Chapter 2

Preliminaries

2.1 Object-oriented programming

Object-oriented programming (OOP) is a programming paradigm that utilizes "objects" to structure applications and computer programs. In OOP, classes act as user-defined data types that serve as blueprints for creating unique instances called objects. These objects can represent real-world entities or abstract concepts. Each class defines both fields and methods, where fields represent the state of an object, and methods describe its behaviors.

Fields in a class are variables that store data related to instances of the class, and they define the state of an object. Class fields, on the other hand, belong to the class itself. Methods are functions encapsulated within the class, enabling code reusability and defining the behaviors associated with an object. Each method within a class begins with a reference to an instance object, making them instance methods.

A method signature consists of the method name coupled with its parameter list. An interface in OOP only contains method declarations, specifying what methods a class must implement but not how to implement them. This allows multiple classes to implement the same interface in different ways, giving programmers the flexibility to change implementations without affecting the code that relies on these interfaces.

The basic features of object-oriented programming include encapsulation, data abstraction, inheritance, and polymorphism, all of which are consistently adhered to by object-oriented programs. *Encapsulation* involves using classes to encapsulate both fields and methods, ensuring that the internal state of these objects is shielded from the outside world—a principle known as information hiding. Methods serve as the public interface through which interaction with object data is controlled, preserving data integrity and restricting unauthorized access. *Data abstraction* allows for the creation of abstract classes or interfaces that define the essential feature of a data type, without revealing the underlying implementation details. This enables users to interact with objects at a higher, more abstract level, which facilitates more comfortable and straightforward usage. *Inheritance* is widely used in object-oriented libraries to promote code reusability and to establish a relationship between supertype and subtype. A subtype inherits fields (if both types are classes) and methods from a supertype, and it also has the ability to override or extend these inherited features. This makes it easy to add functionalities to existing classes or interfaces without modifying them. *Polymorphism* enables objects to be treated as instances of their supertype, leading to simpler code and fewer errors. In object-oriented programs, polymorphism is often implemented through method overriding in subtypes, allowing for different implementations under the same method name.

The *programming to interfaces* principle [26] is one of the most important principles in OOP. This principle is supported by the feature of encapsulation and data abstraction, allowing developers of client code to focus on essential functionalities rather than the need to consider irrelevant implementation details. By focusing on what an object should do rather than how it should do it, this principle enables a higher level of abstraction and decoupling in software systems. Furthermore, when combined with the *design by contract* [27] principle, programming to interface becomes a crucial strategy for managing the complexity of software today. This combination ensures a structured approach to software design, where the focus is on fulfilling clearly defined contracts, thereby enhancing the reliability and maintainability of software systems.

2.2 Java Collection Framework

The Java Collections Framework is a comprehensive set of classes and interfaces provided by Java for working with collections of objects. It provides basic data structures and is among the most widely used libraries. The Java Collection Framework is illustrated in Figure 2.1.

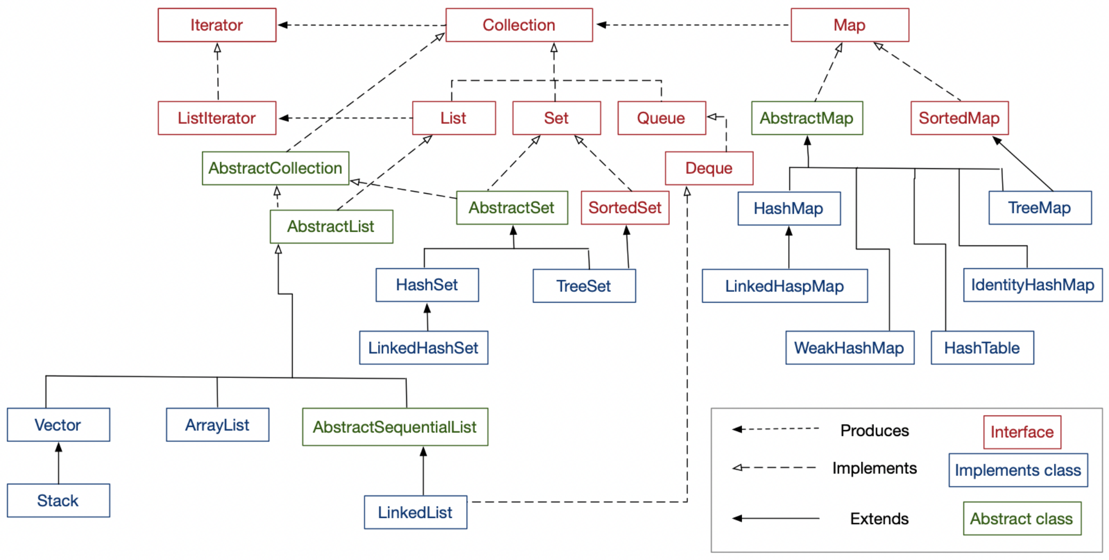


Figure 2.1: Java Collection framework.

The Java Collection Framework has a behavioral subtype hierarchy [28]. The `Collection` interface serves as the topmost type and includes three sub-interfaces: `List`, `Set`, and `Queue`. These sub-interfaces are implemented by some abstract classes and finally lead down to the concrete implementation classes at the bottom of the hierarchy, such as `ArrayList`, `HashSet`, and `LinkedList`, etc.

The `Collection` interface outlines all the fundamental operations for collections, such as `add`, `remove`, and other methods for querying and manipulating a collection of objects.

```
public interface Collection {
    boolean add(Object o);
    boolean addAll(Collection c);
    boolean remove(Object o);
    boolean contains(Object o);
    boolean isEmpty();
    Iterator iterator();
    ...
}
```

Listing 2.1: The part of `Collection` interface.

The `iterator` method, declared within the `Collection` interface, returns an object that implements the `Iterator` interface. The `Iterator` interface is another key component of the Java Collection Framework, offering a way to enumerate all the elements in a collection.

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

Listing 2.2: The `Iterator` interface.

In addition to collections, the framework also includes a variety of `Map` interfaces and classes. These `Map` entities are used to store key/value pairs. Even though a `Map` is not considered as a collection, it is fully integrated into the Java Collections Framework.

The `LinkedList` class is one of the most important classes in the Java Collection Framework. This class serves as a particular case study for the formal verification of objection-oriented libraries, as discussed in Section 3. It was introduced in Java version 1.2 as part of the Java Collection Framework in 1998. Figure 2.1 shows how `LinkedList` fits in the type hierarchy of this framework: `LinkedList` implements the `List` interface, and also supports all general `Collection` methods as well as the methods from the `Queue` and `Deque` interfaces. The `List` interface provides positional access to the elements of the list, where each element is indexed by Java's primitive `int` type. Each element in a `LinkedList` is stored as a separate object referred to as a node, containing data and references to the previous and next elements in the list.

2.3 Theorem prover overview

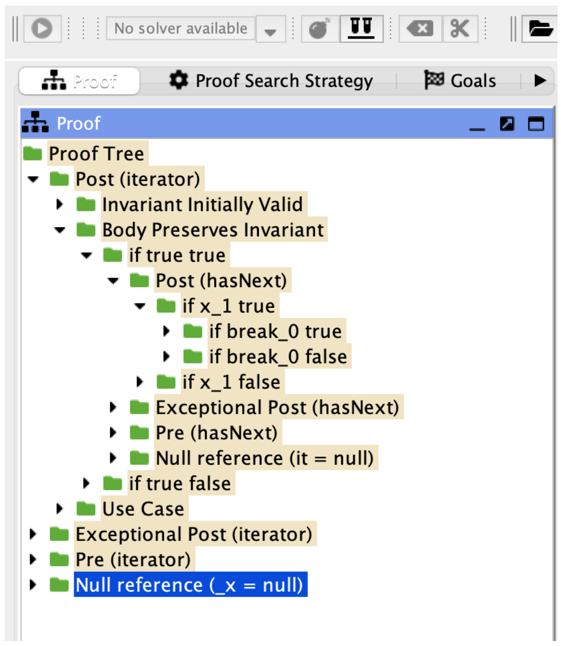
In our setup, we distinguish domain-specific theorem provers from general-purpose theorem provers. The theorems of a domain-specific theorem prover are correct pairs of programs and specifications and thus can be seen as giving axiomatic semantics to programs and specifications. In our case study, we use the state-of-the-art KeY theorem prover, as KeY is tailored to the verification of Java programs. A general-purpose theorem prover, in contrast, is oblivious to the intricate details of programs and the specifications in question: e.g. it is not needed to formalize the semantics of Java nor JML in a general-purpose theorem prover. As a general-purpose theorem prover, we choose the Isabelle/HOL theorem prover. In this section, we will introduce both the KeY theorem prover and the Isabelle/HOL theorem prover, offering an overview of both tools.

2.3.1 KeY theorem prover

JML [24] is a specification language for Java that supports the design-by-contract paradigm. Specifications are embedded as Java comments alongside the program. A method precondition in JML is given by a `requires` clause and a postcondition is given by `ensures`. JML also supports class invariants. A class invariant is a property that all instances of a class should satisfy. In the design by contract setting, each method is proven in isolation (assuming the contracts of methods that it calls), and the class invariant can be assumed in the precondition and must be established in the postcondition, as well as at all call-sites to other methods. To avoid manually adding the class invariant at all these points, JML provides an `invariant` keyword which implicitly conjoins the class invariant to all pre- and postconditions. Method contracts may also contain an `assignable` clause stating the locations that may be changed by the method (if the precondition is satisfied), and an `accessible` clause that expresses the locations that may be read by the method (if the precondition is satisfied). Our approach uses all of the above concepts.

JML also allows annotations of ghost fields and model fields. Ghost fields are virtual fields that become part of the modelled state of an object on the heap, but are never present when actually executing a Java program. Like normal fields, the ghost fields are assigned a default value at object initialization and can be explicitly changed by JML `set` annotations. These annotations occur anywhere in method bodies where otherwise a normal statement can be expected. Model fields are introduced as function symbols, and several axioms are added that allow the definition of model fields to be substituted during proof. An example model field is a *class invariant*, which is implicitly assumed to hold the state of an object between method invocations.

KeY [23] is a semi-interactive theorem prover for Java programs (typically $> 95\%$ of the proof steps are automated). The input for KeY is a Java program together with a formal specification in a KeY-dialect of JML. The user proves the specifications method-by-method. KeY generates appropriate proof obligations and expresses them in a sequent calculus (see Figure 2.2), where the formulas inside the sequent are multi-modal dynamic logic formulas in which Java program fragments are used as the modalities. To reduce such dynamic logic formulas to first-order formulas, KeY symbolically executes the Java program in the modality (it has rules for nearly all sequential Java constructs). Once the program is fully symbolically executed, only formulas without Java program fragments remain. The main reference work on the KeY system is the KeY book [29].



- Proof Tree
  - Post (iterator)
    - Invariant Initially Valid
    - Body Preserves Invariant
      - if true true
        - Post (hasNext)
          - if x_1 true
            - if break_0 true
            - if break_0 false
          - if x_1 false
        - Exceptional Post (hasNext)
        - Pre (hasNext)
        - Null reference (it = null)
      - if true false
    - Use Case
  - Exceptional Post (iterator)
  - Pre (iterator)
  - Null reference (_x = null)

Figure 2.2: Proof tree in KeY(version 2.8.0).

The KeY system consists of three main components: a proof system, a translator of Java programs annotated with JML into proof obligations, and an interactive tool for constructing proofs.

The logic underlying KeY is *JavaDL*, a program logic that directly incorporates Java program fragments. The program logic is a multi-modal logic: $\langle P \rangle \varphi$ expresses that executing the program fragment P definitely terminates and φ holds in the final state; and $[P]\varphi$ expresses *if* the program fragment P terminates, *then* φ holds in the final state. The formula $\varphi \rightarrow \langle P \rangle \psi$ expresses that if φ holds in the initial state, then execution of P terminates in a state for which ψ holds. See [29].

The program logic distinguishes *program variables* from *logical variables*: the value of a program variable can be changed throughout executing a program, whereas a logical variable always has the same value. As logical variables can never be modified by a program fragment, they are used as so-called *freeze variables*.

The proof system that KeY uses to establish the validity of formulas is given as a sequent calculus. A sequent $\varphi_1, \ldots, \varphi_n \Rightarrow \psi_1, \ldots, \psi_m$ consists of n antecendents and m consequents, all formulas of JavaDL, with the usual interpretation: $\varphi_1, \ldots, \varphi_n \Rightarrow \psi_1, \ldots, \psi_m$ means that if all φ_i on the left are true, then at least one ψ_j on the right is true. Derivability of a sequent in the proof system is as usual by means of deduction rules, assembled into a proof tree. Next to the deduction rules for classical first-order logic, the proof system also consists of a large number of other rules.

Deduction rules are given by means of lightweight tactics (called taclets [30]) that perform modifications on the sequent one is proving: e.g. split branches in the proof tree, substitute variables, rewrite terms, or close a branch. There are approximately 1750 rules that implement symbolic execution for Java program fragments, and implement the theories of many sorts: integers, sequences, heaps, location sets, and others.

Of particular interest to us are rules concerning *updates* and *heaps*. Some rules transform modalities with program fragments into so-called update modalities. Updates always terminate and they assign JavaDL terms to program variables. As such, updates cannot assign program variables to side-effectful expressions. Given a formula φ, then $\{\mathtt{x}_1 := t_1 || \ldots || \mathtt{x}_n := t_n\}\varphi$ is a formula where in parallel $\mathtt{x}_i$ are updated to t_i. Updates are simplified by substitution.

There is a hidden and implicit program variable in JavaDL that is the `heap` of heap sort. The heap is used to model the storage of objects, that is, the value of fields associated with object references. From a practical perspective, updates of program variables other than the heap can be thought of as stack variables. Such program variables can *refer* to objects on the heap. Heap updates are tracked by updating the `heap` program variable. Heap updates and program variable updates easily form complex expressions, where both kinds of updates can be intertwined. See [29, Section 2.4.3] and [29, Section 6.4].

Next to the built-in sorts of integers, sequences, heaps, and location sets, are Java types. Each Java type has its own sort in JavaDL that models (infinitely many) references to objects, including the `null` reference. References can be explicitly coerced between sorts, to model subtypes: e.g. every reference can be coerced to a reference of sort `Object`. A heap assigns values to the fields of a non-null reference. Object references are global, but the creation status of each object is a special Boolean field `<created>` local to each heap. An object becomes created in some heap by taking a fresh reference, that has no value yet assigned to its creation field in that heap, and setting that field to `true`. Fields can only be assigned to non-null object references for which their created field is true. A heap is *well-formed* if a finite number of references have `<created>` set to true.

An important aspect of Java programs is method calls. There are three main issues: calling a method may introduce new program variables that shadow older program variables, calling a method may change the heap, and it is possible to call a method on an object for which its exact type is only known at run-time.

To solve the issue of overshadowing older program variables, KeY uses *method frames*; before a method frame is created, it is ensured that old program variables and new program variables do not collide by renaming the program variables of a method body. Since the `this` keyword cannot be renamed, the method frame provides a context in which program fragments evaluate `this` references; it also tracks where the method return value must be placed.

The implicit heap variable is also stored in the method frame, referred to as the before heap. Any heap update within the method is performed on a separate program variable. Statements following a method call are performed using the heap

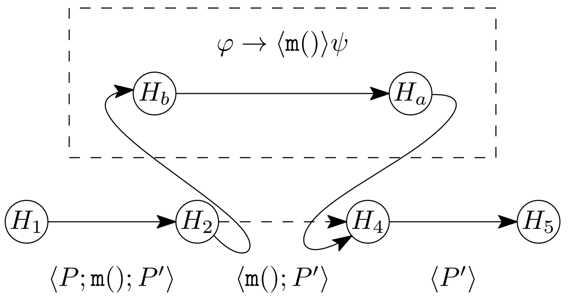

$\varphi \to \langle \mathtt{m()} \rangle \psi$
H_b
H_a
H_1
H_2
H_4
H_5
$\langle P; \mathtt{m()}; P' \rangle$
$\langle \mathtt{m()}; P' \rangle$
$\langle P' \rangle$

Figure 2.3: Heaps are 'threaded' through method calls. In heap H_1 the program fragment P is executed, resulting in heap H_2. Either a contract for method `m()` is employed, which relates the before heap H_b to some heap after the method call H_a; or, the method body is unfolded by wrapping it in a method frame that resolves overshadowing of program variables. In both cases, to call the method, heap H_b is equal to H_2 and heap H_a is equal to H_4. Then the program fragment P' that occurs after the method call is executed using the heap returned by the method.

as it is after the method call completes: there are rules for 'threading' the heap through a method call, see Figure 2.3. There are roughly two ways of treating method calls: unfolding their method body wrapped in a method frame, or using its method contract.

The issue of run-time types is solved using method contracts. Method contracts and other annotations of Java programs are specified in terms of the Java Modeling Language (see [31]) but with KeY-specific extensions (see [29, Chapter 7]). Each method is annotated by a contract that specifies pre- and post-conditions: all implementations of this method should adhere to the contract. The contract determines corresponding correctness formulas in JavaDL: if they are verified, then the corresponding method is correct with respect to its method contract. Consequently, method contracts can be reused in other proofs involving program fragments that call such a method.

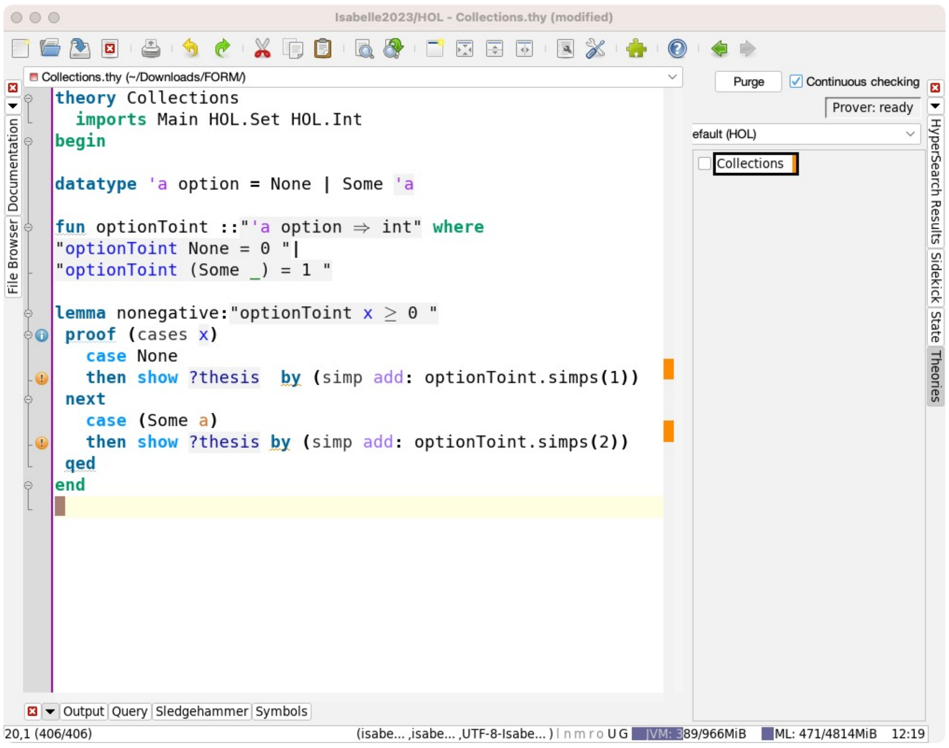
Java programs annotated with JML are entered into the KeY system and the verification begins by generating one or more *proof obligations*. Using the interactive tool, a proof tree is constructed by applying rules until the branches of the tree are closed. Each rule application may result in zero or more branches: in the case of multiple branches, every branch needs to be closed. The tool also provides automation, that automatically applies proof rules using heuristics. This speeds up the verification effort considerably. Finally, after all proof obligations appear as the conclusion of a closed proof tree, the verification effort is done. The proof trees can be stored in a file and reloaded and inspected later.

2.3.2 Isabelle/HOL theorem prover

Isabelle/HOL (Isabelle instantiated with Church's type theory) [32] is an interactive theorem prover that supports the formalization and verification of mathematical theorems and software systems. The main reference for Isabelle/HoL is its reference manual [33, 34].

Isabelle/HOL employs higher-order logic (HOL) to allow for a broader class of mathematical statements to be represented and proven. The underlying logic is conducive to reasoning about functional programming constructs, data types, and even imperative programming paradigms. It comes equipped with a variety of decision procedures and tactical theorem-proving capabilities that automate or semi-automate the verification process [33].

Functional programs in Isabelle/HOL are outlined as shown in Figure 2.4. It defines both datatypes and functions within a theory. Isabelle comes with a large theory library of formally verified mathematics, including elementary number theory, analysis, algebra, and set theory [35].



```
theory Collections
  imports Main HOL.Set HOL.Int
begin

datatype 'a option = None | Some 'a

fun optionToint ::"'a option ⇒ int" where
"optionToint None = 0 "|
"optionToint (Some _) = 1 "

lemma nonegative:"optionToint x ≥ 0 "
 proof (cases x)
   case None
   then show ?thesis  by (simp add: optionToint.simps(1))
 next
   case (Some a)
   then show ?thesis by (simp add: optionToint.simps(2))
 qed
end
```

Figure 2.4: Isabelle/HoL proof system(version Isabelle2023/HOL).

Isabelle/HOL includes a definitional package for data types, as described in the work of Biendarra et al. [36]. The definition mechanism provides so-called *freely generated* data types: the user provides some signature consisting of constants and function symbols and their types, and the system automatically derives (rather than postulates) characteristic theorems. Under the hood, each data type definition is associated with a Bounded Natural Functor (BNF) that admits a non-trivial initial algebra [37]. User-defined datatypes are defined using the *datatype* keyword. The definition usually outlines the constructor functions that can be used to create instances of the datatype, along with the types of arguments these constructors can take.

Functions in Isabelle/HOL are declared using the *fun* keyword. These functions are characteristically pure, meaning they do not have side effects. Functions are usually defined by specifying their behavior over user-defined or built-in datatypes, commonly utilizing pattern matching to unambiguously identify the expected output for specific inputs. Additionally, these functions can also be recursively defined; they may call themselves with altered arguments until a termination condition, often referred to as the base case, is met. This recursion is typically defined with rigorous formal properties to ensure termination and correctness, aligning with the overall goal of theorem proving and formal verification in Isabelle/HOL.

The keywords *theorem* and *lemma* in Isabelle play crucial roles in formalizing and verifying mathematical claims. While a lemma is generally a minor result used as a foundational stepping stone, a theorem is a more significant mathematical proposition that has been rigorously proven based on previously established lemmas, theorems, and axioms. These two keywords are interchangeable and merely indicate the level of importance we assign to a given proposition. Isabelle's rich theory library provides a wide array of pre-proven theorems and lemmas. These resources not only expedite the formal verification of new mathematical claims but also contribute to a robust foundational framework. This framework is invaluable for researchers across multiple disciplines, enabling them to build upon proven results and advance both theoretical and applied research with confidence.