Advanced Programming - Appunti

Francesco Lorenzoni

September 2023

Contents

Chapter 1

Introduction

1.1 19 - Settembre

1.1.1 Info and Contact

Pagina del corso

1.1.2 Framwork

A software **framework** is a collection of common code providing generic functionality that can selectively overryidden or specialized by user code prvoiding specific functionality.

When using *framworks* htere is an **inversion of control**. Differently from what happens when using libraries, the program-flow is dictated by the framework, not the caller.

1.1.3 Design Patterns

A **design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is characterized by:

- **Name**
- **Problem Addressed**
- **Context** - Used to determine applicability
- **Forces** - Constraints or issues that the solution must address
- **Solution** - It must resolve all *forces*

1.2 20 - Settembre

Useful tool, to see preprocessor output, compiling, ecc.

1.2.1 Programming Languages

A **PL** is defined via **syntax**, **semantics** and **pragmatics**[1].

Syntax

Used by the compiler for *scanning* and *parsing*. The *lexical* grammar defines the syntax of token (e.g. "for" blocks, constants)

Semantics

Semantics might be described using natural language, which even if precise, allows amibguousity. Formal approches to semantics definition are:

1. Denotational - Mapping every syntactic entity with a mathematical entity
2. Operational - Defining a computation relation in a form $e \Rightarrow v$, where e is a program
3. Axiomatic - Based on Hoare-triples $Precondition \land Program \Rightarrow Postcondition$

However, they rarely scale to fully-fledged programming languages.

[1] the way in which the PL is intended to be used in practice

Pragmatics

Pragmatics include coding conventions, guidelines for elegant code, etc.

1.2.2 Programming Paradigms

Paradigms belong to languages *pragmatics*, not to the way the language is defined, i.e. not syntax nor semantics.

1. **Imperative**
2. **Object-oriented**
3. **Concurrent** - Processes, communication, ...
4. **Functional**
5. **Logic** - Assertions, relations, *strange sorceries...*

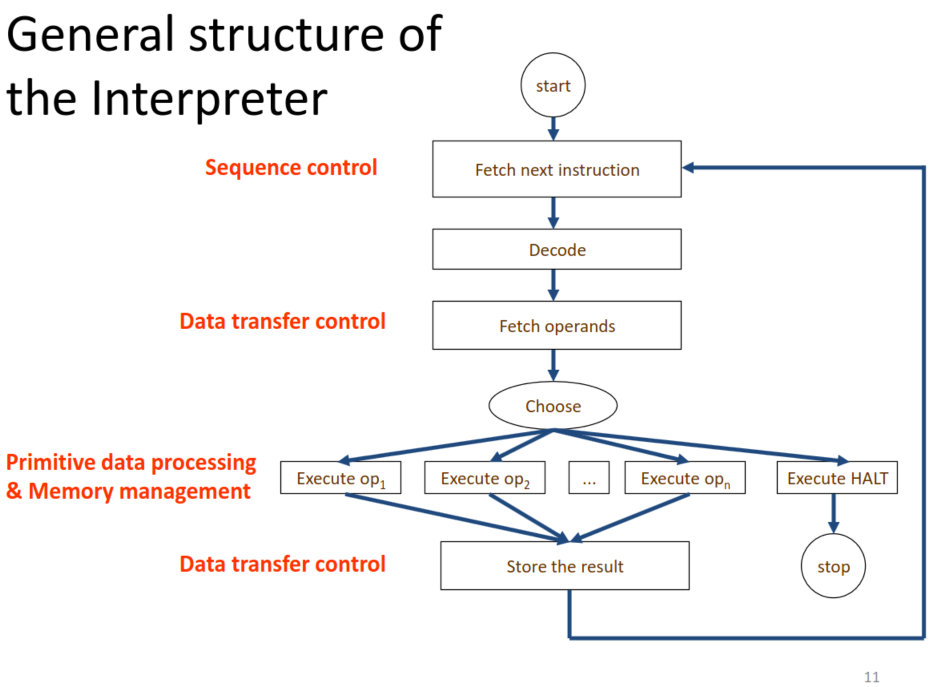
Modern PLs, provide constructs and solutions to program in all these paradigms

1.2.3 Implementing PLs

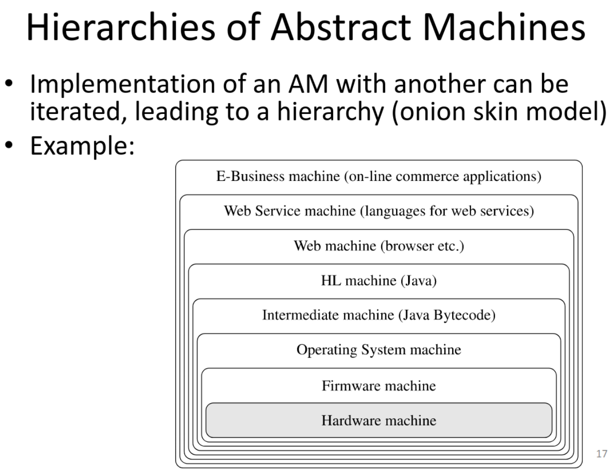
- Programs written in **L** must be *executable*
- Every language **L** implicitly defines and *Abstract Machine* M_L having **L** as a Machine Language
- Implementing M_L on an existing host machine M_O via compilation or interpretation (or both) makes programs written in **L** executables

An **Abstract Machine** M_L for L is a collection of data structures and algorithms which can perform the storage and execution of programs written in L.
Viceversa, M defines a language L_M including all programs which can be executed by the interpreter M.
There is a bidirectional correspondance between machines and languages components.

$$\begin{array}{ccc} \textit{Primitive data processing} & \longleftrightarrow & \textit{Primitive data types} \\ \textit{Sequence control} & \longleftrightarrow & \textit{Control structures} \\ \textit{Data transfer control} & \longleftrightarrow & \textit{Parameter passing and value return} \\ \textit{Memory management} & \longleftrightarrow & \textit{Memory management} \end{array}$$



In computer science one of the main focuses is **abstraction**, as can be seen in this hierarchical scheme.



A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Implementing PLs - Wrap Up

- L High-level programming language
- M_L Abstract machine for L
- M_O host machine

Pure Interpretation

...PIC HERE

M_L is interpreted over M_O. It isn't very efficient, mainly because of fetch-decode phases

Pure Compilation

...PIC HERE

L programs are translated into L_O, the machine laguange of M_O, hence, M_L is not realized at all and the programs are directly executed on M_O.
Compilation is more efficient than *Interpretation*, but produced code is larger

Both

...PIC HERE

All real languages use both *interpretation* and *compilation*,

Some languages, e.g. Java, use an intermediate Abstract Machine, called a *Virtual Machine*, which increases *Portability* and *Interoperability*.

Chapter 2

JVM

25 - Settembre

2.1 Runtime System

Every language defines and **execution model**, which is (partially) implemented by a **runtime system**, providing runtime **support** needed by both *compiled* and *interpreted* programs.

A **Runtime system** includes (eventually):

- Code:
  - in the executing program generated by the compiler
  - running in other threads/processes]
- Language libraries
- Operating system functionalities
- The interpreter/virtual machine itself

Runtime support can be needed for various reasons:

- Memory Management → Stack Management, Heap Management
- I/O operations → File System, Sockets, I/O devices
- Intercation with runtime environment
- Parallel execution (threads/processes)
- Dynamic binding type checking
- Dynamic loading and linking of modules
- Debugging
- ¿Code Generation?
- ¿Verification and Monitoring?

2.1.1 JRE

The **Java Runtime Environmnet** includes **JVM** and **JCL** (*Java Class Library*).

2.2 JVM

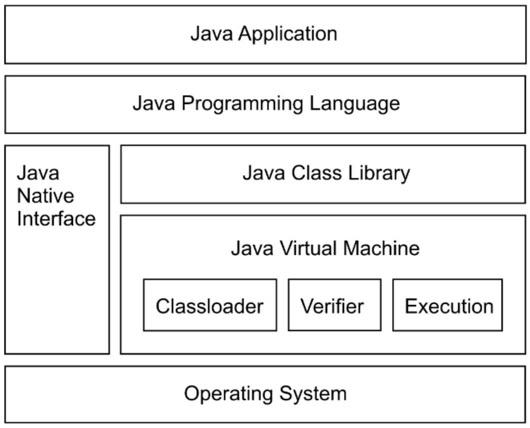
JVM is an *abstract* machine, defined by the documentation, which omits details on stuff like memory layout of runtime data area, garbage-collection, internal optimization, and even the representation of the `null` constant. The JVM specification, instead, defines precisely a machine indipendent *"class file format"* that all JVM implementations must support; it also imposes strong **synctatic** and **structural constraints** on the code in a class file.

The **JVM** is not *register-based*, instead it is a *multi-threaded* ***stack***[1] *based machine*. Id est the JVM pops intructions

[1] Not to be confused with the stack of activation records!

from the top of **operand stack** of the current frame, and pushes their result on the top of the **operand stack**. The **operand stack** is used to:

- ◇ Pass arguments to functions
- ◇ Return results from a function
- ◇ Store intermediate values while evaluating expressions
- ◇ Store local variables



2.2.1 Data types

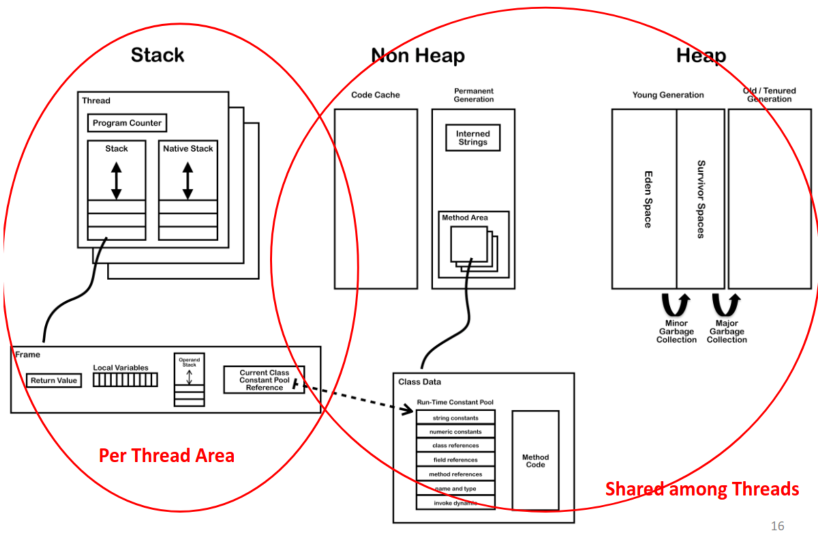
`.class` file are platform independent external represantions, which are represented internally by the JVM using simpler data types, which are implementation dependent.

- ◇ **Primitive types**
  - Numerica integral
  - Numeric floating point
  - boolean (support only for arrays)
  - internal (for exception handling)
- ◇ **Reference types**
  - Class types
  - Array types
  - Interface types

  No type information on local variables at runtime, there are only operand types specified by **opcodes** e.g. `iadd, fadd`, . . .
- ◇

2.2.2 Threads

There are some runtime areas of the JVM related to a single thread, while others are shared among threads



All Java Programs are multithreaded, since there is at least a `main` thread running the user's program, and many

daemons:

- Garbage collector
- Signal Dispatching
- Compilation
- ¿ ... ?

JVM doesn't poses strong implementation constraints, by defining a precise abstract consistency model, including volatiles, allowing non-atomic longs and doubles, distinguishing working-memory and general store.

2.2.3 per-thread Data Areas

- **pc** pointer to next instruction in *method area*
  undefined if current method is native
- **Java stack**: stack of *frames* (or *activation records)*
- **Native stack**: used for invocation of natve functions through the *Java Native Interface* (*JNI*)

Considering the **structure** of **frames**, each one is composed by:

- **Local Variable Array** (32 bits) containing:
  1. Reference to `this`
  2. Method parameters
  3. Local variables
- **Operand stack**
- Reference to **Constant Pool** of current class

Differently from C/C++, where the **linking** phase is done before running an executable, java computes linking **dynamically** at **runtime**; this is achieved using **symbolic** references, which can be resolved using *static* (eager) or *late* (lazy) resolution.
Since the execution of a Java program must **not** depend on the JVM implementation, the JVM always behaves as if the implementation implies *lazy* resolution, even if the actual implementation provides static resolution instead.

2.2.4 shared data areas

Heap

- Memory for objects and arrays
- No explicit deallocation, it is demanded to the garbage collection.

Non-Heap

Memory for objects never deallocated

- Method area
- Interned strings
- Code cache for JIT

Just In Time (JIT) compilation refers to profiling as "hot" code areas of bytecode which may be executed many times, and storing the compiled native code in a cache in the *Non-heap* memory.

Method-area

Here `class` files are loaded. For each class a classloader reference and the following info from the `class` file are stored:

- Runtime Constant Pool
- Field data
- Method data
- Method code

Method area is *shared* among threads! Access to it mustbe *thread safe*.

This should a **permanent** area of the memory, but it may be **edited** when a new class is loaded or when a symbolic link is resolved by dynamic linking.

26 - Settembre

Constant Pool

Contains constants and symbolic references for dynamic binding. It is possible to see the constant pool of a compiled `.class` file using the command:

```
javap -v name.class
```

Displaying something resembling to:

```
  #1 = Methodref          #6.#14         // java/lang/Object."<init>":()V
  #2 = Fieldref           #15.#16        // java/lang/System.out:Ljava/io/PrintStream;
  #3 = String             #17            // Hello World
  #4 = Methodref          #18.#19        // java/io/PrintStream.println:(Ljava/lang/
     String;)V
  #5 = Class              #20            // com/baeldung/jvm/ConstantPool
  #6 = Class              #21            // java/lang/Object
  #7 = Utf8               <init>
  #8 = Utf8               ()V
  #9 = Utf8               Code
 #10 = Utf8               LineNumberTable
 #11 = Utf8               sayHello
 #12 = Utf8               SourceFile
```

2.2.5 Loading

Loading is finding the binary representation of a class or interface type with a given name and creating a class or interface from it.
Class (or Interface) C creation is *triggered* by other classes **referencing** C or by methods (e.g. reflection). If not previously loaded, `loader.loadClass` is invoked.

There are 4 **Classloaders**:

1. **Bootstrap CL**: loads basic Java APIs
2. **Extension CL**: loads classes from standard Java extension APIs
3. **System CL**: loads application classes from *classpath*
   (default application CL)
4. **User Defined CLs**: can be used for:
   - ⋄ runtime classes reloading
   - ⋄ loading network, encrypted files or on-the-fly generated classes
   - ⋄ supporting separation between different groups of loaded classes as required by web servers

Runtime Constant Pool

2.2.6 Linking

Linking includes *verification, preparation, resolution.*

1. **Verification** multiple checks at runtime, e.g. operand stack under/overflows, validity of variable uses and stores, validity arguments type. Details later on
2. **Preparation** Allocation of storage
3. **Resolution**[2] resolve symbol references by loading referred classes/interfaces

Verification is a relevant part of JVM Specification, it is described in 170pp over a total of 600pp. When a class file is loaded there is a *first* verification pass to check formatting, there is a *second* one when a class file is linked regarding only not instruction-dependant checks. During the linking phase there is a data-flow analysis on each method (*third check*), and lastly (*fourth check*) when a method is invoked for the first time.

2.2.7 Initialization

`<clinit>` initialization method is invoked on classes and interfaces to initialize class variables; it also executes static initializers. `<init>` initialization method instead is used for instances.

[2]Optional, it may be postponed till first use by an instruction

Chapter 3

JVM Instr Set & JIT

3.1 Instruction Set

26 - Settembre

Let's consider the instructions **format**. Each instr may have different *"forms"* supporting different kinds of operands. For example there are different forms of `iload` (i.e. push).

Runtime memory contains - Local variable array (frame) – Operand stack (frame) – Object fields (heap) – Static fields (method area)

Note that Java instructions are explicitly typed through **opCodes**, e.g. `dload`,`iload`,`fload`.
opCodes are bytes, allowing only for 256 distinct ones; hence it is impossible to have for each instruction on opCode per type. The JVM specification indicates a selection of which types to support for each op instruction, and not supported types have to be converted; resulting in the Instruction Set Architecture to present non-orthogonality.
Types like `byte`, `char` and `short` are usually converted to `int` when performing computations.

27 - Settembre

3.1.1 Invoking methods

`invokevirtual` causes the allocation of a new frame, pops the arguments from the stack into the local variables of the caller (putting this in 0), and passes the control to it by changing the `pc`.

- A resolution of the symbolic link is performed
- `ireturn` pushes the top of the current stack to the stack of the caller, and passes the control to it. Similarly for `dreturn`, `freturn` ...
- `return` just passes the control to the caller

There are 4 others kinds of method invocation:

- `invokestatic`: call methods with `static` modifier; *this* is not passed
- `invokespecial`: call constructors, `private` methods or *superclass* methods; *this* is always passed
- `invokeinterface`: identical to `invokevirtual`, but used when the method is declared in an interface, thus a different lookup is required
- `invokedynamic`: introduced in Java 7 to support dynamic typing[1]

(...)

3.2 JIT

AOT **Ahead of Time** *Compilation* leads to better performance in general, exploiting hardware features and variables allocation without runtime lookup; While **Interpretation** facilitates interactive debugging and testing: it allows

[1]lambda functions related?

command-line invocation.

JIT aims to get the advantages of both.

JIT differs from *AOT* since it runs in the same process of the application and competes with the app for resources, thus compilation time for JIT is more relevant than for an AOT Compiler. Besides, a JIT compiler doesn't verify classes at compile time, it is a task performed by the JVM at load time. JIT can exploit new optimization possibilities, e.g. *deoptimization* and *speculation*. A JIT takes bytecode as input and outputs machine code that the CPU executes directly.
Wrapping up:

- Code starts executing interpreted with no delay
- Methods that are found commonly executed (*hot*) are JIT compiled
- Once compiled code is available, the execution switches to it.

To identify *hot* methods, there is a **threshold** on two *per-method* counters:

1. Times the method is invoked
2. Times a brach back to the start of a loop is taken in the method

A tradeoff between "fast-to-start-but-slow-to-execute" interpreter vs "slow-to- start-but-fast-to-execute" compiled code is managed by a multi tier system.

(...)

3.2.1 Deoptimization and Speculation

Usually method executions pass in three phases:

$$\text{Interpreter} \longrightarrow \text{Low tier compiler} \longleftrightarrow \text{Optimizing compiter}$$

But sometimes **deoptimization** can happen, i.e. :

$$\begin{tikzcd} \text{Interpreter} \arrow[r] & \text{Low tier compiler} \arrow[r] & \text{Optimizing compiter} \arrow[l, bend left] \arrow[ll, bend left] \end{tikzcd}$$

Chapter 4

Component-Based software

2 - Ottobre

Component software indicates **composite systems** made of **software components**. In short, component software allows reuse, improving reliability[1] and reducing costs.
Bertrand Meyer suggests some guidelines regarding Object-Oriented software construction (1997):

1. modular
2. reliable
3. efficient
4. portable
5. timely

4.1 Definitions

*A **software component** is a unit of composition with contractually specified **interfaces** and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

A **contract**, A specification attached to an **interface** (component specification) that mutually binds the clients and providers of the components.

Context dependencies are specification of the deployment environment and run-time environment. This goes beyond the simple interfaces required and provided which are specified in the contract. Context dependencies include required tools, platform and resources.

Deployed independently means that a component can be plugged or unplugged from an architecture, even at runtime in some cases. It is common-practice to deploy "small components" called connectors, to resolve situations where two components supposed to interact do not provide identical interfaces, creating the need for a intermediary.

composition by third party means that a component may interact with third parties components without knowing the internals of such components.

4.2 Concepts of Component Model

- **Component interface** describes the operations implemented and exposed by the component
- **Composition mechanism** How components can be composed to work together to accomplish a task
- **Component platform** A platfom for the development of the compnents

Concepts should be laguage/paradigm agnostic, laying the ground for language interoperability.

The ancestors of Components are **Modules**, whose support has been introduced in Java 9, but isn't very common. Some concepts related to modules can be found in more modern notions such as classes, components and packages. For example, objects inside a module are visible to each other, but not visible from outside unless exported. Modules

[1]Industries may even require to use *certified* components

worked — pretty much like classes — as abstraction mechanism → *collection of data with operations defined on them.* In OOP the concept of **inheritance**, unknown in modules, is introduced.

4.3 Components and Programming Concepts

Components can be anything and can contain anything, they can be *collections of* classes, objects, functions/algorithms, data structures.

*¡Note that **OOP** ≠ **COP**!*

OOP isn't focused on **reuse**, instead its focus in onto appropriate domain and problem representation.

- ◇ **Component Specification** describes the behavior (as a set of *Interfaces*) of a set of Component Objects and defines a unit of implementation.
- ◇ **Component Implementation** is a realization of *Component Specification* which can be independently deployed[2].
- ◇ **Installed Component** is an installed (i.e. *deployed*) copy of a *Component Implementation.*
  A Component Implementation is deployed by registering it with the runtime environment
- ◇ **Component Object** is an instance of a *Component Implementation.* It is a runtime concept, an object with its own data and unique identity. Ideally, it is the "thing that performs the implemented behaviour". An *Installed Component* may have multiple *Component Objects*

Some examples of successful components are Plugin architectures, Microsoft's Visual Basic, Operating Systems, Java Beans, and others. It is clear that components can be purchased by independent providers and deployed by the clients, and that multiple components can coexist in the same installation. Besides, components exist on a level on abstraction where they directly mean something to the deploying client.

Recalling the comparison with modules, while modules are usually seen as part of a program, *components are parts of a system.*

[2] It does **not** mean that it cannot have *dependencies* nor that it must be a *single file*

Chapter 5

Java Beans

5.1 3 - Ottobre

"A **Java Bean** is a *reusable* software component that can be manipulated visually in a builder tool."

Typically a Bean has a GUI representation but is not necessary. What is necessary instead for a class to be recognized as a bean is that it:

- has a public constructor with no arguments
- implements `java.io.Serializable`
- is in a `jar` file with a *manifest* file that contains: `Java-Bean: True`

Beans can be **assembled** to build a new bean or application, writing clue code to wire beans together. *Connection-oriented* programming is based on the **Observer** or (*Publish-Subscribe*) paradigm. *Observers* come into play when there is a $1 : N$ dependency between objects and one of them changes state, creating the need for the others to be notified and updated. Beans must be able to run in a *design environment* allowing the user to customize aspect and behaviour. Beans provide support for some standard features:

1. **Properties** e.g. color. **Bounded** properties generate an *event* of type `PropertyChangeEvent`, while **constrained** can only change value if none of the registered *observers* "poses a veto", by raising an *exception* when they receive the `PropertyChangeEvent` object.
2. **Events**: The **Observer** pattern is based on *Events* and *Events listeners*. An *event* is an object created by an *event source* and propagated to the registered *event listeners*. Sometimes event **adaptors** can be placed between source and listener, which might implement queuing mechanism, filter events, demuxing from many sources to a single listener.
   - **Design Patterns for Events**

     ```java
     public void add<EventListType>(<EventListType> a)
     public void remove<EventListType>(<EventListType> a)
     ```
3. **Customization**
4. **Persistence**
5. **Introspection**: process of analyzing a bean to determine capabilities. There are implicit methods based on *reflection*, *naming conventions* and *design patterns*, but can be simplified by explicitly defining info for the builder tool in the `<BeanName>BeanInfo` class. Such class allows exposition of features, specifying customizer class, segregate feats in normal/expert mode, and some other stuff.
   - **Design Patterns for Simple Methods**

     ```java
     public <PropertyType> get<PropertyName>();
     public void set<PropertyName>(<PropertyType> a);
     ```
   - **Design Patterns for Simple Methods**

     ```java
     public java.awt.Color getSpectrum (int index);
     public java.awt.Color[] getSpectrum ();
     public void setSpectrum (int index, java.awt.Color color);
     public void setSpectrum (java.awt.Color[] colors);
     ```

Chapter 6

Reflection

9 - Ottobre

6.1 Introduction and Definitions

Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. Another dimension of reflection is if a program is allowed to **read only**, or also to **change** itself.

- **Introspection** is the ability of a program to observe and therefore reason about its own state
- **Intercession** is the ability for a program to modify its own execution state or alter its own interpretation or meaning
- **Reification** is the mechanism of encoding execution state into data, which is needed by both *introspection* and *intercession*

.

Structural reflection is concerned with the ability of the **language** to provide a complete *reification* of both the *program* executed and its *abstract data types*.
Behavioral reflection is concerned instead with the reification of its[1] *semantics* & *implementation* (processor) and the data and implementation of the *run-time system*.

6.2 Uses and drawbacks

6.2.1 Uses

- *Class Browsers* need to be able to enumerate the number of classes
- *Visual Development Environments* can exploit type info available in reflection to aid the developer in writing correct code
- *Debuggers* need to be able to examine private members on classes
- *Test Tools* exploit reflection to ensure a high level of code coverage in a test suite
- *Extensibility Features* an app may make use of external, user-defined classes by creating instances of extensibility objects.

6.2.2 Drawbacks

- **Performance Overhead**
- **Security Restrictions**
- **Exposure of internals**

6.3 Reflection in Java

Java supports **introspection** and **reflexive invocation**, but not *code modification*.

[1] referred to a **language**

6.3.1 Introspection

The JVM mantains for every type an associated object of type `java.lang.Class` which "*reflects*" the type it represents, acting as entry point for reflection, since it provides all info needed:

- Class name and modifiers
- Extended superclasses and implemented inferfaces
- Methods, fields, constructors, etc.

To retrieve such `java.lang.Class` object it is sufficient to do `Object.getClass()`. `Class` objects are constructed automatically by the JVM as classes are loaded.

Using `java.util.reflect.*` it is possible also to retrieve class **Members** i.e. *fields, constructors* and *methods*. The extensive `java.util.reflect.*` API provides many *methods* to achieve this which will not be reported here.
There is a class for each Member

- `java.util.reflect.Field`: access type info and set/get values.
- `java.util.reflect.Method`: type info for parameters and return type; invoking method on a given object.
- `java.util.reflect.Constructor`: note that constructors have no return values and invocation creates a new instance of the given class.

6.3.2 Program Manipulation

By now we have talked only about **introspection** in java, but reflection can be used also to create objects of a type not known at compile time, or to access members (access fields or invoke methods) unknown at compile time.

Chapter 7

Annotations

9 - Ottobre

In java, `static`,`private`,... modifiers are *meta-data* describing properties of program elements. **Annotations** can be understood as (user-) definable modifiers. They are composed by one or two parts:

1. **name**
2. finite number of **attributes** i.e. `name=value`. There may be no attributes.

The syntax is the following:

```
    @annName             // e.g. Override
    @annName{constExp}   // shorthand for @annName{value=constExp}
    @annName{name_1 = constExp_1, ..., name_k = constExp_k}
```

`constExp` are expression which can be evaluated at *compile time*. Besides, attributes have a *type*, thus the supplied values have to convertible to that type.
Annotations can be applied to almost any syntactic element, from packages to parameters and any type use.

7.1 Defining annotations

```
@interface InfoCode {
    String author ();
    String date ();
    int ver () default 1;
    int rev () default 0;
    String [] changes () default {};
}
```

This defines the custom annotation InfoCode, imposing some fields possibly with default values. It can be used as follows:

```
@InfoCode(author="Beppe", date="10/12/07")
    public class C {
    public static void m1() { /* ... */ }
    @InfoCode(author="Gianni",
        date="4/8/08", ver=1, rev=2)
    public static void m2() { /* ... */ }
}
```

Chapter 8

Polymorphism

Polymorphism basically means *"many forms"*, where *forms* are **types**. Thus there may be *polymorphic* function names, or *polymorphic* types.

There are many "flavors" of polymorphism, many variations. Two main kinds opposed to each other are *ad hoc* and *universal* polymorphism, which however, may coexist:

- **ad hoc** PM indicates that a single function name denotes different algorithms, determined by the actual types.
- **universal** PM indicates a single algorithm (solution) applicable to objects of different types.

When PM is taken into account, it is crucial to consider when happens the **binding** between a function name and the actual code to be executed:

- compile time; *static/early binding*
- linking time
- execution time; *late/dynamic binding*

In general the earlier the binding happens, the better (for debugging reasons). If the binding spans over more phases (e.g. *overriding* in Java), as a convention we consider the **binding time** the last phase.

8.1 Classification

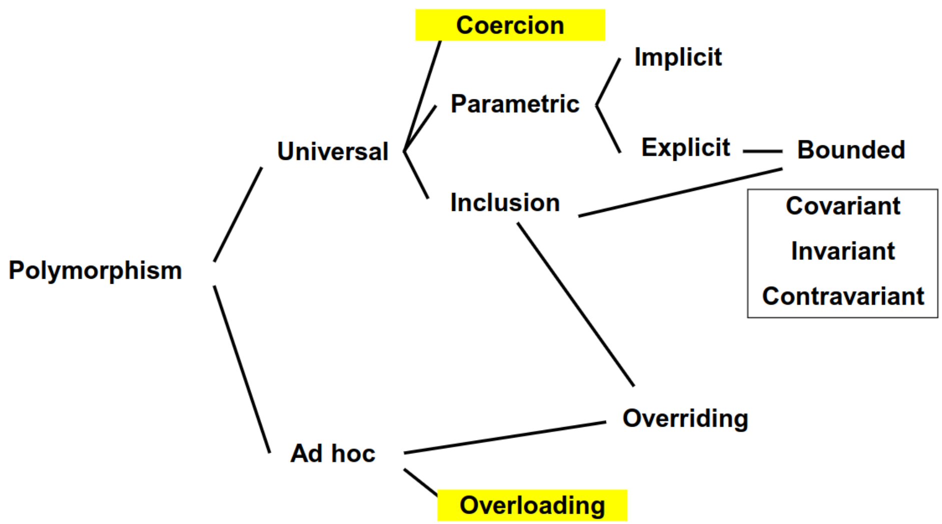


Figure 8.1: Polymorphism classification

8.1.1 Overloading

Overloading is present in every language for basic operators $+ - *$..., and sometimes is supported for user-defined functions, and in some languages it is even allowed the overloading of primitive operator by user-defined functions.

Since this falls under the **ad hoc** polymorphism family, the code to be executd is determined by the type of the arguments; the binding can either happen at *compile* or at *runtime*, depending on the typing of the language, whether it is static or dynamic.

```
// C language doesn't allow overloading for user-defined functions
int sqrInt(int x) { return x * x; }
double sqrDouble(double x) { return x * x; }

// Overloading in Java & C++
int sqr(int x) { return x * x; }
double sqr(double x) { return x * x; }
```

Haskell introduces **type classes** for handling overloading in presence of type inference

8.2 Coercion

Coercion is the automatic (implicit) conversion of an object to a different type, opposed to casting which is explicit instead. Coercion allows a code snippet to be applied of arguments of different (convertible) types. Sometimes coercion is allowed only if there is no **information loss**.

```
double sqrt(double x){...}
double d = sqrt(5) // applied to int
```

´

8.3 Inclusion Polymorphism

Inclusion polymorphism is also known as *subtyping polymorphism* or ***inheritance***. It is ensured by *Barbara Liskov*'s **substitution principle**:

A subtype object can be used in any context where a supertype object is expected

Methods and fields defined in a superclass may be invoked and accessed by subclasses if not redefined (see *Overriding*).

8.4 Overriding

In Java a method m of a class A can be redefined in a subclass B of A.

Overriding introduces ad hoc polymorphism in the universal polymorphism of inheritance. Notice that overriding requires the final binding to happen at runtime: it happens through the lookup done by `invokevirtual` in the JVM.

8.5 C++ v Java

```
class A {
 public:
  virtual void onFoo() {}
  virtual void onFoo(int i) {}
};
class B : public A {
 public:
  virtual void onFoo(int i) {}
};
class C : public B {};
int main() {
  C* c = new C();
  c->onFoo();
  // Compile error - doesn't exist
}
```

The equivalent code in Java compiles, because in java invokes the function `onFoo()` with no arguments defined in the superclass A. In C++ instead, the function onFoo(int i) defined in B is found and stops the search, but there is arguments type mismatch, thus it doesn't compile. This happens because in C++ the method lookup is based on the method *name*, not on its *signature*.

8.6 C++ Templates

They are similar to *Generics* in Java, they are used as function and class templates each concrete instantiation produces a copy of the generic code, specialized for that type: monomorphization. In java Generics, instead, **type erasure** happens at runtime, i.e. type variables `T` are replaced by `Object` variables.
Templates support parametric polymorphism and type parameters can also be primitive types (unlike Java generics)

```
    template <class T> // or <typename T>
    T sqr(T x) { return x * x; }
```

Assuming to invoke `sqr(T x)` on variables of different types, the compiler will generated a specific code for each type used. This works even on user-defined types; check the following code for an example:

```
class Complex {
 public:
  double real;
  double imag;
  Complex(double r, double im) : real(r), imag(im){};
  Complex operator*(Complex y) {  // overloading of *
    return Complex(real * y.real - imag * y.imag,
                   real * y.imag + imag * y.real);
  }
};

Complex cc = sqr(c); // legal and produces a function "Complex sqr(Complex x) {...}"
```

It is important to check for type ambiguosity; in the following example, it is highlighted a case where it's not clear whether it is `i` to be converted to `long` or `m` to `i`.

```
template <class T>
T GetMax(T a, T b) {  return (a > b) ? a : b; }
...
n = GetMax(l, m);        // ok: GetMax<long>
// v = GetMax(i, m);     // no: ambiguous
v = GetMax<int>(i, m);   // ok
```

8.6.1 Macros

Macros can be exploited to achieve *polymorphism* and can have the same effect of the templates, but notice that macros are executed by the preprocessor[1] and are only **textual substition**, there is no parsing, no static analysis checks or whatsoever.

```
#define sqr(x) ((x) * (x))
int a = 2;
int aa = sqr(a++); // int aa = ((a++) * (a++));
// value of aa? aa contains 6  :(

#define fact(n) (n == 0) ? 1 : fact(n-1) * n
// compilation fails because fact is not defined
```

16 - Ottobre

8.6.2 Specialization

A template can be **specialized** by defining a template with the same name but with more specific parameters (*partial specialization*) or with no parameters (*full specialization*). This is kinda similar to *Overriding*, leaving to the compiler the choice of the most appropriate template.

```
/* Primary template */
template <typename T> class Set {
        // Use a binary tree
    };
/* Full specialization */
template <> class Set<char> {
        // Use a bit vector
```

[1]Macro expansion can be seen using the option *-E* when compiling

```
    };
/* Partial specialization */
template <typename T> class Set<T*> {
        // Use a hash table
    };
```

Templates can be used by a compiler to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled.

Template compilation happens *on demand*: the code of a template is not compiled until an instantiation is required, however in case of *fully-specialized* template, the compiler treats the template as a function, thus it generates its code **regardless** whether it is ever used or not.

Note that in C/C++ while method *prototypes* usually are in a separate `.h` file, the compiler needs the template *declaration* and *definition* in the same place to instatiate it.

Chapter 9

Generics

Generics are instance of *Universal Polymorphism* with explicit parameters (see Fig 8.1).

9.1 Methods

```
public static <T> T getFirst(List<T> list)
```

Invocations of generic methods must instantiate all type parameters, either explicitly or implicitly. Some sort of *type inference* is applied in case of implicit instantiation.

```
class NumList<E extends Number> {
      void m(E arg) {
            arg.intValue();   // OK, since...
            // Number and its subtypes support intValue()
         }
   }
```

Type parameters can also be **bounded** as in the above example, allowing methods (and fields) defined in the **bound** to be invoked on objects of the type parameter `T`.
There may be various kinds of type bounds:

```
<TypeVar extends SuperType>
   // UPPER bound; SuperType and any of its subtype are ok.
<TypeVar extends ClassA & InterfaceB & InterfaceC & ...>
   // MULTIPLE UPPER bounds
<TypeVar super SubType>
   // LOWER bound; SubType and any of its supertype are ok
```

Unlike C++ where *overloading* is resolved and can **fail** after instantiating a template, in Java **type checking** ensures that overloading will succeed.

9.2 Inheritance and Arrays

There are two major issues which came up along with generics. The first one regards **inheritance**; consider the following example:

Since `Integer` is a *subtype* of `Number`,
is `List<Integer>` *subtype* of `List<Number>`?

NO!

In a formal way, *subtyping is invariant* for Generic classes. Informally, given `A,B` concrete types, `MyClass<A>` has no relationship to `MyClass<B>`, even if `A,B` have one.
On the other hand if `A extends B` and are *generic* classes, then `A<C> extends B<C>` for any type `C`. For example, `ArrayList<Integer> extends List<Integer>`.

Note that the common parent of `MyClass<B>` and `MyClass<A>` is `MyClass<?>`.

Let's now discuss **covariance** and **contravariance**, with the aid of a few examples.

```
List<Integer> lisInt = new ...;
List<Number> lisNum = new ...;
lisNum = lisInt; // ??? - Reassign pointer
lisNum.add(new Number(...)); // NOT ALLOWED
listInt = lisNum; // ??? - Reassign pointer
Integer n = lisInt.get(0); // NOT ALLOWED
```

`List<Integer>` is neither a subtype or a supertype of `List<Number>`, thus the above operations aren't allowed. However there are *read-only* and *write-only* situations where they may be allowed.

```
RO_List<Integer> lisInt = new ...;
RO_List<Number> lisNum = new ...;
lisNum = lisInt; // ???
Number n = lisNum.get(0); // OK
```

It is ok to *read* a **supertype** starting from a **subtype**.

covariance *is safe if the type is* ***read-only***

```
WO_List<Integer> lisInt = new ...;
WO_List<Number> lisNum = new ...;
lisInt = lisNum; // ???
lisInt.add(new Integer(...)); // OK
```

It is ok to *write* a **subtype** in the place of from a **supertype**.

contravariance *is safe if the type is* ***write-only***

17 - Ottobre

Other languages

In the case of **C#**, generic classes can be marked with the keyword `out` (*covariant*) or `in` (*contravariant*), otherwise the class is invariant. In **Scala** the same happens, but with the `+` or `-` operators.

Let's now discuss **arrays**.
Let `A extends B`, then `A[] extends B[]` even if instead `Array<A>` is not related to `Array<B>`.

Thus, *arrays in Java are* ***covariant****.*

However there is a counterpart, since this allows rule-breaking assignments which are allowed by the compiler but which lead to a runtime `ArrayStoreException`. This happens because the dynamic type of an array is checked at runtime. Knowing this, for each array update, a runtime check is performed by the JVM which throws the exception if needed.

```
Apple[] apples = new Apple[1];
Fruit[] fruits = apples;       // Ok, covariance
fruits[0] = new Strawberry(); // Compiles!
// Throws ArrayStoreException at runtime
```

After compilation Generic are all **type-erasured** to `Object` or to their first *bound*, if present. This choice has been made mainly for compatibility with legacy code, leading all instances of the same generic type to have the same type at runtime; i.e.

```
List<String> lst1 = new ArrayList<String>();
List<Integer> lst2 = new ArrayList<Integer>();
assert(lst1.getClass() == lst2.getClass())
```

9.2.1 Generic Arrays

What about *arrays of generics*? Such arrays in Java are **not allowed**, because every array update needs a runtime check which is impossible to perform on generics, since at runtime generics are all of the same type due to *type-erasure*.

9.3 Wildcards

Wildcards are strongly related to the topic of *covariance* and *contravariance*.
As briefly mentioned before, wildcards are the only relationship between generic classes.

To use *wildcards*, the **PECS** principle is applied: ***P**roducer **E**xtends, **C**onsumer **S**uper*.

- `? extends T` to **get** values from a *Producer*: **covariance** allowed
- `? super T` to **insert** values into a *Consumer*: **contravariance** allowed
- Never use `?` when both insertion and retrieving is needed, `T` is sufficient and way more appropriate.

Wildcards improve type-safety, allowing a program to fail at *compile-time* instead of *runtime*.

```
List<Apple> apples = new ArrayList<Apple>();
List<? extends Fruit> fruits = apples;
fruits.add(new Strawberry()); // COMPILING FAILS
```

9.4 Generics Limitations

- Cannot instantiate Generics with primitive types:

```
    ArrayList<int> a = ...                          // compile error
```

- Cannot create instances of type parameters
- Cannot declare static fields whose types are type parameters

```
    public class C<T>{ public static T local; ...}
```

  Because static fields are represented in the **unique** representation of the class in the dedicated static memory area of the JVM for classes
- Cannot use casts or instanceof with parameterized types

```
    mylist instanceof ArrayList<Integer>            // fails
    mylist instanceof ArrayList<?>                  // OK
```

- Cannot Create arrays of parameterized types
- Cannot create, catch, or throw objects of parameterized types
- Cannot overload a method where the formal parameter types of each overload erase to the same raw type.

```
    public class Example {                          // does not compile
       public void print(Set<String> strSet) { }
       public void print(Set<Integer> intSet) { } }
```

Chapter 10

Standard Templates Library

17 - Ottobre

The goal of STL is to represent algorithms in as general form as possible without compromising efficiency. There is an extensive use of **templates**, **overloading** and **iterators**, which are used for decoupling algorithms from containers, and can be seen as an abstraction of pointers.
STL is very different from the *Java Collection Library* since it does **not** use *dynamic binding* and is **not** *object oriented*; instead the STL uses only *static binding* and *inlining*.

10.1 Main Entities

- **Container** collection of *typed* objects
- **Iterator** Generalization of pointer or address; used to step through the elements of collections
- **Algorithm** initializing, sorting, searching, and transforming contents of containers
- **Adaptor** Convert from one form to another e.g. iterator from updatable container; or stack from list
- **Function Object** Form of closure (class with "operator()" defined)
- **Allocator** encapsulation of a memory pool

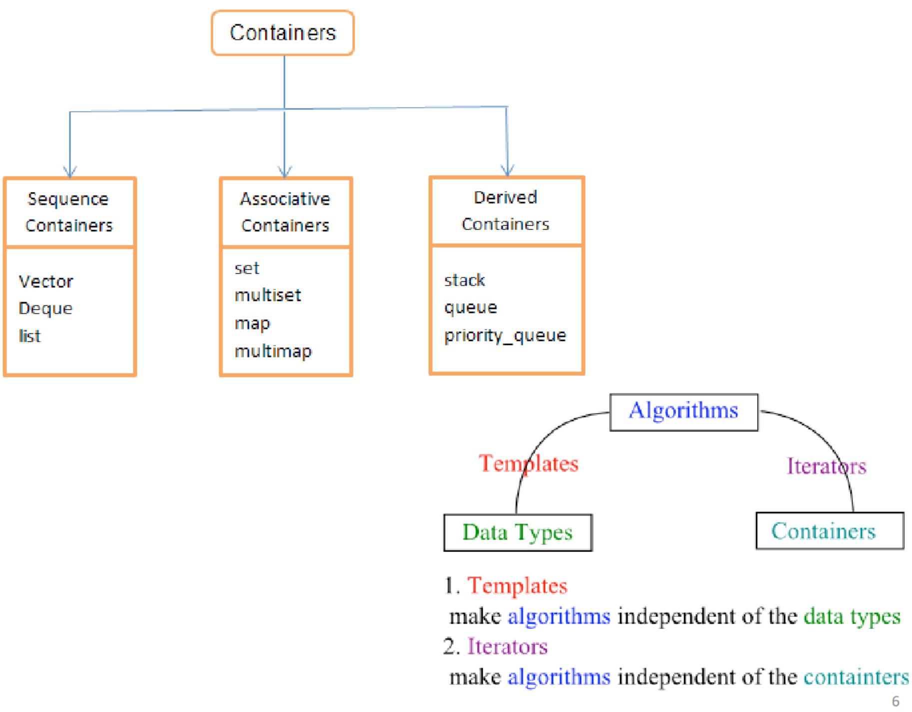


Figure 10.1: TIKZ RE-DO : STL Containers

10.2 Iterators

Since algorithms cannot be used *directly* on different kinds of collections, Iterators come in handy by providing a **uniform**, **linear** access to elements of different collections.

In **Java** iterators are supported by the *JCF*[1] through the interface `Interface<T>`. They are related to an **instance** of a class and are usually defined as *nested classes*, more precisely *non-static private member classes*.
Collections equipped with iterators must `implements` `Iterable<T>` interface.

18 - Ottobre

In **C++** there is no `next/hasNext()` function, standard `++` `--` operators are used instead.
In case of *arrays* pointers can be trivially used, since `int v[]` is no different `int *v`[2]. In the case of `vector` instead, an actual *iterator* may be instantiated, but the operator `++` stays the same.

```
vector<int> vec;
vector<int>::iterator v = vec.begin();
while( v != vec.end()) {
   cout << "value of v = " << *v << endl;
   v++;
}
```

Every class in C++ has its own iterator; more specifically, containers define and expose a type named iterator in the container's **namespace**, allowing the semantic value of `iterator` to change according to the context.

10.2.1 C++ iterators implementation

Typically iterators are implemented as `struct` and provide a visit of the container, retaining information about the **state** of the visit, e.g. pointer to next element, remaining elements, and so on. Note that in case of *trees* or *graphs* the visit's state may not be trivial to be represented.

```
template <class T>
struct v_iterator {
  T* v;
  int sz;
  v_iterator(T* v, int sz) : v(v), sz(sz) {}
  // != implicitly defined
  bool operator==(v_iterator& p) { return v == p->v; }
  T operator*() { return *v; }
  v_iterator& operator++() {  // Pre-increment
    if (sz)
      ++v, --sz;
    else
      v = NULL;
    return *this;
  }
  v_iterator operator++(int) {  // Post-increment!
    v_iterator ret = *this;
    ++(*this);  // call pre-increment
    return ret;
  }
};
```

Container	*insert/erase* **beginning**	**middle**	**end**
vector	linear	linear	amortized constant
list	constant	constant	constant
deque	amortized constant	linear	amortized constant

Table 10.1: **Guaranteed** time complexity for iterators

To achieve transparency to third-party algorithms STL assumes *constant* time for every operation, and allows 5 types of operators:

[1] *Java Collection Framework*
[2] At least from an "accessing values" point of view, there are some differences in terms of static/dynamic allocation of memory.

- *Formard iterators* only dereference and pre/post increment
- *Input (and Output) iterators* same as *formard iterators* but with possible issues when dereferencing
- *Bidirectional iterators* dereference, pre/post increment and decrement
- *Random access iterators* same as Bidirectional but allow also integer sum $(p + n)$ and difference $(p - q)$

Each category defines only the functions which take constant time. Not all iterators are defined for all containers, e.g. since *random access* takes *linear* time on lists, there is no *random access* iterator on *lists.*

10.2.2 Invalidation

When a container is *modified*, iterators *may* become **invalid**: no "exception" is thrown, iterators can still be used, but their behaviour is **undefined**. ***Not*** *every operation* invalidates iterators, it depends on the operation and on the container.

The main limiting aspect of STL's iterators is that they provide a **linear view** of the container, allowing the definition of operations only on one-dimensional containers; thus, if it is needed to access the organization of the container (e.g. tree custom visist), the only way-to-go is to define a custom iterator which behaves as desired.

10.3 C++ specific features

10.3.1 Inheritance

STL relies on typedefs combined with namespaces to implement genericity, the programmer always refers to `container::iterator` to know the type of the iterator. Note that there is no relation among iterators for different containers (!), if not a semantically abstract one. The reason for this is **performance**: without *inheritance*, types are resolved at compile time and the compiler may produce better and optimized code. On the other hand sacrificing inheritance may lead to lower expressivity and lack of type-checking; in fact, STL relies only on coding conventions: when the programmer uses a wrong iterator the compiler complains of a bug in the library.

10.3.2 Inlining

C++ standard has the notion of **inlining** which is a form of semantic macros. Inline methods should be available in header files and can be labelled *inline* or defined within class definition, invocation on such methods is type-checked and then it is replaced by the method body. The compiler tends to (automatically?) inline methods with small bodies and without iteration; it is able to determine types at compile time and usually does inlining of function objects.

10.3.3 Memory management

STL abstract from the specific memory model using a concept named **allocators**. All the information about the memory model is encapsulated in the **Allocator** class. Each container is parametrized by such an allocator to let the implementation be unchanged when switching memory models.

10.3.4 Potential Problems

The problem may be error checking: almost all facilities of the compiler fail with STL resulting in lengthy error messages that ends with error within the library

Chapter 11

Functional Programming

Functional Programming languages radicate their roots in the Church's model of computing known as ***lambda calculus***. Such model is based on the notion $\Lambda -$ *parametrized expressions*, with the focus on defining mathemtical functions in a constructive and effective way. The computation proceeds by substituting parameters into expressions.

Functional programming languages such as *Lisp, Scheme, FP, ML, Miranda* and *Haskell* aim to implement Church's lambda calculus in the form of a programming language which does everything needed by **composing functions**, thus no *mutable* ***state*** and no *side effects*.

FPL[1] needs some key features which are often absent in *imperative* languages:

- 1^{st}-class order and **high-order** functions: Functions can be *denoted*, passed as *arguments* to other functions, *returned* as result of function invocation
- **Recursion** opposed to *"control variables"*
- **Powerful list facilities**: Recursive functions exploit recursive definition of lists
- **Polymorphism** typically universal parametric implicit, which plays a key role when handling containers/collections.
- **Fully general aggregates**: there is a wide use of tuples and records, besides, data structures cannot be modified (*no state!*), they have to be re-created.
- **Structured function returns** allow to pass more meaningful information to the caller, avoiding the need for "side-effects".
- **Gargabe collection**

23 - Ottobre

11.1 FP language families

1. **LISP**: currently most used for *AI* after Python. Original LISP is no longer used, the current standard is *Common LISP* which introduced statical scope opposed to the dynamic one of *Original LISP* ; another version is called *Scheme*
2. **ML**: Common languages of this family are *Standard ML, Caml,OCaml, F#*. These are compiled languages, but intended for interactive use. ML results from the combination of Lisp and Algol-like features, including Garbage collection, Abstract data types, Module system and Exceptions
3. **Haskell**: Many features are shared with *ML* languages, but with some differences.
   - Type inference, Implicit parametric polymorphism, Ad hoc polymorphism (**overloading**) with type classes
   - **Lazy** evaluation, Tail recursion and continuations
   - **Purely functional** $\rightarrow$ precise management of side effects

[1] Short for ***F**unctional **P**rogramming **L**anguages*

11.2 Haskell basics

Basic types

- ◇ `Unit`
- ◇ `Booleans`
- ◇ `Integers`
- ◇ `Strings`
- ◇ `Reals`
- ◇ `Tuples`
- ◇ `Lists`
- ◇ `Records`

Note that basic types are written with the first letter Uppercased.

Other types

- ◦ Patterns
- ◦ Declarations
- ◦ Functions
- ◦ Polymorphism
- ◦ Type declarations
- ◦ Type Classes
- ◦ Monads
- ◦ Exceptions

Haskell provides an interactive read-eval-print interpreter (`ghci`): many examples are available in the lecture's slides, here we will discuss only some more interesting ones.

Variables (**names**) are bound to expressions, *without* evaluating them (because of *lazy evaluation*); the scope of the binding is the rest of the session.

```
ghci> let a = 3    -- 'let' can be omitted
ghci> b = a + 2
ghci> b
5
ghci> a = a + 1    -- okay, until here
ghci> a            -- infinite recursion
-- CTRL+C Manual Interrupt
ghci> x = 1:x
ghci> x            -- infinite ',1' print
```

Moving onto **anonymous functions** i.e. `\x -> ...` lambda notation

```
ghci> (\x -> x+1)5       -- apply 5 to anon function
6
ghci> f = (\x -> x+1)
ghci> f 5                -- brackets () can be omitted
6
ghci> h = \(x,y) -> x+y -- tuple Pattern instead of single variable
ghci> h (3,4)            -- brackets are needed here
7
ghci> h 3 4              -- brackets are needed here
-- ERROR
ghci> :t f
f :: Num a => a -> a
ghci> :t h
h :: Num a => (a, a) -> a
```

To declare explicit functions instead, the syntax is quite simple

```
f (x,y) = x+y --argument must match pattern (x,y)
```

```
   reverse xs =       -- linear, tail recursive
      let rev ( [], accum ) = accum
         rev ( y:ys, accum ) = rev ( ys, y:accum )
      in rev ( xs, [] )
```

24 - Ottobre

11.3 More on Haskell features

Let's recall that Haskell is a **lazy** language, thus functions and data constructor don't evaluate arguments until they actually need them.

```
   myData = [1,2,3,4,5,6,7]
   twiceData = [2 * x | x <- myData]
   -- [2,4,6,8,10,12,14]
```

```
    twiceEvenData = [2 * x| x <- myData, x `mod` 2 == 0]
    -- [4,8,12]
```

```
ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
      [10,11,12,14,16,17,18,20] -- more predicates
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
      [16,20,22,40,50,55,80,100,110] -- more lists
length xs = sum [1 | _ <- xs] -- anonymous (dont care) var
      -- strings are lists...
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

Datatype declearations

```
    data Color = Red | Yellow | Blue
    data Atom = Atom String | Number
    data List = Nil | Cons (Atom, List)

    -- General form:
    data <name> = <clause> | ... | <clause>
    <clause> ::= <constructor> | <contructor> <type>
```

```
    -- also possible to define Recursive data types
    data Tree = Leaf Int | Node (Int, Tree, Tree)

    Node(4, Node(3, Leaf 1, Leaf 2), Node(5, Leaf 6, Leaf 7))

    -- it is possible to use constructors in pattern matching
    sum (Leaf n) = n
    sum (Node(n,t1,t2)) = n + sum(t1) + sum(t2)
```

Besides it is possible to match different cases with a specific `case` statement; note that **Indendation** in case statement *MATTERS*

```
    data Exp = Var Int | Const Int | Plus (Exp, Exp)

    case e of
       Var n -> ...
       Const n -> ...
       Plus(e1,e2) -> ...

    -- Indendation in case statement MATTERS
```

11.3.1 Function Types

`f :: A -> B` means that:

$$\forall x \in A f(x) = \begin{cases} \exists y = f(x) \in B \\ \textit{run forever} \end{cases}$$

In other words, if $f(x)$ terminates, then $f(x) \in B$. In ML, functions with type A $\rightarrow$ B can throw an exception or have other effects, but **not** in Haskell.

11.3.2 Loops and Recursion

In FP `for` and `while` iterative loops are replaced by **recusive** subroutines calling themselves directly or indirectly (*mutual recursion*).

```
length' [] = 0
length' (x:s) = 1 + length'(s)
   -- definition using guards and pattern matching
   -- take' n lst returns first n elements of a list
```

```
take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
| n <= 0 = []
take' _ [] = []
take' n (x:xs) = x : take' (n-1) xs
```

11.3.3 Higher-Order functions

Functions that take other functions as arguments or return a function as a result are **higher-order** functions.

```
applyTwice :: (a -> a) -> a -> a    -- function as arg and res
applyTwice f x = f (f x)

> applyTwice (+3) 10 => 16
> applyTwice (++ " HAHA") "HEY" => "HEY HAHA HAHA"
> applyTwice (3:) [1] => [3,3,1]

applyTwice' f = f.f                 -- equivalent definition
:t (.)
> (.) :: (b -> c) -> (a -> b) -> a -> c
```

```
   -- define the operator |> which inverts the order between function and argument
   (|>) a f = f a
   (|>) :: t1 -> (t1 -> t2) -> t2
   -- Seems dull right?
   -- Look at the following example

   -- Here, the order of invocation is the same,
   -- but the second "infix" form is (might be) more readable
   > length ( tail ( reverse [1,2,3]))
      2
   > [1,2,3] |> reverse |> tail |> length
      2
```

```
(+) :: Num a => a -> a -> a
> let f = (+) 5 // partial application
>:t f ==> f :: Num a => a -> a
> f 4 ==> 9
elem :: (Eq a, Foldable t) => a -> t a -> Bool
> let isUpper = (`elem` ['A'..'Z'])
>:t isUpper ==> isUpper :: Char -> Bool
> isUpper 'A' ==> True
> isUpper '0' ==> False
```

Combinators

`map` *combinator* applies argument function to each element in a collection.

```
   map :: (a -> b) -> [a] -> [b]
   map _ [] = []
   map f (x:xs) = f x : map f xs
```

`filter` takes a collection and a boolean predicate, and returns the collection of the elements satisfying the predicate. It is defined as follows:

```
   filter :: (a -> Bool) -> [a] -> [a]
   filter _ [] = []
   filter p (x:xs)
      | p x = x : filter p xs
      | otherwise = filter p xs
```

And can be applied in the following way

```
> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
> filter (==3) [1,2,3,4,5]
[3]
> filter even [1..10]
[2,4,6,8,10]
> let notNull x = not (null x)
in filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]
[[1,2,3],[3,4,5],[2,2]]
```

reduce (`foldl`,`foldr`): takes a collection, an initial value, and a function, and combines the elements in the collection according to the function.

```
-- folds values from end to beginning of list
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
-- folds values from beginning to end of list
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
-- variants for non-empty lists
foldr1 :: Foldable t => (a -> a -> a) -> t a -> a
foldl1 :: Foldable t => (a -> a -> a) -> t a -> a
```

Let's provide some examples:

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs

maximum' :: (Ord a) => [a] -> a
maximum' = foldr1 (\x acc -> if x > acc then x else acc)

reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []

product' :: (Num a) => [a] -> a
product' = foldr1 (*)
product' = foldr (*) 1
-- Notice that product' [] returns 1 !

filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc)[]

head' :: [a] -> a
head' = foldr1 (\x _ -> x)
last' :: [a] -> a
last' = foldl1 (\_ x -> x)
```

11.3.4 Recursion and Optimization

From a theoretical point of view recursion and iteration are equivalently expressive, and typically one is preferred over the other depending on the problem being faced to make the code more intuitive. In general a procedure call is *much more expensive* than a conditinoal branch, however FP compilers can perform many optimizations and produce better code, especially for known blocks; for this reason the use of combinators such as `map`,`reduce`,`filter`, `foreach`,... is strongly encouraged.

Tail-recursive functions are functions in which no operations follow the recursive call(s) in the function, thus the function returns immediately after the recursive call, allowing the compiler to reuse the subroutine's frame on the run-time stack, since the current subroutine state is no longer needed. Besides many compilers instead of re-invoking the function, simply jump to the beginning of the function.

Let's provide the classic example of Fibonacci to illustrate how to convert a normal recursive function to its tail-recursive correspondant one:

```haskell
-- typical Fibonacci
fib = \n -> if n == 0 then 1
   else if n == 1 then 1
      else fib (n - 1) + fib (n - 2)

fibTR = \n -> let fibhelper (f1, f2, i) =
   if (n == i) then f2
      else fibhelper (f2, f1 + f2, i + 1)
   in fibhelper(0,1,0)
```

Notice that fibTR takes only $\mathcal{O}(n)$ since it builds the Fibonacci sequence starting from 1 to n, while the more canonical approach calculates multiple times the same values, and starts from n until 1 is reached.

`foldl` is tail-recursive, `foldr` is not. But because of laziness Haskell has **no** tail-recursion optimization. Despite this, it provides a more efficient variant of foldl called `foldl'` where f is evaluated **strictly**.
Strictly means that the compiler evaluates the accumulator at *each step* of the folding process, ensuring that intermediate values are not built up as *unevaluated thunks* i.e. Haskell's term for delayed computations. Due to its *strictness* `fold'` is less likely to cause **space leaks** (see note below), and it generally has better performance for many common folding operations.

"Space leaks" mahy happen when a program retains references to data that should no longer be needed, preventing the garbage collector from reclaiming memory. This can lead to inefficient memory usage and, in some cases, cause the program to run out of memory. This (generally) occurs only when handling large data sources or *infinite data structures* (which are allowed in Haskell); space leaks may be very hard to debug, tools like memory profilers and heap profiling can be helpful in identifying them

Chapter 12

λ Lambda calculus

Due to Haskell's **laziness**, functions and data constructors don't evaluate their arguments until they need them. In several languages there are forms of lazy evaluations (`if-then-else`, shortcutting `&&` and `||`).

12.1 Syntax

```
t ::= x | λ x.t | t t | (t)
```

- `x` variable, name, symbol,...
- λ `x.t` abstraction, defines an anonymous function
- `t t'` application of function `t` to argument `t'`

We say that an occurrence of x is **free** in a term t if it is not in the body of an abstraction $\lambda x.t$, otherwise it is **bound**; λx instead is a **binder**. Examples $\lambda z.\lambda x.\lambda y.x(yz)$
$(\lambda x.x)x$

Terms without free variables are **combinators**. Identity function: $id = \lambda x.x$
First projection: $fst = \lambda x.\lambda y.x$.

β-Reduction

β-reduction, i.e. *function application*, also called **redex**:

$$(\lambda x.t)t' = t[t'/x]$$

Examples

$$(\lambda x.x)y \longrightarrow y \tag{12.1}$$
$$(\lambda x.x(\lambda x.x))(ur) \longrightarrow ur(\lambda x.x) \tag{12.2}$$
$$(\lambda x.(\lambda w.xw))(yz) \longrightarrow \lambda w.yzw \tag{12.3}$$
$$(\lambda x.xx)(\lambda x.xx) \longrightarrow (\lambda x.xx)(\lambda x.xx) \tag{12.4}$$

12.2 Functions and lambdas

A definition of a function with a single argument associates a name with a λ-abstraction, while a function with several arguments is equivalent to a sequence of λ-abstractions

```
f x = <exp> -- is equivalent to
f = λ x.<exp>

f(x,y) = <exp> -- is equivalent to
f = λ x. λ y.<exp>

-- Curriend and uncurried functions
curry :: ((a, b) → c) → a → b → c
curry f x y = f(x,y)
```

```haskell
uncurry :: (a → b → c) → (a, b) → c
uncurry f (x,y) = f x y
```

12.3 Well-known functions

- T = λt.λf.t -- first
- F = λt.λf.f -- second
- and = λb.λc.bcF
- or = λb.λc.bTc
- not = λx.xFT
- test =λl.λm.λn.lmn

```
and T F
→ (λb.λc.bcF) T F
→ (λc.TcF) F
→ TFF
→ F
```

```
not F
→ (λx.xFT) F
→ FFT
→ T
```

```
test F u w
→ (λl.λm.λn.lmn) F u w
→ (λm.λn.Fmn) u w
→ (λn.Fun) w
→ Fuw
→ w
```

Figure 12.1: Church Booleans using λ-calculus

$$
\begin{aligned}
pair &= \lambda f.\lambda s.\lambda b.b\ f\ s \\
fst &= \lambda p.p\ T \\
snd &= \lambda p.p\ F
\end{aligned}
$$

Pair function

$$
\begin{aligned}
&fst(pairuw) \\
&\quad\longrightarrow\quad (\lambda p.p\ T)(pair\ u\ w) \\
&\quad\longrightarrow\quad (pair\ u\ w)\ T \\
&\quad\longrightarrow\quad (\lambda f.\lambda s.\lambda b.b\ f\ s)\ u\ wT \\
&\quad\longrightarrow\quad (\lambda s.\lambda b.b\ u\ s)\ w\ T \\
&\quad\longrightarrow\quad (\lambda b.b\ u\ w)\ T \\
&\quad\longrightarrow\quad T\ u\ w \\
&\quad\longrightarrow\quad u
\end{aligned}
$$

Church ***Numerals***

$$0 = \lambda s.\lambda z.z \tag{12.5}$$
$$1 = \lambda s.\lambda z.s\ z \tag{12.6}$$
$$2 = \lambda s.\lambda z.s\ (s\ z) \tag{12.7}$$
$$3 = \lambda s.\lambda z.s\ (s(s\ z)) \tag{12.8}$$

Numerals n takes a function s as argument and returns the n-th composition of s with itself, s^n.

e.g. $succ = \lambda n.\lambda s.\lambda z.s(nsz)$

12.4 Fix-point Y combinator

The following *fix-point combinator* $\boldsymbol{Y}$, when applied to a function R, returns a **fix-point** of R, i.e. $R(YR) = YR$

$$Y = (\lambda y.(\lambda x.y(x\ x))(\lambda x.y(x\ x))) \tag{12.9}$$

$$
\begin{aligned}
YR &= (\lambda x.R(x\ x))(\lambda x.R(x\ x)) \\
&= R((\lambda x.R(x\ x))(\lambda x.R(x\ x))) \qquad (12.10) \\
&= R(YR)
\end{aligned}
$$

12.5 Evaluation ordering

Consider the two following ways of evaluating a redex, but remember that *regardless* of the evaluation order, the evaluation result is only one, and it is **unique**[1].

[1]Proved by Church and Rosser

Applicative order evaluation implies eager evaluation of arguments before applying them to the function

```
(λ x.(+ x x)) (+ 3 2)
→ (λ x.(+ x x)) 5
→ (+ 5 5)
→ 10
```

Normal order evaluation implies functions to be evaluated first, and delay argument evaluation only when needed. Note that this may lead to multiple re-evaluations of the same argument.

```
(λ x.(+ x x)) (+ 3 2)
→ (+ (+ 3 2) (+ 3 2))
→ (+ 5 (+ 3 2))
→ (+ 5 5)
→ 10
```

Haskell realizes **lazy evaluation** by using **call by need** parameter passing: an expression passed as argument is bound to the formal parameter, but it is evaluated *only if* its value is **needed**. Besides, the argument is evaluated *only* the **first time**, using the **memoization** technique: the result is saved and further uses of the argument do not need to re-evaluate it.

Combined with **lazy data constructors**, this allows to construct *potentially **infinite** data structures* and to call *infinitely recursive* functions without necessarily causing non-termination.

Note: lazy evaluation works fine with purely **functional languages**. Side effects such as IO operations force the programmer to reason about the order in which things happen, which not predictable in lazy languages. We will address this fact when introducing Haskell's IO-*Monad*.

12.6 Post-lecture Takeaway message

While discussing with the professor after the lecture, an important intuition emerged about evaluation an memoization.

```
a = 5
b = a + 3
```

b would evaluate to 8 but it is not evaluated until it is strictly necessarily.

```
a = 5
b = a + 3
a = 2
-- b?
```

Someone may think that due to lazy evaluation, b would now evaluate to 5. However, this is ***NOT*** Haskell's case. Due to **memoization**, even if `b = a + 3` doesn't get evaluated, the current value of `a` is memoized and its re-definition doens't affect b evaluation. Thus this snippet code leads `b` to be evaluated as `8`, regardless of `a` redefinition.

```
a = 5
b = a + 3
a = 2
b
> 8
```

Chapter 13

Type Inference

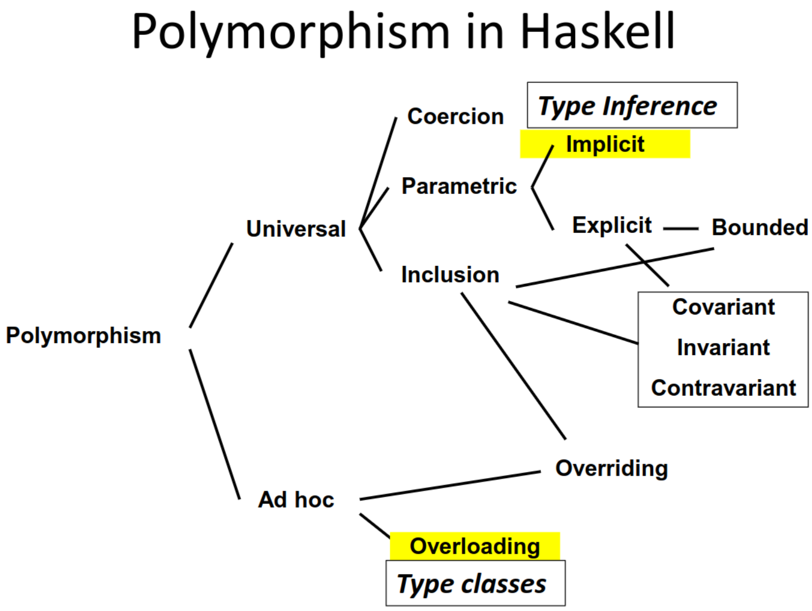


Figure 13.1: Haskell Polymorphism Recap

13.1 Overloading

Haskell allows **overloading** even of **primitive types**: the code to be executed is determined by the type of the arguments, leading to have *early binding* in *statically* typed languages or *late binding* in *dynamically* typed languages.

In Haskell we can write the following, but what is the type?

```
sqr x = x * x
```

When considering overloading besides arithmetic, we find that some functions are **fully polymorphic**:

```
length :: [w] → Int
```

While others not so much; for example, *membership* works only for types that support equality, while *sorting* works only for types which support *ordering*.

```
member :: [w] → w → Bool
sort :: [w] → [w]
```

13.2 Type Classes

Type Classes solve many overloading problems concerning arithmetic and equality (and similar properties) support.

The idea is to generalize ML's eqtypes to arbitrary types and provide concise types to describe overloaded functions, so no exponential blow-up (i.e. defining functions for every possible combination of type arguments).
Type classes allow users to define functions using overloaded operations —e.g. square, squares, and member— and to declare new collections of overloaded functions: equality and arithmetic operators are not privileged built-ins. Haskell's solutions fits perfectly within type inference framework.

The intuition is that a sorting function may allow to be passed a comparison `cmp` operator as argument, thus making the function parametric.

```
qsort:: (a → a → Bool) → [a] → [a]
qsort cmp [] = []
qsort cmp (x:xs) = qsort cmp (filter (cmp x) xs) ++ [x] ++
qsort cmp (filter (not.cmp x) xs)
```

Developing this idea, consider rewriting the parabola function to take operators as argument

```
parabola x = (x * x) + x
parabola' (plus, times) x = plus (times x x) x
```

Here the extra parameter is a ***dictionary*** that provides implementations for the overloaded ops. These implies rewriting calls to pass appropriate implementations for plus and times:

```
y = parabola'(intPlus,intTimes) 10
z = parabola'(floatPlus, floatTimes) 3.14
```

1. Type class declarations
   i. Define a set of operations, give it a name
   ii. Example: `Eq a` type class • operations `==` and `\=` with `type a →a → Bool`
2. Type class instance declarations
   i. Specify the implementations for a particular type
   ii. For `Int` instance, `==` is defined to be integer equality
3. Qualified types (or Type Constraints) Concisely express the operations required on otherwise polymorphic type `member:: Eq w =>w →[w] →Bool`

implementation summary

1. Each overloaded symbol has to be introduced in at least one type class
2. The compiler translates each function that uses an overloaded symbol into a function with an extra parameter: the dictionary.
3. References to overloaded symbols are rewritten by the compiler to lookup the symbol in the dictionary.
4. The compiler converts each type class declaration into a dictionary type declaration and a set of selector functions.
5. The compiler converts each instance declaration into a dictionary of the appropriate type.
6. The compiler rewrites calls to overloaded functions to pass a dictionary. It uses the static, qualified type of the function to select the dictionary.

13.2.1 Compositionality

```
class Eq a where
(==) :: a → a → Bool
instance Eq Int where
(==) = intEq -- intEq primitive equality
instance (Eq a, Eq b) => Eq(a,b) where
(u,v) == (x,y) = (u == x) && (v == y)
instance Eq a => Eq [a] where
(==) [] [] = True
(==) (x:xs) (y:ys) = x==y && xs == ys
(==) _ _ = False
```

13.2.2 Compound Translation

13.2.3 Subclasses

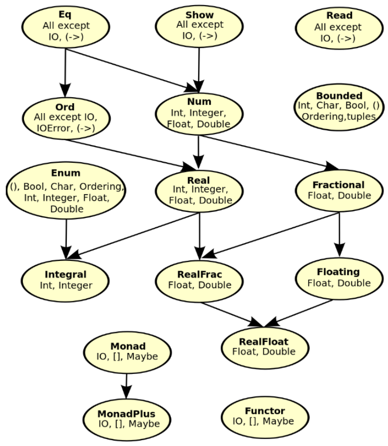


Figure 13.2: Haskell Subclasses relationships

A subclass declaration expresses this relationship:

```haskell
    class Eq a => Num a where
    (+) :: a → a → a
    (*) :: a → a → a
```

• With that declaration, we can simplify the type of the function

```haskell
memsq :: (Eq a, Num a) => a → [a] →
   Bool
memsq x xs = member (square x) xs
```

13.2.4 Deriving

For Read, Show, Bounded, Enum, Eq, and Ord, the compiler can generate instance declarations automatically.

```haskell
data Color = Red | Green | Blue
   deriving (Show, Read, Eq, Ord)

Main>:t show
show :: Show a => a → String
Main> show Red
"Red"
Main> Red < Green
True
Main>:t read
read :: Read a => String → a
Main> let c :: Color = read "Red"
Main> c
Red
```

13.2.5 Numeric Literals

```haskell
class Num a where
   (+) :: a → a → a
   (-) :: a → a → a
   fromInteger :: Integer → a
   -- Even literals are overloaded.
   -- 1 :: (Num a) => a
   ...

inc :: Num a => a → a
inc x = x + 1
```

Advantages

Numeric literals can be interpreted as values of any appropriate numeric type, for example: 1 can be an Integer or a Float or a user- defined numeric type.

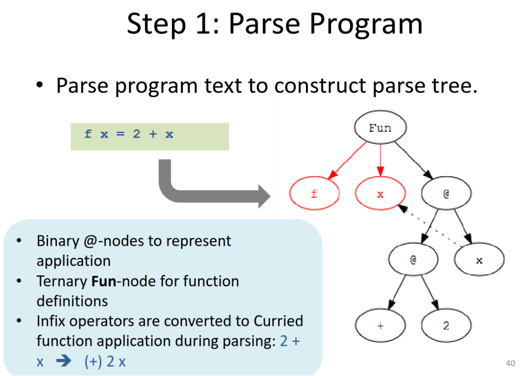
13.2.6 Missing Notes

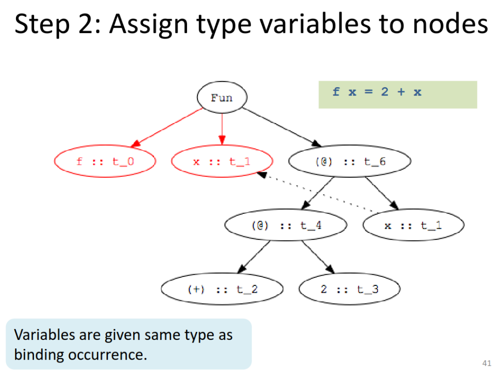
Look at slides 34...64 for more on Type Inference.

13.3 Inferencing types

In standard type checking the compiler examine body of each function and uses declared types to check agreement; type inference instead consists in examining code without type information, and infer the most general types that could have been declared

13.3.1 Steps schematics





Constraints can be deduced from (function) *Application* nodes `f x` and from *Abstractions* `f x = e`.

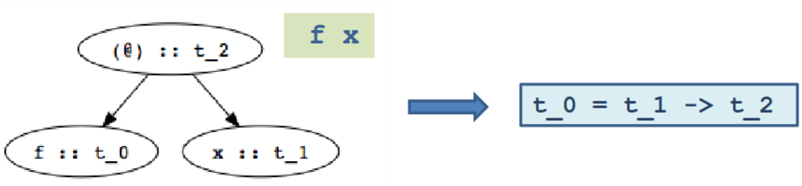


Figure 13.2: Deducing constraints from function application

- ◇ Type of `f` (`t_0` in figure) must be *domain* ⟶ *range*.
- ◇ **Domain** of `f` must be type of argument `x` (`t_1`)
- ◇ **Range** of f must be result of application (`t_2`)
- ◇ **Constraint**: `t_0 = t_1` →`t_2`

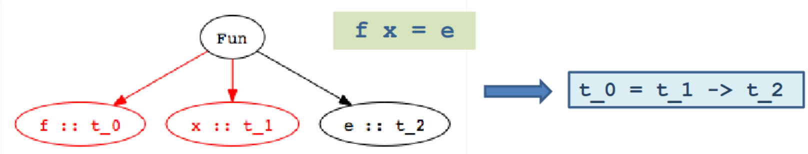


Figure 13.3: Deducing constraints from abstractions

- ◇ Type of `f` (`t_0`) must *domain* ⟶ *range*
- ◇ **Domain** is type of abstracted variable `x` (`t_1`)
- ◇ **Range** is type of function body `e` (`t_2`)
- ◇ **Constraint**: `t_0 = t_1` →`t_2`

Steps summary

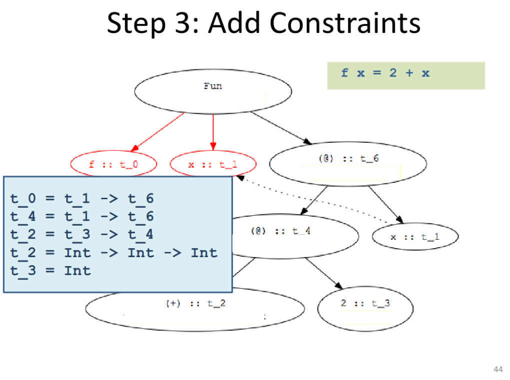
1. Parse program to build parse tree
2. Assign type variables to nodes in tree
3. Generate constraints:
   i. From environment: constants (`2`), built-in operators (`+`), known functions (`tail`).
   ii. From shape of parse tree: e.g., application and abstraction nodes.
4. Solve constraints using unification
5. Determine types of top-level declarations

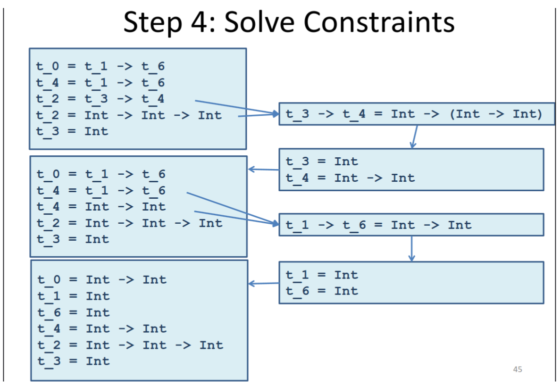
13.3.2 Polymorphism

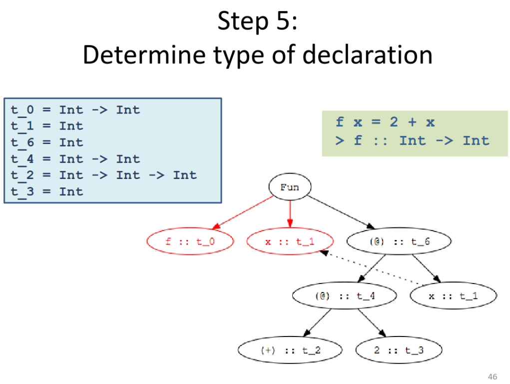
In general **unconstrained** type variables become **polymorphic types**; for instance, in the example below `t_4` is unconstrained, hence we get a polymorphic type:

```
f g = g 2
> f :: (Int → t_4) → t_4
```

For functions with multiple clauses, i.e. *polymorphic datatypes*, for each clause a separate type is inferred, and then the resulting types are combined by adding constraints such as that all clauses have the same type. In case of *recursive calls*: the function has same type as its definition.








```
append ([],r) = r
append (x:xs, r) = x : append (xs, r)
```

1. Infer type of each clause
   i. First clause:
   ```
   > append :: ([t_1], t_2) → t_2
   ```
   ii. Second clause:
   ```
   > append :: ([t_3], t_4) → [t_3]
   ```
2. Combine by equating types of two clauses
   ```
   > append :: ([t_1], [t_1]) → [t_1]
   ```

13.3.3 Overloading

In presence of **overloading** (*Type Classes*), type inference infers a **qualified type** `Q => T`

- T is a Hindley Milner type, inferred as seen before
- Q is set of type class predicates, called a constraint

```
example :: Ord a => a → [a] → Bool
example z xs =
   case xs of
      [] → False
      (y:ys) → y > z || (y==z && ys == [z])
```

In the example **Type** `T` is `a → [a] → Bool` while the **Constraint** `Q` is `{ Ord a, Eq a, Eq [a]}`. `Q` later simplifies[1] to `Ord a`

- `Ord a` because `y>z`
- `Eq a` because `y==z`
- `Eq [a]` because `ys == [z]`

[1]According to some rules not discussed here

13.4 Type Constructors

Type Classes are *predicates* over *types*, while [**Type**] **Constructor Classes** are *predicates* over *type constructors*.

For example, consider three versions of the `map` function (implementation is omitted): the basic one for lists, one for trees and one for `Maybe`.

```
map :: (a → b) → [a] → [b]
mapTree :: (a → b) → Tree a → Tree b
mapMaybe :: (a → b) → Maybe a → Maybe b
```

They all share the same structure, thus they can all be written as

```
fmap:: (a → b) → g a → g b
```

where `g` is a function from *types to types*, i.e. a **type constructor**; it is: `[-]` for lists, `Tree` for trees, and `Maybe` for options.

13.4.1 Functor

This pattern can be captured in a constructor class `Functor`. A **constructor class** is simply a type class where the predicate is over a type constructors rather than on a type:

```
class Functor g where
   fmap :: (a → b) → g a → g b
```

Compare with the definition of a *standard type class*:

```
class Eq a where
   (==) :: a → a → Bool
```

So, wrapping up, we can instantiate `Functor` on all three data structures, and then simply use the *overloaded* symbol `fmap`, instead of `map`, `mapTree` and `mapMaybe`.

```
class Functor f where
   fmap :: (a → b) → f a → f b
instance Functor [] where // [] is an instance of Functor
   fmap f [] = []
   fmap f (x:xs) = f x : fmap f xs
instance Functor Tree where // Tree is an instance of Functor
   fmap f (Leaf x) = Leaf (f x)
   fmap f (Node(t1,t2)) = Node(fmap f t1, fmap f t2)
instance Functor Maybe where // Maybe is an instance of Functor
   fmap f (Just s) = Just(f s)
   fmap f Nothing = Nothing
```

Chapter 14

Monads

14.1 Type Constructors

14.1.1 Towards Monads

Often type constructors can be thought of as defining *"boxes"* for values, and `Functors` with `fmap` allow to apply functions inside such *"boxes"*.
`Monad` is a constructor class introducing operations for *putting a value* into a "box" (`return`) and *composing* functions that return "boxed" values (`bind`)

"Monads" are type constructors that are instances of `Monad`

14.1.2 Maybe

A function `f :: a →Maybe b` is a partial function from `a` to `b`.

```
father :: Person → Maybe Person -- partial function
mother :: Person → Maybe Person -- (lookup in a DB)
maternalGrandfather :: Person → Maybe Person
maternalGrandfather p =
case mother p of
Nothing → Nothing
Just mom → father mom -- Nothing or a Person
```

14.1.3 Bind operator

We introduce a higher order operator to compose partial functions in order to "propagate" undefinedness automatically.

The bind operator will be part of the definition of a monad.

```
y >= g = case y of
    Nothing → Nothing
    Just x → g x

(>>=) :: Maybe a → (a → Maybe b) → Maybe b
```

`do{}` is an alternative equivalent syntax, more *imperative-like*.

```
bothGrandfathers p =
  father p >>=
  (\dad → father dad >>=
    (\gf1 → mother p >>=
      (\mom → father mom >>=
        (\gf2 → return (gf1, gf2)))))

bothGrandfathers p = do
  dad <- father p
  gf1 <- father dad
  mom <- mother p
```

```
    gf2 <- father mom
    return (gf1, gf2)
```

14.2 Monads as *

14.2.1 ...containers

```
class Monad m where -- definition of Monad type class
   return :: a → m a
   (>>=) :: m a → (a → m b) → m b -- "bind"
   ... -- + something more + a few axioms
```

The monadic constructor can be seen as a container: let's see this for `lists`

```
map :: (a → b) → [a] → [b] -- seen. "fmap" for Functors
return :: a → [a] -- container with single element
return x = [x]
concat :: [[a]] → [a] -- flattens two-level containers
   Example: concat [[1,2],[],[4]] = [1,2,4]
(>>=) :: [a] → (a → [b]) → [b]
xs >>= f = concat(map f xs)
Exercise: define map and concat using bind and return
```

14.2.2 ... computations

```
class Monad m where -- definition of Monad type class
   return :: a → m a
   (>>=) :: m a → (a → m b) → m b -- "bind"
   (>>) :: m a → m b → m b -- "then"
   ... -- + something more + a few axioms
```

A value of type m a is a "computation returning a value of type a"

For any value, there is a computation which "does nothing" and produces that result. This is given by function return

Given two computations x and y, one can form the computation x ¿¿ y which intuitively "runs" x, throws away its result, then runs y returning its result

Given computation x, we can use its result to decide what to do next. Given f: a -¿ m b, computation x ¿¿= f runs x, then applies f to its result, and runs the resulting computation.

Note that we can define then using bind:

```
x >> y = x >>= (\_ → y)
```

eturn, bind and then define basic ways to compose computations • They are used in Haskell libraries to define more complex composition operators and control structures (sequence, for-each loops, ...) • If a type constructor defining a library of computations is monadic, one gets automatically benefit of such libraries

Example: MAYBE • f:a -¿ Maybe b is a partial function • bind applies a partial function to a possibly undefined value, propagating undefinedness Example: LISTS • f:a -¿ [b] is a non-deterministic function • bind applies a non-deterministic function to a list of values, collecting all possible results

14.3 IO Monad

14.3.1 FP pros & cons

Pros

- Concise and powerful abstractions
  - higher-order functions, algebraic data types, parametric polymorphism, principled overloading, ...
- Close correspondence with mathematics
  - Semantics of a code function is the mathematical function
  - Equational reasoning: if x = y, then f x = f y
  - Independence of order-of-evaluation (Confluence, aka Church-Rosser)

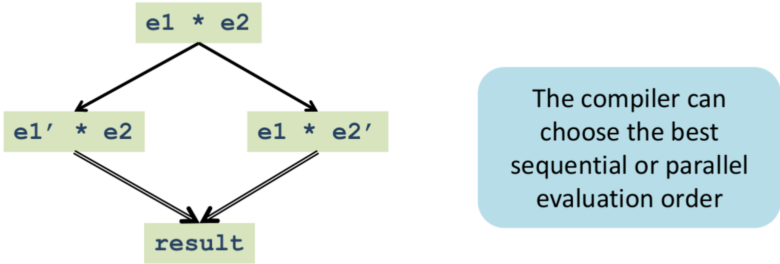


Figure 14.1: Evaluation order freedom

Cons

- Input/Output
- Imperative update
- Error recovery (eg, timeout, divide by zero, etc.)
- Foreign-language interfaces
- Concurrency control

Besides, recall that the whole point of a running a program is to **interact** with the external environment and affect it

14.3.2 Towards IO

To overcome the problem of interaction, an approach is to add imperative constructs to the language, for instance:

```
res = putchar 'x' + putchar 'y'
```

Seems easy right? Well, in fact no, because in lazy languages like Haskell, the evaluation order is **undefined**; so, in the previous example, which char will be printed first, `x` or `y`? The answer is not trivial for Haskell. However it is not an impossible problem. Haskell's approach is to exploit the concept of **Monads**.

Recall that the bind operator ¿¿= forces a **sequence** between the evaluation of terms; the IO monad exploits this and defines monadic values which are called **actions**, and prescribes how to compose them *sequentially*

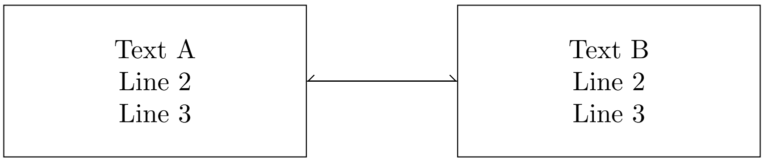


Figure 14.2: ¡caption¿

Before Monads

Before Monads there were **Streams**, which allowed a program to send stream of requests to OS and receive stream of responses, or the user could supply **continuations** to I/O routines to specify how to process results.
However, both of these approaches revealed to be not so useful.

14.3.3 Key Ideas - Monadic I/O

`IO` is a type constructor, instance of `Monad`, and a value of type `(IO t)` is an *action* (i.e. computation) that, when **performed**, may do some input/output before delivering a result of type `t`

- `return` returns the value without making I/O
- `then (>>)` [and also `\lstinline`bind (¿¿=)——] composes two actions sequentially into a larger action
- The only way to perform an action is to call it at some point, directly or indirectly, from `Main.main`, which is the standard entry point for Haskell programs.

An **action** is a *first-class* value, and **evaluating** has *no effect*: **performing** the action has the *effect*.

The actual meaning of this statement is unclear even to the professor ☺

```
return :: a → IO a
return a = \w → (a,w)
(>>=) :: IO a → (a → IO b) → IO b
(>>=) m k = \w → case m w of (r,w') → k r w'
```

By writing `case m w ...` we force the evaluation of `m`, resulting in the application of `k` to `r w'` to be performed (evaluated?) *after* the evaluation of `m`.

14.3.4 >>= and >>combinators

Operator is called **bind** because it binds the result of the left-hand action in the action on the right. Performing compound action `a >>= \x→b` :

1. performs action `a`, to yield value `r`
2. applies function `\x→b` to `r`
3. performs the resulting action `b{x <- r}`
4. returns the resulting value `v`

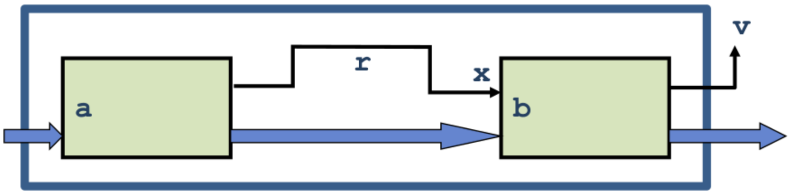


Figure 14.3: Bind Combinator

The **then** combinator `(>>)` instead does sequencing when there is no value to pass:

```
m >> n = m >>= (\_ → n)
```

14.3.5 Restrictions

In pure Haskell, there is no way to transform a value of type `IO a` into a value of type `a`.
Suppose you wanted to read a configuration file at the beginning of your program:

```
configFileContents :: [String]
configFileContents = lines (readFile "config") -- WRONG!
useOptimisation :: Bool
useOptimisation = "optimise" 'elem' configFileContents
```

The problem is that `readFile` returns an `IO String`, not a `String`.

Possible workarounds

1. Write entire program in IO monad. But then we lose the simplicity of **pure** code.
2. Escape from the IO Monad using a function from `IO String →String`. But this is **disallowed**!

We know the configuration file will *not change* during the program, so it doesn't matter ***when*** we read it.
This situation arises sufficiently often that Haskell implementations offer one last unsafe I/O primitive: `unsafePerformIO`
.

```
unsafePerformIO :: IO a → a
configFileContents :: [String]
configFileContents = lines(unsafePerformIO(readFile "config"))
```

The operator has a deliberately long name to *discourage* its use. Besides, its use comes with a proof obligation: a promise to the compiler that the *timing* of this operation relative to all other operations doesn't matter.

It is called *unsafe* because it breaks the soundness of the type system; thus, claims that Haskell is type safe are valid only when `unsafePerformIO` is **not** used.

14.4 Summary

◇ A complete Haskell program is a single IO action called main. Inside IO, code is **single-threaded**.

- Big IO actions are built by gluing together smaller ones with `bind (>>=)` and by converting pure code into actions with return.
- IO actions are first-class. They can be passed to functions, returned from functions, and stored in data structures; so, it is easy to define new "glue" combinators.
- The IO Monad allows Haskell to be pure while efficiently supporting side effects.
- The type system separates the *pure* from the *effectful* code.

Comparison

- In languages like ML or Java, the fact that the language is in the IO monad is baked in to the language. There is no need to mark anything in the type system because it is everywhere.
- In Haskell, the programmer can choose when to live in the IO monad and when to live in the realm of pure functional programming.
- So it is not Haskell that lacks imperative features, but rather the other languages that lack the ability to have a statically distinguishable pure subset.

Chapter 15

Lambdas

15.1 Java 8

```
List<Integer> intSeq = Arrays.asList(1,2,3);
intSeq.forEach(x → System.out.println(x));
// equivalent syntax
intSeq.forEach((Integer x) → System.out.println(x));
intSeq.forEach(x → {System.out.println(x);});
intSeq.forEach(System.out::println); //method reference
```

Note that local variables used inside the body of a lambda must be `final` or *effectively* `final`, or have to be static.

```
int var = 10; // must be [effectively] final
intSeq.forEach(x → System.out.println(x + var));
// var = 3; // uncommenting this line it does not compile
```

```
public class SVCExample { // static variable capture
   private static int var = 10;
   public static void main(String[] args) {
      List<Integer> intSeq = Arrays.asList(1,2,3);
      static int var = 10;

      intSeq.forEach(x → System.out.println(x + var));
      var = 3; // OK! it compiles
}}
```

15.2 Functional Interfaces

Java 8 *lambdas* are instances of *functional interfaces*, which are java interfaces with exactly *one* ***abstract*** method.

```
public interface Comparator<T> { //java.util
   int compare(T o1, T o2);
}
public interface Runnable { //java.lang
   void run();
}
public interface Consumer<T>{ //java.util.function
   void accept(T t)
}
public interface Callable<V> {//java.util.concurrent
   V call() throws Exception;
}
```

The lambda is invoked by calling the only abstract method of the functional interface; lambdas can be interpreted as instances of anonymous inner classes implementing the functional interface.

For instance, recalling the forEach presented earlier, the corresponding interface is the following. Note that it must be checked that the lambda matches the `forEach` signature defined in the interface:

```
intSeq.forEach(x -> System.out.println(x));

// List<T> extends Iterable<T>
interface Iterable<T>{ //java.lang
   default void forEach(Consumer<? super T> action)
      for (T t : this)
         action.accept(t);
```

Method References

Method references can be used to pass an existing function in places where a lambda is expected, but their signature needs to match the signature of the functional interface method required.

static	`ClassName::StaticMethodName`	`String::valueOf`
constructor	`ClassName::new`	`ArrayList::new`
specific object instance	`objectReference::MethodName`	`x::toString`
arbitrary object of a given type	`ClassName::InstanceMethodName`	`Object::toString`

Table 15.1: Method references examples

Chapter 16

Streams

Let's consider the properties of **Streams**, and we'll clearly see how they are different from Collections.

- **No storage**
  A stream is *not* a data structure that stores elements; instead, it conveys elements from a *source* [1] through a pipeline of computational operations.
- **Functional** in nature
  An operation on a stream produces a *result*, but does *not* modify its source.
- **Laziness-seeking** Many stream operations can be implemented lazily, exposing opportunities for optimization. Stream operations are divided into **intermediate** (*stream-producing*) operations —which are *always lazy*— and **terminal** (*value-* or *side-effect-producing*) operations.
- Possibly **unbounded**
  While collections have a *finite size*, streams need *not*. Short-circuiting operations such as `limit(n)` or `findFirst()` can allow computations on *infinite streams* to complete in *finite time*.
- **Consumable**
  The elements of a stream are only *visited once* during the life of a stream. Like an Iterator, a new stream must be generated to *revisit* the same elements of the source.
  The Stream is considered *consumed* when a *terminal* operation is invoked. No other operation can be performed on the Stream elements afterwards.

16.1 Pipelines

A typical pipeline contains

1. A **source**, producing (by need) the elements of the stream
2. Zero or more **intermediate** operations, producing streams
3. A **terminal** operation, producing side-effects or non- stream values

Example of typical pattern: `filter / map / reduce`

```
double average = listing // collection of Person
    .stream() // stream wrapper over a collection
    .filter(p -> p.getGender() == Person.Sex.MALE) // filter
    .mapToInt(Person::getAge) // extracts stream of ages
    .average() // computes average (reduce/fold)
    .getAsDouble(); // extracts result from OptionalDouble
```

16.1.1 Sources

Common sources are `Collections` via `stream()` and `parallelStream()` methods, but there are several and various other sources, like `IntStream.range(int, int)`, `Stream.iterate(Object, UnaryOperator)`, `BufferedReader.lines()`, `Random.ints()`, and many others.

[1] e.g. a data structure, an array, a generator function, an I/O channel,...

16.1.2 Intermediate operations

An intermediate operation keeps a stream *open* for further operations. Intermediate operations are lazy, and several of them have arguments of *functional interfaces*, thus **lambdas** can be used.
Exampls are `map(), peek(), distinct(), sorted()`

16.1.3 Terminal operations

A terminal operation must be the *final operation* on a stream. Once a terminal operation is invoked, the stream is consumed and is no longer usable. As said before, the typical approach is to collect values in a data structure, reduce to a value, and lastly print or cause other side effects.
Examples are `reduce()`,`forEach()`,`allMatch()` and others.

`reduce()` is basically our well-known `fold`

16.2 Mutable Reduction

Suppose we want to concatenate a stream of strings:

```
String concatenated = listOfStrings
    .stream()
    .reduce("", String::concat)
```

The above works, but is highly inefficient: it builds one new string for each element, since `Strings` are immutable in Java.
It would be better to "accumulate" the elements in a mutable object (e.g. a `StringBuilder`, a `collection`, ...). In our aid comes the **mutable reduction operation** which is called `collect()`, which requires three functions:

1. a **supplier** function to *construct* new instances of the result container,
2. an **accumulator** function to *incorporate* an input element into a result container,
3. a **combining** function to *merge* the contents of one result container into another.

```
<R> R collect( Supplier<R> supplier,
               BiConsumer<R, ? super T> accumulator,
               BiConsumer<R, R> combiner);
```

```
// NO streams
ArrayList<String> strings = new ArrayList<>();
for (T element : stream) {
    strings.add(element.toString());
}

// with streams and λs
ArrayList<String> strings =
stream.collect(
    () → new ArrayList<>(),          // Supplier
    (c, e) → c.add(e.toString()),    // Accumulator
    (c1, c2) → c1.addAll(c2));       // Combining

// with streams and method references
ArrayList<String> strings = stream.map(Object::toString)
.collect(   ArrayList::new,      // Supplier
            ArrayList::add,      // Accumulator
            ArrayList::addAll);  // Combining
```

However, `collect()` can also be invoked with a `Collector` argument, which encapsulates the functions used as arguments to collect (*Supplier*, *BiConsumer*, *BiConsumer*), allowing for reuse of collection strategies and composition of collect operations.

```
Map<String, List<Person>> peopleByCity =
    personStream.collect(Collectors.groupingBy(Person::getCity));
```

16.3 Parallelism

Streams facilitate parallel execution: stream operations can execute either in serial (default) or in parallel, with the runtime support transparently taking care of using multithreading for parallel execution. If operations *don't* have side-effects, *thread-safety* is **guaranteed** even if *non-thread-safe collections* are used (e.g. `ArrayList`).

Also *concurrent mutable reduction* is supported for parallel streams, however Order of processing stream elements depends on serial/parallel execution and intermediate operations, and may not be predictable.

```
double average = persons   // average age of all males
   .parallelStream()       // members in PARALLEL
   .filter(p -> p.getGender() == Person.Sex.MALE)
   .mapToInt(Person::getAge)
   .average()
   .getAsDouble();

   sorted_listOfIntegers.parallelStream()
   .forEach(e -> System.out.print(e + " "));
   // may print: 3 4 1 6 2 5 7 8
```

16.3.1 Summing up

One should use Parallelism

- When operations are independent, and
- Either or both:
  - Operations are computationally expensive
  - Operations are applied to many elements of

  efficiently splittable data structures

"Always measure before and after parallelizing*!"*

16.3.2 Critical Issues

- **Non-interference**
  - Behavioural parameters (like *lambdas*) of stream operations should *not affect* the source (i.e. *non-interfering behaviour*)
  - Risk of `ConcurrentModificationExceptions`, even in single-threaded execution
- **Stateless behaviours**
  - *Stateless* behaviour for intermediate operations is *encouraged*, as it facilitates parallelism, and functional style, thus maintenance
- **Parallelism and thread safety**
  - For parallel streams with *side-effects*, ensuring thread safety is the programmers' responsibility

```
String concatenatedString = listOfStrings
.stream()
.peek(s -> listOfStrings.add("three")) // DON'T DO THIS!
   // Interference occurs here.
.reduce((a, b) -> a + " " + b)
.get();
```

16.4 Monads in Java

```
public static <T> Optional<T> of(T value)
// Returns an Optional with the specified present non-null value.
<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)
```

If a value is present, `flatMap` applies the provided *Optional-bearing* mapping function to it, return that result, otherwise return an empty `Optional`.

```
static <T> Stream<T> of(T t)
// Returns a sequential Stream containing a single element.
<R> Stream<R> flatMap(
Function<? super T,? extends Stream<? extends R>> mapper)
```

Here `flatMap` returns a `Stream` consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.

Chapter 17

Frameworks and IOC

17.1 Frameworks

A **Software Framework** is a collection of common code providing generic functionality that can be selectively overridden or specialized by user code providing specific functionality.
An **Application Framework** is a software framework used to implement the standard structure of an *application* for a specific development environment.

Examples

1. General Software Frameworks
   i. `.NET`
   ii. `Android SDK`
   iii. `Cocoa`
   iv. `Eclipse`
2. GUI Frameworks
   i. `MFC`
   ii. `Gnome`
   iii. `Qt`
3. Web Frameworks
   i. `ASP.NET`
   ii. `Rails`
   iii. `GWT`
   iv. `Spring`
   v. `Flask`

A framework embodies some *abstract design*, with more behavior built in. In order to use it you need to insert your behavior into various places in the framework either by subclassing or by plugging in your own classes, then the framework's code, which handles the program's **control flow** (the "`main` execution"), then calls your code at these points.
This realizes a very general concept, emphasizing **inversion of control** as opposed to libraries, where the user's code calls the library one, here is the code of the framework that calls the user's one.

17.1.1 Component Frameworks

Componenent Frameworks support development, deployment, composition and execution of components designed according to a given **Component Model**. More specifically, they support **composition/connection** of components according to the mechanisms provided by the *Component Model*, allowing instances to be "plugged" into the component framework itself, and regulating their **interaction**.

IDE and Frameworks

NetBeans is both an `IDE` and supports the `JavaBeans` *Component Framework*.
In general A framework can be supported by several `IDE`s

e.g. `Spring` supported by `Spring Tool Suite` (based on `Eclipse`), `NetBeans`, `IntelliJ IDEA`, `Eclipse`, ...

While an `IDE` can support several frameworks e.g NetBeans supports JavaBeans, Spring, J2EE, Maven, Hibernate, JavaServer Faces, Struts, Qt, ...

17.1.2 Features

Consist of **parts** that are found in many apps of that type

- **Libraries** with APIs (classes with methods etc.)
- Ready-made extensible programs (”**engines**”)
- Sometimes also **tools** (e.g. for development, configuration, content)

They also provide reusable abstractions of code wrapped in a well-defined API, however recall that,unlike in libraries, the overall program’s **flow of control** is *not* dictated by the caller, but by the *framework*.

Frameworks usually support extensibility, either by extending within the framework language — using, subclassing, overriding, implementing interfaces, registering event handlers, ...— or through plug-ins defined in a specific format.

17.2 Inversion of Control

17.2.1 GUI

In *text-based interaction*, the order of interactions and of invocations is decided by the the code, while in the ***GUI**-based interaction*, the *GUI* loop decides when to invoke the methods (listeners), based on the order of events.

```
#ruby
puts 'What is your name?'
name = gets
process_name(name)
puts 'What is your quest?'
quest = gets
process_quest(quest)
```

TEXT

```
require 'tk'
root = TkRoot.new()
name_label = TkLabel.new() {text "What is Your Name?"}
name_label.pack
name = TkEntry.new(root).pack
name.bind("FocusOut") {process_name(name)}
quest_label = TkLabel.new() {text "What is Your Quest?"}
quest_label.pack
quest = TkEntry.new(root).pack
quest.bind("FocusOut") {process_quest(quest)}
Tk.mainloop()
```

GUI

Figure 17.1: Text vs GUI interaction

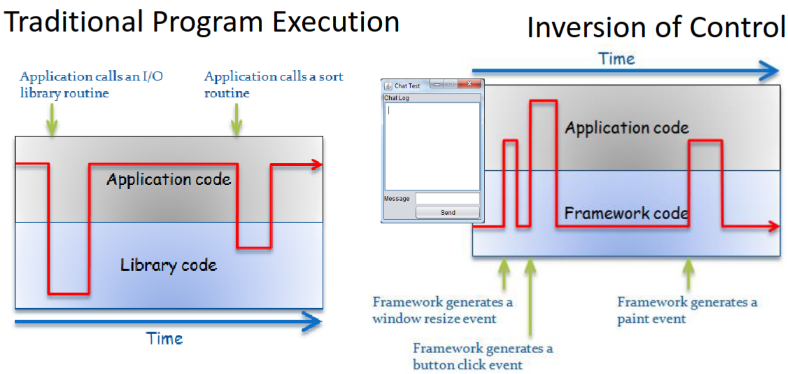


Figure 17.2: IoC: Library vs Framework approach

17.2.2 Containers

Often Frameworks provide **containers** for deploying *components*: a container may provide at *runtime functionalities* needed by the components to execute.

For examples EJB containers are responsible of the persistent storage of data and of the availability of EJB’s for all authorized clients.

17.3 Loosely Coupled Systems

Good *OO Systems* should be organised as network of interacting objects, keeping in mind as a goal to have *high **cohesion***, *low **coupling***.
Low coupling has as key advantages

◇ Extensibility
◇ Testability
◇ Reusability

17.3.1 Dependecy Injection

When discussing **IoC** in Frameworks, *"Control"* does not refer only to control flow, but also control over *dependencies,coupling, configuration.*

We can make a few considerations on IoC with respect to dependencies:

◇
◇ something outside a component handles:
  – configuration (properties)
  – wiring / dependencies (components)
◇ component-oriented
◇ removes coupling
  – coupling of configuration and dependencies to the point of use
  – coupling of component to concrete dependent components
◇ somewhat contrary to encapsulation

17.4 Trade Monitor

Let's discuss this example to see how all of this comes into practice.

A trader wants that the system rejects trades when the exposure reaches a certain limit

Thus the component (class) `TradeMonitor` provides a method `TryTrade` (below) which checks the condition, accessing *current exposure* and *exposure limit* from a `DAO` (*Data Access Object*), a persistent storage.

```
public bool TryTrade(string symbol , int amount){
   int limit = limitDao.GetLimit(symbol);
   int exposure = limitDao.GetExposure(symbol);
   return (exposure + amount > limit) ? false : true;
   }
```

How can we limit dependencies among the two components?

17.4.1 Interfaces - Refactoring 1

Let's consider a possible refactoring, introducing **interface** and implementation separation, which still has a static dependency on `DAO` :

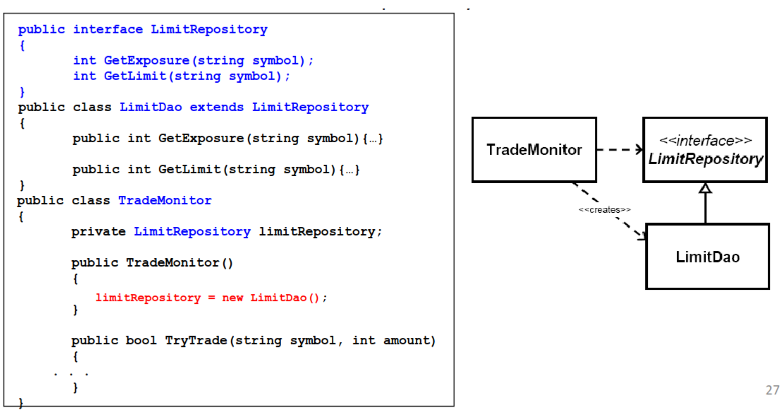


Figure 17.3: Refactoring 1

17.4.2 Factory - Refactoring 2

Here we introduce a **factory** which resolves the previous problem, but `LimitDao` is still tightly coupled, but to `Factory`.

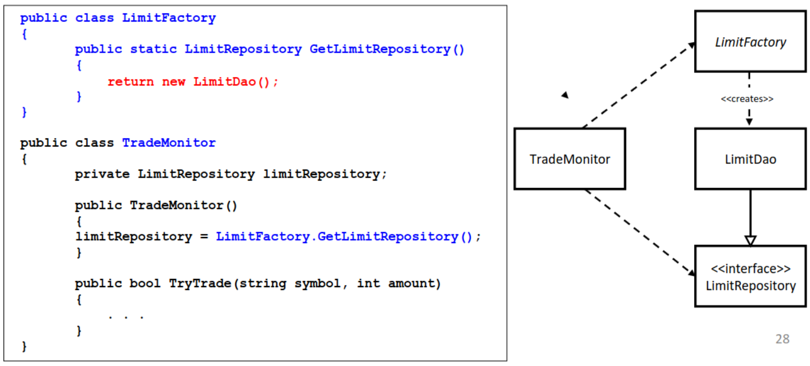


Figure 17.4: Refactoring 2

17.4.3 ServiceLocator - Refactoring 3

Introduce a `ServiceLocator`. This object acts as a (static) registry for the `LimitDao` you need, giving us extensibility, testability, reusability.
However ote that an external `Assembler` sets up the registry.

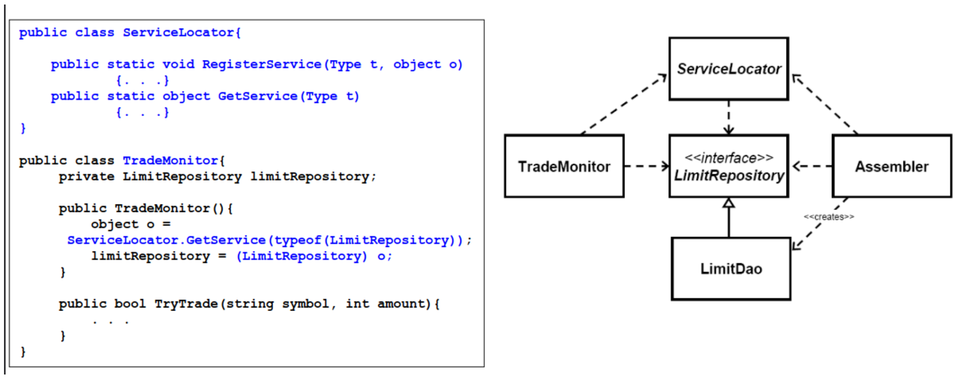


Figure 17.5: Refactoring 3

Pros

- ◇ The Service Locator pattern succeeds in decoupling the TradeMonitor from the LimitDao
- ◇ Allows new components to be dynamically created and used by other components later
- ◇ It can be generalized in several ways, eg. to cover dynamic lookup

Cons

- ◇ Every component that needs a dependency must have a reference to the service locator
- ◇ All components need to be registered with the service locator
- ◇ If bound by name:
  - Services can't be type-checked
  - Component has a dependency to the dependent component names
  - if many components share an instance but later you want to specify different instance for some, this becomes difficult
- ◇ If bound by type can only bind one instance of a type in a container
- ◇ Code needs to handle lookup problems

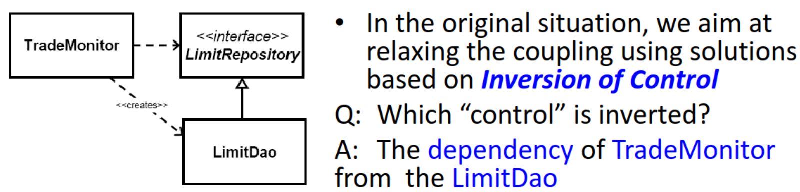
17.5 Dependency Injection

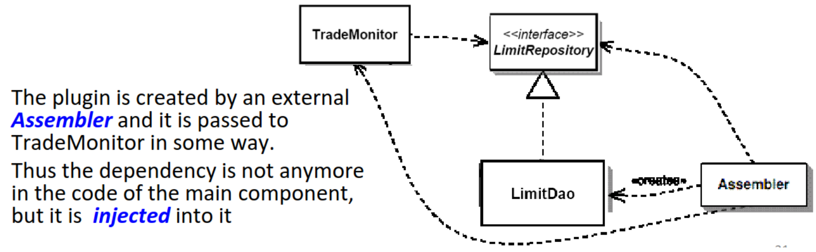
Dependency injection allows avoiding *hard-coded* dependencies (strong coupling) and changing them, and allows selection among multiple implementations of a given dependency interface at run time. It can be achieved through:

1. Setter injection
2. Constructor injection
3. (Interface injection)

Both **Service Locator** and **Dependency Injection** provide the desired decoupling, but let's compare the two solutions:

- ◇ With service locator there is no **IoC**, since the desired component is obtained after request by the `TradeMonitor` to the `Locator`; this makes the application still depending on the locator.





◇ With dependency injection there is *no explicit request*: the component appears in the application class.

Inversion of control a bit harder to understand, however, it is easier to find dependencies of component if *Dependency Injection* is used

Check *constructors* and *setters*
vs
Check *all invocations* to `Locator` in the source code

17.6 Designing Frameworks

Frameworks are normally implemented in an object- oriented language such as Java. It is important to learn to analyze a potential software family, identifying its possible common and variable aspects, and evaluating alternative framework architectures.

A possible idea is to start from a known divide-and-conquer algorithm such as:

Listing 17.1: Example pseudocode of a Divide-and-Conquer algorithm

```
function solve (Problem p) returns Solution {
   if isSimple(p)
      return simplySolve(p);
   else
      sp[] = decompose(p);
      for (i= 0; i < sp.length; i = i+1)
         sol[i] = solve(sp[i]);
      return combine(sol);
}
```

We can apply known techniques and patterns to **define** a *framework* for a **software family**. Instances of the defined framework, obtained by standard extension mechanism, will be concrete algorithms of the *family*.

17.6.1 Terminology

◇ **Frozen Spot**
  common (shared) aspect of the software family
◇ **Hot Spot**
  variable aspect of the family
◇ **Template method**
  concrete method of base (abstract) class implementing behavior common to all members of the family
◇ A hot spot is represented by a group of abstract ***hook*** *methods*.
◇ A template method calls a *hook method* to invoke a function that is specific to one family member —Inversion of Control—.
◇ A hot spot is realized in a framework as hot spot subsystem:

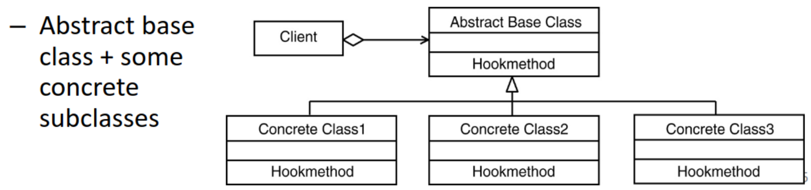


Figure 17.6: *Hotspot* implementation

Principles

1. The unification principle [Template Method Design Pattern]
   i. It uses inheritance to implement the hot spot subsystem
   ii. Both the template methods and hook methods are defined in the same abstract base class
   iii. Hook methods are implemented in subclasses of the base class
2. The separation principle [Strategy Design Pattern]
   i. It uses delegation to implement the hot spot subsystem
   ii. The template methods are implemented in a concrete context class; the hook methods are defined in a separate abstract class and implemented in its subclasses
   iii. The template methods delegate work to an instance of the subclass that implements the hook methods

17.6.2 Template Method design pattern

It is one of the behavioural pattern of the *Gang of Four*; Its intent is to define the skeleton of an algorithm in an operation, *deferring* some steps to subclasses: A **template method** belongs to an *abstract* class and it defines an algorithm in terms of *abstract* operations that subclasses **override** to provide *concrete behavior*.

Template methods call, among others, the following operations:

1. **concrete** operations of the abstract class $\longrightarrow$ fixed parts of the algorithm
2. **primitive** operations, $\longrightarrow$ abstract operations that subclasses have to implement
3. **hook** operations $\longrightarrow$ provide default behavior that subclasses may override if necessary.
   A hook operation often does nothing by default.

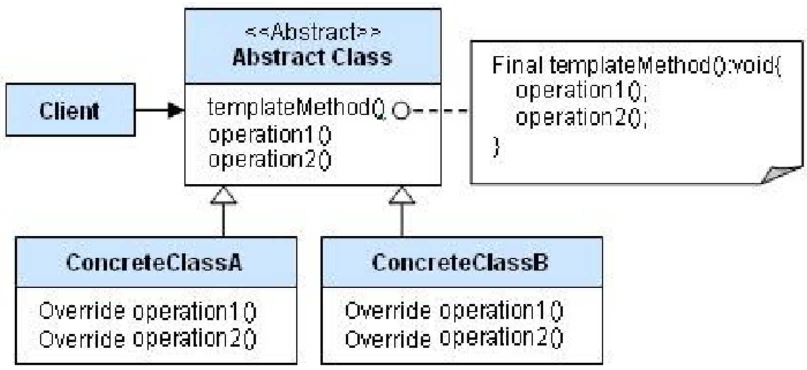


Figure 17.7: Template method

Applying *Unification Principle*

Let's consider the result of applying *unification principle* to the example code 17.1 provided before.

```
-- hotspots
function solve (Problem p) returns Solution { -- templatemethod
   if isSimple(p)
      return simplySolve(p);
   else
      sp[] = decompose(p);
      for (i= 0; i < sp.length; i = i+1)
         sol[i] = solve(sp[i]);
      return combine(sol);
}
```

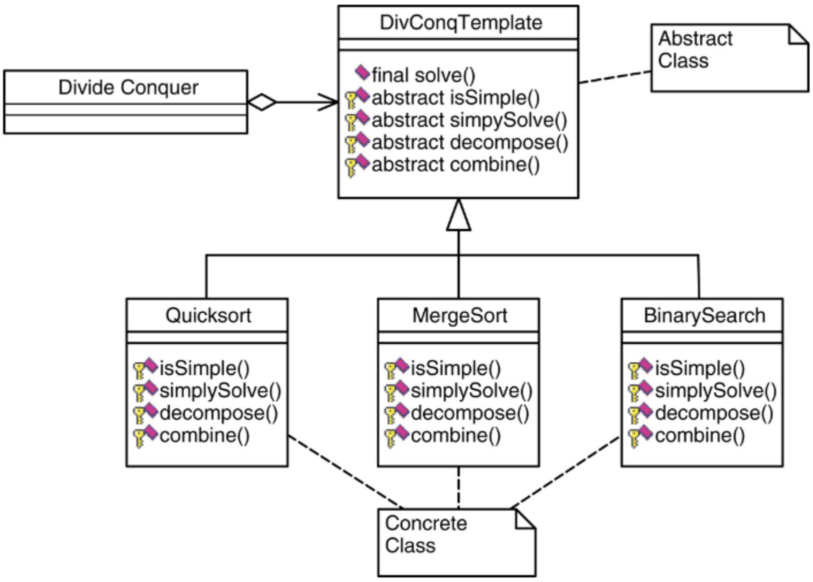
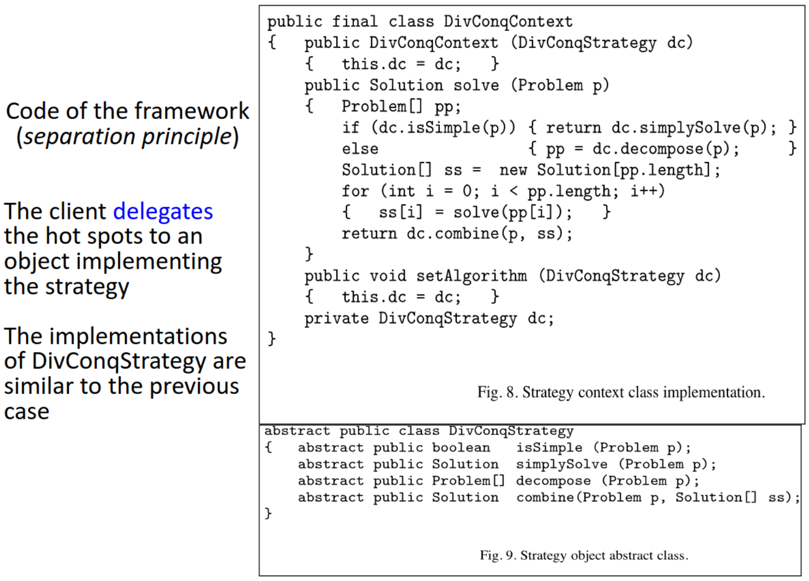


Figure 17.8: The generic schema of a Divide-and-Conquer *Template Method* designed Framework

17.6.3 Strategy design pattern

Another one of the behavioural pattern of the *Gang of Four*; Its intent is to allow to select (part of) an algorithm at runtime, leading the client to use an object implementing the interface and invoking methods of the interface for the hot spots of the algorithm.

Applying *Separation Principle*




```
public final class DivConqContext
{   public DivConqContext (DivConqStrategy dc)
    {   this.dc = dc;   }
    public Solution solve (Problem p)
    {   Problem[] pp;
        if (dc.isSimple(p)) { return dc.simplySolve(p); }
        else                { pp = dc.decompose(p);     }
        Solution[] ss =  new Solution[pp.length];
        for (int i = 0; i < pp.length; i++)
        {   ss[i] = solve(pp[i]);   }
        return dc.combine(p, ss);
    }
    public void setAlgorithm (DivConqStrategy dc)
    {   this.dc = dc;   }
    private DivConqStrategy dc;
}
```

Fig. 8. Strategy context class implementation.

```
abstract public class DivConqStrategy
{   abstract public boolean   isSimple (Problem p);
    abstract public Solution  simplySolve (Problem p);
    abstract public Problem[] decompose (Problem p);
    abstract public Solution  combine(Problem p, Solution[] ss);
}
```

Fig. 9. Strategy object abstract class.

Figure 17.9: Result of applying the Separation principle on the example code

17.7 Development by generalization

Recalling what said earlier, we try to address:

Learning to analyze a potential software ***family****, identifying its possible common and variable aspects, and evaluating alternative framework architectures. Framework design involves incrementally* ***evolving*** *a design rather than discovering it in one single step*

Where the *evolution* consists of examining **existing designs** for family members, identifying the **frozen** and **hot spots** of the family, and ultimately **generalizing** the program structure to enable *code reusing* for frozen spots and multiple *different implementations* for each hot spot.

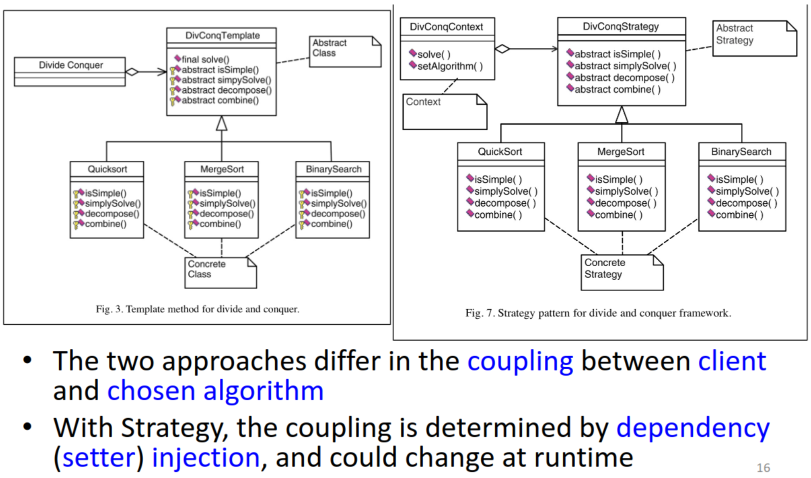


Figure 17.10: Comparison between the two pattern's schemas

We will consider an example based on binary tree traversals, and discuss each generalization step.

17.7.1 Identifying Frozen and Hot spots

Frozen Spots, which are fixed for the whole family:

1. The structure of the tree, as defined by the BinTree hierarchy
2. A traversal accesses every element of the tree once, but it can stop before completing
3. A traversal performs one or more visit actions accessing an element of the tree
   meaning that there may be different and multiple actions after visiting a node, since it may represent the end of a left subtree visit, a right subtree visit or a root.

Let's identify possible **Hot Spots**, which have to be fixed in each element of the family.

1. Variability in the visit operation's **action**: a function of the current node's value and the accumulated result
2. Variability in **ordering** of the visit *action* with respect to subtree traversals; Should support *preorder*, *postorder*, *in-order*, and their combination.
3. Variability in the **tree navigation** technique. Should support any access order.
   not only left-to-right, depth-first, total traversals

17.8 Visitor Pattern

The **Visitor** pattern guarantees *separation* between algorithm and data structure.

The data structure can be made of different types of components (*ConcreteElements*), and each component implements an `accept(Visitor)` method. The `Visitor` defines one visit method for each type, including the navigation logic in itself. At each step, the correct visit method is selected by **overloading**.

```
public interface BinTreeVisitor
{   abstract void visit(Node t);
    abstract void visit(Nil t);
}
```

```
abstract public class BinTree
{   public void setValue(Object v)  { }        // mutators
    public void setLeft(BinTree l)  { }        //  default
    public void setRight(BinTree r) { }
    abstract public void accept(BinTreeVisitor v); // accept Visitor
    public Object  getValue() { return null; } // accessors
    public BinTree getLeft()  { return null; } //   default
    public BinTree getRight() { return null; }
}
```

```
public class Node extends BinTree
{   public Node(Object v, BinTree l, BinTree r)
    {   value = v; left = l; right = r; }
    public void setValue(Object v)  { value = v; }  // mutators
    public void setLeft(BinTree l)  { left = l;  }
    public void setRight(BinTree r) { right = r; }
    // accept a Visitor object
    public void accept(BinTreeVisitor v) { v.visit(this); }
    public Object  getValue() { return value; }    // accessors
    public BinTree getLeft()  { return left;  }
    public BinTree getRight() { return right; }
    private Object  value;    // instance data
    private BinTree left, right;
}
```

```
public class Nil extends BinTree
{   private Nil() { }   // private to require use of getNil()
    // accept a Visitor object
    public void accept(BinTreeVisitor v) { v.visit(this); }
    static public BinTree getNil() { return theNil; } // Singleton
    static public BinTree theNil = new Nil();
```

Figure 17.11: Visitor Pattern applied to the BinTree visit

Even if in the *Visitor* pattern, as in the *Template Method* pattern, an abstract class is defined and later implemented by subclasses which provide concrete behaviour, in the *Visitor* pattern such classes are ***intended*** to be used directly by *clients*, while in the *Template Method* pattern they are ***intended*** to be called by the *Frozen Spots* inside the abstract class itself, not by *clients*.

Chapter 18

Java Memory Model

A **memory model** for *multithreaded* systems specifies how mem actions in a program will appear to execute to the programmer, i.e. —more specifically— which value each read of a memory location may return.
Every hardware and software interface of a system that admits multithreaded access to shared memory **requires** a memory model. such model determines the transformations that the system can apply to a program

In the case of high-level programming languages such as Java the memory model determines

1. the transformations the compiler may apply to a program when **producing bytecode**
2. the transformations a VIrtual-Machine may apply to bytecode when **producing native code**
3. the **optimizations** that hardware may perform on the native

Besides, the model also impacts the programmer, since such transformations determine the possible outcomes of a program.
Without a well defined memory model for a programming language, it is impossible to known what the legal results are for a program in such language.

When we programming *"correctly"* in Java, using `volatile` keywords and related constructs, we can however —in some sense— ignore the memory model

Memory Hierarchy In modern architectures memory is stratified ranging from mass memory (hard disks) to `CPU` registers, passing through different cache levels (`L1,L2,L3`), obtaining a **memory hierarchy**; depending on `CPU` architectures, cache levels may be shared or not among cores.

18.1 Java Memory Model

For incorretly synchronized programs, the behaviour is *bounded* by a well-defined notion of **causality**, so the semantics are *not* completely undefined as they were in the early (pre `Java 5 -` 2004) versions of the memory model.
The causality constraints are *strong enough* to *respect* the **safety** and **security properties** of java, and *weak enough* to *allow* standard compiler and hardware **optimizations**.

18.1.1 Runtime Data Areas

Local -*primitive* type- **variables** of methods are allocated on thread stacks, and *cannot* be accessed by other threads; *Objects* are instead allocated on the Heap.

For what concerns the distribution of data in Java, it may be spread orthogonally around the memory hierarchy, i.e. anything can go anywhere.
This leads to two key issues:

1. **Visibility** of variable updates
2.

(...)

18.2 `volatile` modifier

`volatile` is a modifier that can only be applied to fields of a class, and intuitively it declares that a field can be modified by multiple threads[1]. The JMM guarantees that the write of a volatile variable is visible when it is read. An *implementation* should guarantee that the new value is flushed from the cache to the RAM, if a read happens "after" a write.

What does it mean that *"a read happens after a write"*?

This will be discussed later on in Sec 18.3, however, When threads do *not* use any synchronization mechanism, their behavior is described as the sequence of performed read/write **actions**, along with the results of the read operations; such sequence has a **partial ordering** whose *legitimacy* is checked by the JMM which aims to ensure that a read results in the actual last value written in this partial ordering.

Notice that `volatile` doesn't solve **Data Races**, which need synchronization mechanism; the typical example is incrementing a shared counter, which actually consists of three operations, `read`, increment the value read, and `write` it; such actions are **not** performed *atomically*[2], thus a second thread may read the "old" before the first one writes the updated one, resulting in only one incrementation instead of two.

Monitors

Monitors are the default Java synchronization mechanisms. Every object has a monitor exposing a lock which can be held only by one thread at a time. methods and ... with the `synchronized` modifier are guarded by the lock.

18.3 Describing thread behaviour

The JMM has no explicit global ordering of all actions by time consistent with each thread's perception of time, and has no global store.
Executions are instead described in terms of memory **actions**, **partial orders** on these actions, and a **visibility function** that assigns a write action to each read action.

Actions

1. Volatile read
2. Volatile write
3. lock
4. TODO

An execution of a *single-threaded* program fixes a total order $\leq_{po}$ on its *actions*, called **program order**; while for a *multi-threaded* the program order consists in the union the program order of its threads, so it does not relate actions of different threads.
An execution of a *multi-threaded* program is **sequentially consistent** if there is a total order of its actions consistent with the program order —and such that each read has the value of the last write—.
For *datarace-free*[3] mt-programs, the JMM guarantees that only **sequential consistent** executions are legal.

JMM has been designed to guarantee three things:

1. Promise for programmers
   Sequential consistency must be sacrificed to allow optimizations, but it still holds for datarace-free program
2. Promise for security
   Values should not appear "out of thin air", allowing for information leakage
3. Promise for compilers
   HW and SW optimizations should be applied without violating (both) the first two requirements

TODO

[1] Clearly incompatible with `final`
[2] So... which operations are atomic and which are not?
[3] Also called <u>*corretly*</u> or <u>*well*</u> *synchronized programs*

```
int r1
```

Instr reordering may be performed as long as it guarantees sequential consistency in the *single thread*

TODO

Listing 18.1: Thread 1

```
r1 = x;
y = r1;
```

Listing 18.1: Thread 2

```
r2 = y;
x = r2;
```

`x = y = 0` initially; can we obtain `r1 == r2 == 42` at the end?
*"Well **no**, but actually **yes**..."*
In some situations the Runtime environment may *guess* that, at some point, `x` evaluates to `42`: we say that `42` comes *"out-of-thin-air"*. Then it checks by looking at the two thread instructions if it may happen that `r1 == r2 == 42`:

"Yes! So x actually really evaluates to 42! I guessed right! ☺"

This was an accepted guess before the JMM introduced with Java 5 (2004), but currently such claims at runtime are forbidden.

18.3.1 Synchronization order

Each execution of a program is associated with a **synchronization order** $\leq_{so}$ which is a total order over all synchronization actions satisfying:

1. Consistency with program order
2. Read to a volatile variable v returns the value of the write to v that is ordered last before the read by the synchronization order.

TODO

Formally Defining Data Races Two accesses x and y form a data race in an execution of a program if they are from different threads, they conflict, and they are not ordered by happens-before.

A program is said to be correctly synchronized or datarace-free if and only if all sequentially consistent executions of the program are free of data races.

The first requirement for the JMM is to ensure sequential consistency for correctly synchronized or datarace free programs
Programmers should not worry about code transformations for datarace-free programs. TODO

Executions - Formal Definition

$E = (P, A, \leq_{po}, \leq_{so}, W, V, \leq_{sw}, \leq_{hb})$

1. TODO TODO

$E = (P, A, \leq_{po}, \leq_{so}, W, V, \leq_{sw}, \leq_{hb})$ is a **well-formed execution** if:

1. Each read of a var x sees a write to x, and all reads and writes of volatile variables are volatile actions
2. Synchronization order is consistent with program order and mutual exclusion
3. The execution obeys intra-thread consistency
4. The execution obeys intra-thread and happens-before consistency on TODO

Now, which *well-formed executions* are **legal**? **Legal executions** are built iteratively: in each iterations, the JMM commits a set of memory actions; actions can be committed if they occur in some well-behaved...

A well-formed $E = (P, A, \leq_{po}, \leq_{so}, W, V, \leq_{sw}, \leq_{hb})$ is validated by *commiting* actions in A; if all actions of A are committed, then E is legal. There must exists a sequence of subsets of a

$$C_0 C_1 = A$$

and one $\{E_i\}$ of well-formed executions such that each E_i witnesses the actions in C_i

TODO many slides have been skipped

18.4 Intructions reordering

Chapter 19

RUST

Rust is a general purpose, system programming language with a focus on safety, especially **safe concurrency**, supporting both *functional* and *imperative* paradigms. Its main goal is to *ensure safety without penalizing efficiency*. `C/C++` provide more control but less safety, while `Python/Haskell` provide less control but more safety. **Rust** aims to get the best of both worlds, providing both **control** and **safety**.

Despite its syntax resemblance to C/C++, in a deeper sense Rust is closer to the ML family languages; in fact almost every part of a function body is an expression, include `if-then-else` constructs, which returns a value.

19.1 Key Points

Rust, similarly to C, compilates to **object code** for bare-metal performance, but it supports **memory safety**: programs can *dereference* only previously allocated pointers that have not been freed, and *out-of-bound* array accesses not allowed; Besides, the **overhead** introduced is very low, since it's the *compiler* which checks that memory safety rules are followed, and there's *no garbage collection*, so zero-cost abstraction in managing memory.
This is achieved through and **advanced type system** and three key concepts to prevent memory corruption:

1. Ownership
2. Borrowing
3. Lifetime

Again, Rust is designed to be **memory safe** even in the presence of concurrency, and guarantees the following properties **statically**, meaning that if the program *compiles* it will *never manifest a violation* of these properties:

- No `null` pointers
  longrightarrow accessing a variable which does not hold a value
- No dangling pointers
  longrightarrow Pointers to invalid memory location
  - Pointers to explicitly deallocated objects;
  - Pointers to locations beyond the end of an array;
  - Pointers to objects allocated on the stack;
- No double frees
  longrightarrow A memory location in the heap is reclaimed twice
- No data races
  longrightarrow unpredictable results in concurrent computations
- No iterator invalidation

19.2 `null` and Primitive types in Rust

A `null` value does **not** exist in Rust, so in some way it must address the problem of accessing a variable which does not hold a value.
Data values can only be initialized through a fixed set of forms, requiring their inputs to be already initialized, and if any branch of code fails to assign a value to the variable, we get a **compile time error**.

Static/global variables must be initialized at declaration time.

Nullable types, are managed with a generic `Option<T>`, playing the role of Haskell's `Maybe` or Java's `Optional`

```
enum std::option::Option<T> {
    None,
    Some(T)
}
```

19.2.1 Primitive Types

Listing 19.1: Rust primitive types

```
// Numeric types:
i8 / i16 / i32 / i64 / isize
u8 / u16 / u32 / u64 / usize
f32 / f64

bool
char // (4-byte unicode)
```

- **Type inference** for variables declarations with let
- **No overloading** for literals: type annotations to disambiguate
- **Tuples** like in Haskell
- **Arrays** with fixed length.
  out-of-bound access is checked at **runtime**, but it's just a single comparison, its overhead is negligible

19.3 Memory Management

As usual, Rust uses a **stack** of activation records, and a **heap** for dynamically allocated data structures.

The user is forced to be *aware of where* the data are stored: there is no **implicit boxing**[1].

```
fn main() {
    let x = 3; // 'let' allocates a variable on the stack
    let y = Box::new(3); // y is a reference to 3 on the heap
    println!("x == y is {}", x == *y); // "x == y is true"
    }
```

To avoid the overhead of a Garbage collection mechanism and the possible subtle errors introduced a programmer to whom memory management is delegated, Rust provides *deterministic management of resources*, with very low overhead, using **RAII** (*Resource Acquisition Is Initialization*).

By *default*, Rust variables are **immutable**, and their usage is statically checked by the compiler. `mut` is used to declare a resource as mutable.

[1]Act of *boxing* an `int` in `Integer`, or extracting an *int* from *Integer*

Listing 19.2: Compilation *error*

```
fn main() {
    let a: i32 = 0;
    a = a + 1;
    println!("a == {}", a);
}
```

Listing 19.2: Compilation ✓

```
fn main() {
    let mut a: i32 = 0;
    a = a + 1;
    println!("a == {}", a);
}
```

The *Resource Acquisition Is Initialization* (**RAII**) programming idiom states that Resource *allocation* is done during object *initialization*, by the constructor, while resource *deallocation* (**release**) is done during object destruction (specifically **finalization**), by the destructor.

19.3.1 Ownership

This approached is adopted in modern `C++`: small objects are allocated on *stack*, while larger resources are on the *heap* –or elsewhere– and are **owned** by an object on the *stack*, who is responsible for *releasing* the resource in its destructor.
Each resource has a **unique owner**.

Rust supports RAII in a *strict* way through an **ownership system**, based on the concepts of *ownership* and *borrowing*.

Ownership

O1 - Every value is *owned* by a variable, identified by a name (possiby a path);
O2 - Each value has *at most* *one owner* *at a* *time*;
O3 - When the owner goes *out-of-scope*, the value is *reclaimed* / destroyed.

By default, an assignment between variables has a **move semantics**: the ownership is moved from the RHS to the LHS

```
fn main() {
    let x = Box::new(3);
    let _y = x; // underscore to avoid 'unused' warning
    println!("x = {}", x); // error!
    }
```

For primitive types and types implementing the **Copy trait**, assignment has a **copy semantics**;

Here O2 is satisfied because a new value is created

```
    fn main() {
        let x = 3;
        let _y = x;
        println!("x = {:?}", x); // OK
    }
```

```
    fn main() {
        let x = Option::Some(3);
        let _y = x;
        println!("x = {:?}", x); // OK
    }
```

The same move semantics apply also for parameter passing: Any value passed to the function will be reclaimed when it returns, as the formal parameters gets out of scope; only returned values can survive.

tuples allow to return more

```
struct Dummy { a: i32, b: i32 }
fn foo() {
    let mut res = Box::new(Dummy {
        a: 0,
        b: 0
});
take(res);
println!("res.a = {}", res.a); \\ compilation error
}
fn take(arg: Box<Dummy>) {...}
```

When invoking `take(res)` the ownership of `Dummy` is moved from `res` to `arg`: when `take()` returns `arg` goes out of scope, so the resource gets freed automatically, making it no longer usable in `println`: this result in a **compilation error**. To use again the resource, we would have to make take return it, i.e. `res = take(res)`.

This looks rather limiting, but allows to completely avoid the *Double-free* problem: memory is freed automatically when the owner goes out of scope, and by rule O2, each value has only one owner.

Rust does not allow explicit memory allocation

19.3.2 Borrowing

Since Ownership rules in some case may be too restrictive, **borrowing** is introduced: a resource can be *borrowed* from its owner via assignment or parameter passing. To guarantee memory safety, borrowing rules ensure that *aliasing*[2] and *mutability cannot* ***coexist***.
Values can be passed

1. by immutable reference ⟶ `x = &y`
2. by mutable reference ⟶ `x = &mut y`
3. or by value ⟶ `x = y`

[2]Both the owner and the borrower can access the resource. More generally indicates that there are multiple ways to access a resource on the heap.

Borrowing

About mutable and immutable references:

O1 - At most **one** *mutable* reference to a resource can exist at any time

O2 - If there is a *mutable* reference, **no** *immutable* references can exist

O3 - If there is **no** *mutable* reference,**several** *immutable* references to the same resource can exist

During borrowing, ownership is reduced or suspended:

O4 - Owner *cannot* free or **mutate** its resource while it is *immutably borrowed*

O5 - Owner *cannot* even **read** its resource while it is *mutably borrowed*

19.3.3 Strings

String types

1. `String` does not require to know the length at compilation time, thus allocated on the *heap*.
2. `&str` size must be known statically, allocated on the *stack*.

Method `String::from()` allocates memory on the heap: it takes an argument of type `&str` and returns a `String`.

A String object has three components:

1. a reference to the heap location containing the character sequence
2. capacity (unsigned integer)
3. length (unsigned integer)

`String` does not implement `Copy`, thus assignment is subject to move semantics; assignment creates a copy of length, capacity and reference, but not of the char sequence in the heap.

19.3.4 Lifetime

A **lifetime** is a construct that the borrow checker uses to ensure the validity of the *borrowing rules* 19.3.2. Lifetimes are associated with each individual ownership and borrowing: a lifetime *begins* when the ownership starts, and *ends* when it is moved / destroyed, while for borrowings, it ends where the borrowed value is accessed the last time.

Lifetimes are mostly *inferred*, but sometimes they must be made explicit using the same syntax of generics. Using lifetimes, the compiler checks the validity of the rules of ownership and borrowing in the expected way; in particular, it ensures that –the *owner* of– every borrowed variable/reference has a lifetime that is longer than the borrower [B4,B5].

Borrowed (reference) formal parameters (arguments, return value) of a function have a lifetime, and if borrowed values are returned, each *must* have a lifetime.
The compiled tries to infer output lifetimes according to the following rules, but when not sufficient explicit lifetimes are necessary:

Lifetime

R1 - The lifetimes of the borrowed paramers are, by default, all **distinct**

R2 - If there is exactly **one input** lifetime, it will be assigned to **each output** lifetime

R3 - If a method has **more than one input** lifetime, *but* **one** of them is `&self` or `&mut self`, then this lifetime is assigned to **all output** lifetimes

```
fn longest(s1: &str, s2: &str) → &str { //does not compile
    if s1.len() > s2.len() { s1 }
    else { s2 }
}
```

Here the lifetime of the parameters depends on whether `s1` or `s2` is returned, so the compiler cannot infer the lifetime of the output parameters; hence, an **explicitly named lifetime** for input parameters is requires, as in the following snippet.

```
fn longest<'a>(s1: &'a str, s2: &'a str) → &'a str {
    if s1.len() > s2.len() { s1 }
    else { s2 }
```

19.4 More on Types

19.4.1 Enums

```
enum RetInt {
      Fail(u32),
      Succ(u32)
   }
fn foo_may_fail(arg: u32) -> RetInt {
      let fail = false;
      let errno: u32;
      let result: u32;
      ...
      if fail {
            RetInt::Fail(errno)
         } else {
            RetInt::Succ(result)
         }
   }
```

```
#[derive(Debug)] // needed to print
enum Tree<T> {
   Empty,
   Node(T, Box<Tree<T>>, Box<Tree<T>>)
}

fn main() {
   let tree = Tree::Node(
      42,
      Box::new(Tree::Node(
         0,
         Box::new(Tree::Empty),
         Box::new(Tree::Empty)
      )),
      Box::new(Tree::Empty));
   println!("{:?}", tree);
   //>Node(42, Node(0, Empty, Empty), Empty)
}
```

`println!("{:?}", tree);` indicates to print `tree` in *"debug mode"*.

19.4.2 Pattern Matching

```
   let x = 5; // try others...
   match x {
      1              => println!("one"),
      2              => println!("two"),
      3|4            => println!("three or four"),
      5..=10         => println!("five to ten"),
      e @ 11..=20    => println!("{}", e),
      i32::MIN..=0   => println!("less than zero"),
      21..           => println!("large"),
      _              => println!("???"),
   }
```

19.4.3 Classes

Rust is **not** *Object Oriented* and there is **no inheritance**, instead it pushes for composition over inheritance.

```
#[derive(Debug)]
struct Rectangle { // class
      width: u32, // instance variable
      height: u32,
   }
impl Rectangle { // methods
      fn area(&self) -> u32 { // first argument is this
            self.width * self.height
            // self.width = 20; // <- illegal, self is immutable
         }
   }
fn main() {
      let rect1 = Rectangle {
            width: 30,
            height: 50,
         };
      println!(
      "The area of the rectangle is {} square pixels.", rect1.area()
      );
   }
```

19.4.4 Traits

Traits are equivalent to *Type Classes* in Haskell and to *Concepts* in C++20, similar to Interfaces in Java. A trait can include *abstract* and *concrete* (default) **methods**, but <u>not</u> fields or variables. A struct can implement a trait providing an implementation for at least its abstract methods

```
impl <TraitName> for <StructName>{ ... }
```

The `#[derive]` clause can be used –if possible– to derive automatically an implementation of a trait.

Rust supports **bounded universal explicit polymorphism** with **generics**, as in Java, where bounds are one or more traits. Generic functions may have the generic type of parameter bound by one or more traits, so within such a function, the generic value can only be used through those traits, allowing for generic function to be **type-checked** when defined, as it happens in Java, unlike C++ templates.
• However, implementation of Rust generics uses *monomorphization* and is similar to typical implementation of C++ templates, where a separate copy of the code is generated for each instantiation, in contrast with the type erasure scheme of Java.

19.5 Smart Pointers

Smart pointers act as a pointer but with additional metadata and capabilities, and are typically structs, implementing `Deref` (`*`) and `Drop` (*reclaiming* when out of scope).

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
    }
```

`Box<T>` Allow to store a data of type T on the heap, with no performance overhead. Deref (*) returns the value. Optional by coercion.

TODO
What does *"Optional by coercion"* mean?

Rc`<T>` supports immutable access to resource with reference counting. Method `Rc::clone()` *doesn't clone!* It simply returns a new reference, incrementing the counter, whose value can be obtained by `Rc::strong_count`; when the counter is 0 the resource is reclaimed.

`RefCell<T>` supports shared access to a mutable resource through the **interior mutability** pattern. It provides two methods `borrow()` and `borrow_mut()` which return a smart pointer (`Ref<T>` or `RefMut<T>`). `RefCell<T>` keeps track of how many `Ref<T>` and `RefMut<T>` are active, and *panics* if the ownership/borrowing rules are invalidated. Its implementation is single-threaded, and it is typically used with `Rc<T>` to allow multiple accesses.

Type	Sharable?	Mutable?	Thread Safe?
&	yes *	no	no
&mut	no *	yes	no
Box	no	yes	no
Rc	yes	no	no
Arc	yes	no	yes
RefCell	yes **	yes	no
Mutex	yes, in Arc	yes	yes

* but doesn't own contents, so lifetime restrictions.
** while there is no mutable borrow

Figure 19.1: Pointers comparison

19.6 Functional elements

Closures can capture non-local variables in three ways, corresponding to ownership, mutable and immutable borrowing; this is reflected in the trait they implement: `FnOnce`, `FnMut` and `Fn`. The trait implemented is inferred. With move before || FnOnce is enforced.

```
let greater_than_x = |y| y > x; // Parameters within ||
let vector = vec![1, 2, 3, 4, 5]; // stream-like
vector.iter()
    .map(|x| x + 1)
    .filter(|x| x % 2 == 0)
    .for_each(|x| println!("{}", x));
```

19.6.1 Race conditions

TODO Listen to lecture

19.6.2 Sync and Send

Send and Sync are two strongly related traits regarding multithreading: an error is signaled by the compiler if the ownership of a value *not* implementing Send is passed to another thread; for a value to be referenced by multiple threads, it has to implement Sync.

$$\mathtt{T}\textit{implements}\mathtt{Send} \Leftrightarrow \mathtt{\&T}\textit{implements}\mathtt{Sync} \tag{19.1}$$

Examples

- Arc<T> is the thread-safe version of Rc<T> which implements Send and Sync
- Mutex<T> supports mutual exclusive access to a value via a lock. It is both Send and Sync, and typically wrapped in Arc