = T_i$ for some $i \in 1..n$. Hence, since also $\Gamma_P \vdash \sigma \circ$ holds, from the corresponding rule we have that $\sigma(i^C) = h_i : \nu_i^{1..n}$ with $f = f_i, \langle \Gamma_P, \emptyset \rangle \vdash \nu_i : T_i$. Hence we can conclude that $e' = \sigma(i^C)(f)$ is well-typed and has type $T$.

- If we applied second rule, then $e = \text{null}$ and $e' = \text{nullExc}$, hence the thesis immediately holds.

- If we applied the third rule, then it is easy to see that the thesis holds by inductive hypothesis.

**Theorem 2** Given a program $P$, and a configuration $\langle e, \sigma \rangle$, where $e$ is neither a value nor nullExc,
if \( \vdash P \circ \Gamma_p \vdash (e, \sigma) : T \),
then there exists \( (e', \sigma') \) s.t. \( (e, \sigma) \sim_p (e', \sigma') \).

The proof is again by induction on the structure of (run-time) expressions and we show as an example how the proof for the case of field selection works.

Assume that \( \vdash P \circ \Gamma_p \vdash (e, f, \sigma) : T \). Then, the type \( T \) for \( e, f \) has been deduced by applying typing rule for field selection in Fig.3, hence \( (\Gamma_p, \emptyset) \vdash e : C \).

Let us consider the different forms \( e \) may have.

- If \( e \) is a value, then, since \( (\Gamma_p, \emptyset) \vdash e : C \) holds, we can conclude that \( e \) is either of the form \( i^C \) or \( \text{null} \). Hence we can apply either the first or the second rule for field selection in Fig.2 and the thesis holds.
- If \( e = \text{null} \), then we can apply propagation rule for exception in field selection (omitted in Fig.2), and the thesis holds.
- If \( e \) is an expression which is neither a value nor \( \text{null} \), then by inductive hypothesis we can apply third rule for field selection in Fig.2, and the thesis holds.

### 4.5 Adding inheritance

Adding inheritance to MINJAVA means that in the syntax the production for class declarations changes as shown below, while other productions remain the same.

\[
\text{class} \quad ::= \quad \text{class } C \text{ extends } C' \{ \text{fields meths} \}
\]

We assume that there exists a distinguished class name \( \texttt{Object} \) which can be used as parent class in a class declaration, but not as the name of a declared class. This class name will denote the root class in the inheritance hierarchy, which has no parent class and (for simplicity) no fields and no methods.

The execution model does not change by adding inheritance, that is, reduction rules still are those given in Fig.2. However, the definitions of the functions which return the fields and the method bodies of a class need to be modified in order to take into account inherited fields and methods as well, as shown below.

- \( \text{Fields} \_p(C) \) are all the fields of class \( C \) in program \( P \) (either directly declared or inherited), if any, undefined otherwise.
- \( \text{Body} \_p(C, m) \) is
  - \( \langle x_1 \ldots x_n, e \rangle \) if class \( C \) is declared in \( P \) and declares a method with name \( m \), parameters \( x_1 \ldots x_n \) and body \( e \);
  - \( \text{Body} \_p(C', m) \), if class \( C \) is declared in \( P \), declares no method with name \( m \) and has parent class \( C' \);
  - undefined otherwise.

Note that the function \( \text{Body} \) formalizes method look-up as informally described in Sec.3.2; note also that in case either class \( \texttt{Object} \) is reached, or in case of cycles in the inheritance hierarchy the function turns out to be undefined (this will be prevented by the type system).

We illustrate now how the type system must be changed in order to take into account inheritance. First of all, now for typechecking components of a program we also need a \textit{subtyping} relation among types (since, in general, an expression of a type \( T \) which is a subtype of \( T' \) can be used in contexts of type \( T' \)). This relation will be determined by the inheritance hierarchy induced by a program. More precisely, if \( \prec_p \) is the inheritance hierarchy induced by a program (that is, the transitive closure of the \texttt{extends} relation), then we will denote by \( \leq_p \) the corresponding subtyping relation, that is, the reflexive and transitive closure of the \texttt{extends} relation, extended to primitive types which are only subtypes of themselves. Hence a class type environment becomes a pair \( \langle \leq, C_i : CT_i \rangle \) where \( \leq \) is a reflexive and transitive relation.

In Fig.5 we give the new typing rules which take into account subtyping. We only show rules which need to be changed w.r.t. those in Fig.3.

In the typing rule for programs, there is now the additional requirement that the inheritance hierarchy induced by the program must be acyclic (this is denoted by the judgment \( \vdash \prec_p \circ \texttt{Acyclic} \)).

In the typing rule for classes, the class type is now constructed by updating the fields and methods inherited from the parent with the fields and methods declared in the class. The two predicates \( \texttt{NoHiding}(FST, FST') \) and \( \texttt{OKOverriding}(MST, MST') \) are defined as follows.

- \( \texttt{NoHiding}(FST, FST') \) holds if \( \text{Def}(FST) \cap \text{Def}(FST') = \emptyset \), that is, the heir class cannot declare a field with the same name of an inherited field. This is a simplification we assume in MINJAVA, which does not hold in full Java where fields can be hidden.
Program \[ \Gamma \vdash C : CT_i^{\leq 1.n} \] \[ \vdash P \] \[ P = \text{class}_1 \ldots \text{class}_n \]

Class \[ \Gamma \vdash C : \langle \text{FST}', \text{MST}' \rangle \]
\[ \Gamma \vdash \text{fields} : \text{FST} \]
\[ \langle \Gamma, C \rangle \vdash \text{mets} : \text{MST} \]
\[ \Gamma \vdash \text{class} C \text{ extends } C't' \{ \text{fields mets} \} : \langle \text{FST}'[\text{FST}], \text{MST}'[\text{MST}] \rangle \]

NoHiding(\text{FST}, \text{FST}') \text{, OKOverriding(\text{MST}', \text{MST})}

Method \[ \Gamma \vdash T, T_1, \ldots, T_n \text{ type} \]
\[ \langle \Gamma, C \rangle \vdash T \{ x_1, \ldots, x_n \} \{ e ; \} : \langle T, T_1 \ldots T_n \rangle \]

Field assignment \[ \langle \Gamma, \Pi \rangle \vdash e_1 : C \]
\[ \Gamma \vdash C : \langle f_i : T_i^{\leq 1.n}, \text{this} : C \rangle \]
\[ \langle \Gamma, \Pi \rangle \vdash e_2 : T_2 \]
\[ \Gamma \vdash T_2 \leq T \]
\[ f_i = f, T_i = T \]

Method call \[ \langle \Gamma, \Pi \rangle \vdash e : C \]
\[ \Gamma \vdash C : \langle -, m_j : \text{MT}_j^{\leq 1.m} \rangle \]
\[ \langle \Gamma, \Pi \rangle \vdash e_1 : T'_i \]
\[ \Gamma \vdash T'_i \leq T_i \]
\[ \forall i \in 1..n \]
\[ m_j = m \]
\[ \text{MT}_j = \langle T, T_1 \ldots T_n \rangle \]

Sequence with local declaration \[ \langle \Gamma, \Pi \rangle \vdash e_1 : T_1 \]
\[ \Gamma \vdash T_1 \leq T \]
\[ \langle \Gamma, \Pi[x : T] \rangle \vdash e_2 : T_2 \]
\[ \langle \Gamma, \Pi \rangle \vdash T x = e_1 ; e_2 : T_2 \]

Figure 5: Typing rules for MiniJava with inheritance

- **OKOverriding(\text{MST}, \text{MST}')** holds if, for each \( m \in \text{Def(MST)} \), if \( m \in \text{Def(MST}') \) then \( \text{MST}(m) = \text{MST}'(m') \), that is, an heir class can declare a method with the same name of an inherited method (in this case the inherited method is overridden), but the new method must have the same parameter and return types. This corresponds to the rule that holds in full Java too. However, in MiniJava we assume no overloading (a class cannot have two methods with the same name), hence methods types are mappings from method names into method types, while in full Java overloading of methods is allowed (a class can have more than one method with the same name and different parameter types), hence a methods type should be modeled as a mapping from pairs consisting of method names and sequences of parameter types into return types.

In the typing rule for methods, the body of the method is now allowed to have a subtype of the return type.

In the typing rule for field assignments, the expression on the right side is now allowed to have a subtype of the field type.

In the typing rule for method calls, arguments are now allowed to have subtypes of the parameter types.

Finally, in the rule for sequences with local declaration, the expression which is used for initializing the variable is now allowed to have a subtype of the type declared for the variable.

An alternative way for expressing the type system with subtyping would be to keep rules for expressions as they are and to add the subsumption rule:

\[ \langle \Gamma, \Pi \rangle \vdash e : T_1 \]
\[ \Gamma \vdash T_1 \leq T_2 \]
\[ \langle \Gamma, \Pi \rangle \vdash e : T_2 \]

However, this alternative formulation would not work when considering, e.g., static fields and methods and overloading, since in these cases the “minimal” type of an expression matters (for instance, in the rule for field selection, in case of hiding we could non deterministically select the field in the parent and the field in the heir).

The subject reduction and progress property for MiniJava with inheritance are analogous to those previously stated; however, in the subject reduction property the type is not preserved now by reduction, but we can obtain an expression of a subtype. We show the new formulation below.
Theorem 3 Given a program $P$, and a configuration $(e, \sigma)$,

$$
\begin{align*}
\Gamma \vdash P \land \Gamma \vdash (e, \sigma) : T, (e, \sigma) \leadsto (e', \sigma')
\end{align*}
$$

then either $\Gamma \vdash (e', \sigma') : T'$, with $\Gamma \vdash T' \leq T$, or $e' = \text{nullExc}$. 

5 Typing issues

In this section, we illustrate other features related to typing in Java-like languages, namely exceptions, casts and run-time checks, and overloading. Moreover, we discuss some alternative design choices, as contravariant and covariant rules in overriding and type systems where classes and types are kept distinct.

5.1 Checked exceptions

Many languages, both object-oriented and not, provide an exception mechanism for handling error situations. The basic two ingredients of this mechanism are the following.

- The language provides a linguistic construct for throwing, or raising, an exception; in Java it has the form
  
  ```
  \text{throw} \ \text{excp}
  ```

  where `excp` is an expression of an exception type (that is, a subclass of the predefined class `Exception`); the intuitive meaning is that some error situation has been encountered, and normal execution flow is abandoned.

- The language provides a linguistic construct for catching exceptions; in Java it has the form

  ```
  \text{try} <\text{block}>
  \text{catch} (E_1 <\text{excp}_1>) <\text{block}_1>
  ...
  \text{catch} (E_n <\text{excp}_n>) <\text{block}_n>
  ```

  where $E_1, \ldots, E_n$ are exception types. The execution of a `try` block can be informally described as follows. The block `<block>` is executed. If its execution terminates normally (that is, without throwing exceptions), then the execution of the `try` block is terminated. If during the execution of `<block>` some exception is thrown, say of type $E$, then the `catch` clauses are considered in sequential order. If a `catch` clause is encountered, say $E_i$, s.t. $E_i$ is (a supertype of) $E$, then the exception is caught and the corresponding block `<block_i>` is executed; otherwise, the exception is propagated to the enclosing block.

Note that the usual subtyping mechanism works, hence an exception can be caught by a `catch` clause where an exception of a supertype is expected.

An additional feature provided in Java together with the above illustrated exception mechanism is the fact that (some) exceptions are checked.

This means that a method must explicitly declare by a `throws` clause all the exception types which are possibly thrown in its body, otherwise there is a static error.

As introductory example, we show an implementation of stacks which uses Java arrays and handles the two possible error situations (pop on empty stack and push on full stack) by defining two exceptions.

```java
class StackByArray implements Stack {
    final static int maxLength = 15;
    int[] elems = new int[maxLength];
    int length = 0;

    public int pop () throws EmptyStackException {
        if (length==0) throw new EmptyStackException();
        return elems[--length];
    }

    public void push (int e) throws FullStackException {
        if (length == maxLength)
            throw new FullStackException();
        elems[length++] = e;
    }
}
```

4We do not describe the optional `finally` part in these notes.
5It is also possible to define unchecked exceptions in Java, by declaring heir classes or either `Error` or `RuntimeException`.

35
Exceptions in Java are objects, instances of (heirs of the) predefined class `Exception`. Hence, we must also include in the program the following declarations.

```java
class EmptyStackException extends Exception{}
class FullStackException extends Exception{}
```

class stacksTest{
    public static void main (String[] args) throws EmptyStackException{
        Stack s = new StackByArray();
        for (int i=1;i<20;i++)
            try { s.push(i); }
            catch (FullStackException excp) {
                System.out.println("Full Stack"); break;
            }
        for (int i=1;i<5;i++) { s.pop(); }
    }
}
```

The programmer which writes a method must, for each call of another method which declares an exception in its `throws` clause, do one of the following, otherwise there is a static error:

- either include the call in a try-block and catch the exception, providing an handling block;
- or declare (a superclass of) the exception in the `throws` clause of the method being defined.

The first choice corresponds to the fact that the programmer knows which alternative action to take in case this error situation is encountered. Note that a particular case of alternative action is to raise another exception which is declared in the `throws` clause of the method being defined.

The second choice corresponds to the fact that the programmer simply leaves the handling of the error situation to the caller of the method being defined.

In the example above, only for illustrating the two choices, the exception `FullStackException` has been handled by writing an user-defined error message, whereas the exception `EmptyStackException` has been not handled, hence must be declared in the `throws` clause of `main`. In case the latter exception is thrown during execution, a default handler takes place.

As other objects, exceptions may have fields which can be used for storing additional information on the error situation which has been raised (e.g., the predefined class `Exception` has a field of type `String` which can be used for storing an error message).

The fact that a class must declare in the `throws` clauses of its methods the exceptions which may be thrown is very important from a methodological point of view, since in this way a client of the class is informed about the error situations which can possibly be encountered using the class. In other words, `throws` clauses are part of the specification of the class. Handling such error situations will be a task of the client. Moreover, since exceptions are actually `checked`, `throws` clauses are part of the type information related to a class, hence, referring to the formal type system defined in Sec.4.3 and 4.5, they must be included in method types.

Typing constraints related to checked exceptions are based on the overall principle that, as for ordinary objects, an exception of type `E` can be used as an exception of a supertype. More in detail the following rules hold.

- As already said, for each exception `E` which is thrown and not caught in a method body, the method must declare in its `throws` clause (a supertype of) the exception `E`. Note, however, that from the methodological point of view it is always better to declare in the `throws` clause exception types as much specific as possible, in order to provide more information for the client.

- When a method in a parent class is redefined in a heir class, then, for each exception `E` in the heir’s declaration, the `throws` clause in the parent must declare a (supertype of) `E` (this also hold for an abstract parent class which declares the method abstract or an implemented interface). Why does this constraint hold?

- Catch clauses are considered in sequential order. Hence, it makes no sense (and it is a static error in Java) to have a `catch` clause for a type before a `catch` clause for a subtype.

We show now, as an example, a version of the list implementation introduced in Sec.3.3 enriched by handling error situations by exceptions.

```java
class BadIntervalException extends Exception {
    BadIntervalException (int a, int b) {
        super(b+"<"+a);
    }
}
```
class EmptyListException extends Exception {}

class EmptyList extends List {
    boolean isEmpty () { return true; }
    int head () throws EmptyListException { throw new EmptyListException(); }
    List tail () throws EmptyListException { throw new EmptyListException(); }
}

class IntervalList extends List {
    private int a, b;

    IntervalList (int a, int b) throws BadIntervalException {
        if (b < a) throw new BadIntervalException(a, b);
        else {
            this.a = a; this.b = b;
        }
    }
    boolean isEmpty () { return false; }
    int head () { return a; }
    List tail () {
        if (b > a) {
            try {
                return new IntervalList(a + 1, b);
            }
            catch (BadIntervalException excp) { throw new InternalError(); } />
        else return new EmptyList();
    }
}

class NonEmptyList extends List {
    private int head;
    private List tail;
    NonEmptyList (int head, List tail) {
        this.head = head; this.tail = tail;
    }
    boolean isEmpty () { return false; }
    int head () { return head; }
    List tail () { return tail; }
}

abstract class List {
    abstract boolean isEmpty ();
    abstract int head () throws EmptyListException;
    abstract List tail () throws EmptyListException;
}

Exercise 12 Assume to add in MINIJAVA exceptions, with the following syntax (we only show the productions which change):

$$ET \ ::= \ C_1, \ldots, C_n$$
$$meth \ ::= \ T m(T_1 x_1, \ldots, T_n x_n)\{e\} \text{ throws } ET$$
$$e \ ::= \ \ldots | \text{throw } C | \text{try } e \text{ catch } C_1 : e_1 \ldots \text{catch } C_n : e_n$$

The metavariable $ET$ ranges over sets of exceptions (class names). We assume for simplicity that only class names (corresponding to exception names) may appear in $\text{throw}$ expressions, and that all class names can be used as exceptions (e.g., we ignore the predefined class $\text{Exception}$).

Show which extensions are needed in reduction and typing rules. More precisely:

1. add reduction rules for $\text{try}$ expressions (a $\text{throw } C$ expression will be considered a final result);
2. describe how the definition of method types should be changed, adding a new component corresponding to the $\text{throws}$ clause;
3. describe how typing rules for class and method declarations should change (in particular, the definition of $\text{OKOverriding}(-,-)$);
4. assuming that now the judgment of well-formedness of an expression becomes \( (\Gamma, \Pi) \vdash e : (T, ET) \), meaning that the expression \( e \) has type \( T \) and may throw exceptions in \( ET \), in the context of class type environment \( \Gamma \) and local type environment \( \Pi \), modify existing typing rules and add typing rules for \textit{throw} and \textit{try} expressions.

**Exercise 13** Consider the following Java abstract class corresponding to the partial functions from integers to integers. The method \texttt{apply} returns the result of applying a function to a given argument, and throws the exception \texttt{UndefinedException} if the function is undefined.

```java
class UndefinedException extends Exception {
}
abstract class Function {
    abstract int apply (int arg) throws UndefinedException;
}
```

Define three heir classes (with suitable constructors) corresponding, respectively:

- to the identity functions which are defined only from some value \( k \),
- to the functions which, given another function \( f \), give on each argument the same result of \( f \) when \( f \) is defined, some fixed value \( k \) otherwise,
- to the functions which return, on each argument, the maximum among the results of two given functions (undefined only if both are undefined).

Moreover, add (where?) an instance method which tests whether two functions are equal in a given interval \([n, m]\) (empty if \( n > m \)), that is, return either the same result or are both undefined on each argument in \([n, m]\).

Solution:

1. class Identity extends Function {
   private int k;
   Identity (int k) {this.k = k;}
   int apply (int arg) throws UndefinedException {
      if (arg < k) throw new UndefinedException();
      return arg;
   }
}
2. class Total extends Function {
   private Function f;
   private int k;
   Total (Function f, int k) {this.f = f;this.k = k;}
   int apply (int arg) {
      try { return f.apply(arg);}
      catch (UndefinedException e) {return k;}
   }
}
3. class Max extends Function {
   private Function f,g;
   Max (Function f, Function g) {this.f = f;this.g = g;}
   int apply (int arg) throws UndefinedException {
      int fres = 0; int gres = 0;//initialization is useless but required by Java
      try {fresh = f.apply(arg);}
      catch (UndefinedException e) {return g.apply(arg);}
      try {gres = g.apply(arg);}
      catch (UndefinedException e) {return fres;}
      return Math.max(fres, gres);
   }
}
4. The method can be defined in \textit{Function} as a derived operation, by using \texttt{apply}.
   abstract class Function {
   abstract int apply (int arg) throws UndefinedException;
   boolean equalsIn (Function other, int n, int m) {
      for (int i=n;i<=m;i++) {
   
```
try {
    myres = apply(i);
    try {
        if (myres!=other.apply(i)) return false;
        catch (UndefinedException e) {return false;}
    }
    catch (UndefinedException e1) {
        try {
            other.apply(i);return false;
        }
        catch (UndefinedException e2) {}
    }
    return true;
}

5.2 Casts and run-time type checks

Some object-oriented languages, as Java, allow the programmer to explicitly convert an expression to a different type. A cast expression in Java has the form \((T) e\) where \(T\) is a type and \(e\) is an expression. The intuitive meaning is that expression \(e\), which has a type, say \(T_s\) for “static type”, is considered as an expression of type \(T\).

Let us consider in more detail the case in which both \(T_s\) and \(T\) are classes. Then, in Java, in order the expression to be well-typed, \(T_s\) and \(T\) must be in the inheritance relation, or the same class.

There are, hence, two kinds of casting among classes:

- from a class to a superclass, called casting up, safe casting or widening,
- from a class to a subclass, called casting down, unsafe casting or narrowing.

At run-time, evaluation of a cast expression \((T) e\) consists in, first, evaluating \(e\), getting an object \(i^D\) of a given class, say \(D\) (for “dynamic type”); then, checking whether the dynamic type \(D\) is a subtype of \(T\). In case it is, then the cast is successful and the result of the evaluation is \(i^D\). In case it is not, then the exception ClassCastException is thrown. Casting up is always successful (why?).

From the methodological point of view, casting down can be used to access the richer interface of a supertype.

```java
Rectangle r = new Cuboid();
int v = ((Cuboid) r).volume();
```

Casting up can be useful whenever the static type of an expression matters in the execution, as happens in Java in hiding and overloading. However, casting up does not affect the dynamic type of an expression, hence dynamic binding. This is illustrated by the following example.

```java
class One {
    String str = "One";
    void show () {
        System.out.println(str);
    }
}
class Two extends One {
    int str = 2;
    void show () {
        System.out.println(str);
    }
}

class CastingTest {
    static void test (One one) {
        System.out.println("test(One)");}
    static void test (Two two) {
        System.out.println("test(Two)");}
    public static void main (String[] args) {
        Two two = new Two();
        One one = two;
        System.out.println(((One)two).str);//One
    }
}
```
Another related feature which is supported by some object-oriented languages is the possibility of run-time tests on the dynamic type of an object. An example is the `instanceof` operator in Java; the expression `e instanceof T`, where `e` is an expression and `T` is an object type (that is, a class or an interface), evaluates to `true` if and only if the dynamic type of `e` is a subtype of `T`. Hence, this operator can (and should) be used before a down cast.

Analogous operators are the `typecase` statements in the Theta language and the “reverse assignment” in Eiffel (see [2] Pag.10, [1] Sec.2.5).

Note that using casts and tests on the dynamic type is useful for circumventing unnecessary constraints posed by the type system, but can be considered “impure”, since in this way the object abstraction is violated, dynamic failures are introduced and, more importantly, code becomes less extensible (indeed using tests on the dynamic type corresponds to reintroduce “programming by cases” in the sense of Sec.3.3 in the object-oriented approach).

**Exercise 14**

- Add reduction and typing rules for casts and `instanceof` operator in the formal definition of MINIJAVA.
- Find a motivation for the fact that casts among classes are only allowed when the two classes `T_s` and `T` are in the inheritance relation. What about the case in which either `T_s` or `T` are interfaces? Suggest which should be a reasonable rule, and why.

### 5.3 Type constraints in overriding

See also [2] 2.4, 3.2.5, [1] 2.6, 2.7.

Assume that a heir class `H` overrides a method `m` of the parent class `P`.

In MINIJAVA, as it is in full Java and in many other object-oriented languages, we have imposed the constraint that the method in `H` must have exactly the same parameter and return types it has in `P`. An analogous restriction holds for instance variables (in MINIJAVA we have for simplicity forbidden the redeclaration of a field in a subclass, while in full Java this is interpreted as `hiding`, see Sec.3.7).

We will refer to type systems that restrict the types of methods and instance variables in subclasses to be identical to those in superclasses as *invariant* type systems.

However, this choice may look too restrictive; indeed in many situations it seems reasonable to make more specific the parameter and/or result type of a method, or the type of a field.

**Return types** Let us first consider the case of the return type of a method. In many situations the programmer may know that a certain method always returns an object of a given type, even though the type system restricts the programmer from writing this type as the return type (see also [2] 3.2.1).

A typical example is that of methods whose intended return type is that of the receiver, as happens for methods which return a (either shallow or deep) copy of the receiver. Let us consider our running example of rectangles and cuboids.

```java
class Rectangle {
    int length, width;
    Rectangle clone() { //illegal in Java
        Rectangle clone = super.clone(); //(*) illegal in Java
        clone.length = length;
        clone.width = width;
        return clone;
    }
}

class Cuboid extends Rectangle {
    int height;
    Cuboid clone() { //illegal in Java
        Cuboid copy = super.clone(); //(*) illegal in Java
        copy.height = height;
        return clone;
    }
}
```
We cannot override the method `clone` (which is a predefined method in `Object` with return type `Object`) with a more specific return type, as would be natural in this case. Note that of course we could declare methods with different names (e.g., `rectangleClone` and `cuboidClone`, respectively), but we lose the benefit of inheritance (and also, the old method is invoked in calls inside inherited and not redefined methods). In newer versions of C++ it is allowed to specialize return type in overriding methods. In Java, we must keep the same return type and use a down cast for, e.g., assigning the result to a variable of type `Cuboid`.

Note, moreover, a related problem that we will consider in more detail later: even in case it is allowed to write a more specific return type in overriding methods, the assignments (*) would be illegal and we should insert a down cast (we assume that the predefined method `clone` in `Object` returns an instance of the class of the receiver).

### Parameter types

We consider now situations in which it seems intuitively desirable to write a more specific type for a parameter. Typically this happens for *binary methods*, that is, methods which have a parameter whose intended type is the same as that of the receiver (see also [2] 3.2.2, [1] 2.6,2.7).

Consider, e.g., the method `equals` in class `Rectangle`, already shown in Sec.2.2, which tests equality among rectangles. In the heir class `Cuboid`, it seems reasonable to redefine the method with a more specific parameter type, as shown below. We use Java syntax for uniformity, but we are assuming a typing rule different from that of Java, that is, allowing more specific parameter types when overriding a method.

```java
class Rectangle {
    ...
    boolean equals (Rectangle r) {
        return length == r.length && width == r.width;
    }
}
class Cuboid extends Rectangle {
    ...
    boolean equals (Cuboid c) {
        return length == r.length && width == r.width && height == c.height;
    }
}
```

This typing rule looks intuitively correct, and is actually the rule taken in the Eiffel language. However, this rule makes the type system unsound. Consider, indeed, the following code fragment:

```java
Rectangle r = new Rectangle();
Rectangle r1 = new Cuboid();
System.out.println(r1.equals(r)); // well-typed
```

The method call turns out to be statically well-formed, since `r1` and `r` have static type `Rectangle` and class `Rectangle` has a method `equals (Rectangle)` (Rectangle). However, at run-time, if we assume that the method is overridden in `Cuboid`, then the version in `Cuboid` is invoked, since `r1` has dynamic type `Cuboid`. As a consequence, we get an error, since the object denoted by `r` has no `height` field.

**Exercise 15** Formally show this fact using MiniJAVA reduction rules.

Hence the rule which allows to declare more specific types for the parameters in overriding is not sound. This rule is called *covariant* since the parameter types are allowed to change in the same direction as the object type. The *contravariant* rule, instead, which allows parameter types to change in the opposite direction as the object type (that is, to become more general), can be proved to be sound.

We motivate now the fact that the contravariant rule is correct, while the covariant is not, considering an analogous case, that of functional types. Assume that we want to define when a functional type, that is, the type of the functions which take arguments of type $T_a$ and give a result of type $T_r$, written $T_a \rightarrow T_r$, is a subtype of another functional type, say $T'_a \rightarrow T'_r$. Then, the contravariant rule can be formalized as follows:

\[
\frac{T'_a \leq T_a \quad T_r \leq T'_r}{T_a \rightarrow T_r \leq T'_a \rightarrow T'_r}
\]

This rule expresses the fact that a function from a type $T_a$ to another type $T_r$ can also be regarded as a function from a more specific type into a more general type. To see that this is intuitively correct, consider a function $f: [3, 7] \rightarrow [7, 9]$: $f$ can also be regarded as a function from $[4, 6]$ in $[6, 10]$. 

41
Hence, this rule looks intuitively correct, and, indeed, a type system with this rule turns out to be sound. However, the rule is not completely satisfactory from the methodological point of view; for instance, we could not write in Cuboid the new definition of equals we would like to have.

Another solution to this problem is that of consider class and type two distinct notions. In most object-oriented languages, as we have seen, class names are types and the inheritance and the subtyping relation among classes coincide; that is, class Heir is a subtype of class Parent if and only if Heir inherits from Parent.

However, it is perfectly possible to design languages where the two notions are independent (examples are the language used in [2] and Objective Caml). In these languages, the type of an object is its operational interface, that is, the set of its fields (if visible) and methods with their types. A class declaration uniquely determines the type of its instances, but different classes may implement the same type, hence objects with the same type can be instances of different classes. We will discuss in more detail these languages in Sec.5.5. If class and type are two different notions, then also the inheritance and subtyping notion turn out to be independent. Given two classes C1 e C2 which generate objects of type T1 and T2, respectively, then it may obviously happen that T2 is subtype of T1, but C2 is not heir of C1. For what concerns the converse situation (C2 heir of C1 but T2 non subtype of T1), this depends on the typing rules of the language. The implication always holds in MINIJAVA (Exercise: informally check this), but does not hold, for instance, if we allow the contravariant rule in method overriding. Indeed, what happens in this case is that Cuboid inherits from Rectangle, hence the code is reused, but the type of objects generated by Cuboid is not a subtype of the type of objects generated by Rectangle.

This solution is technically correct but partly unsatisfactory since we could not use, referring to the example, cuboids in contexts of type Rectangle, as it looks intuitively desirable.

Let us finally consider the Java solution. As already said, Java takes the invariant rule in overriding. However, situations like that of the example can be partially handled in Java by using method overloading.

Indeed, in Java, both overriding and overloading of methods are allowed. In the example, the method equals in Cuboid does not override that in Rectangle, but it is as a matter of fact a new method which just happens to have the same name. Class Cuboid has indeed two methods equals, which take arguments of type Rectangle and Cuboid, respectively.

```java
boolean equals (Rectangle r) //old
boolean equals (Cuboid c) //new
```

Hence, the code

```java
Rectangle r = new Rectangle();
Rectangle r1 = new Cuboid();
System.out.println(r1.equals(r)); //statically well-formed
```

does not raise in Java any run-time error; the version of equals which is invoked is, correctly, that in Rectangle. Indeed, overloading resolution is static, that is, takes place before execution and is driven by the static types of arguments, see Sec.5.4. However, in other cases the fact that overloading resolution is static does not lead to the intuitively expected choice of the method version to be invoked. Consider for instance:

```java
Cuboid c = new Cuboid();
Rectangle r1 = new Cuboid();
System.out.println(c.equals(r1)); //version in Rectangle is invoked
```

The version in Rectangle is still invoked, even though the two objects to be compared are in this case two cuboids, since the static type of r1 is Rectangle. The version in Cuboid will be invoked in languages supporting multimethods [8, 5], that is, methods in which the selection of the version to be invoked depends on the dynamic type of not only the receiver, but all arguments.

See also [2] 2.5 for an analogous example.

However, this can be simulated in Java if the programmer explicitly uses the instanceof operator, see Sec.5.2, that is, declaring in Cuboid two methods equals, as shown below.

```java
public boolean equals (Rectangle r) {
    if (r instanceof Cuboid) { return equals((Cuboid)r);}
    else return super.equals(r);
}

public boolean equals (Cuboid c) {
    return super.equals(c) && height == c.height;
}
```
Note the use of downcasting, see Sec. 5.2, for invoking the version with parameter of type Cuboid; also note the use of super, see Sec. 3.6, for invoking the version of equals in Rectangle.

The example above illustrates a quite general situation, that is, the instanceof operator can often be used in Java to simulate multimethods, whenever the action to be taken depends on the dynamic type of arguments. However, as already discussed, this solution can be considered “impure” especially since extensibility of code is not preserved (e.g., adding a new heir class to Rectangle we should rewrite code).

Field types See also [2] 5.1.3, 5.1.4.

Also for fields it seems intuitively reasonable in many situations to be able to declare more specific types in heir classes. Consider the following example:

```java
class Rectangle {
    int length, width;
}
class Cuboid extends Rectangle {
    int height;
}
class RectanglePair {
    Rectangle first;
    Rectangle second;
}
class CuboidPair extends RectanglePair {
    Cuboid first;
    Cuboid second;
}
```

The fact that the fields in CuboidPair should have more specific type seems intuitively desirable. However, if we allow this, then the type system becomes not sound, as shown by the following code fragments (assuming break of type Break, cp of type CuboidPair):

```java
class Break {
    void m (RectanglePair rp) {
        rp.first = new Rectangle(); // should be dynamically checked
    }
}
Break break;
...
break.m(cp); // well-formed since cp can be an argument for m
cp.first.height // well-formed since cp.first has type Cuboid but dynamic error!!
```

It is easy to show by an analogous example that also allowing more generic types in fields of subclasses leads to an unsound type system (Exercise: write such an example). Hence, we can conclude that field types must be invariant in subtypes.

Note, moreover, that the problem illustrated above actually holds in Java for array types (see [2] 5.1.4 for details); that is, Java typing rules for array types are not sound, and this is compensated in Java by inserting dynamic checks when assigning array elements (as could be done also for fields, as indicated in the comment in the code above).

Note, also, that, as happens for the covariant rule for method overriding, the problem above would be avoided if we consider types and classes two distinct notions; indeed, in this case CuboidPair would no longer be a subtype of RectanglePair.

Type constraints on subclasses See also [2] Chapter 6. In the examples above we have seen that it would have been possible to allow the more specific method parameter types and the more specific field types in the subclass if we had no longer considered subtyping to be subclassing. However, even under this assumption, it is not possible in the general case to allow more specific method parameter types and different field types in a subclass, due to the fact that other methods can refer to them via self. Consider the following example:

```java
class RectanglePair {
    Rectangle first;
    Rectangle second;
    void setFirst (Rectangle r) {
        first = r;
    }
}
```
void setSecond (Rectangle r) {
    second = r;
}

void doSomething() {
    Rectangle r;
    ...
    setFirst(r);
}

class CuboidPair extends RectanglePair{
    Cuboid first;
    Cuboid second;
    void setFirst (Cuboid c) {
        first = c;
    }
}

The method setFirst in CuboidPair overrides that in RectanglePair with a more specific parameter type. However, CuboidPair also inherits the method doSomething, and the call to setFirst becomes in this way ill-typed. Analogously, CuboidPair declares the field second with a more specific type; however, CuboidPair also inherits the method setSecond, and the assignment to second becomes in this way ill-typed.

5.4 Overloading

See also [2] 2.5.
Some object-oriented languages, including Java and C++, allow overloading of methods, that is, a class can have among its (either inherited or directly declared) methods more than one method with the same name but different parameter types. As already explained in Sec.1.3, this is an overloading with static resolution, that is, the method selection takes place at compile-time, whereas late binding in method calls can be seen as a form of overloading with dynamic resolution restricted to the first argument (the receiver).

We describe now more precisely how overloading resolution happens in Java.
Let us consider a method call e.m(e₁,...,eₙ) and assume that e has type T and e₁,...,eₙ have types T₁,...,Tₙ, respectively.
The algorithm for selecting the so-called “most specific” method can be described as follows.

1. All methods are searched which are applicable, that is, have name m, are declared in a supertype (class or interface) T₀ of T and have parameter types T₀₁,...,T₀ₙ which are supertypes of T₁,...,Tₙ, respectively.

2. Then, among all the applicable methods, the most specific is selected, if any. More precisely, for each pair of applicable methods, say

   a method declared in T' with parameter types T'₁,...,T'ₙ, and a method declared in T'' with parameter types T''₁,...,T''ₙ

   if the former is more specific of the latter, that is, T' is a subtype of T'' and T'ᵢ is a subtype of T''ᵢ for each i, then the less specific method is eliminated.

3. If, iterating the procedure, at the end only one method remains, then this is the method which is selected, otherwise there is a static error (the call is ambiguous).

Note that the type of the call is determined by the return type of the most specific method, hence overloading resolution can not only affect the dynamic semantics, but even the static correctness of programs.
Ambiguity can arise, for instance, in situations as in the following example, where we assume Heir subclass of Parent:

class C {
    void m (Parent p, Heir h) {...}
    void m (Heir h, Parent p) {...}
}

in a call c.m(h,h) where c has type C and h has type Heir; moreover, since in the definition of “more specific” also the type where the method is declared is taken into account, also in situations as in the following example:
class Parent {
  int m (Parent p) { return 0; }
  int m (Heir h) { return 1; }
}

class Heir extends Parent {
  int m (Parent p) { return 2; }
}

public class OverloadingTest {
  public static void main(String argv[]) {
    Parent p1 = new Parent();
    Parent p2 = new Heir();
    Heir h = new Heir();
    System.out.println(p2.m(p2)); // 2
    System.out.println(p1.m(h)); // 1
    // System.out.println(h.m(h)); // ambiguous, static error
  }
}

The example above also illustrates that overloading and overriding are two distinct notions: once the method to be associated to a call has been selected at compile time (depending on the static types of the receiver and the arguments), the version which is invoked at run-time is determined by dynamic binding.

5.5 Languages where classes are distinct from types

In Fig.6 we show how to modify MINIJAVA in order to have types distinct from classes. Classes are defined as before; however, types are no longer class names but are now either primitive types or object types, which are pairs of fields and methods types; moreover, it is possible to declare type identifiers. However, note that type declarations are introduced only as abbreviations, since we consider now structural equivalence among types (two types are equal if they have, recursively, equal subcomponents). Accordingly with the discussion in Sec.5.3, an object type is a subtype of another if: it has possibly more fields and methods; common fields have exactly the same type; common methods have more specific return type and more generic parameter types. This can be formally expressed by the following rules:

\[
\frac{\forall j \in 1..h \vdash MT_j \leq MT'_j}{\vdash \{ f_i : T_i^{j\in 1..m}, m_j : MT_j^{j\in 1..h} \} \leq \{ f_i : T_i^{j\in 1..n}, m_j : MT'_j^{j\in 1..k} \}} \quad m \leq n, h \leq k
\]

\[
\frac{\vdash T \leq T' \quad \forall i \in 1..n \vdash T_i' \leq T_i}{\vdash \langle T, T_1 \ldots T_n \rangle \leq \langle T', T_1 \ldots T'_n \rangle}
\]

\[\vdash \text{boolean} \leq \text{boolean} \quad \vdash \text{int} \leq \text{int}\]

Note that the subtyping relation does no longer depend on the class type environment.

We briefly discuss now the language SOOL used throughout [2], referring to this book (in particular, read examples in Sec.1.2 and 1.3, and Chapter 10) for more details. See also [1] 3.1 In this language types and classes are different, as in the modification

Figure 6: Syntax of a language with types distinct from classes
of MINIJAVA we have shown below. There are some (non significant) differences in syntax, that we illustrate by the following example where we write our running example of rectangles and cuboids with the syntax of SOOL (exercise: find and describe all these differences):

```lisp
RectangleType = ObjectType { setLength : Integer -> Void;
  setWidth : Integer -> Void;
  area : Void -> Integer }

CuboidType = ObjectType { setLength : Integer -> Void;
  setWidth : Integer -> Void;
  setHeight : Integer -> Void;
  area : Void -> Integer;
  volume : Void -> Integer }

class Rectangle {
  length : Integer := 0;
  width : Integer := 0;
  function setLength (length:Integer) : Void is
    {self.length := length}
  function setWidth (width:Integer) : Void is
    {self.width := width}
  function area () : Integer is
    {return self.width * self.length}
}

class Cuboid inherits Rectangle {
  height: Integer := 0;
  function setHeight (height:Integer) : Void is
    {self.height:= height}
  function volume () : Integer is
    {return self <= area() * self.height}
}
```

The example above is written in the abbreviated SOOL syntax; see [2] for the expanded version (exercise: write the example above in the expanded version).

A much more important difference is that SOOL also allows anonymous class expressions, analogously to what happens in functional languages such as ML where we can write anonymous function expressions (see [2]). Moreover, there are some differences in the type system. The most important is that object types only contain methods, that is, fields are considered to be always hidden from outside the object, but are inherited in subclasses (note that this is different from the protected visibility level in Java). As a consequence, three different types are considered (see [2] Def.10.2.1):

- **object types**, which are sequences of methods with their types, and are used as types of objects;
- **class types**, which are types of class expressions, and are pairs consisting of a sequence of fields with their types and a sequence of methods with their types;
- **visible object types**, which are formally equal to class types and are used as types of self.

Another less significant difference is that reference types are explicitly introduced as types for variables. Note also that SOOL has no constructors, but they can be simulated by defining functions which return classes (see [2]). For instance, a constructor with two integer parameters in class Rectangle can be simulated as follows:

```lisp
class Rectangle (l : Integer, w : Integer) {
  length : Integer := l;
  width : Integer := w;
  ...
}
```

### 5.6 MyType

See also [1] 2.8, 3.4, 3.5, [2] 3.2.2, Chapter 16.

In many situations, class definitions are recursive in the sense that the definition of a class can use the type of the objects which are instances of the class (the class itself in a language where classes and types coincide). We have already shown examples of this situation in Sec.5.3. For instance, let us consider again the following example where the return type of a method is the type of the objects of the class.
class Rectangle {
    int length, width;
    Rectangle clone() {
        Rectangle clone = super.clone(); //(*) illegal in Java;
        clone.length = length;
        clone.width = width;
        return clone;
    }
}

class Cuboid extends Rectangle {
    int height;
    Cuboid clone() { //illegal in Java
        Cuboid clone = super.clone(); //(*) illegal in Java
        clone.height = height;
        return clone;
    }
}

Note that, even in case it is allowed to write a more specific return type in overriding methods, the lines (*) would be illegal and we should insert a down cast. This problem can be fixed in Java by the programmer by using constructors.

class Rectangle {
    int length, width;
    Rectangle (int length, int width) {
        this.length = length;
        this.width = width;
    }
    Rectangle clone() {
        return new Rectangle(length, width);
    }
}

class Cuboid extends Rectangle {
    int height;
    Cuboid (int length, int width, int height) {
        super(length, width);
        this.height = height;
    }
    Cuboid clone() {
        return new Cuboid(length, width, height);
    }
}

Note also that, in case the heir class does not add instance variables we would like the return type to change “for free”.

We have also already seen examples in Sec.5.3 in which the parameter type of a method is the type of the objects of the class. An analogous problem is encountered when implementing recursive data structures where an object is expected to have object subcomponents of the same type.

Let us consider for instance the following class declaration which implements linked nodes (the example is the same of [2] 3.2.2 but we use a Java syntax).

class Node {
    protected int value;
    protected Node next;
    int getValue () { return value; }
    Node getNext () { return next; }
    void setValue (int value) { this.value = value; }
    void setNext (Node next) { this.next = next; }
}

Assume now that we want to define a subclass which implements doubly linked nodes.

class DoubleNode extends Node {
    // next should either automatically or manually become a DoubleNode
    // getNext should either automatically or manually now return a DoubleNode
}
protected DoubleNode previous;
DoubleNode getPrevious() { return previous;
void setPrevious(DoubleNode previous) {this.previous = previous;
void setNext (DoubleNode next) {//would be overloading in Java
    super.setNext(next);
    next.setPrevious(this);
}

In all these examples, we would like the type of the instances of a class to “automatically” change in subclasses. This can be obtained by introducing a special keyword MyType which represents the type of this. Having this feature, the examples above could be written as follows:

class Rectangle {
    int length,width;
    MyType clone() {
        Mytype clone = super.clone();//well-typed now
        clone.length=length;
        clone.width = width;
        return clone;
    }
}

class Cuboid extends Rectangle {
    int height;
    Mytype clone() {
        Mytype clone = super.clone();//well-typed now
        clone.height=height;
        return clone;
    }
}

Rectangle r = new Rectangle();
Rectangle r1 = r.clone();
Cuboid c = new Cuboid();
Cuboid c1 = c.clone()

class Node {
    protected int value;
    protected MyType next;
    int getValue () { return value;}
    MyType getNext () { return next;}
    void setValue (int value) {this.value = value;}
    void setNext (MyType next) {this.next = next;}
}

class DoubleNode extends Node {
    protected MyType previous;
    MyType getPrevious() { return previous;
    void setPrevious(MyType previous) {this.previous = previous;
    void setNext (MyType next) {//well-typed now
        super.setNext(next);
        next.setPrevious(this);
    }
}

The object types corresponding to the above classes are as follows:

RectangleType =  ObjectType { clone : Void -> MyType} 
CuboidType =  ObjectType { clone : Void -> MyType} 
NodeType =  ObjectType { getValue : Void -> Integer;
    getNext: Void -> MyType; 
    setValue: Integer -> Void; 
    setNext : MyType -> Void} 
DoubleNodeType =  ObjectType { getValue : Void -> Integer; 

getNext: Void -> MyType;
getPrevious: Void -> MyType;
setValue: Integer -> Void;
setNext : MyType -> Void;
setPrevious : MyType -> Void}

The object type corresponding to Cuboid turns out to be a subtype of the object type corresponding to Rectangle. However, the object type corresponding to DoubleNode is not a subtype of the object type corresponding to Node, since MyType is used as parameter and field type (see at the end of [2] 3.2.2 for an example showing that considering the object type corresponding to DoubleNode to be a subtype of the object type corresponding to Node leads to a run-time error).

Indeed, the rule for subtyping in presence of MyType can be informally expressed as follows (see [2] 16.2 for the formalization):

if we have two object types

Type1 = ObjectType { TypeExpr1(MyType) }
Type2 = ObjectType { TypeExpr2(MyType) }

where TypeExpr1(MyType), TypeExpr2(MyType) are two type expressions which can contain MyType, and assuming that Type1 is a subtype of Type2 we are able to prove that TypeExpr1(Type1) is a subtype of TypeExpr2(Type2), then we can conclude that Type1 is a subtype of Type2.

For instance, in the example above, assuming DoubleNode to be a subtype of Node, we should have that

ObjectType { getValue : Void -> Integer;
  getNext: Void -> DoubleNode;
  getPrevious: Void -> DoubleNode;
  setValue: Integer -> Void;
  setNext : DoubleNode -> Void;
  setPrevious : DoubleNode -> Void}

is a subtype of

ObjectType { getValue : Void -> Integer;
  getNext: Void -> Node;
  setValue: Integer -> Void;
  setNext : Node -> Void;
  setPrevious : Node -> Void;

and this is not the case, since the type of the method setNext changes in a covariant way. Thus we lose subtyping in this case, hence it will be ill-typed, for instance, to assign an expression of type DoubleNode to a variable of type Node, but we are at least able to write DoubleNode as a subclass of Node.

Note also that the rules for typing expressions must take into account MyType, in particular (informally and roughly, see [2] 16.2 for the formalization) we have the following rules.

• A class is typechecked assuming that MyType is the corresponding object type (which can contain occurrences of MyType); the type of self is MyType (or its “visible” version in the case in which fields are considered to be not visible outside the object). For instance, the call next.setPrevious(this) in class DoubleNode can be proved to be well-typed by the following argument:
  – next has type MyType, which in this case is DoubleNode;
  – in the type DoubleNode there is a method setPrevious : MyType -> Void;
  – this has type MyType, hence the call is well-typed.

• When a method call or field access is typechecked, its type is obtained by replacing MyType by the type of the receiver. For instance, the type of a call node.getNext() with node of type Node can be deduced by the following argument:
  – node has type Node;
  – in the type Node there is a method getNext : Void -> MyType;
  – hence, the call is well-typed and has type Node.

The MyType construct is, for instance, supported in the Eiffel language (see [2] 4.2), where, however, since subtyping is determined by inheritance, the type system is not sound.
References


