AP_Notes

Riccardo Paoletti

January 2020

Premises

This document was made taking infos from slides provided by prof. Andrea Corradini(`andrea@di.unipi.it`) at `http://pages.di.unipi.it/corradini/Didattica/AP-19/`. They can contains errors and typos, due to the fact that everyone can make a mistake. Hope you find them useful.

Contents

1 Languages and Abstract Machines. Compilation and interpretation schemes.

A **programming language** is defined by

- **sintax**, the form of a program. How expressions, commands, declarations, and other constructs must be arranged to make a well-formed program;
- **semantics**, the meaning of (well-formed) programs. How a program may be expected to behave when executed;
- **pragmatics**, the way a programming language is intended to be used in practice.

The **Sintax** is used by the compiler for scanning and parsing.
The **Semantics** is usually described precisely, but informally, in natural language.
The **Pragmatics** includes coding conventions, guidelines and also the description of the supported programming paradigms(A **Paradigm** is a style of programming, characterized by particular selection of key concepts and abstractions).

To implement a programming language **L**, programs written in **L** must be executable. Every language **L** implicitly define an abstract machine $M_{\mathbf{L}}$(a collection of data structures and algorithms which can perform the storage and execution of programs written in **L**) having **L** as a machine language. Implementing $M_{\mathbf{L}}$ on an existing host machine $M_{\mathbf{O}}$(via compilation, interpretation or both) makes programs written in **L** executable.

Viceversa, each abstract machine **M** defines a language $L_{\mathbf{M}}$ including all programs(particular data on which the interpreter can act) which can be executed by the interpreter of **M**.

The components of **M** are realized using data structures and algorithms implemented in the machine language of $M_{\mathbf{O}}$, in two possible ways

- The interpreter of **M** coincides with the interpreter of $M_{\mathbf{O}}$, so **M** is an extension of $M_{\mathbf{O}}$;
- The interpreter of **M** is different from the interpreter of $M_{\mathbf{O}}$, so **M** is interpreted over $M_{\mathbf{O}}$.

This process can be iterated, leading to a hierarchy(ex. WebApp→WebService → Browser → Java → Java Bytecode → Operating System Machine → Firmware machine → Hardware machine).

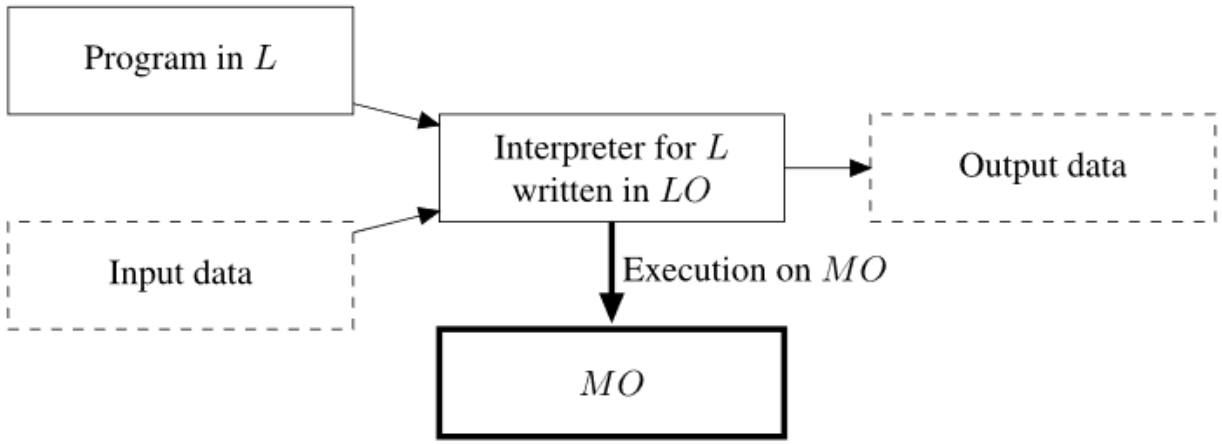

Figure 1: Pure Interpretation of a machine $M_{\mathbf{L}}$ over the host machine $M_{\mathbf{O}}$

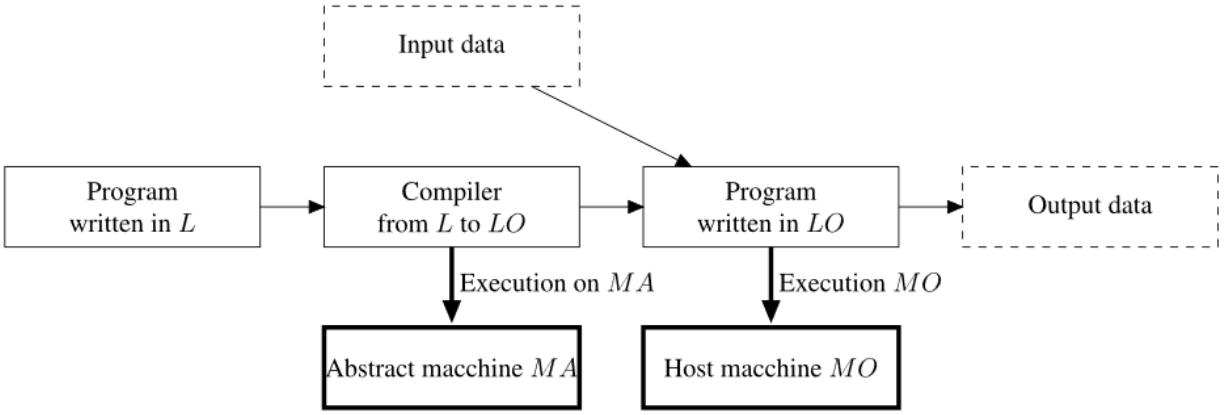

Figure 2: Program written in $\mathbf{L}$ is translated in equivalent program in $L_{\mathbf{O}}$ and then executed in $M_{\mathbf{O}}$ without even realize $M_{\mathbf{L}}$

1.1 Compilation and Interpretation

Compilation leads to better performance in general

- Allocation of variables without variable lookup at run time;
- Aggressive code optimization to exploit hardware features.

It is prohibitive to use because of the enormous size of the compiled program.

Interpretation facilitates debugging and testing

- Better diagnostic for programming problems;
- Procedures can be invoked at command line by user;
- Variables can be inspected and modified by user.

All implementations of programming languages use both. Can be modeled identifying an intermediate abstract machine $M_{\mathbf{I}}$ with language $L_{\mathbf{I}}$. A program in $\mathbf{L}$ is compiled to a program in $L_{\mathbf{I}}$ and then it is executed by an interpreter for $M_{\mathbf{I}}$.

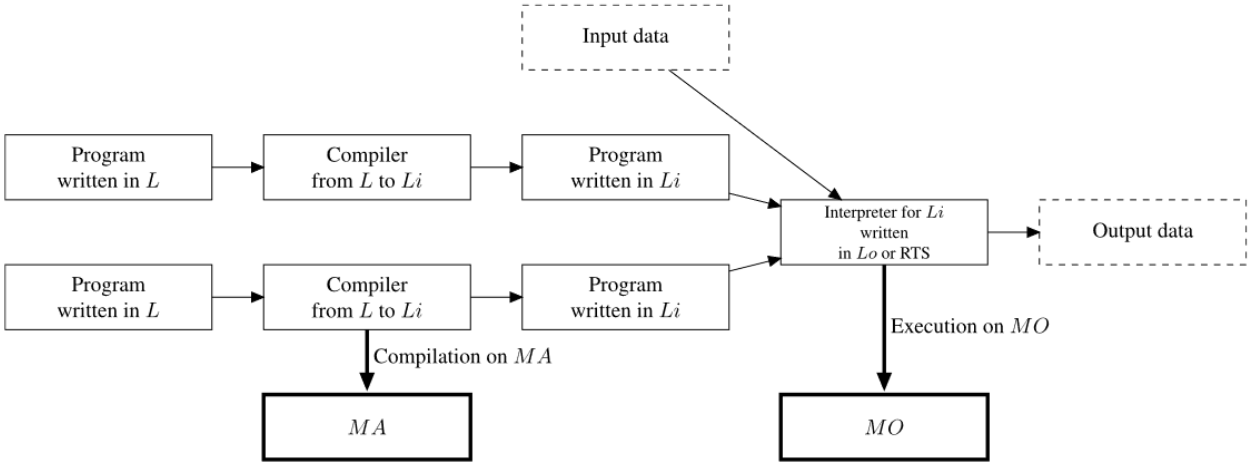

Figure 3: Final Scheme

1.2 Abstract Machines

Several languages adopt as intermediate machine a **Virtual Machine** (Pascal, Java...).

The compiler generates the intermediate program, then the virtual machine interprets the intermediate program.

Examples of this are

- **Microsoft** compilers generates **CLI** code (Common Intermediate Language;
- **C, Fortran, Haskell** compilers produces **LLVM IR** (Intermediate representation);
- **Java** compiler produces **Java Bytecode**.

This leads to many benefits like **portability**(ex. compile JAVA source code and execute bytecode on any platform with JVM) and **interoperability**(ex. for any language L, just provide a compiler to JVM bytecode. Then it could exploit JAVA libraries).

1.3 Other compilation Schemes

Other compilation schemes are composed of **pure compilation** and **static linking**(ex. Fortran), where library routines are separately linked with the object code of the program.

Using **pure compilation** with **static linking**, we can produce a **binary standalone executable** containing static pre-compiled libraries, generally available in Operating System.

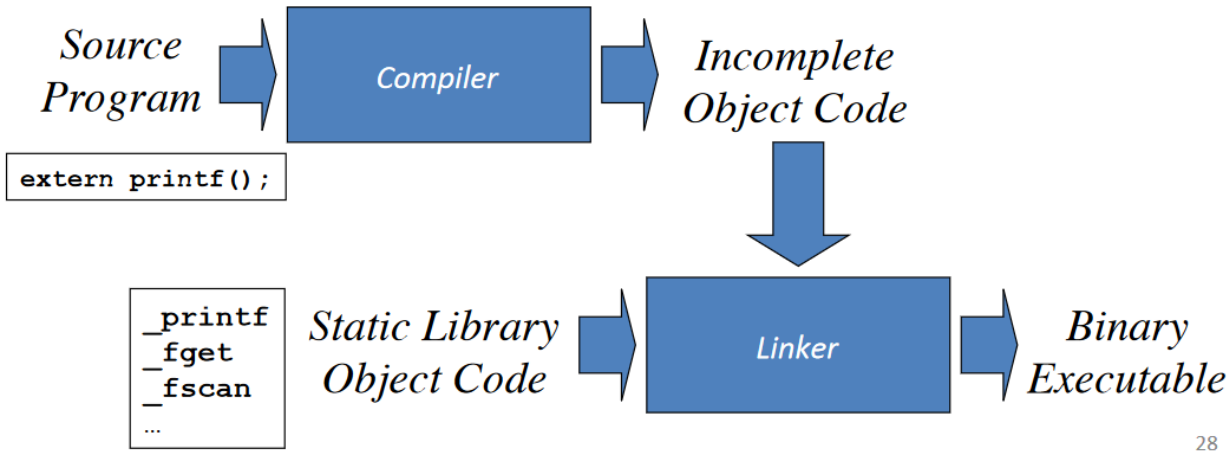

Figure 4: Pure Compilation and Static Linking

Compilation, Assembly and Static Linking

This schema facilitates the debugging of the compiler.

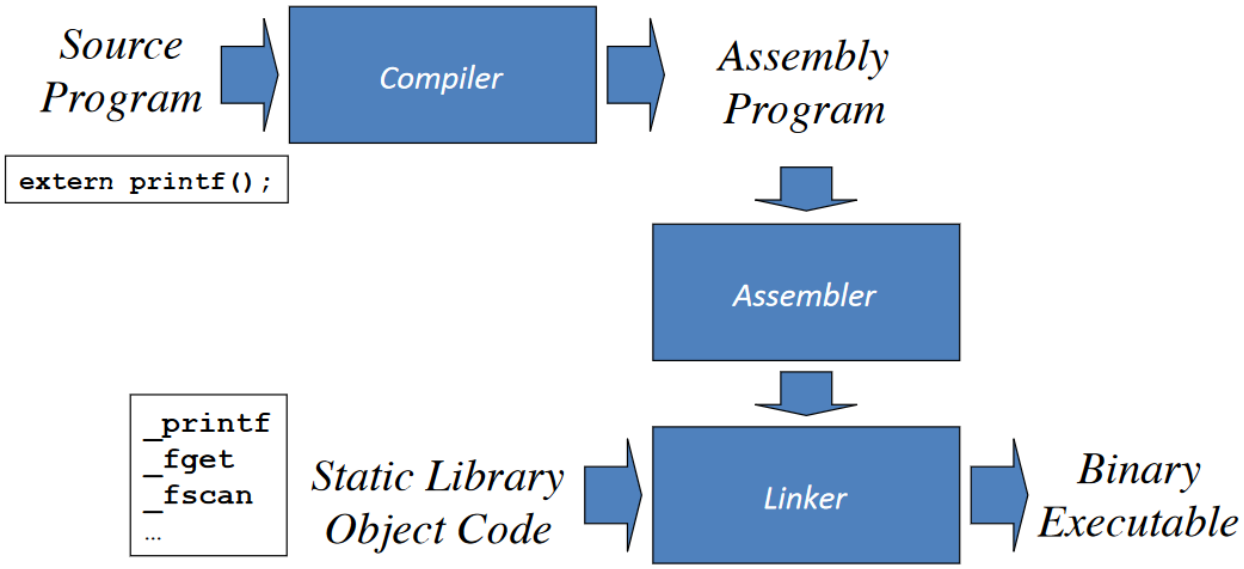

Figure 5: Compilation, Assembly and Static Linking

Compilation, Assembly and Dynamic Linking

Dynamic Libraries(DDL, .so, .dylib) are linked at run-time by the OS(becoming a stub in the executable).

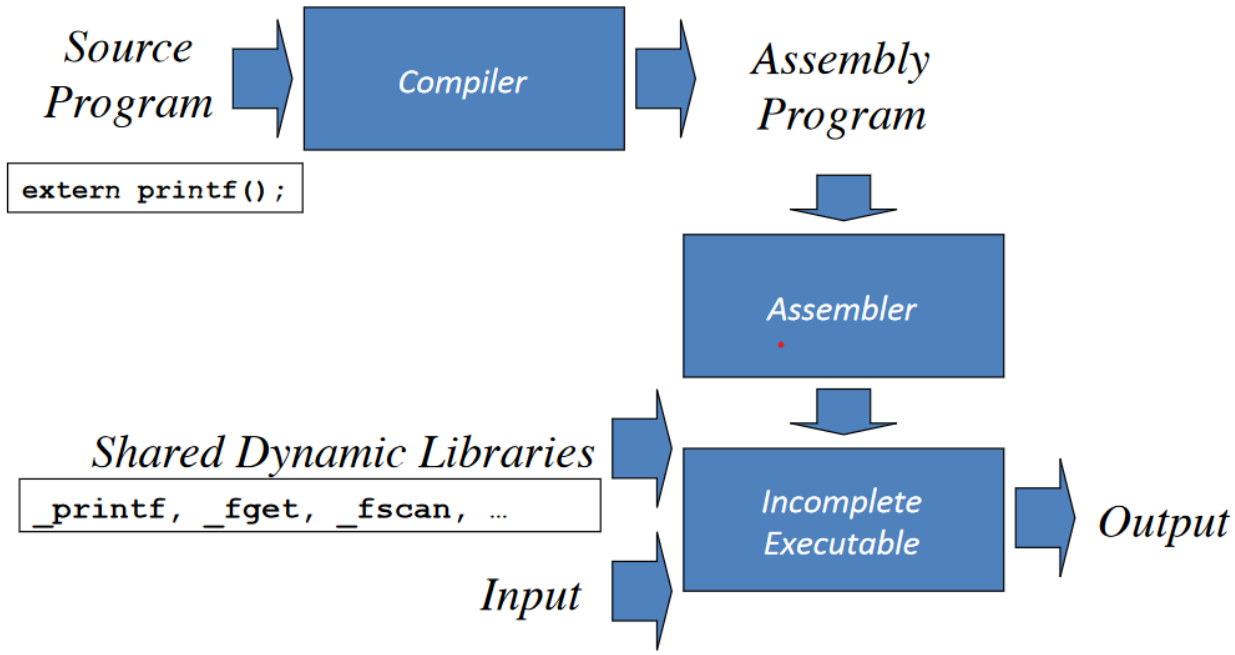

Figure 6: Compilation, Assembly and Dynamic Linking

Preprocessing

Most C and C++ compilers use **preprocessor** to import header files and expand macros.

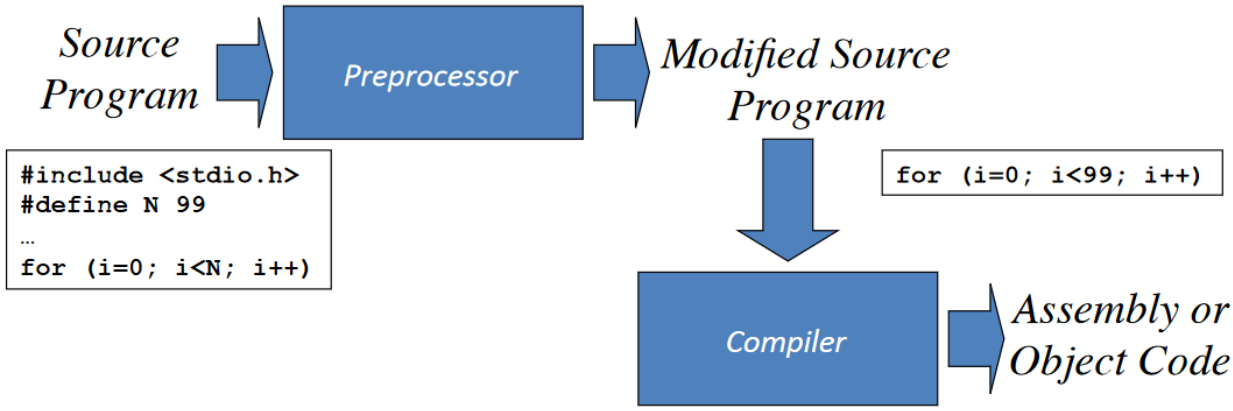

Figure 7: Preprocessing

2 Runtime System and Java Virtual Machine

2.1 Runtime System

Every programming language defines an **execution model**. A **runtime system** implements such execution model, providing support during the execution of corresponding programs. **Runtime Support** is needed by both interpreted and compiled programs, even if less by the latter.

The **runtime system** can be made of

- Code generated by the compiler;
- Code executed in other threads during program execution(ex. Garbage Collector);
- Language libraries;
- Operating System Functionalities;
- The interpreter/virtual machine itself.

The **runtime system** also is needed for

- Memory management, stack pop/push, heap allocation/free;
- Input/Output
- Interaction with runtime environment, variables/active entities etc...
- Parallel execution;
- Dynamic type checking and binding;
- Dynamic loading and linking of modules;
- Debugging;
- Code generation and Optimization;
- Verification and monitoring.

2.2 Java Runtime Environment

It includes all what is needed to run compiled Java programs(JVM and Java Class Library). Let's focus on the JVM cause it provides most of the functionalities of the language.

2.3 Java Virtual Machine

The **JVM** is an **abstract machine** in the true sense of the word.

The **JVM specification** does not give implementation details like memory layout of runtime data area, garbage-collection algorithm, internal optimization(can be dependent on target OS/platform, performance requirements, etc.).

The **JVM specification** defines a **machine independent *class file format*** that all JVM implementations must support.
The JVM imposes strong syntactic and structural constraints on the code in a class file. Any language with functionality that can be expressed in terms of a valid class file can be hosted by the JVM.

The **JVM** is a *multi-threaded stack based machine*. Its **instructions** implicitly take arguments from the top of the **operand stack** and then store there the results.

The **Operand Stack** is used to pass to and return argument from a method, store intermediate results and local variables.

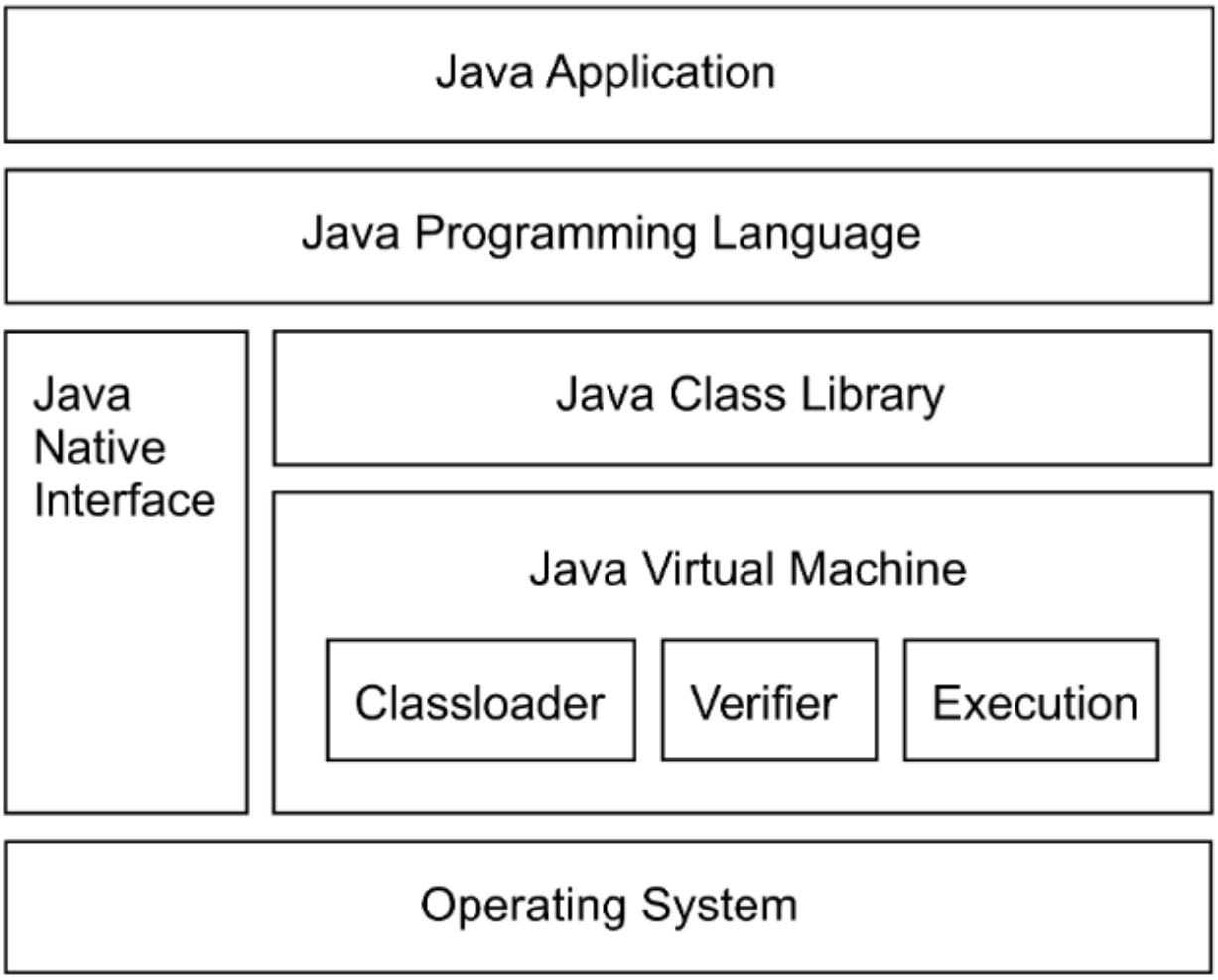

Figure 8: JVM Hierarchy

Java Classes are loaded by the JVM from files with extension .class that are the result of the compilation by Java compiler.

JVM data types can be **primitive**(byte, short, int, long, char, float, double, boolean, returnAddress) or **reference**(class types, array, interface). No type information on local variable are given at runtime. The type of operands are specified by **opcodes**(ex. iadd, fadd, ...).

Object representation is left to implementation(including *null*). This adds an extra level of indirection, so that you need **pointers** to instance data and class data in order to make garbage collection easier. It must also include **mutex lock implementation** and **Garbage Collector state flags**.

JVM Runtime Data Areas

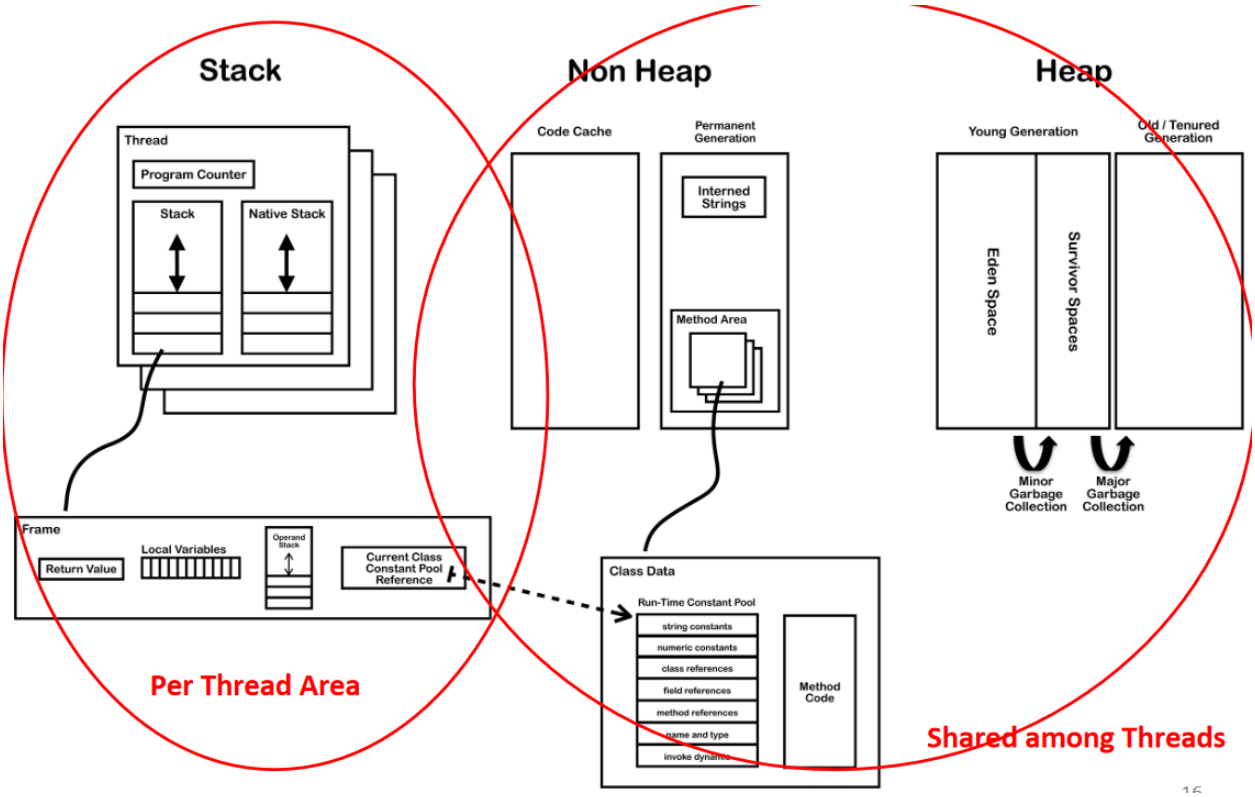

Figure 9: JVM Runtime Data Areas

2.3.1 Dynamic Linking

In C/C++ typically multiple objects are linked together to produce an executable or *dll*. During the linking phase, **symbolic references** are replaced with an actual **memory addressed** relative to the final executable.

In Java this linking phase is done **dynamically** at runtime. When a Java class is compiled , all references to variables and methods are stored in the **class' constant pool** as symbolic reference.

The JVM implementation can choose when to resolve the symbolic reference(**Eager, or static resolution**, when the class file is verified after being loaded, or **Lazy, or late resolution**, when the symbolic reference is first time used). The JVM has to behave as if the resolution is lazy and throw any error at this point.

Binding is the process in which the **entity**(field, method or class), identified by the symbolic reference, is replaced by the direct reference, and happens only once with all the references. If the reference belongs to a class that has not been resolved, it will be loaded.

2.3.2 Threads

JVM allows **multiple threads** per application(*main* included). Once created, a thread is invoked with ***start()***, which invokes the ***run()*** method of the thread instance.

Several threads are started by the system for Garbage Collection, signal dispatching, compilation etc... Threads can be supported by time slicing and/or multiple processors.

They have **shared access** to **heap** and **persistent memory**.

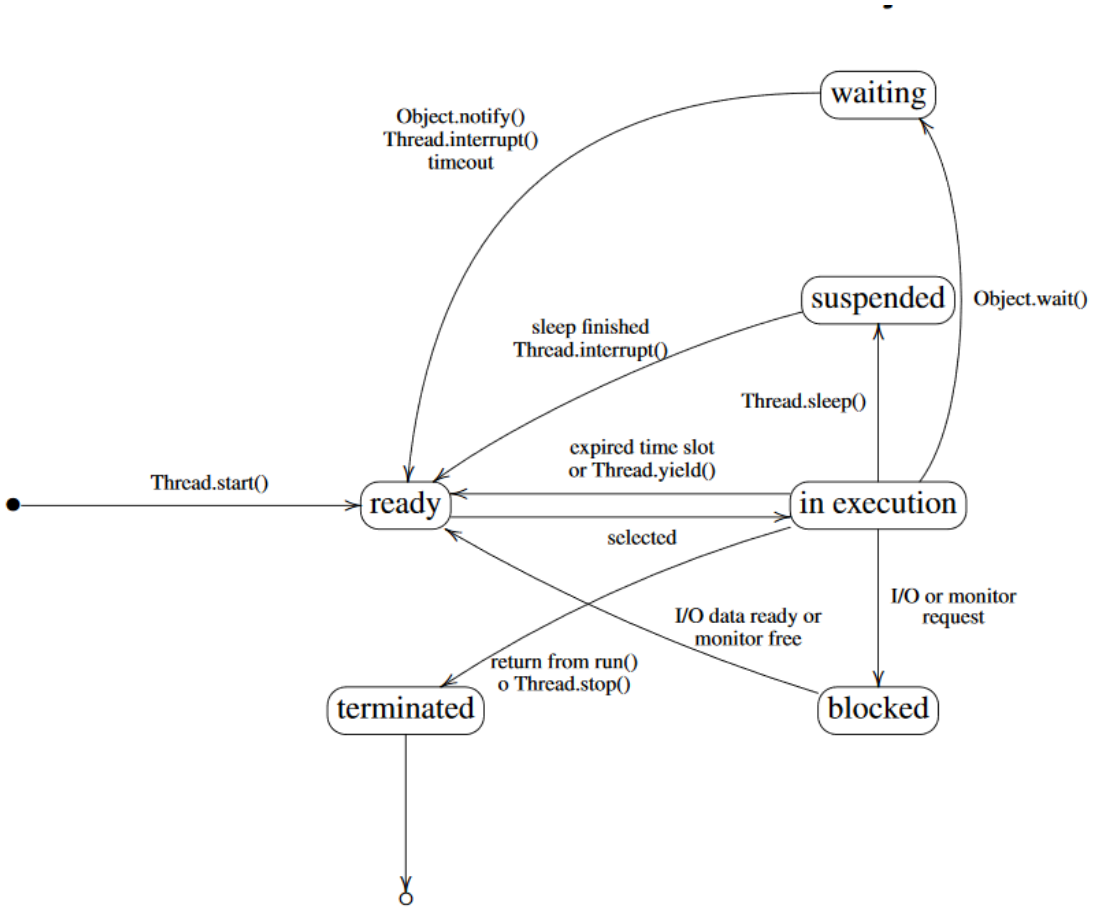

Figure 10: Thread Life Cycle

2.3.3 Per Thread Data Areas

Per Thread Data Areas are

- **PC**, program counter. Pointer to next instruction in *method area*(undefined if current method is inactive);
- **Java Stack**, a stack of frames(or *activation records*). A new frame is created every time a method is invoked, and it's destroyed when the method completes;
- **Native Stack**, used for invocation of native functions, through the **Java Native Interface**(ex. C functions).

Frames

A **Frame** has the following structure

- **Local Variable Array**, 32 bits. Contains reference to *this*(if instance method), the method parameters and the local variables;
- **Operand Stack**, for evaluation of methods and expressions;

- **Reference to Constant Pool**, of current class. It helps to support dynamic linking. It contains constants and symbolic references used for dynamic binding. They are suitably tagged
  - **Numeric Literals**(Integer, Float, Long, Double);
  - **String Literals**(Utf8);
  - **Class References**(Class);
  - **Field References**(Fieldref);
  - **Method References**(Methodref, InterfaceMethodref, MethodHandle);
  - **Signatures**(Name and Type).

  Operands in bytecodes often are indexes in the constant pool. This reference is used to construct the **runtime constant pool** upon class creation. All references in the runtime constant pool are initially symbolic, and are derived from the *.class* file. Class names are those returned by *Class.getName()*. Field and Method references are made of name, descriptor and class name.

2.3.4 Shared Among Threads Data Areas

Heap

It is shared among threads and

- **Contains memory for objects and arrays**;
- **Free only by Garbage Collector**;

Non-Heap

It is shared among threads and contains

- **Memory for objects which are never freed**, needed for JVM execution;
- **Method Area**, the memory where class files are loaded.
  For each **Class**
  - **ClassLoader Reference**;
  - **Runtime Constant Pool**;
  - **Field Data**(Name, Type, Modifiers, Attributes);
  - **Method Data**. For each Method
    - **Name**;
    - **Return Type**;
    - **Parameter Types**;
    - **Modifiers**;
    - **Attributes**;
    - **Method Descriptor**, that contains a sequence of zero or more parameter descriptors, in brackets, and a return descriptor, or V for void(ex. *(IDLjava/lang/Thread;)Ljava/lang/Object;* for *Object m(int i, double d, Thread t)* {...}).
  - **Method Code**. For each Method
    - **Bytecodes**;
    - **Operand Stack Size**;

- **Local Variable Size**;
- **Local Variable Table**;
- **Exception Table**;
- **Line Number Table**(which line of source code corresponds to which bytecode instruction).

For each **Exception Handler**(try/catch/finally clause)

- **Start Point**;
- **End Point**;
- **PC offset** for handler code;
- **Constant pool index** for exception class being caught.

NB: Method Area is shared among threads, it has to be thread-safe;

- **Interned Strings**;
- **Code chace for JIT**(Java Just-In-Time compiler). During code interpretation, when it finds code areas that are executed regularly, it compiles them to native code and stores them in the code cache.

Class File Structure

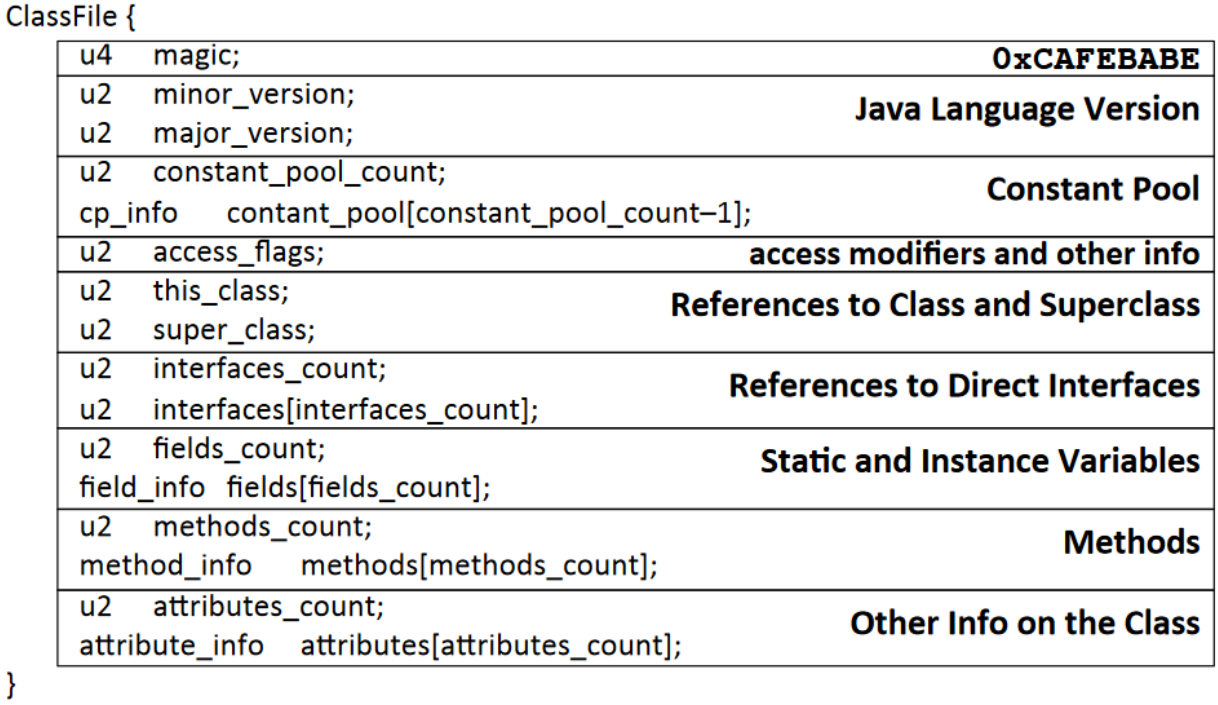

Figure 11: Class File Structure

FieldType term	Type	Interpretation
`B`	`byte`	signed byte
`C`	`char`	Unicode character code point in the Basic Multilingual Plane, encoded with UTF-16
`D`	`double`	double-precision floating-point value
`F`	`float`	single-precision floating-point value
`I`	`int`	integer
`J`	`long`	long integer
`L` *ClassName* `;`	`reference`	an instance of class *ClassName*
`S`	`short`	signed short
`Z`	`boolean`	`true` or `false`
`[`	`reference`	one array dimension

Figure 12: Field Type descriptors

Loading, Linking, Initializing and Finalization

Loading: finding the binary representation of a class or interface type with a given name and creating a class or interface from it. Once a class referencing another class has triggered its creation, the JVM creates the **array classes** and check whether they are already loaded or not. If not, invoke the appropriate *loader.loadClass*. Each class is tagged with the initiating loader, and **loading constraint** are checked during loading to ensure that the same name denotes the same type in different loaders.

Linking: taking a class or interface and combining it into the run-time state of the Java Virtual Machine so that it can be executed. This phase is in three step

- **Verification**. It's mainly done during the load and link process, because we have no guarantee that the class file was generated by a Java compiler. Boost runtime performances. Steps
  - **Step 1** - when the class file is loaded, the file is properly formatted and all its data is recognized by the JVM;
  - **Step 2** - when the class file is linked, all checks that do not involve instructions are made. Typical checks are that *final* classes are not subclasses and *final* method are not overridden, that every class, except Object, has a superclass, and that all field references in the constant pool have valid names, valid classes and a valid type descriptor;
  - **Step 3** - still during linking, dataflow analysis on each method are performed. It has to ensure that at any given point in the program
    - operand stack has always same size and contains same types of objects;
    - no local variable is accessed unless it knows to contain an appropriate value;
    - methods are invoked with appropriate parameters;
    - fields are assigned with appropriate values;

    - all opcodes have appropriate type arguments on the operand stack and in the local variable;
    - a method must not throw more exceptions then it admits;
    - a method must end with a return value or throw instruction;
    - methods must not use a half of a two word value.
  - **Step 4** - the first time a method is actually invoked a virtual pass whose checking is done by the JVM instructions to ensure that the referenced method or field exist in the given class and that the currently executing method has access to the referenced method or field.
- **Preparation**, allocation of storage;
- **Resolution**, optional, resolve symbol references by loading referred classes/interfaces, otherwise, it is postponed till the class is used by an instruction.

Initialization: executing the class or interface initialization method $< clinit >$ to initialize class variables. Static initializers are executed, while the superclass need to be initialized prior. It happens on direct use of method, constructor or field. The initialization method for instances is $< init >$, and it is traduced with the instruction **invokespecial** that can be invoked only on uninitialized instances. When a static method is called, only the class declaring that method is initialized.

Finalization: invoked just before Garbage Collection, not well specified when or in which thread. Not automatic invocation of super's method.

The JVM starts up by loading an initial class using the **bootstrap classloader**. The class is linked and initialized, and the *main* method is invoked. This will trigger the loading, linking and initialization of additional classes and interfaces.

Class Loader Hierarchy is composed of

- **BootStrap ClassLoader**, loads basic Java APIs;
- **Extension ClassLoader**, loads classes from standard Java Extension APIs;
- **System ClassLoader**, default application class loader, loads application classes from the classpath;
- **User Defined ClassLoader**, can be used to load application classes
  - for runtime reloading;
  - from different sources or generated on the fly;
  - for supporting classes separation in groups as required by web servers.
- **ClassLoader hooks**: *findClass*(build a byte array), *defineClass*(turns an array of bytes into a class object) and *resolveClass*(links a class).

2.3.5 Practic Example: SimpleClass

Disassembling

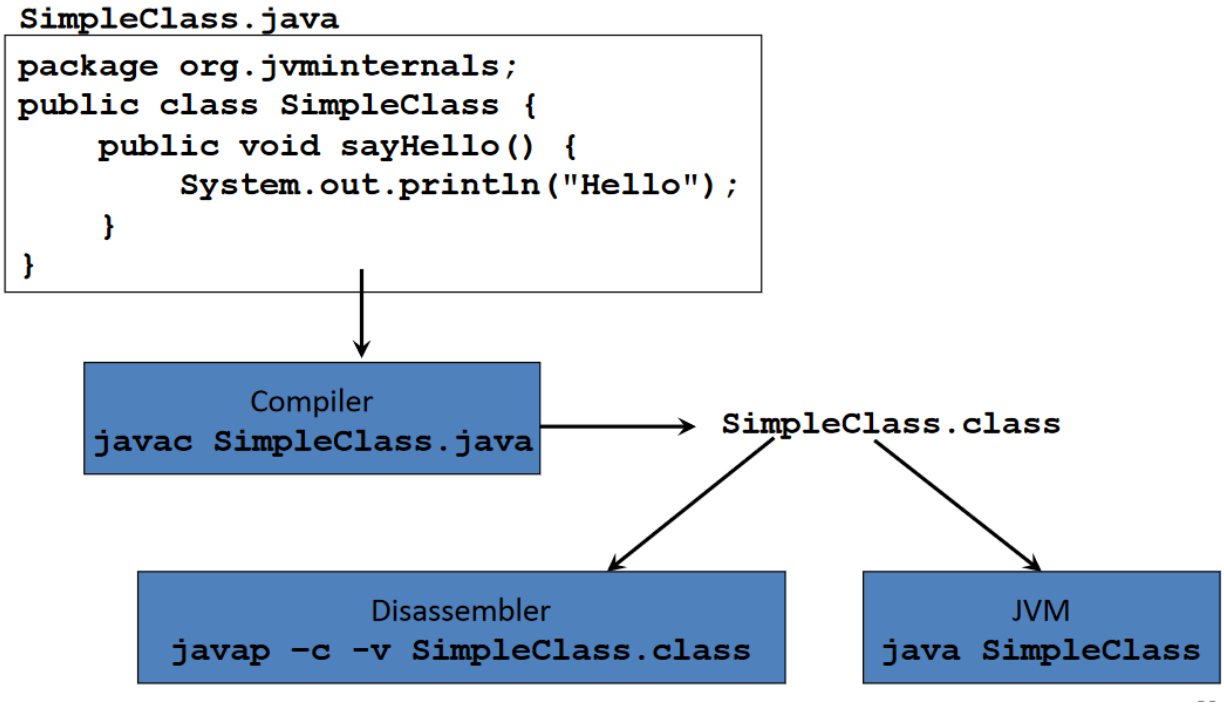

Figure 13: Disassembling SimpleClass

Constructor and Method

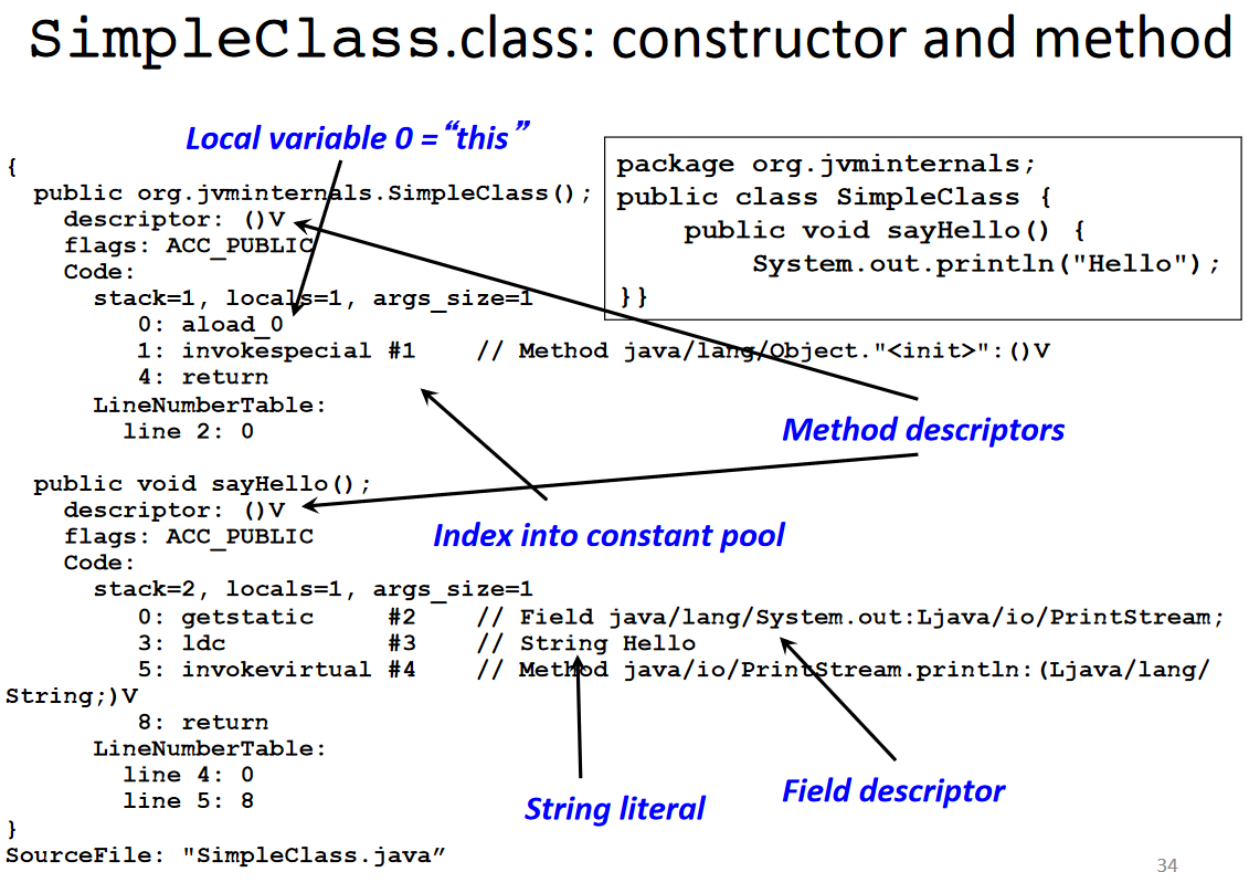

Figure 14: Constructor and Method

The Constant Pool

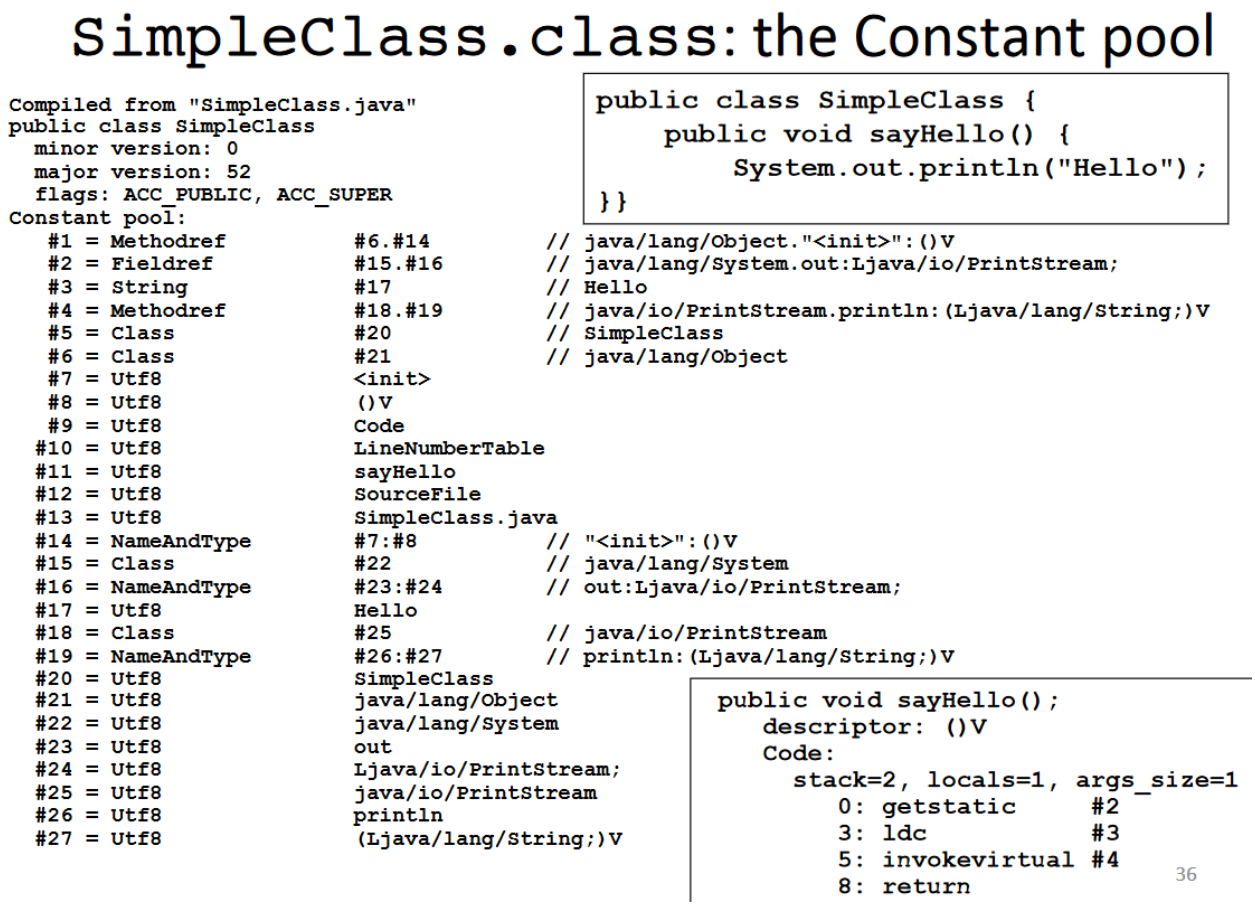

Figure 15: Constant Pool

Bytecode Execution

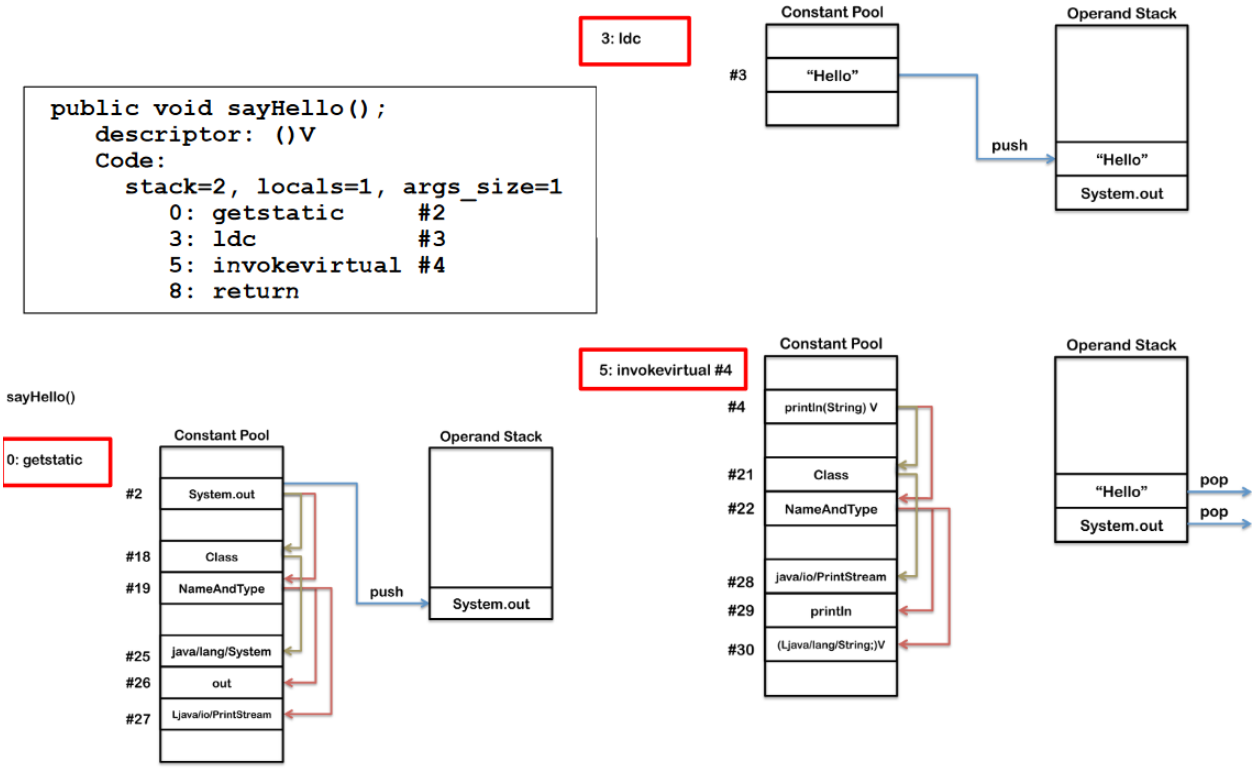

Figure 16: Bytecode Execution

2.4 The JVM instruction set

The JVM interpreter performs a classic fetch/execute loop where it

- automatically calculates **PC** and **fetches** the opcode at PC;
- if there are any operands it fetches them also;
- **executes** the action for the opcode;
- **repeat** until there's no more to do.

A **Java Virtual Machine instruction** consists of an opcode specifying the operation to be performed, followed by zero or more operands embodying values to be operated upon.

Different types of operation

- **Load** and **Store**($operandstack <-> localvars$);
- **Arithmetic**;
- **Type conversion**;
- **Object creation** and **manipulation**;
- **Operand stack manipulation**;
- **Control transfer**;
- **Method invocation** and **return**;

- **Monitor entry/exit**.

Each instruction has **different forms** supporting different kinds of operands. For example instruction *iload_0* pushes the local variable 0 to the top of the operand stack, but *iload_1* pushes local variable 1 and so on...

The **Runtime Memory** is composed by

- **Local Variable Array** in the frame;
- **Operand stack** in the frame;
- **Object fields** in the heap;
- **Static fields** in the method area.

Each JVM stack instruction implicitly takes arguments from the top of the operand stack of current frame and store the results of the operation to the top of the stack also.

The **Operand Stack** is used also to store intermediate results of calculations and to store the local variables.

There are three addressing modes in the JVM

- **Immediate** - with constants that are part of the instruction;
- **Indexed** - accessing variables from local variable array;
- **Stack** - retrieving values from the operand stack using *pop*.

The JVM instructions are also implicitly typed, so there's a different opcode for every different type of instruction (ex. *iload*, *lload*, *fload*, *dload*, *aload*)

A ***load*** instruction takes something from the args/locals area and pushes it onto the top of the operand stack. A ***store*** instruction pops something from the operand stack and places it in the args/locals area.

Since opcodes are bytes, there could be only 256 of them so it's impossible to have one opcode for each type for each instruction. Type support for instruction has to be carefully chosen, and types that are not supported have to be converted. For this reason there's almost no support for *byte*, *char* and *short* and it must be used *int* as computational type.

Actual type	Computational type	Category
`boolean`	`int`	1
`byte`	`int`	1
`char`	`int`	1
`short`	`int`	1
`int`	`int`	1
`float`	`float`	1
`reference`	`reference`	1
`returnAddress`	`returnAddress`	1
`long`	`long`	2
`double`	`double`	2

Figure 17: Actual and Computational types in the Java Virtual Machine

invokevirtual is used to invoke a function. It 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***...

return just passes the control to the caller.

There are also other kinds of method invocation

- ***invokestatic*** - for calling static methods. Arguments are copied from local variable 0;
- ***invokespecial*** - for calling **constructors**, **private methods** and **superclass methods**. *this* is always passed;
- ***invokeinterface*** - same as *invokevirtual* but used when the method is declared in an interface;
- ***invokedynamic*** - supports dynamic typing.

Objects are manipulated essentially like data of primitive types, but through references using the corresponding instructions(ex. *areturn*).

putfield and ***getfield*** are used to manipulate instance variables using the offset of the field in the class to access the field in *this*.

newarray command allows to create a new array of the type specified as argument and with dimension taken from the top of the operand stack. It is used like it is a references, hence using *a* as a prefix for the instructions *store* and *load*.

Compilation of switches can be made in two ways

- Using ***tableswitch*** - used for simple switches when the cases are not sparse;
- Using ***lookupswitch*** - used when the cases are sparse. Each case is a pair $< value : address >$, instead of an offset int the table of addresses. Cases are sorted, so binary search can be used.

NB: only switches on integer values are supported, so for the other types convertions are needed.

dup instruction copy the first element of the operand stack and pushes it. ***dup2_x1*** duplicates the long on the top of the stack and inserts it into the operand stack below the original value.

athrow is used to throw exceptions. It looks in the method for a catch block for the thrown exception using the **exception table**. If it exists, stack is cleared and control passes to the first instruction, otherwise the current frame is discarded and the same exception is thrown on the caller. If no method catches the exceptions, the thread is aborted.

try/catch compilation is like any other method. The difference is in the exception table that records the boundaries of *try* and they are used by *athrow* to dispatch the control.

Other instructions

- Handling synchronization: ***monitorenter***, ***monitorexit***;
- Verifying instances: ***instanceof***;
- Checking a cast operation: ***checkcast***;
- No operation: ***nop***.

Limitations

- max number of entries in **constant pool**: **65535**;
- max number of **fields**, **methods** and **direct superinterfaces**: **65535**;
- max number of **local variables** in the local variables array of a frame: **65535**;

- max **operand stack** size: **65535**;
- max number of **parameter** of a method: 255;
- max length of fields and methods name: **65535** characters;
- max number of **dimensions** in an **array**: **255**;

```
System.out.println("Result = "+i);

getstatic    #3; // Field System.out:Ljava/io/PrintStream;
new          #4; // class StringBuffer
dup
invokespecial    #5; // StringBuffer."<init>":()V
ldc              #6; // String Result =
invokevirtual    #7; // StringBuffer.append:(LString;)LStringBuffer
iload_1
invokevirtual    #8; // StringBuffer.append:(I)LStringBuffer;
invokevirtual    #9; // StringBuffer.toString:()LString;
invokevirtual    #10;// PrintStream.println:(LString;)V
```

```
System.out.print("Result = ");
System.out.println(i);

getstatic       #3; //Field System.out:Ljava/io/PrintStream;
ldc             #4; //String Result =
invokevirtual   #5; //Method PrintStream.print:(LString;)V
getstatic       #3; //Field System.out:LPrintStream;
iload_1
invokevirtual   #6; //Method PrintStream.println:(I)V
```

Figure 18: Coding Example

3 Software Components

3.1 Introduction

Why **component-based software**? Reliability, Re-use, Scaling...Much better.

Component Software: composite system made of software components.

A **software component** is a unit of composition with **contractually specified interface** and **explicit context dependencies** only. It can be **deployed independently** and can be subject to **composition by third parties**. It is not accessible once deployed and it's used as a black box.

A **contract** is a specification attached to an interface that mutually binds the clients and providers of the components(functional and non-functional aspects).

Context dependencies - specification of the deployment environment and run-time environment.

Deployed independently means it is connected with other components possibly deployed by **third parties** via interfaces and **late binding** is performed, so that the dependencies are resolved at load or run-time.

Modules are **main features** of programming languages for supporting development of large applications. They can be seen also like **abstraction mechanisms**: collections of data with operations defined on them.

Several component-related concepts were already present in modules. **Modules** are parts of a **program**, **Components** are parts of a **system**. Modules have been replaced by classes in OO programming.

Component forms

- **Component Specification** - The specification of a unit of software that describes the behavior of a set of *Component Objects* and defines a unit of implementation. Behavior is defined as a set of *interfaces*. A *Component Specification* is realized as a *Component Implementation*;
- **Component Interface** - A definition of a set of behaviors that can be offered by a *Component Object*;
- **Component Implementation** - A realization of *Component Specification*, which is independently deployable. It can be installed and replaced independently of other components;
- **Installed Component** - An installed(or deployed) copy of a *Component Implementation*. A *Component Implementation* is deployed by registering it with the runtime environment. This enables the runtime environment to identify the *Installed Component* to use when creating an instance of the component, or when running one of its operations;
- **Component Object** - An instance of an *Installed Component*, with it's own data and unique identity. It performs the implemented behavior.

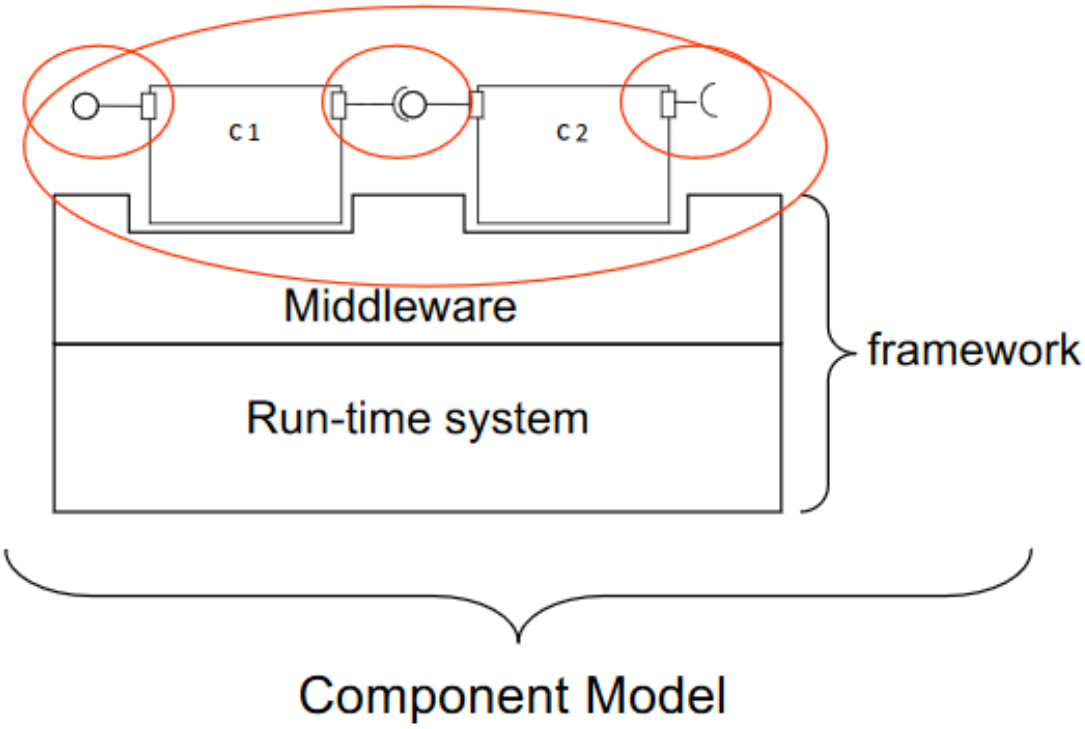

Figure 19: Component Based Software Engineering Schema

Components Final Panoramic

In all cases there is an **infrastructure** providing rich foundational functionality for the addressed domain.

Components can be purchased from **independent providers** and deployed by clients.

The components provide services that are substantial enough to make **duplication of their development** too difficult or **not cost-effective**.

Multiple components from different sources **can coexist** in the same installation.

Components exist on a **level of abstraction** where they directly mean something to the deploying client.

With Visual Basic, this is obvious – a **control** has a direct visual representation, displayable and editable properties, and has meaning that is closely attached to its appearance.

With **plugins**, the client gains some explicable, high-level feature and the plugin itself is a user-installed and configured component.

3.2 The SUN approach, Java Beans

A **Java Bean** is a **reusable software component** that can be **manipulated visually** in a **builder tool**. Any Java class can be recognized as a bean provided that

- Has a **public default constructor**(no arguments);
- Implements the interface ***java.io.Serializable***;
- Is in a **jar** file with *manifest file* containing **Java-Bean: True**;

Java beans are components. They can be assembled to build a new bean using glue code to wire beans together. Common features

- Support for **properties**, for customization and programmatic use;
- Support for **events**, can be used to connect several beans;
- Support for **customization**;
- Support for **persistence**, a bean can be customized in an application, saved and reloaded with the same state;
- Support for **introspection**, a builder tool can analyze how a bean works.

Java Beans have **simple properties**, with a type and possibly a getter and a setter. Properties can be identified by **Introspection**, a feature of java beans that allow to analyze them and gather infos like class type, name and methods.

Java Beans **design pattern**:

- from pair of methods
  - ***<PropertyType> get<PropertyName>***
  - ***set<PropertyName>(<PropertyType> a)***

  I can infer existence of property named ***<propertyName>*** of type ***<PropertyType>***.
- for events, from
  - ***add<EventListType>(<EventListType> a)***
  - ***remove<EventListType>(<EventListType> a)***

  I can infer that the object is source of the ***<EventListType>*** event. If ***add*** throws ***java.util.TooManyListenersException*** then it is an ***Unicast Event***.

A **Bound property** generates an event when the property is changed. The event is of type ***PropertyChangeEvent*** and is sent to objects that previously registered an interest in receiving such notifications.

A **Constrained property** can only change value if none of the registered observers *poses a veto*. The event is of type ***PropertyChangeEvent*** and is sent to objects that previously registered an interest in receiving such notifications. Those objects are able to veto the proposed change by raising an exception.

3.3 Java Reflection

Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution.

A system may support reflection at different levels: from simple information on types to reflecting the entire structure of the program.

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.

Both aspects require a mechanism for encoding execution state as data: providing such an encoding is called **reification**.

Structural reflection is concerned with the ability of the language to provide a complete reification of both the program currently executed as well as its abstract data types.

Behavioral reflection is concerned with the ability of the language to provide a complete reification of its own semantics and implementation(processor) as well as the data and implementation of the run-time system.

If it is possible to perform an operation without using reflection, then it is preferable to **avoid using it**, because Reflection brings

- **Performance Overhead** - Reflection involves types that are dynamically resolved, thus JVM optimizations can not be performed, and reflective operations have slower performance than their non-reflective counterparts;
- **Security Restrictions** - Reflection requires a runtime permission which may not be present when running under a security manager. This affects code which has to run in a restricted security context, such as in an Applet;
- **Exposure of Internals** - Reflective code may access internals (like private fields), thus it breaks abstractions and may change behavior with upgrades of the platform, destroying portability.

Java supports **Introspection** and **Reflexive Invocation**, but **not Code Modification**. To access infos about a particular type, Java maintains a special object ***java.lang.Class*** that can be accessed and read. **Class objects** are constructed automatically by the JVM as classes are loaded. They provide access to the information read from the class file.

Members in Java are **fields**, **methods** and **constructors**. For each member the **Java Reflection API** provides support to retrieve type and operations of the member. More in details

- **Member**: interface;
- **Field**: class. Fields have a type and a value;
- **Method**: class. Methods have return values, parameters and can throw exceptions;
- **Constructor**: class. Constructors are like methods but have no return values and the invocation of a constructor creates a new instance of an object for a given class.

Certain operations(changing private fields, etc...) are forbidden even if invoked through reflection. The programmer can allow these invocations defining fields and methods *accessible*. The class ***AccessibleObject*** provides methods to see if a field is accessible for this object and to make some fields accessible.

3.4 Java Annotations

From **Modifiers** to **Annotations**: Modifiers in Java are meta-data describing properties of program elements. They are **reserved keywords**(*static*, *final*, *public*...), thus wired-in in the language.

Annotations can be seen as user-definable modifiers. They are made of a name and a finite number of attributes, i.e. *name* = *value* pairs, possibly 0. *value* is an expression that can be evaluated at compile time, and types of declaration and calling have to be compatible.

Annotations can be applied to almost any syntactic element(package, class, interface, field, method, constructor, parameter, type, etc...), in any number.

The Java compiler defines a small set of **predefined annotations**

- ***@Override*** - Makes explicit the intention of the programmer that the declared method overrides a method defined in a superclass. The compiler can issue a warning if no method is overridden;
- ***@Deprecated*** - Declares that the annotated element is not necessarily included in future releases of the Java API. Typically applied to methods, but also to classes and interfaces;
- ***@SuppressWarnings*** - Instruct the compiler to avoid issuing warnings for the specified situations;
- ***@FunctionalInterface*** - Declares an interface to be functional.

User-defined annotations are ignored at compile time but can be used by other tools. Programmers can define new annotations(for documentation, for processing after compilation, for inspecting annotation placed at runtime, etc...) with a sintax similar to interfaces but with the ***@interface*** tag.

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

Figure 20: Annotation Declaration Example

Each **method** determines the **name** of an attribute and its **type**(the return type). A default value can be specified for each attribute(as for ver, revand changes). Attribute types can only be **primitive**(String, Class, Enum, Annotation, ...) or an array of those types. Additionally(like any interface) an ***@interface*** can contain constant declarations(with explicit initialization), internal classes and interfaces and enumerations but rarely used.

Annotation definitions can be annotated in turn, to describe their meta-data. Some predefined meta-annotations:

- ***@Target*** - Constrains the program elements to which the annotation can be applied. The value type is ***annotation.ElementType[]***, an **enum** including *ANNOTATION_TYPE*, *CONSTRUCTOR*, *FIELD*, *LOCAL_VARIABLE*, *METHOD*, *PACKAGE*, *PARAMETER*, *TYPE_PARAMETER* and *TYPE_USE*;

- ***@Retention*** - Till when should the annotation be present. Three options(values of enum **RetentionPolicy**): *SOURCE*, *CLASS*(default) and *RUNTIME*;
- ***@Inherited*** - Marker annotation. The annotation is inherited by subclasses.

Annotations in class files can be exploited by appropriate tools for **program analysis**. Package ***javax.annotation.processing*** provides a Java API for writing such tools. Retrieval of annotations at runtime occurs through the Reflection API.

3.5 Frameworks and Inversion of Control: Decoupling components; Dependency Injections; IoC Containers

Software Framework: a collection of common code providing a generic functionality that can be selectively overridden or specialized by user code providing a specific functionality.

Application Framework: a software framework used to implement the standard structure of an application for a specific development environment.

Examples

- **GUI Frameworks**
  - **MFC** - Microsoft foundation class library. C++ object-oriented library for Windows;
  - **Gnome** - Written in C. Mainly for Linux;
  - **Qt** - Cross-platform. Written in C++.
- **Web Frameworks** - Based on **Model-View-Controller** design pattern
  - **ASP.NET** - by Microsoft for web sites, web applications and web services;
  - **GWT** - Google Web Toolkit;
  - **Rails** - Written in Ruby. Provides default structures for databases, web services and web pages;
  - **Spring** - for Java-based enterprise web applications;
  - **Flask** - micro-framework in Python. Highly extensible.
- **Concurrency Frameworks**
  - **Hadoop Map/Reduce** - software framework for applications which process big amounts of data in-parallel on large clusters(thousands of nodes) in a fault-tolerant manner.
    - **Map** - Takes input data and converts it into a set of tuples(key/value pairs);
    - **Reduce** - Takes the output from *Map* and combines the data tuples into a smaller set of tuples.

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. The framework's code then **calls your code** at these points.

Inversion of control : as opposed to libraries is the code of the framework that calls the code of the user.

Component Frameworks : Frameworks that support development, deployment, composition and execution of components designed according to a given **Component Model**.

They support the development of **individual components**, enforcing the design of **precise interfaces** and the composition/connection of components according to the mechanisms provided by the **Component Model**.

They allows **instances** of these components to **be plugged** into the component framework itself and provide **prebuilt functionalities**, such as useful components or automated assembly functions that automatically instantiate and compose components to perform common tasks.

The **component framework** establishes **environmental conditions** for the component instances and regulates the **interaction** between component instances.

All frameworks can be **extended** to satisfy app-specific functionalities. Common ways of extension are **within the framework**(subclassing, interfaces, event handlers...) or through **plug-ins** to load extra code in a specific format.

3.5.1 Invertion of Control

With Frameworks the **Inversion of Control** becomes dominant. The **application architecture** is often **fixed**, even if customizable, and determined by the Framework.

When using a framework, one usually just implements a few callback functions or specializes a few classes, and then invokes a single method or procedure. The framework does the remaining work for you, invoking any necessary client callbacks or methods at the appropriate time and place.

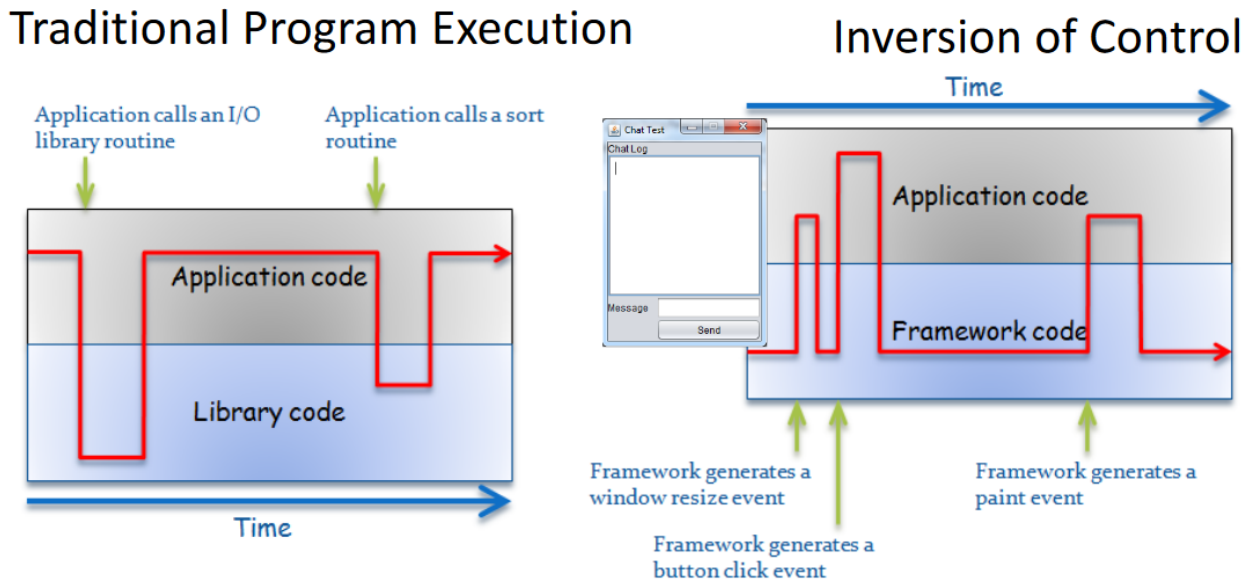

Figure 21: Invertion of Control

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

Good OO Systems should be organised as **network of interacting objects**, in order to achieve **high cohesion** and **low coupling**. Advantages of low coupling are **extensibility**, **testability** and **reusability**.

We discuss **Dependency Injection** and other techniques to achieve it.

Dependency injection

- **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**

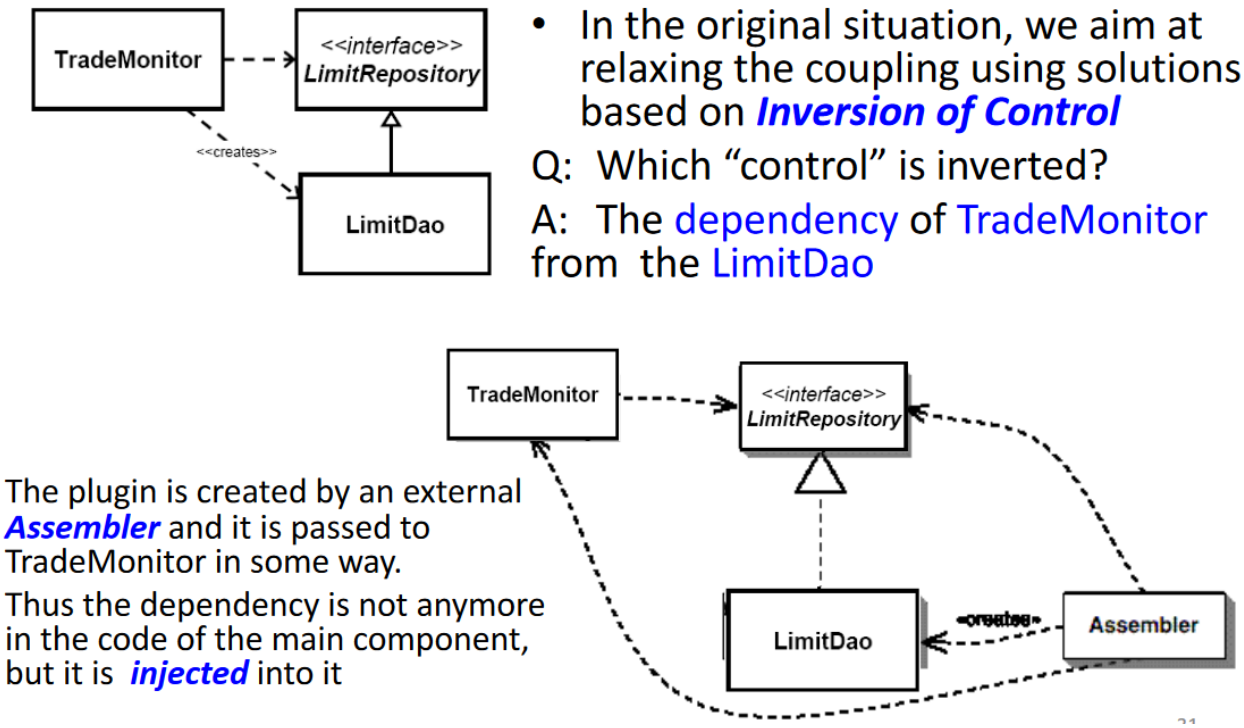

Figure 22: Dependency Injection Schema

Dependency injection allows avoiding hard-coded dependencies(strong coupling) and changing them. Allows selection among multiple implementations of a given dependency interface at run time.

Three forms

- **Setter injection** - based on **setter methods**. Idea: add a setter and leave creation and resolution to others. Widely used in Spring. It's very simple and often already available, but it's possible to create partially constructed objects or change dependencies at runtime;
- **Constructor injection** - use the **constructor**. Widely used in PicoContainer. Also this approach is simple and avoid partially constructed objects. Nevertheless, coding becomes much more difficult to write and to develop;
- **Interface injection**.

There are still some open issues like *"who creates the dependencies?"*, *"what if I have some initialization code to run after dependencies?"* and so on. **Ioc Containers** solve these issues. They have configuration, often external, create objects, ensure that all the dependencies are satisfied and provide life cycle support.

Other possible solutions are **Reflection**(can be used to determine dependencies, reducing the need for config files) and **auto-wiring**(automatic wiring between properties of a bean and other beans based, like name or type).

Example: Spring

The objects that form the backbone of a Spring application are called **beans**. A bean is an object that is instantiated, assembled, and otherwise managed by a **Spring IoC container** (***ApplicationContext***). Bean definition contains the information called **configuration metadata**, which is needed for the container to know the following

- How to **create** a bean;
- Bean's **life cycle** details;
- Bean's **dependencies**.

The configuration metadata can be supplied to the container in three possible ways:

- **XML** based configuration file(the standard);
- **Annotation**-based configuration;
- **Java**-based configuration.

The **Spring container** is at the core of the Spring Framework. it will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction.

The Spring container uses **Dependency Injection** to manage the components that make up an application. It gets its instructions on what objects to instantiate, configure, and assemble by reading the **configuration metadata** provided.

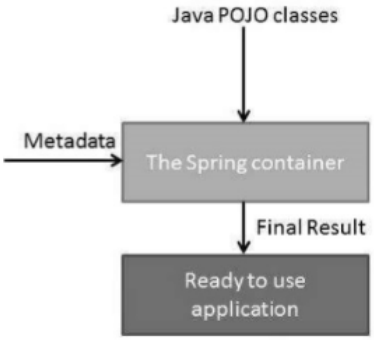

Figure 23: Spring Behavior Schema

The diagram in figure 23 represents a high-level view of how Spring works. The **Spring IoC container** makes use of **Java POJO classes** and **configuration metadata** to produce a **fully configured** and **executable system or application**.

4 Polymorphism

4.1 Classification

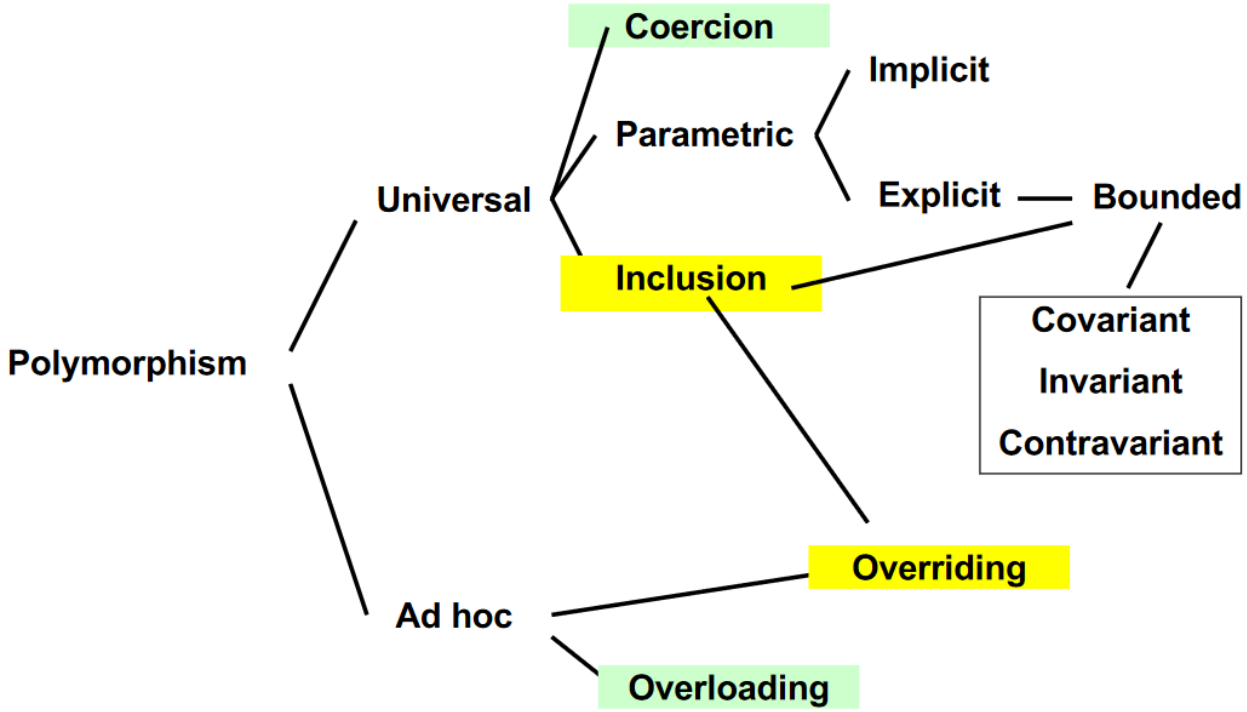

Figure 24: Polymorphism Classification

With **ad hoc** polymorphism the same function name denotes different algorithms, determined by the actual types.
With **universal** polymorphism there is only one algorithm: a single (universal) solution applies to different objects.

Ad hoc and universal polymorphism **can coexist**.

The **binding** of the function name with the actual code to execute can be

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

If it spans over different phases, the binding time is the last one.
The earlier the better, for debugging reasons.

Overloading(ad hoc polymorphism): function with the same name, but with different parameter types. They are inferred by the type of the parameters passed. Present in all languages, at least for built-in arithmetic operators: +, *, -, ... Sometimes supported for user defined functions(Java, C++, ...).

Coercion(Universal Polymorphism): automatic conversion of an object to a different type. Opposed to casting, which is explicit. Thus the same code is applied to arguments of different types.

Inclusion polymorphism: Also known as subtyping polymorphism, or just inheritance. Polymorphism ensured by (Barbara Liskov')**Substitution principle**: an object of a subtype (subclass) can be used in any context where an object of the supertype(superclass) is expected.

Overriding: [Java] A method m(...) of a class A can be redefined in a subclass B of A.

Dynamic binding: overriding introduces ad hoc polymorphism in the universal polymorphism of inheritance. Resolved at runtime by the lookup done by the invokevirtual operation of the JVM.

```
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
}
```

```
class A {

    public void onFoo() {}
    public void onFoo(int i) {}
}

class B extends A {

    public void onFoo(int i) {}
}

class C extends B {
}

class D {
public static void main(String[] s)
{
        C c = new C();
        c.onFoo();
                    //Compiles !!
    }                                15
}
```

Figure 25: Overloading + Overriding: C++ vs Java

[**Java**] overloading is type-checked by the compiler. Overriding is resolved at runtime by the lookup done by invokevirtual.

[**C++**] Dynamic method dispatch: C++ adds a v-table to each object from a class having virtual methods. The compiler does not see any declaration of *onFoo* in **C**, so it continues upwards in the hierarchy. When it checks **B**, it finds a declaration of void *onFoo(int i)*, so it stops lookup and tries overload resolution, but it fails due to the inconsistency in the arguments. *void onFoo(int i)* hides the definitions of *onFoo* in the superclass. Solution: add *using A::onFoo;* to class **B**.

4.2 Polymorphism in C++: inclusion polymorphism and templates

Difference between Java Generics

- **[C++] Templates**

  - Function and class templates. Type variables;
  - Each concrete instantiation produces a copy of the generic code, specialized for that type;
- **[Java] Generics**
  - Generic method and classes. Type variables;
  - Strongly type checked by the compiler;
  - Type erasure: type variable are *object* at runtime.

Function Templates in C++ support **parametric polimorphism**. Type parameter can also be primitive types(unlike Java). The **compiler/linker** automatically generates **one version for each parameter type** used by a program. Parameter types are inferred or indicated explicitly.

Macros can be used for polimorphism in simple cases. They are executed by the preprocessor, while templates are esecuted by the compiler. **Preprocessor** makes **only textual substitutions**, neither parsing or static analysis.

Problems with Macros are that the side effects are multiplied because of the copying made, and also recursion is not possible.

Template parameters can also include **expressions** of a particular type. In this case the value of the parameter is determined at compile time

A template can be **specialized(also partially)** defining another template with **same name** and **more specific parameters or no parameters**. This is similar to the Java Overriding. At compile time, the most specific applicable templates is chosen.

Implementation: a template is **compiled on demand**(at first instantiation, *Static Binding*). Compiler chooses the **best match templates** and creates a **template instance**. The **overloading resolution** is done after substitution and it fails if some operator is not defined for the type instance.

In C/C++ usually function **prototypes** are collected in a **header file(.h)** while the actual **definitions** are in a **separate file(.c/.cpp)**. In **template case**, compiler need both prototype and definition to instantiate it, so **compilation cannot be separated**. If the same template function is defined in different files that are compiled and then linked, there will be only one instantiation per type of template function.

4.3 Java Generics, Type bounds and subtyping, Subtyping and arrays in Java, Wildcards, Type erasure

Java Generics $->$ Universal, Parametric, Explicit Polymorphism

Classes, Interfaces and Methods can have **type parameters** that can be used arbitrarily in the definition. They can be instantiated providing arbitrary(reference) type arguments.

Methods can introduce their own type parameters, and when the method is invoked, all type parameters must be instantiated, explicitly or implicitly(type inference).

Type parameters can be bounded to be of a particular type(ex. *E extends Number*. Only classes that matches the bound can be used as type argument. There are several types of bound

- ***upper bound*** - *"TypeVar extends SuperType"*. SuperType and any of its subtypes are OK;
- ***multiple upper bounds*** - *"TypeVar extends ClassA & InterfaceB & InterfaceC & ..."*;
- ***lower bound*** - *"TypeVar super SubType"*. 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.

Some rules about Generics

- Given two concrete types **A** and **B**, **MyClass<A>** has no relationship to **MyClass<B>**, regardless of whether or not **A** and **B** are related;
- Formally: subtyping in Java is invariant for generic classes;
- **Note**: The common parent of **MyClass<A>** and **MyClass<B>** is **MyClass<?>**;
- On the other hand, as expected, if **A** extends **B** and they are generic classes, for each type **C** we have that **A<C>** extends **B<C>**.

Covariance means that the compatibility of two types implies the compatibility of the types dependent on them:

the compatibility of T1 to T2 implies the compatibility of A(T1) to A(T2) the type A(T) is covariant.

If the compatibility of T1 to T2 implies the compatibility of A(T2) to A(T1), then the type A(T) is contravariant.

If the compatibility of T1 between T2 does not imply any compatibility between A(T1) and A(T2), then A(T) is invariant.

Arrays

Let **Type1** be a subtype of **Type2**. According with Java rules, **Array¡Type1¿** and **Array¡Type2¿** are not related by subtyping.

But instead... in Java, if **Type1** is a subtype of **Type2**, then **Type1**[] is a subtype of **Type2**[], thus **Java arrays are covariant**.

Java(and also C#, .NET) fixed this rule before the introduction of generics.
Why? Think to *void sort(Object[] o)*. Without covariance, a new sort method is needed for each reference type different from Object, but sorting does not insert new objects in the array, thus it cannot cause type errors if used covariantly.

Problems with array covariance: Even if it works for sort, **covariance may cause type errors** in general.
Apple[] apples = new Apple[1];
Fruit[] fruits = apples; – ok, covariance
fruits[0] = new Strawberry(); – compiles!
This breaks **the general Java rule**: For each reference variable, the **dynamic type(type of the object referred by it) must be a subtype of the static one(type of declaration)**.

Java's design choices: The dynamic type of an array is known at runtime. During execution the JVM knows that the array bound to fruits is of type Apple[]. Every array update includes a run-time check. Assigning to an array element an object of a non-compatible type throws an ***ArrayStoreException***.

Type Erasure

All type parameters of generic types are transformed to ***Object*** or to their first bound after compilation. Main Reason is **backward compatibility** with legacy code. Thus at run-time, **all the instances of the same generic type have the same type**.

Every Java array update includes run-time check, but Generic types are not present at runtime due to type erasure, thus **Arrays of generics are not supported in Java**. In fact they would cause type errors not detectable at runtime, breaking Java strong type safety.

Wildcards

Wildcard = anonymous variable "?"

Wildcards are used when a type is used exactly once, and the name is unknown. They are used for **use-site variance**(not **declaration-site variance**).

Syntax of wildcards:

- ***<? extends Type>***, denotes an unknown subtype of **Type**;
- ***<?>***, shorthand for ***<? extends Object>***;
- ***<? super Type>***, denotes an unknown supertype of **Type**.

The **PECS principle**: **P**roducer **E**xtends, **C**onsumer **S**uper.
When should wildcards be used?

- Use ***<? extends T>*** when you want to get values(from a producer): **supports covariance**;
- Use ***<? super T>*** when you want to insert values(in a consumer): **supports contravariance**;
- Do not use ***<?>***(**T** is enough) when you both obtain and produce values.

Limitations of Java Generics

Mostly due to Type Erasure.
Cannot

- **instantiate generic types with primitive types;**
- **create instances of type parameters;**
- **declare static fields whose types are type parameters;**
- **use *casts* or *instanceof* with parameterized types;**
- **create arrays of parameterized types;**
- **create, catch, or throw objects of parameterized types;**
- **overload a method where the formal parameter types of each overload erase to the same raw type.**

4.4 The Standard Template Library: an overview

Goal: **represent algorithms in form as general as possible without compromising efficiency**.

The STL makes extensive use of **templates** and **overloading**, and only uses **static binding**(and inlining): **not object oriented**, **no dynamic binding** – very different from Java Collection Framework.

It uses **iterators** for **decoupling** algorithms from containers. Iterators are seen as **abstraction of pointers**.

Excellent example of generic programming, generated code is very efficient.

Main entities in STL

- **Container**: Collection of typed objects(array, vector, deque, list, set, map ...);
- **Iterator**: Generalization of pointer or address. Used to step through the elements of collections;
- **Algorithm**: initialization, sorting, searching, and transforming of the contents of containers;
- **Adaptor**: Converts from one form to another;
- **Function object**: Form of closure(class with "operator()" defined);
- **Allocator**: encapsulation of a memory pool.

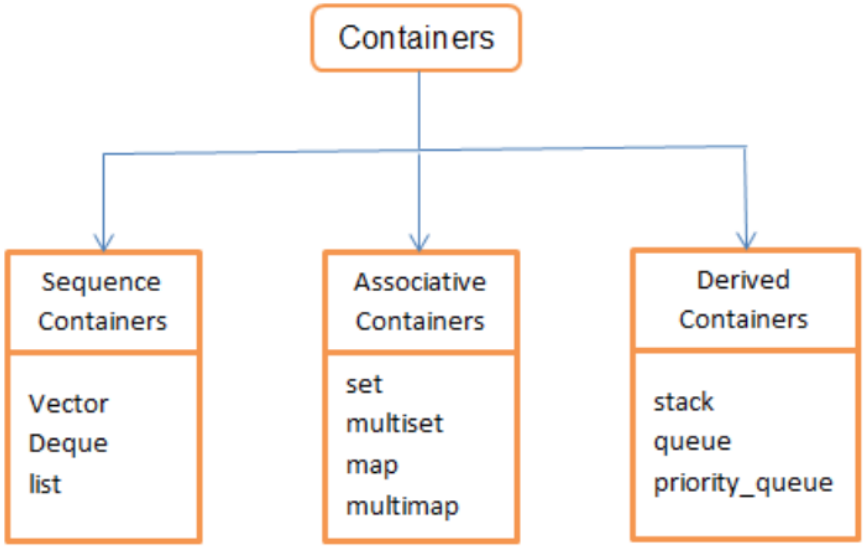

Figure 26: Containers Tree

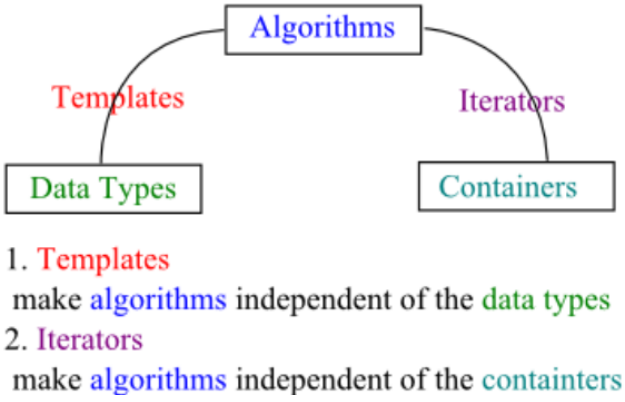

Figure 27: Algorithms decoupling Tree

STL relies on C++ **namespaces**. Containers expose a type named ***iterator*** in the container's namespace(ex. *std::vector<std::string>::iterator*).

Each class implicitly introduces a new namespace, and the iterator type name assumes its meaning depending on the context.

Container	***insert/erase overhead at the beginning***	***in the middle***	***at the end***
Vector	linear	linear	amortized constant
List	constant	constant	constant
Deque	amortized constant	linear	amortized constant

Figure 28: Operation Costs

How can we control complexity of algorithms and guarantee that code behaves as expected? **The solution proposed by STL is assume that iterators implement all operations in constant time.**

Containers may support different iterators depending on their structure

- **Forward iterators**: only dereference(operator*), and pre/post-increment operators(operator++). Provide for one-directional traversal of a sequence;
- **Input and Output iterators**: like forward iterators but with possible issues in dereferencing the iterator(due to I/O operations). Are like forward iterators but **do not guaratee these properties** of forward iterators
  - that an input or output iterator can be saved and used to start advancing from the position it holds a second time;
  - that it is possible to assign to the object obtained by applying * to an input iterator;
  - that it is possible to read from the object obtained by applying * to an output iterator;
  - that it is possible to test two output iterators for equality or inequality(== and != may not be defined).
- **Bidirectional iterators**: like forward iterators with pre/post-decrement(operator–). Provide for traversal in both directions;
- **Random access iterators**: like bidirectional iterators but with integer sum(p + n) and difference (p – q). Provide for bidirectional traversal, plus bidirectional *long jumps*;
- **Any C++ pointer type, *T**, obeys all the laws of the random access iterator category**.

Iterators heavily rely on operator overloading provided by C++.

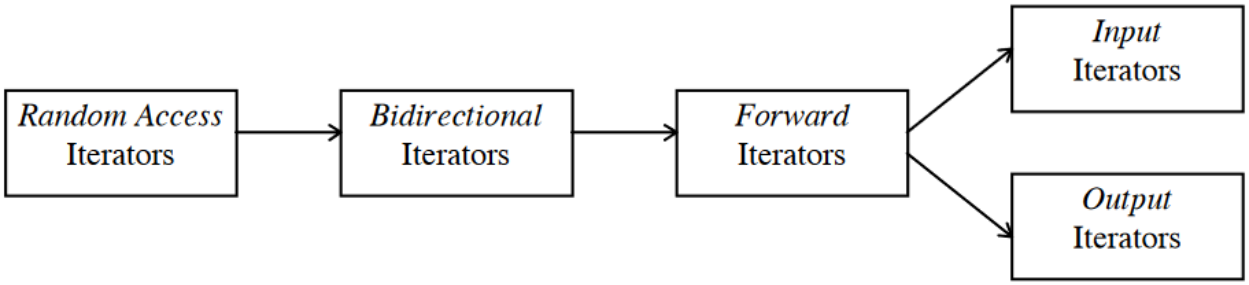

Figure 29: Five categories, with decreasing requirements

Each category has only those functions defined that are realizable in **constant time**.

Not all iterators are defined for all categories: since **random access** takes **linear time** on lists, random access iterators **cannot be used with lists**.

When a container is modified, iterators to it can become **invalid**: the result of operations on them is not defined. Which iterators become invalid depends on the operation and on the container type.

Container	*operation*	*iterator validity*
vector	inserting	reallocation necessary - all iterators get invalid
		no reallocation - all iterators before insert point remain valid
	erasing	all iterators after erasee point get invalid
list	inserting	all iterators remain valid
	erasing	only iterators to erased elements get invalid
deque	inserting	all iterators get invalid
	erasing	all iterators get invalid

Figure 30: Iterators validity

Iterators can help to see containers as linear, but then we can define only algorithms operating on single dimension containers. If it is necessary to walk the container in a different way, it is necessary to define a new iterator.

Under The Hood...

Iterators

Iterators are implemented by containers. Usually are implemented as ***struct***(classes with only public members). An iterator implements a visit of the container and retains inside information about the state of the visit(i.e. in the vector the pointer to the current element and the number of remaining elements). The state may be complex in the case of non linear structures such as graphs.

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. ***There is no relation among iterators for different containers!***. The reason for this is **PERFORMANCE**. Without inheritance types are resolved at compile time and the compiler may produce better code! This is an extreme position: sacrificing inheritance may lead to **lower expressivity** and **lack of type-checking**. STL relies only on coding conventions: when the programmer uses a wrong iterator the compiler complains of a bug in the library.

Inlining

STL relies also on the compiler. C++ standard has the notion of **inlining** which is a form of semantic macros. A method invocation is type-checked then it is replaced by the method body. Inline methods should be available in header files and can be labelled *inline* or defined within class definition. Inlining isn't always used: the compiler tends to inline methods with small bodies and without iteration. The compiler is able to determine types at compile time and usually does inlining of function objects.

Memory Management

STL abstracts 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. STL's own memory management strategies are used when no other allocator is specified by the user.

Potential Problems

The main problem with STL is **error checking**. Almost all facilities of the compiler fail with STL resulting in lengthy error messages that ends with error within the library. The generative approach taken by C++

compiler also leads to possible **code bloat**. Can be a problem if the working set of a process becomes too large.

5 Functional Programming

5.1 Introduction

Key Idea: do everithing by comping functions. No mutable state. No side effects.

Main Features

- **1st class and high-oder functions** - Functions can be denoted, passed as arguments to functions and returned as result of function invocation;
- **Recursion** - takes the place of iteration;
- **Powerful list facilities** - recursive functions exploit recursive definition of list;
- **Polymorphism** - relevance of containers/collections;
- **Fully general aggregates** - Wide use of tuples and records. Data structures cannot be modified, have to be re-created;
- **Structured function returns** - no side effects, thus the only way for functions to pass information to the caller;
- **Garbage collection** - In case of static scoping, unlimited extent for locally allocated data structures and locally defined functions. They cannot be allocated on the stack.

5.2 Evaluation Strategies on Lambda Calculus

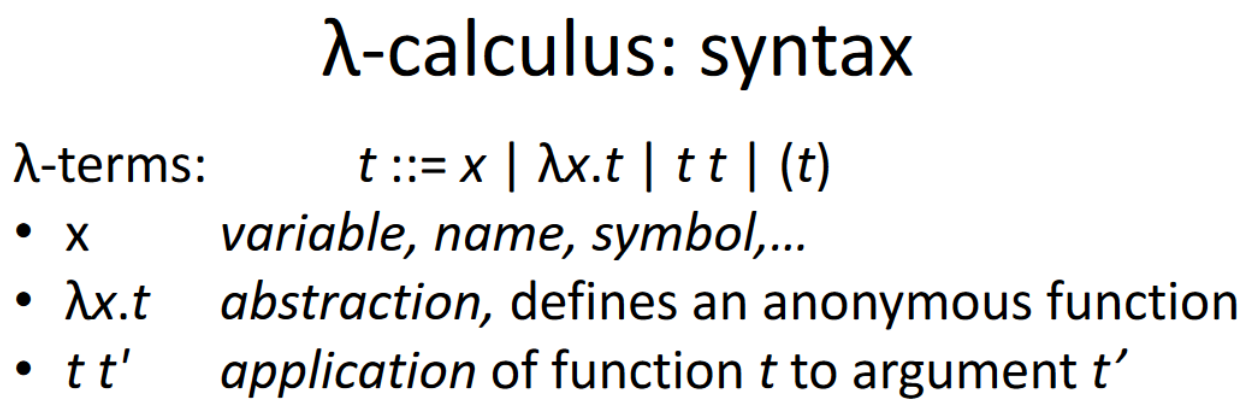

Figure 31: Lambda calculus sintax

Terms can be represented as abstracts syntax trees.

An occurrence of **x** is **free** in a term **t** if it is not in the body of an abstraction $\boldsymbol{\lambda x.t}$, otherwise it is **bound**. $\boldsymbol{\lambda x}$ is a **binder**.

Terms without free variables are **combinators**.

Operational Semantics

$\boldsymbol{\beta}$**-reduction** = function application.

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

Despite the simplicity, we can encode in λ-calculus most concepts of functional languages

- Functions with several arguments(a sequence of λ-abstractions;
- Booleans and logical connectives;
- Integers and operations on them;
- Pairs and tuples;
- Recursion(A recursive function definition(like factorial) can be read as a **higher-order transformation** having a function as first argument, and the desired function is its fix-point).

Applicative and Normal Order evaluation

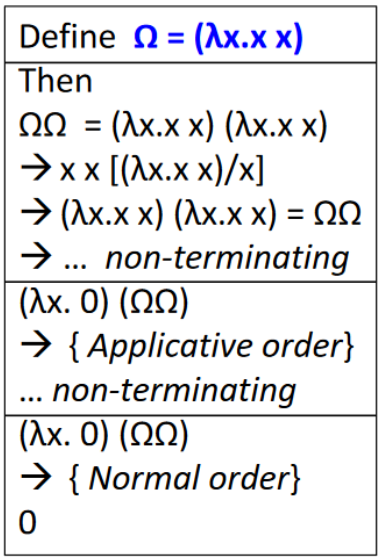

Figure 32: Lambda calculus execution orders

Applicative Order evaluation: Arguments are evaluated before applying the function(Eager evaluation), with parameter passing by value.

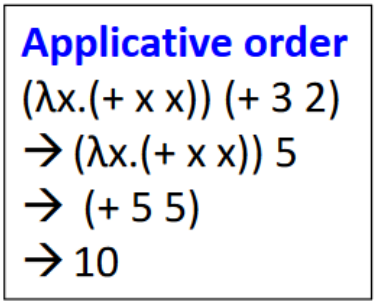

Figure 33: Lambda calculus Applicative Order Example

Normal Order evaluation: Function evaluated first, arguments if and when needed. Sort of parameter passing by name. Some evaluation can be repeated.

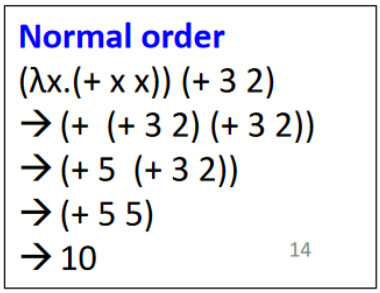

Figure 34: Lambda calculus Normal Order Example

Church-Rosser: If evaluation terminates, the result(**normal form**) is unique. If some evaluation terminates, normal order evaluation terminates.

5.3 Calling by sharing, by name and by need

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. The argument is evaluated only the first time, using **memoization**: the result is saved and further uses of the argument do not need to re-evaluate it.

Call by Sharing: parameter passing of data in the reference model. The value of the variable is passed as actual argument, which in fact is a reference to the (shared) data. Essentially this is call by value of the variable.

Call by Name: the actual parameter is copied wherever the formal parameter appears in the body, then the resulting code is executed. Thus the actual parameter is evaluated a number of times that depends on the logic of the program. Since the actual parameter can contain names, it is passed in a closure with the environment at invocation time (called a thunk).

Haskell ->Reference Model.

Java ->Value Model for built-in types, Reference Model for class instances.

C++ ->Value Model for value types, Reference Model for reference types.

6 Haskell

6.1 Introduction to Haskell, Laziness, Basic and compounds types, Patterns and declarations, Function declarations

Basic Types

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

Binding variables: **variables(names)** are bound to **expressions**, without evaluating them(because of **lazy evaluation**).

Patterns can be used in place of variables.

Haskell is a **lazy** language. Functions and data constructors don't evaluate their arguments until they need them(thanks to Normal Order evaluation and Call by Need).

6.2 Higher-order functions, Recursion

In Haskell, $\boldsymbol{f :: A -> B}$ means for every $x \in A$, **f(x)** = some element $\boldsymbol{y = f(x) \in B}$, or **run forever**. In words, "if **f(x)** terminates, then $\boldsymbol{f(x) \in B}$".

Note: if **f** is a standard binary function, **'f'** is its infix version, if **x** is an infix (binary) operator, **(x)** is its prefix version.

In functional programming, for and while loops are replaced by using **Recursion**: subroutines call themselves directly or indirectly(mutual recursion).

Functions that take other functions as arguments or return a function as a result are **higher-order functions**. They can be used to support alternative syntax(ex. from functional to stream-like). Any **curried function** with more than one argument is **higher-order**: applied to one argument it returns a function.

map: applies argument function to each element in a collection.

filter: takes a collection and a boolean predicate, and returns the collection of the elements satisfying the predicate.

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

Iteration and **recursion** are equally powerful in theoretical sense: Iteration can be expressed by Recursion and vice versa. Recursion is the natural choice when the solution of a problem is defined in terms of simpler versions of the same problem, as for tree traversal. In general a procedure call is much more expensive than a conditional branch. Thus recursion is in general less efficient, but good compilers for functional languages can perform good code optimization. Use of **combinators**, like map, reduce (foldl, foldr), filter, foreach,... are strongly encouraged, because they are highly optimized by the compiler.

Tail-recursive functions are those in which no operations follow the recursive call(s) in the execution, thus the function returns immediately after the recursive call. A tail-recursive call could **reuse the subroutine's frame** on the run-time stack(since the current one is no longer needed), simply eliminating the push (and pop) of the next frame. In addition, we can do more for **tail-recursion optimization**: the compiler replaces tail-recursive calls by jumps to the beginning of the function.

Tail-call: a function returns calling another function, not necessarily itself. Optimization is still possible, reusing the stack frame(**Note**: Number/size of parameters can differ).

- For example

```
reverse [] = [] -- quadratic
reverse (x:xs) = (reverse xs) ++ [x]
```

can be rewritten into a tail-recursive function:

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

Equivalently, using the **where** syntax:

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

Figure 35: Example of optimization of a tail-call

Remove the work after the recursive call and include it in some other form as a computation that is passed to the recursive call.

foldl* is tail-recursive**, ***foldr* is not**. But because of laziness, Haskell has no tail-recursion optimization. ***foldl' is a variant of ***foldl*** where **f** is evaluated strictly. It is more efficient.

6.3 Type classes in Haskell

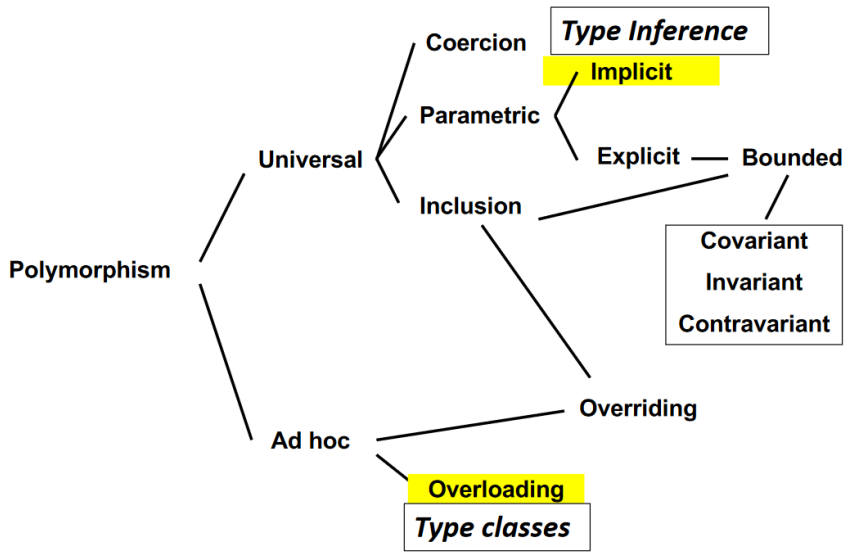

Figure 36: Haskell Type Classes Polymorphism

Overloading: present in all languages, at least for built-in arithmetic operators. Haskell allows overloading of primitive operators. The code to execute is determined by the type of the arguments, thus

- **early binding** in statically typed languages;
- **late binding** in dynamically typed languages.

Type Classes provide concise types to describe overloaded functions. Them allow users to define functions using overloaded operations, to declare new collections of overloaded functions.

To define a **type class** you have to define also a **set of operations** for the type class(ex. *Eq a*, type class with operations == and \=). Then if you want to **implement the type class**, you have to provide an implementation for all the operations that were defined before. At this point, it is allowed to define **constraints** on function parameters to implement the operations of a particular type class.

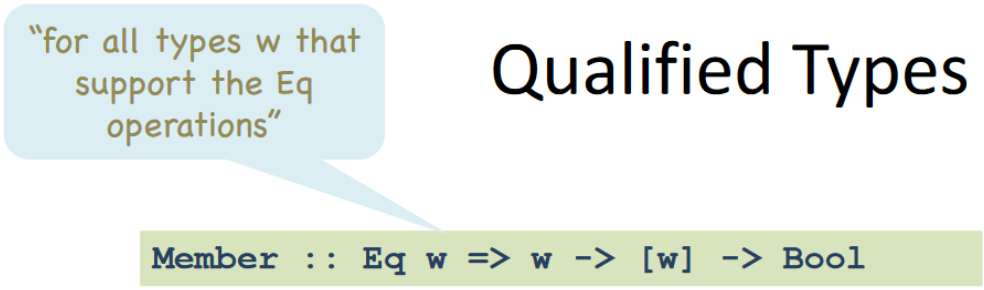

Figure 37: Haskell Qualified Types

When you compile a function defined for specific type class operations, at compile time, the function is turned into another function that takes **an additional argument** that is a **dictionary** that contains all the required operations that have to be implemented(ex. A value of type $(Num\ n)$ is a dictionary of the Num operations for type n).

References to overloaded symbols are rewritten by the compiler to **lookup the symbol in the dictionary**. The compiler converts each **type class declaration** into a **dictionary type declaration** and a set of **selector functions**.

It also converts each **instance declaration** into a **dictionary of the appropriate type**, rewrites calls to overloaded functions to **pass a dictionary** and uses the **static, qualified type** of the function to **select the dictionary**.

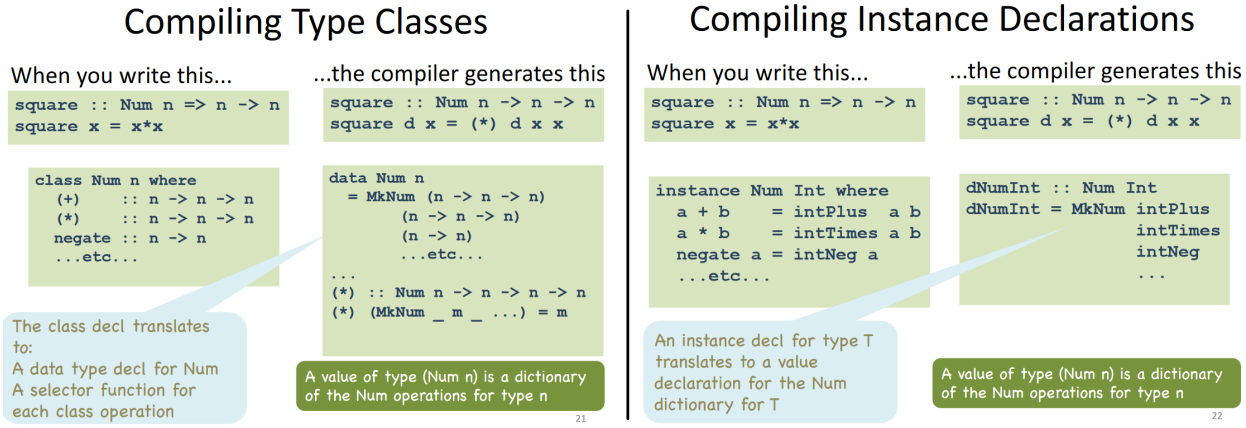

Figure 38: Haskell Type Classes Compilation

Common Type Classes are

- **Eq**: equality;
- **Ord**: comparison;
- **Num**: numerical operations;
- **Show**: convert to string;
- **Read**: convert from string;
- **Testable, Arbitrary**: testing;
- **Enum**: operations on sequentially ordered types;
- **Bounded**: upper and lower values of a type.

6.4 The Maybe constructor and composition of partial functions

6.4.1 Type Constructor Classes

Type Classes are predicates over types. **Type Constructor Classes** are predicates over type constructor. They allow to define overloaded functions common to several type constructors(ex. *map* function is usable on many Haskell types).

All ***map*** functions share the same structure:
`fmap:: (a → b) → g a → g b`
where ***g*** is a function from types to types, i.e. a **type constructor**.

This pattern can be captured in a constructor class **Functor**.

6.4.2 Functor e fmap

Functor is simply a type class where the predicate is over a **type constructor** rather than on a **type**.

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

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor [] where
  fmap f [] = []
  fmap f (x:xs) = f x : fmap f xs

instance Functor Tree where
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Node(t1,t2)) = Node(fmap f t1, fmap f t2)

instance Functor Maybe where
  fmap f (Just s) = Just(f s)
  fmap f Nothing = Nothing
```

Figure 39: Haskell Functor constructor class and some instances

6.4.3 Maybe and partial functions

Monads are constructor classes introducing operations for putting a value into a "box" and compose functions that return "boxed" values.

Maybe

```
data Maybe a = Nothing | Just a
```

A value of type **Maybe a** is a possibly undefined value of type **a**. A function `f:: a → Maybe b` is a partial function from a to b.

Partial Functions

Once defined, partial functions can be **composed** introducing a **high order operator** capable of propagate undefinedness automatically.

```
y >>= g  = case  y of          -- y "bind" g
               Nothing -> Nothing
               Just x -> g x

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

Figure 40: Haskell bind operator

The **bind** operator will be part of the definition of a *Monad*.

```
(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
maternalGrandfather p = mother p >>= father

bothGrandfathers :: Person -> Maybe(Person, Person)
bothGrandfathers p =
       father p >>=
           (\dad -> father dad >>=
               (\gf1 -> mother p >>=
                   (\mom -> father mom >>=
                       (\gf2 -> return (gf1,gf2) ))))
```

Figure 41: Haskell partial function composition

6.5 Monads in Haskell

```
class Monad m where
        return :: a -> m a
        (>>=)  :: m a -> (a -> m b) -> m b -- "bind"
         ... -- + something more
```

Figure 42: Haskell Monad definition

m is a type constructor.
m a is the type of **monadic values**.

```
instance Monad Maybe where
        return :: a -> Maybe a
        return x = Just x
        (>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
        y >>= g  = case  y of
                     Nothing -> Nothing
                     Just x -> g x
```

Figure 43: Haskell Maybe Monad definition

bind shows how to propagate undefinedness.

6.5.1 Undestanding Monads as containers

The monadic constructor can be seen as a ***container***. Let's see it 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)
```

Figure 44: Haskell list monad

6.5.2 Undestanding Monads as 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
```

Figure 45: Haskell monad as computations

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.

return, **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.

7 Functional programming in Java 8

7.1 Lambdas in Java 8

They enabled to use a sort of **functional paradigm** in Java, in particular being able to pass behaviors as well as data to functions and introducing **lazy evaluation** with stream processing. They also helped to write cleaner code and facilitated **parallel programming**.

`x → System.out.println(x)` is a lambda expression that defines an anonymous function(method) with one parameter named **x**.

At **compile time**, the compiler first converts a lambda expression into a function, compiling its code, then it generates code to call the compiled function where needed.

Design decision: Java 8 lambdas are instances of ***functional interfaces***. A functional interface is a Java interface with exactly one abstract method(ex. Runnable). Functional interfaces can be used as **target type** of lambda expressions(**type of variable** to which lambda is assigned and **type of formal parameter** to which lambda is passed).

The compiler uses type inference **based on target type**: arguments and result types of the lambda must match those of the unique abstract method of the functional interface, and the lambda is invoked calling this method.

Lambdas can be seen as **instances** of **anonymous inner classes** implementing the functional interface.

```
public class ThreadTest {// using functional interface Runnable
  public static void main(String[] args) {
    Runnable r1 = new Runnable() { // anonymous inner class
      @Override
      public void run() {
        System.out.println("Old Java Way");
      }
    };

    Runnable r2 = () -> { System.out.println("New Java Way"); };

    new Thread(r1).start();
    new Thread(r2).start();
  }
}
```

Figure 46: Java Lambda example with Runnable functional interface

Lambda expressions in conjunction with **default methods** of interfaces and **Java collection framework** to achieve **backward compatibility** with existing published interfaces.

Method references can be used to pass an existing function in places where a lambda is expected(ex. `System.out::println`).

Lambdas can, in principle, be compiled as instances of anonymous inner classes. Neither JLS 8 nor JVMS 8 prescribe a specific compilation strategy for lambdas, so it is left to the designer of the compiler, which can exploit this freedom on behalf of efficiency.

7.2 The Stream API in Java 8

The `java.util.stream` package provides utilities to support functional-style operations on **streams of values**. Streams differ from collections in several ways:

- **No storage**: a stream is not a data structure that stores elements; instead, it conveys elements from a source(a data structure, an array, a generator function, an I/O channel,...) 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** and **terminal(*value-* or *side-effect-producing*) operations**. Intermediate operations are always lazy;
- **Possibly unbounded**: 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.

A typical pipeline contains

- A **source**, producing (by need) the elements of the stream;
- Zero or more **intermediate operations**, producing streams;

- A **terminal operation**, producing side-effects or non-stream values.

```
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
```

Figure 47: Java Stream example

A Stream is processed through a **pipeline of operations**. It starts with a **source**, then Intermediate methods are performed on the Stream elements. These methods produce Streams and are not processed until the **terminal method** is called. The Stream is considered **consumed** when a terminal operation is invoked. No other operation can be performed on the Stream elements afterwards. A Stream pipeline may contain some **short-circuit methods**(which could be intermediate or terminal methods) that cause the earlier intermediate methods to be **processed** only until the short-circuit method can be evaluated.

Stream popular sources are

- from a Collection via the `stream()` and `parallelStream()` methods;
- from an array via `Arrays.stream(Object[])`;
- from static factory methods on the stream classes, such as `Stream.of(Object[])`, `IntStream.range(int, int)` or `Stream.iterate(Object, UnaryOperator)`;
- the lines of a file can be obtained from `BufferedReader.lines()`;
- streams of file paths can be obtained from methods in **Files**;
- streams of random numbers can be obtained from `Random.ints()`;
- infinite streams can be obtained with generators, like **generate** or **iterate**.

Peek function does not affect the stream and is typically used for debugging.

Streams are available **only** for reference types, int, long and double, so for minor primitive types you need a cast.

To collect elements in efficient way → `collect()`, it's a **mutable reduction operation**. It requires three functions

- **supplier**: function to construct a new instance of result container;
- **accumulator**: function to incorporate an input element into a result container;
- **combining function**: function to merge the contents of one result container into another.

Method **collect()** can be invoked with a **Collector** argument: a Collector encapsulates the functions used as arguments to collect, allowing for reuse of collection strategies and composition of collect operations.

Stream API facilitates parallelism through **parallelStream()**. The runtime support takes care of using multithreading for parallel execution in a transparent way. If operations doesn't have side effects, thread safety is guaranteed, even with non-thread safe data structures.

When to use parallel streams? When operations are **independent**, **computationally expensive** and are applied to many elements of **efficiently splittable data structures**.

A useful tool is the **Splititerator**, that is the parallel equivalent of the **Iterator**. It describes a possibly infinite collection of elements with support for **sequentially advancing**, **applying** of an action to next

element or all remaining ones and for **splitting off** some portion of the input into another Splititerator which can be processed in parallel.

8 Scripting Languages and Python

8.1 Overview of Scripting Languages

8.1.1 Common Characteristic

Common Characteristics of scripting languages are

- both **batch**(file) and **interactive**(console) use(compiled/interpreted line by line);
- **economy of expression**(concise syntax, avoid top level declarations);
- **lack of declarations**;
- **simple** default **scoping rules**, or simple explicit declared scoping rules;
- **dynamic typing**;
- **flexible typing**(variable interpreted differently depending on the context);
- **easy access to system facilities**(I/O, file manipulation, process management...);
- sophisticated **pattern matching and string manipulation**(based on extended regular expressions);
- **high level data types**(associative arrays implemented as hash table);
- **quick development cycle**(able to include state-of-the-art features).

8.1.2 Problem Domains

Some general purpose languages are widely used for scripting, while some scripting languages are intended for general use, with features supporting "programming in the large". Nevertheless most scripting languages have principal use in well **defined problem domains**.

Shell languages

Shell Languages have features designed for interactive use. Provide many mechanisms to manipulate file names, arguments, and commands, and to glue together other programs. Typical mechanisms supported are

- **Filename and Variable Expansion**;
- **Tests**, **Queries**, and **Conditions**;
- **Pipes** and **Redirection**;
- **Quoting** and **Expansion**;
- **Functions**;
- **The #! Convention**.

Text processing and report generation

sed: Unix's stream editor. No variables, no state: just a powerful filter. Processes one line of input at a time, the first matching command is executed. `s/_/_/` substitution command.

awk: adds variables, state and richer control structures, also fields and associative arrays.

Perl("practical extraction and report language"): Unix-only tool, meant primarily for text processing. Over the years has grown into a large and complex language, ported to all operating systems. Also fast

enough for much general purpose use, and includes separate compilation, modularization, and dynamic library mechanisms appropriate for large-scale projects.

Mathematics and statistics

Maple, **Mathematica** and **Matlab**(Octave): commercial packages successor of APL. Extensive support for numerical methods, symbolic mathematics, data visualization, mathematical modeling. Provide scripting languages oriented towards scientific and engineering applications.

Languages for statistical computing: **R**(open source) and **S**. Support for multidimensional arrays and lists, array slice operations, call-by-need, first-class functions and unlimited extent.

"Glue" languages and General Purpose Scripting

Rexx(1979) is considered the first of the general purpose scripting languages.

Perl and **Tcl** are roughly contemporaneous. Perl was originally intended for glue and text processing applications, Tcl was originally an extension language, but soon grew into glue applications.

Python was originally developed by Guido van Rossum at CWI in Amsterdam, the Netherlands, in the early 1990s. Recent versions of the language are owned by the Python Software. All releases are Open Source and Object oriented.

Ruby was developed in Japan in early 1990: "a language more powerful than Perl, and more object-oriented than Python". English documentation published in 2001. Smalltalk-like object orientation.

Extension Languages

Most applications accept some sort of commands. Commands are entered textually or triggered by user interface events such as mouse clicks, menu selections, and keystrokes. An **extension language** serves to increase the usefulness of an application by allowing the user to create new commands, generally using the existing commands as primitives. Extension languages are an essential feature of sophisticated tools. Adobe's graphics suite (Illustrator, Photoshop, InDesign, etc.) can be extended (scripted) using JavaScript, Visual Basic(on Windows), or AppleScript. To admit extension, a tool must–incorporate, or communicate with, an **interpreter** for a scripting language. It provides hooks that allow scripts to call the tool's existing commands and allows the user to tie newly defined commands to user interface events. With care, these mechanisms can be made independent of any particular scripting language.

8.1.3 Innovative Features

Most scripting languages **do not require variables to be declared**.

- **Perl** and **JavaScript**, permit optional declarations, sort of compiler-checked documentation;
- **Perl** can be run in a mode that requires declarations. With or without declarations, most scripting languages use dynamic typing;
- The interpreter can perform type checking at run time, or coerce values when appropriate;
- **Tcl** is unusual in that all values, even lists are represented internally as strings.

Nesting and scoping conventions vary quite a bit.

- **Scheme**, **Python** and **JavaScript** provide the classic combination of nested subroutines and static(lexical) scope;
- **Tcl** allows subroutines to nest, but uses dynamic scope;
- Named subroutines(methods) do not nest in **PHP** or **Ruby**. **Perl** and **Ruby** join **Scheme**, **Python**, **JavaScript**, in providing first class anonymous local subroutines;

- Nested blocks are statically scoped in **Perl**. In **Ruby** they are part of the named scope in which they appear;
- **Scheme**, **Perl**, **Python** provide for variables captured in closures;
- **PHP** and the major glue languages(**Perl**, **Tcl**, **Python**, **Ruby**) all have sophisticated namespace mechanisms for information hiding and the selective import of names from separate modules.

String and Pattern Manipulation.

- **Regular expressions** are present in many scripting languages and related tools employ extended versions of the notation(**sed**, **awk**, **Perl**, **Tcl**, **Python**, and **Ruby**). **Grep**, the stand-alone Unix is a pattern-matching tool;
- Two main groups. The first group includes **awk**, **egrep**(the most widely used of several different versions of grep), the **regex routines of the C standard library**, and older versions of **Tcl**. These implement **REs** as defined in the POSIX standard. Languages in the second group follow **Perl**, which provides a large set of extensions, sometimes referred to as "**advanced REs**".

As we have seen, scripting languages don't generally require (or even permit) the **declaration of types for variables**. Most perform **extensive run-time checks** to make sure that values are never used in inappropriate ways. Some languages(e.g., Scheme, Python, and Ruby) are relatively strict about this checking. When the programmer wants to convert from one type to another he must say so explicitly.

Numeric types: "numeric values are simply numbers"

- In **JavaScript** all numbers are double precision floating point;
- In **Tcl** are strings;
- **PHP** has double precision float and integers;
- To these **Perl** and **Ruby** add **bignums**(arbitrary precision integers);
- **Python** has complex numbers;
- **Scheme** has rationals;

Composite types: mainly associative arrays(based on hash tables)

- **Perl** has fully dynamic arrays indexed by numbers, and hashes, indexed by strings. Records and objects are realized with hashes;
- **Python** and **Ruby** also have arrays and hashes, with slightly different syntax;
- **Python** also has sets and tuples;
- **PHP** and **Tcl** eliminate distinction between arrays and hashes. Likewise **JavaScript** handles in a uniform way also objects.

Object Orientation

- **Perl 5** has features that allow one to program in an object-oriented style;
- **PHP** and **JavaScript** have cleaner, more conventional-looking object-oriented features;
- **Python** and **Ruby** are explicitly and uniformly object-oriented;
- **Perl** uses a value model for variables: objects are always accessed via pointers;
- In **PHP** and **JavaScript**, a variable can hold either a value of a primitive type or a reference to an object of composite type.These languages provide no way to speak of the reference itself, only the object to which it refers;
- **Python** and **Ruby** use a uniform reference model;
- They are types in **PHP**, much as they are in **C++**, **Java**, or **C#**;

- Classes in **Perl** are simply an alternative way of looking at packages (namespaces);
- **JavaScript**, remarkably, has objects but no classes. Its inheritance is based on a concept known as prototypes;
- While **Perl**'s mechanisms suffice to create object-oriented programs, dynamic lookup makes both **PHP** and **JavaScript** more explicitly object oriented;
- Classes are themselves objects in **Python** and **Ruby**, much as they are in **Smalltalk**.

8.2 Introduction to Python: Basic and Sequence Data types, Dictionaries, Control Structures, List Comprehension

The first assignment to a variable creates it. Variable types don't need to be declared, Python figures out the variable types on its own.

Basic data types are

- **integer** - default for numbers;
- **floats**;
- **strings**;

Binding a variable in Python means setting a name to hold a reference to some object. **Assignment creates references**, not copies(like Java). A variable is created the first time it appears on the left side of an assignment expression. An object is deleted (by the garbage collector) once it becomes unreachable. Names in Python do not have an intrinsic type, **objects have types**. Python determines the type of the reference automatically based on what data is assigned to it.

Sequence data types are

- **Tuples** - a simple immutable ordered sequence of items. Can be of mixed types. The fact that a tuple is immutable makes it faster than a list.
- **Strings** - Immutable.
- **Lists** - Mutable ordered sequence of items of mixed types. Can be modified in place.

Dictionaries store a mapping between a set of keys and a set of values. **Keys** can be of any immutable hashable type, cannot contain mutable components. **Values** can be any type. Values and keys can be of different types in a single dictionary. Dictionaries are **unordered**. New entry might appear anywhere in the output(**Dictionaries work by hashing**).

True and **False** are constants. Other values are treated as equivalent to either True or False when used in conditionals. **False**: zero, None, empty containers. **True**: non-zero numbers, non-empty objects.

```
x = true_value if condition else false_value
```

lazy evaluation: First **condition** is evaluated. If True, **true_value** is evaluated and returned. If False, **false_value** is evaluated and returned.

List Comprehension
A powerful feature of Python is the power to generate a list by applying a function to every member of the original list.
`[expression for name in list]`. Where **expression** is some calculation or operation acting upon the variable **name**. For each member of the list, the list comprehension sets **name** equal to that member, and calculates a new value using **expression**. It then collects these new values into a list which is the return value of the list comprehension. If the elements of list are other collections, then **name** can be replaced by a collection of names that match the "shape" of the list members.

Filtered List Comprehension
`[expression for name in list if filter]`
Filter determines whether **expression** is performed on each member of the list. While processing each member of the list, first check if **filter** is satisfied. If the **filter** returns False, the element is omitted from the new list.

8.3 Python: Function definition, Positional and keyword arguments of functions, Functional Programming in Python, Iterators and Generators, Using higher order functions: Decorators

Essentials

- Functions are first-class objects;
- All functions return some value (possibly None);
- Function call creates a new namespace;
- Parameters are passed by object reference;
- Functions can have optional keyword arguments;
- Functions can take a variable number of **args** and **kwargs**;
- Higher-order functions are supported.

```
def sum(n,m):
    """ adds two values """
    return n+m

def print_args(*items): # arguments are put in a tuple
    print(type(items))
    return items

#-------------------------------------

def sum(n,m=5): # default parameter
    """ adds two values, or increments by 5 """
    return n+m
#-------------------------------------

def print_kwargs(**items): # args are put in a dict
    print(type(items))
    return items
```

Figure 48: Python parameter passing

As everything in Python, also functions are objects, of class **function**. Functions can be passed as argument and returned as result. Main combinators(**map**, **filter**) are predefined: allow standard functional programming style in Python.

Some modules for functional programming in Python are

- **functools**: Higher-order functions and operations on callable objects;
- **itertools**: Functions creating iterators for efficient looping.

Decorators
A **decorator** is any callable Python object that is used to modify a **function**, **method** or **class definition**. A decorator is passed the original object being defined and returns a modified object, which is then bound to the name in the definition. (Function) Decorators exploit Python **higher-order features**: Passing functions as argument, nested definition of functions, returning function. Widely used in Python (system) programming, it supports several features of meta-programming.

```
import functools
def decorator(func):
    @functools.wraps(func)     #supports introspection
    def wrapper_decorator(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value
    return wrapper_decorator
```

Figure 49: Python decorator structure

Besides passing arguments, the **wrapper** also forwards the result of the decorated function. Supports introspection redefining `__name__` and `__doc__`.

Other common uses of decorators are

- **Debugging**: prints argument list and result of calls to decorated function;
- **Registering plugins**: adds a reference to the decorated function, without changing it;
- **Check the user is logged**;
- **@staticmethod** and **@classmethod**: make a function invocable on the class name or on an object of the class.

Namespace
A **namespace** is a mapping from names to objects, typically implemented as a dictionary. Name **x** of a module **m** is an attribute of **m** accessible(read/write) with "*qualified name*"**m.x**.

Scope
A **scope** is a textual region of a Python program where a namespace is **directly accessible**, i.e. reference to a name attempts to find the name in the namespace. Scopes are determined statically, but are used dynamically. During execution at least three namespaces are directly accessible, searched in the following order

- the scope containing **the local names**;
- the scopes of any enclosing functions, containing **non-local**, but also **non-global** names;
- the **next-to-last scope** containing the current module's **global names**;
- the **outermost scope** is the namespace containing **built-in names**.

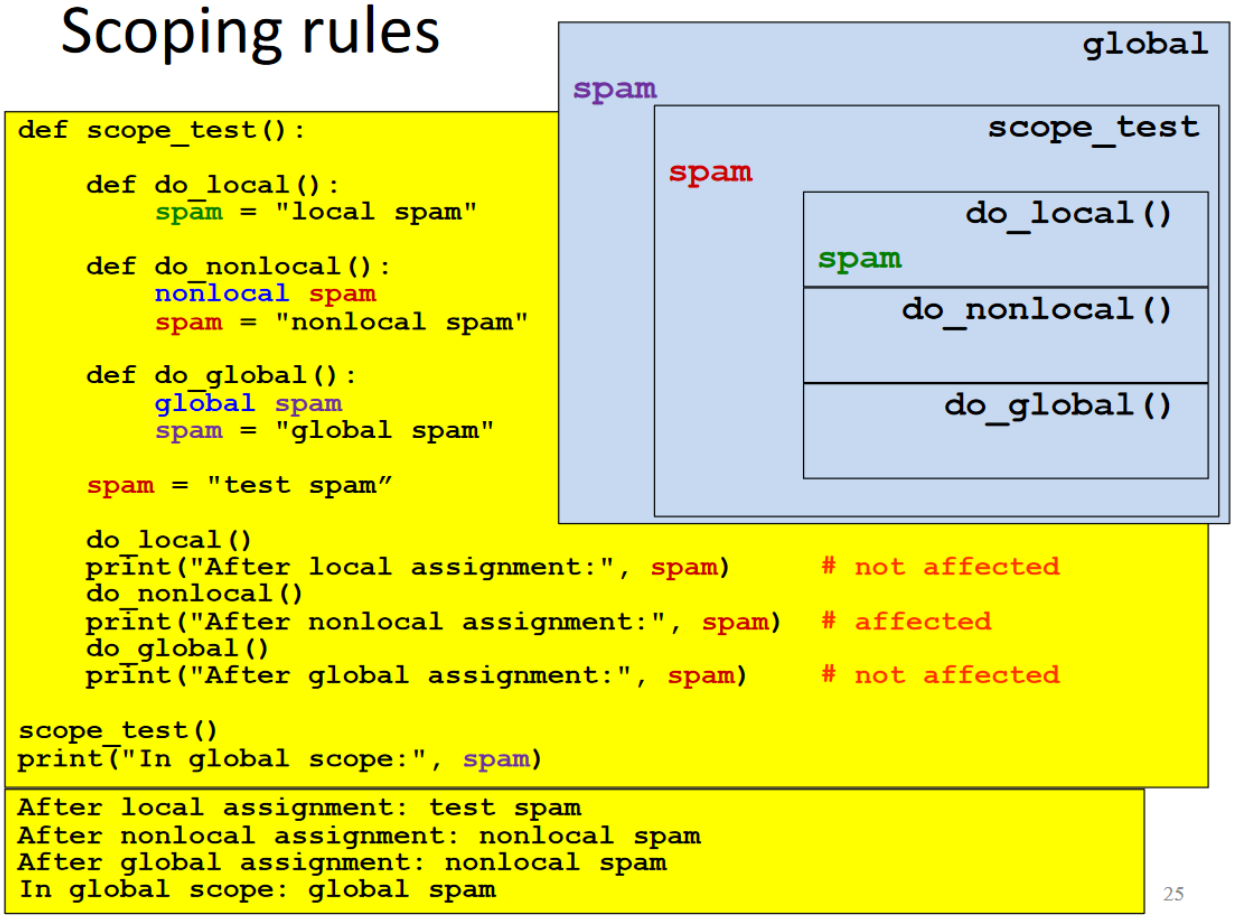

Figure 50: Python scoping rules

Python supports closures: Even if the scope of the outer function is reclaimed on return, the non-local variables referred to by the nested function are saved in its attribute `__closure__`.

An **iterator** is an object which allows a programmer to traverse through all the elements of a collection(**iterable** object), regardless of its specific implementation. In Python they are used implicitly by the **FOR** loop construct. Python iterator objects required to support two methods

- **__iter__**: returns the iterator object itself. This is used in FOR and IN statements;
- The **next** method: returns the next value from the iterator. If there is no more items to return then it should raise a **StopIteration** exception.

Remember that an iterator object **can be used only once**. It means after it raises *StopIteration* once, it will keep raising the same exception.

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the **yield** statement whenever they want to return data. Each time the **next()** is called, the generator resumes where it left-off(it remembers all the data values and which statement was last executed).
Anything that can be done with generators can also be done with class based iterators(not vice-versa).
What makes generators so compact is that the **__iter__()** and **next()** methods are created automatically.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise **StopIteration**.

8.4 Python: Classes and Instances, Single and Multiple Inheritance, Magic Methods for operator overloading, Modules definition and importing

A **class** is a blueprint for a new data type with specific internal **attributes**(like a struct in C) and internal functions(**methods**).

When the class definition is left, a **class object** is created, bound to **className**, on which two operations are defined: **attribute reference** and **class instantiation**. Attribute reference allows to access the names in the namespace in the usual way.

A **class instance** introduces **a new namespace nested in the class namespace**: by visibility rules all names of the class are visible. If no **constructor** is present, the syntax of class instantiation is `className()`: the new namespace is empty.

A class can define a set of **instance methods**, which are just functions. The first argument, usually called **self**, represents the **implicit parameter**(*this* in Java). A method must access the object's attributes through the **self** reference(eg.*self.x*) and the class attributes using **className.<attrName>**(or **self.__class__.<attrName>**). The first parameter must not be passed when the method is called. It is bound to the target object(but it can be passed explicitly).

Any function **with at least one parameter** defined in a class can be invoked on an instance of the class with the dot notation. A name **x** defined in the (namespace of the) instance is accessed as **par-0.x**(i.e.,usually **self.x**). A name **x** defined in the class is accessed as **className.x**(or **self.__class__.x**).

A **constructor** is a **special instance method** with **name = __init__**. The first parameter **self** is bound to the new object.
Note: at most ONE constructor(no overloading in Python!).

Method overloading: you can define special instance methods so that Python's built-in operators can be used with your class.

A class can be defined as a **derived** class. No need of additional mechanisms: the namespace of derived is nested in the namespace of **baseClass**, and uses it as the next non-local scope to resolve names. All instance methods are automatically **virtual**: lookup starts from the instance (namespace) where they are invoked. Python supports **multiple inheritance**.

Private instance variables(not accessible except from inside an object) **don't exist in Python**.
Convention: a name prefixed with underscore(e.g._*spam*) is treated as non-public part of the API(function, method or data member). It should be considered an implementation detail and subject to change without notice.
Any **name with at least two leading underscores and at most one trailing underscore**, like ***__spam***, is textually replaced with ***_class__spam***, where class is the current class name. Name mangling is helpful for letting subclasses override methods without breaking intra class method calls.

Static methods are simple functions defined in a class with **no self** argument, preceded by the **@staticmethod** decorator. They are defined inside a class but they cannot access instance attributes and methods. They can be called through both the class and any instance of that class. **Benefits of static methods**: they allow subclasses to customize the static methods with inheritance. Classes can inherit static methods without redefining them.

Class methods are similar to static methods but they have a first parameter which is the class name. Definition must be preceded by the **@classmethod** decorator. Can be invoked on the class or on an instance.

CPython(original interpreter of Python, written in C) manages memory with a **reference counting** and a **mark&sweep** cycle collector scheme. **Reference counting**: each object has a counter storing the number of references to it. When it becomes 0, memory can be reclaimed.

- **Pros**: simple implementation, memory is reclaimed as soon as possible, no need to freeze execution passing control to a garbage collector;

- **Cons**: additional memory needed for each object, cyclic structures in garbage cannot be identified (thus the need of mark&sweep).

Updating the **refcount** of an object has to be done atomically. In case of **multi-threading** you need to synchronize all the times you modify *refcounts*, or else you can have wrong values. Synchronization primitives are quite expensive on contemporary hardware. Since almost every operation in *CPython* can cause a *refcount* to change somewhere, handling *refcount* with some kind of synchronization would cause spending almost all the time on synchronization. As a consequence → **GIL**.

8.5 The Global Interpreter Lock (GIL)

The *CPython* interpreter assures that only one thread executes Python bytecode at a time, thanks to the **Global Interpreter Lock**. The current thread must hold the **GIL** before it can safely access Python objects. This simplifies the CPython implementation by making the object model (including critical built-in types such as **dict**) **implicitly safe** against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, **at the expense of much of the parallelism afforded by multi-processor machines**.

However the GIL can degrade performance even when it is not a bottleneck. The **system call overhead** is significant, especially on multicore hardware. Two threads calling a function may take twice as much time as a single thread calling the function twice. The GIL can cause I/O-bound threads to be scheduled ahead of CPU-bound threads. And it prevents signals from being delivered. Some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing. Also, **the GIL is always released when doing I/O**.

Past efforts to create a "free-threaded" interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case. It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain. Guido van Rossum has said he will reject any proposal in this direction that slows down single-threaded programs.