\\introduction

21st century: complex software? =
What/Who is a Software Architect? =
Run-Time Systems and the JVM =
Software/Application Framework =
Framework Features =
OO Software Framework =
Examples of Frameworks =
Framework Design =

\\Languages and Abstract Machines. Compilation and interpretation schemes

What is the definition of Programming Languages? =
What are a Programming Paradigms? How many types exist and how to implement one? =
How to implement a Programming Language? =
What is an Abstract Machines? How to implement it? =
The general structure of the Interpreter
Execution models: What is Pure Interpretation/Compilation?
Virtual Machines as Intermediate Abstract Machines =
What are the advantages of the intermediate abstract machine (examples for JVM) =
Other Compilation Schemes

\\Runtime Systems and the JVM

a. JVM internals

Runtime system =
What is the JVM? =
What is the execution model of a JVM? =
JVM Data Types =
Object Representation
Threads =
Per Thread Data Areas =
Structure of frames =
Dynamic Linking =
Data Areas Shared by Threads: (Non)Heap =
JIT compilation =
Method Area/data/code =
Class file structure + FiledType Descriptor
Disassembling Java files: javac, javap, java
SimpleClass.class: constructor and method
The constant pool =
Loading, Linking, and Initializing =
JVM Startup =
ClassLoader Hierarchy =
Runtime Constant Pool =
Linking =
Verification Process
Initialization =
Finalization =
JVM Exit =

b. The JVM Instruction Set

JVM interpreter loop =
JVM Instruction Set and properties =
Runtime memory =
JVM Addressing Modes =
Instruction-set: typed instructions =
Opcode “pressure” and non-orthogonality =
Specification of an instruction
Compiling Constants, Local Variables, and Control Constructs
int vs. double: lack of opcodes for double requires longer bytecode
Accessing literals in the Constant Pool
Parameter passing: Receiving Arguments
Invoking Methods
Working with objects
Accessing fields (instance variables)
Compiling switches
Throwing Exceptions
JVM limits =

\\ Software Components

Why component-based software? =
Software components features =
Contract =
Explicit context-dependency =
What does it mean deployed independently =
Basic concepts of a Component Model =
What about Modules? (before components)
Scoping Rules for Modules =
Modules vs. Components =
What are Components? =
OOP vs COP =
Component Forms =
Some successful components =

b. Software Components: JavaBeans

Components in Java EE
JavaBeans and JavaBeans API
JavaBeans common features, What kind of properties a bean can have? =
Design time vs. run-time =
Java Beans introspection =
Design Patterns =
Show a Pattern Template =
The 23 Design Patterns of the Gang of Four
A Sample Pattern: Singleton (Creational) =
Pro and cons of Singleton =
Connection-oriented programming and Observer pattern =
Event Adaptors =
What is a Bound/Constrained Property? =
What is Java EE and its architecture?, How can we interact with java beans? =
Java EE Application Servers =
Java EE Containers =
Life Cycle of Java EE Application (RIVEDO)
Dynamic web contents =
Java Servlet/Java Server Pages (JSP) =
Enterprise Java Beans 3.2 (2013) =
Pro and cons of EJB =
EJB: Session Beans, How can we interact with java beans? =
EJB: Message Driven Beans =
different interfaces of Session Beans =
Life Cycle of Stateless/Stateful Session and Message-Driven Beans =
Java Persistence API (JPA) =

c. Reflection in Java

What is Reflection? =
What is Introspection/Intercession? =
Structural and behavioral reflection =
Which Uses of Reflection? =
Drawbacks of Reflection =
Reflection in Java =
How could you retrieve a Class Objects? =
Class object =
What can you obtain inspecting a Class? =
How to Discover Class members? =
How can you Work with Class members? =
How to use Reflection for Program Manipulation? =
Accessible Objects =

d. Annotations in Java

Annotations =
Structure of Annotations =
Which elements can be annotated? =
Predefined annotations =
Define and use your own annotations =
Recovering annotations through the Reflection API =

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

Inversion of control =
Frameworks vs Libraries =
Components, Containers, and IoC =
Loosely coupled systems: advantages and techniques
Dependency injection (RIVEDO) =
Data Access Object (DAO)
ServiceLocator =
ServiceLocator vs Dependency Injection =
Who creates the dependencies? What if we need some initialization code that must be run after dependencies have been set? What happens when we don’t have all the components? =
Four levels for understanding frameworks
Some terminology…=
Frozen Spot
Hot Spot
Which are the principles for Framework Construction?
Template Method DP =
Strategy DP (behavioral) =
Unification vs. separation principle/Template method vs. Strategy DP =

\\ Polymorphism

a. A classification

Universal vs. ad hoc polymorphism =
Binding time =
Classification of Polymorphism
Ad hoc polymorphism: overloading =
Universal polymorphism: Coercion =
Inclusion polymorphism =
Overriding =

b. Polymorphism in C++: inclusion polymorphism and templates

Parametric polymorphism/generic programming, What happens with generic at runtime/after compilation?
Function Templates in C++ =
Templates vs Macros in C++ =
Template (partial) specialization =
C++ Template implementation and instantiation

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

Java Generics: Explicit Parametric Polymorphism
Generic methods =
How many Type Bounds?
Explain the concept (with written example) of covariance and contravariance in a language with universal polymorphism and explain in what cases their use is safe =
Start by Java rules =
Limitations of Java Generics, What are the problems between arrays and generics in Java? =
A digression: Java arrays =
Recalling "Type erasure" =
Wildcards for covariance =
The “PECS principle” =

d. The Standard Template Library

Main entities in STL =
Iterators in Java =
C++ namespaces =
Categories/Classifying iterators =
Iterator validity =
Iterator Limits =
Which language mechanisms it is based STL upon? =
Iterators: small struct =
STL Inheritance =
Inlining =
Memory management =
STL problem =

\\ Functional Programming

Functional Programming Concepts =
The LISP family of languages =
Haskell, Lazyness in haskell
Applicative and Normal Order evaluation
Describe the different kind of parameter passing strategies

d. Call by sharing, by name, and by need

L-Values vs. R-Values and Value Model vs. Reference Model
References and pointers =
Parameter Passing by Sharing =
Call by name & Lazy evaluation =

\\Haskell

Laziness =
List Comprehensions
Datatype Declarations
Datatypes and Pattern Matching
Case Expression
From loops to recursion =
Higher-Order Functions =
the map/filter/reduce combinator =
On efficiency =
Tail-Recursive Functions =
Tail-Call Optimization =

d. Type classes in Haskell

Polymorphism in Haskell
Ad hoc polymorphism: overloading =
Type Classes =
Intuition =
Extra parameter dictionary =
Subclasses

e. The Maybe constructor and composition of partial functions

Type Constructor Classes =
What is "Functor" in haskell?
Constructor Classes
What relationships there are between functor and maybe type class?
The Maybe type constructor =
Composing partial function =
Monads as containers =

f. Monads in Haskell

Pros of Functional Programming =
The IO Monad =
Implementation of the IO monad =
The Bind Combinator (>>=)
The then (>>) Combinator
The “do” Notation
References
Monad (I/O) Restriction
Comparison

\\ Functional programming in Java 8

Lambdas in Java=
Implementation of Java 8 Lambdas, How are lambda expressions implemented? How the java compiler manages lambdas? =
What are Functional Interfaces ?=
Default Methods =
From Lambdas to Bytecode, How the java compiler manges lambdas? =

b. The Stream API in Java 8

Pipelines =
Anatomy of the Stream Pipeline =
Stream sources
Intermediate Operations
Terminal Operations
Types of Streams
Differences between reduce in functional programming and collect in the Java Stream API
From Reduce to Collect: Mutable Reduction
Infinite Streams
Parallelism
When to use a parallel stream?

\\ Scripting Languages and Python

Characteristics of Scripting Languages
Problem Domains
Language features
Why Python?, Is Python more OO or more functional, according to your opinion?
Dynamic typing
Useful commands
The dir() Function
Import and Modules
Whitespace
Assignment
Sequence Types
Negative indices
Slicing: Return Copy of a Subset (1)
The ‘in’/+ Operator
Operations on Lists Only
Tuples vs. Lists
Dictionaries: Like maps in Java
Creating and accessing dictionaries
Updating Dictionaries
Removing dictionary entries
Assert
The range() function
List Comprehensions
Filtered List Comprehension
Functions in Python
Decorators
Namespaces and Scopes
OOP in Python
Defining a class (object)
Creating a class instance
String representation
(Multiple) Inheritance
Encapsulation (and "name mangling"/storpiatura)
Static and Class methods, What does the @staticmethod decorator do?
What are iterators?
Generators and coroutines

e. The Global Interpreter Lock (GIL).

Garbage collection in Python
Handling reference counters
The Global Interpreter Lock (GIL)
Alternatives to the GIL?
Builtins & Libraries
Differences between component, packages and classes
Scoping Rules for Modules
What are Components?

\\introduction 02

21st century: complex software? = CLI and single computer are not enough, and data becomes from multiple sources. We are moving from Object-Oriented Model to Component Oriented Model which allows structuring a complex application into pieces and reusable components. Key ingredients are: Advanced features, extending programming languages; Component models, to ensure reusability; Frameworks, to support efficient development of (component-based) applications; Execution environments, runtime support for dynamic systems

What/Who is a Software Architect? = A new role which creates/defines/chooses an application framework, creates the component design according to a component model, understand the interactions and dependencies among components, select the environment based on cost/performance criteria, organize the development process.

Run-Time Systems and the JVM = RTSs provide a Virtual Execution Environment (e.g JVM) interfacing a program in execution with the OS, with support for Memory Management, Thread Management, Exception Handling, Security, AOT and JIT Compilation,…

Software/Application Framework = Software Framework is a collection of common code providing generic functionality that can be overridden. While Application Framework is a software framework used to implement the standard structure of an application for a specific development environment.

Framework Features = They need to provide reusable code wrapped in a well-defined API, helps to solve recurring design problems, drives solutions with default behavior. Unlike in libraries, the program's flow of control is not dictated by the caller, but by the framework.

OO Software Framework = They consist of a set of abstract classes, by inheriting from pre-existing classes in the framework and composing and subclassing the existing classes

Examples of Frameworks = General software frameworks: .NET, Windows platform. Provides language interoperability; Android SDK, supports the development of apps in Java (but does not use a JVM!); Spring, Cocoa, … . With GUI: Gnome, Written in C; mainly for Linux Web Application Frameworks [based on Model-View-Controller design pattern]; ASP.NET, by Microsoft for web sites, web applications, and web services; Google Web Toolkit (GWT), Rails, … . Concurrency: Hadoop Map/Reduce - software framework for applications that process big amounts of data in-parallel on large clusters in a fault-tolerant manner.

Framework Design = Requires a deep understanding of the problem domain, mastering of software (design) patterns, OO methods and polymorphism in particular.

\\Languages and Abstract Machines. Compilation and interpretation schemes 03

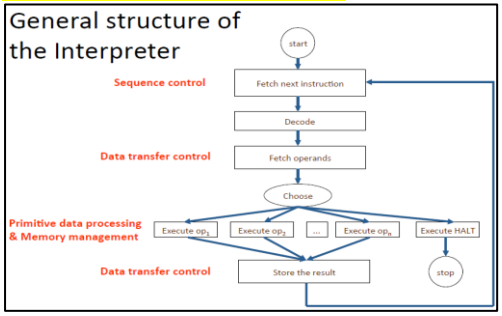
What is the definition of Programming Languages? = A PL is defined via: **Syntax**, so how expressions, commands, declarations, and other constructs must be arranged to make a well-formed program and Used by the compiler for scanning and parsing; **Semantics**, how a program may be expected to behave when executed on a computer; **Pragmatics**, the way in which the PL is intended to be used, including coding conventions and guidelines for structuring of code.

What are a Programming Paradigms? How many types exist and how to implement one? = A paradigm is a style of programming, characterized by a particular selection of key concepts and abstractions. Examples are: Imperative, OO, Concurrent, Functional, Logic programming.

How to implement a Programming Language? = Every language L (must be executable) implicitly defines an Abstract Machine ML having L as machine language. So, Implementing the abstract machine (ML) on an existing host machine (MO) makes programs written in L executable.

What is an Abstract Machines? How to implement it? = Given a programming language L, an Abstract Machine ML is a collection of data structures and algorithms which can perform the storage and execution of programs written in L. Each abstract machine can be implemented in hardware or in firmware, but if high-level this is not convenient in general. An Abstract machine M can be implemented over a host machine MO using data structures and algorithms implemented in the (AbstMach) ML of the latter. If the interpreter of the abstract machine M coincides with the interpreter of host machin (HostMach) MO, then M is an extension of (HostMach) MO and other components of the machines can differ. While, if The interpreter of M is different from the interpreter of (HostMach) MO, then M is interpreted over (HostMach) MO other components of the machines may coincide.

The general structure of the Interpreter

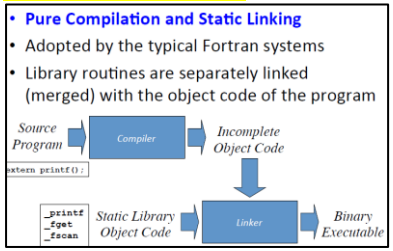


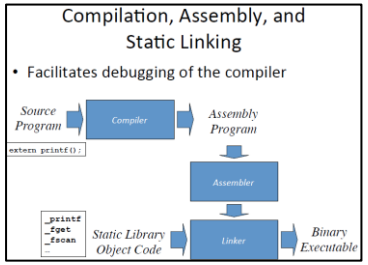
Execution models: What is Pure Interpretation/Compilation? = (AbstMach) ML is interpreted over (HostMach) MO: every instruction of (AbstMach) ML is translated in real-time into a set of instructions of (HostMach) MO. It is not very efficient but allows to run programs on-the-go, facilitating interactive debugging and testing. Programs written in L are translated into equivalent programs written in LO, the machine language of (HostMach) MO. The translated programs can be executed directly on (HostMach) MO, hence (AbstMach) ML is not realized at all. The execution becomes more efficient, but the produced code is larger. Implementations of programming languages use both: compilation (= translation) from external to internal representation and interpretation for I/O operations (runtime support). It can be modeled by identifying an Intermediate Abstract Machine MI with language LI where a program in L is compiled to a program in LI which will be executed by an interpreter for MI.

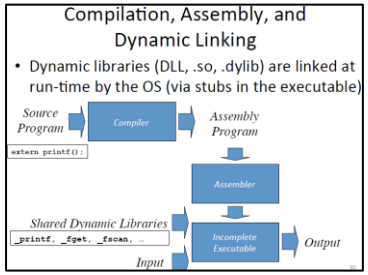
Virtual Machines as Intermediate Abstract Machines = Adopted by Pascal, Java, functional, and script languages. The compiler generates an intermediate program, which will run on the VM.

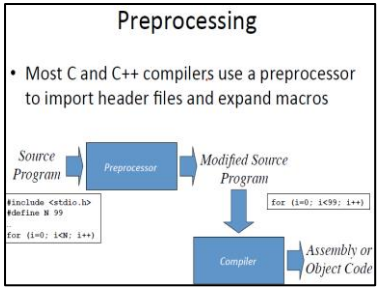
What are the advantages of the intermediate abstract machine (examples for JVM) = Portability: Compile the Java source, distribute the bytecode and execute on any platform equipped with JVM. Interoperability: for a new language L, just provide a compiler to JVM bytecode; then it could exploit Java libraries.

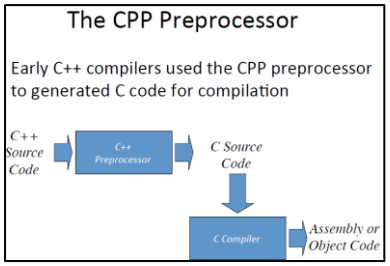
Other Compilation Schemes











\\Runtime Systems and Introduction to the JVM 04-06 a. JVM internals

Runtime system = Every programming language defines an execution model implemented by a runtime system, providing support during the execution. It can be made of code in the executing program, language libraries, OS functionalities, the interpreter/virtual machine itself. For instance, Java Runtime Environment includes JVM and a Java Class Library to run compiled Java programs.

What is the JVM? = The JVM is an AM. Its specification does not give implementation details like garbage-collection algorithm or internal optimization. Thanks to a machine-independent "class file format" that all JVM must support, the JVM imposes strong syntactic and structural constraints on the code in a class file.

What is the execution model of a JVM? = JVM is a multi-threaded stack-based machine. JVM instructions work implicitly taking arguments from the top of the operand stack to put their result on the top. The latter is used to return a result from a method, store intermediate results while evaluating expressions, store local variables.

JVM Data Types = it has Primitive types like:

- numeric integral: byte, short, int, long, char
- numeric floating point: float, double
- Boolean (support only for arrays)
- internal, for exception handling: returnAddress

And Reference types like class types, array and interface types. All with No-type information on local variables at runtime and types of operands specified by opcodes (eg: iadd, fadd,.)

Object Representation = Left to the implementation, including a concrete value of null. This adds an extra level of indirection with pointers to instance data and class data, making garbage collection easier. The representation must include mutex lock and GC state (flags).

Threads = JVM allows multiple threads per application. Created as instances of Thread invoking start() (which invokes run()). There are some (daemon) system threads for GC, Signal dispatching, Compilation,…. They have shared access to the heap and persistent memory.

Per Thread Data Areas = it has: a Program Counter, pointer to next instruction in method area; The java stack, a stack of frames (or activation records), created/destroyed to each method invoked and which does not require that frames are allocated contiguously; and The native stack, used for the invocation of native functions, through the Java Native Interface

Structure of frames = It has: a Local Variable Array (32 bits) containing reference to this (if instance method), method parameters, local variables; Operand Stack to support evaluation of expressions and method; and a Reference to Constant Pool of the current class

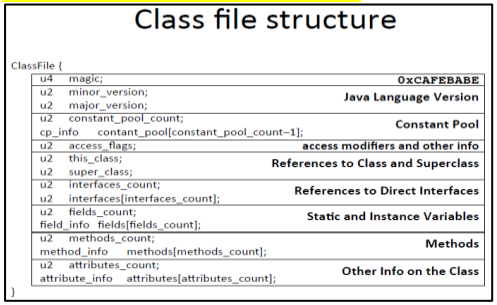
Dynamic Linking = In Java this is done dynamically at runtime: during compilation, all references to variables and methods are stored in the class's constant pool as symbolic references. These can be resolved by the JVM implementation with Eager/static resolution (when the class file is verified after being loaded) or Lazy/late resolution (when the symbolic reference is used for the first time). Binding is the process of the entity (field, method or class) identified by the symbolic reference being replaced by a direct reference. If the symbolic reference refers to a class that has not yet been resolved then this class will be loaded.

Data Areas Shared by Threads: (Non)Heap = It is a memory for objects and arrays. Explicit deallocation is not supported, only by garbage collection. The HotSpot JVM includes four Generational Garbage Collection Algorithms (Z GC). Non-Heap is a memory for objects which are never deallocated, needed for the JVM execution such as method area, interned strings, code cache for JIT.

JIT compilation = JVM profile the code during interpretation, looking for "hot" areas of byte code that are executed regularly. These are compiled to native code then stored in the code cache in non-heap memory.

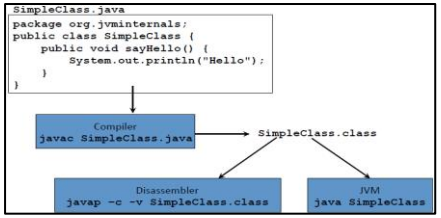
Method Area/data/code = The Method area is the memory where class files are loaded with a Classloader Reference for each, and a Run Time Constant Pool, Field data, Method data, Method code from class file. Method area is shared among threads with a thread-safe access. Changes when a new class is loaded or a symbolic link is resolved by dynamic linking.
Data Per method: Name, Return Type, Parameter Types (in order), Modifiers, Attributes, Method code… A method descriptor contains a sequence of zero or more parameter descriptors and a return descriptor, or V for void descriptor:
Object m(int i, double d, Thread t) {…}
=>
(IDLjava/lang/Thread;)Ljava/lang/Object;
Code Per method: bytecodes, Operand stack size, local variable size, local variable table, exception table, LineNumberTable (which line of source code corresponds to which byte code instruction).

Class file structure + FieldType Descriptor

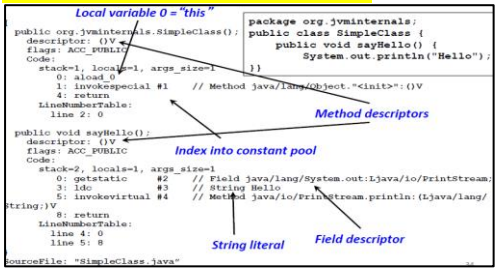


FieldType term	Type	Interpretation
B	byte	signed byte
C	char	Unicode character code point in the Basic Multilingual Plane, encoded with UTF-16
D	double	double-precision floating-point value
F	float	single-precision floating-point value
I	int	integer
J	long	long integer
L *ClassName* ;	reference	an instance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

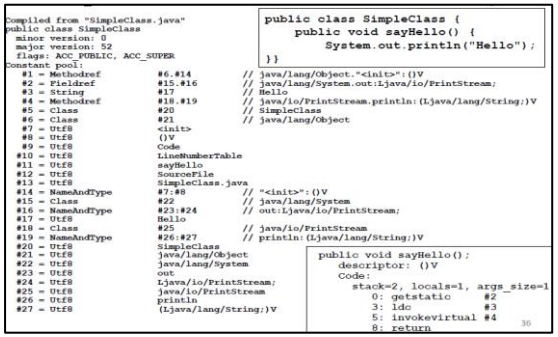
Disassembling Java files: javac, javap, java



SimpleClass.class: constructor and method



The constant pool = Similar to symbol table, but with more info. Contains constants and symbolic references used for dynamic binding, numeric literals (Integer, Float, Long, Double), string literals (Uf8), class references (Class), field references (Fieldref), method references (Mehodref, InterfaceMethodref, MethodHandle), signatures (NameAndType). Operands in bytecodes often are indexes in the constant pool.



Loading, Linking, and Initializing = Loading means finding the binary representation of a class or interface type with a given name and creating a class or interface from it. Linking takes a class or interface and combines it into the run-time state of the Java Virtual Machine so that it can be executed. Initialization executes the class or interface initialization method <clinit>.

JVM Startup = The JVM starts up by loading/linking/initializing an initial class using the

bootstrap classloader. *public static void main(String[])* is invoked. Loading, Class or Interface C creation is triggered either by other class or interface referencing C and by certain methods (eg. reflection). Check whether already loaded and invoke the appropriate loader.loadClass. Each class is tagged with the initiating loader. Loading constraints are checked during loading, to ensure that the same name denotes the same type in different loaders

ClassLoader Hierarchy = Bootstrap Classloader loads basic Java APIs skipping much of the validation that gets done for normal classes. Extension Classloader loads classes from standard Java extension APIs such as security extension functions. System Classloader, the default application classloader, which loads classes from the classpath. User-Defined Classloaders can be used to load application classes.

Runtime Constant Pool = The constant_pool table in the .class file is used to construct the run-time constant pool with all the references initially symbolic. Class names are those returned by Class.getName(). Field and method references are made of name, descriptor, and class name.

Linking = Link = verification during the load and link process because no guarantee that the class file was generated by a Java compiler.

Verification Process
The first step starts when the class file is loaded, the file is properly formatted, and all its data is recognized by the JVM. Then, when the class file is linked, all checks that do not involve instructions and, (pass 3) data-flow analysis on each method, to ensure that at any given point in the program, no matter what code path is taken to reach that point:

- The operand stack is always the same size and contains the same types of objects.
- No local variable is accessed unless it is known to contain a value of an appropriate type.
- Methods are invoked with the appropriate arguments.
- Fields are assigned only using the values of appropriate types.
- All opcodes have appropriate type arguments on the operand stack and in the local variables
- A method must not throw more exceptions than it admits
- A method must end with a return value or throw instruction
- Method must not use one half of a two-word value

The last step is when the first time a method is invoked, a virtual pass whose checking is done by JVM instructions: the referenced method or field exists in the given class and the currently executing method has access to theirs.

Initialization = <clinit> initialization method is invoked on classes and interfaces to initialize class variables. Direct superclass needs to be initialized prior. <init>: initialization method for uninitialized instances, **invokespecial** instruction. For instance:

```
class Super {
    static { System.out.print("Super ");}
}
class One {
    static { System.out.print("One ");}
}
class Two extends Super {
    static { System.out.print("Two ");}
}
class Test {
  public static void main(String[] args) {
    One o = null;
    Two t = new Two();
    System.out.println((Object)o == (Object)t);
  }
}
```

What does java Test print? Super Two False

Finalization = Invoked before GC without specification for "when invoked" and "upon which thread". Each object can be reachable/finalizer-reachable/unreachable and unfinalized/finalizable/finalized.

JVM Exit = classFinalize is similar to object finalization. A class can be unloaded when doesn't exist instances or a class object is unreachable. JVM exits when all its non-daemon threads terminate and/or Runtime.exit/System.exit is secure.

b. The JVM Instruction Set

JVM interpreter loop =

```
do {
        atomically calculate pc and fetch opcode at
pc;
        if (operands) fetch operands;
        execute the action for the opcode;
} while (there is more to do);
```

JVM Instruction Set and properties = it is a 32-bit stack machine, with variable length instruction set with One-byte opcode followed by arguments, Symbolic references, a mix of Compactness vs. performance. JVM Instruction Set are: Load and store, Arithmetic, Type conversion, Object creation/manipulation, Operand stack manipulation, Control transfer, Method invocation and return, Monitor entry/exit. Each instruction may have different "forms" supporting different kinds of operands, e.g. different forms of "iload"

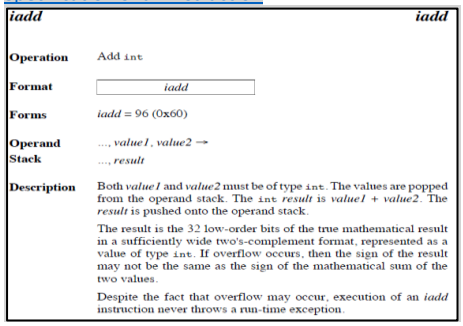
Runtime memory = is composed of: Local variable array (frame), Operand stack (frame), Object fields (heap), Static fields (method area). JVM instructions take arguments from the top of the operand stack to put their result on the top. The latter is used to return a result from a method, store intermediate results while evaluating expressions, store local variables.

JVM Addressing Modes = JVM supports three addressing modes:
– **Immediate**, Constant is part of instruction
– **Indexed**, accessing variables from the local variable array, which means 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 top of the operand stack and places it in the args/locals area
– **Stack**, retrieving values from operand stack using pop

Instruction-set: typed instructions = JVM instructions are explicitly typed: different opCodes for integers, floats, arrays, reference types,… (i/l/f/d/aload)

Opcode "pressure" and non-orthogonality = Since op-codes are bytes, there are at most 256 distinct ones, which mean that it is impossible to have an opcode/instruction, so non-supported types have to be converted, which mean: non-orthogonality of the Instruction Set Architecture

Specification of an instruction



Compiling Constants, Local Variables, and Control Constructs

- Sample Code
- Can compile to

```
void spin() {
        int i;
        for (i = 0; i < 100; i++) {
            ; // Loop body is empty
        }
}
```

```
0 iconst_0
1 istore_1
2 goto 8
5 iinc 1 1
8 iload_1
9 bipush 100
11 if_icmplt 5
14 return
```

```
// Push int constant 0
// Store into local variable 1 (i=0)
// First time through don't increment
// Increment local variable 1 by 1 (i++)
// Push local variable 1 (i)
// Push int constant 100
// Compare and loop if less than (i < 100)
// Return void when done
```

- Pushing constants on the operand stacks
- Incrementing local variable, comparing

int vs. double: lack of opcodes for double requires longer bytecode

- Sample Code
- Can compile to

```
void dspin() {
        double i;
        for (i = 0.0; i < 100.0; i++) {
            ;   // Loop body is empty
        }
}
```

```
0 dconst_0
1 dstore_1
2 goto 9
5 dload_1
6 dconst_1
7 dadd
8 dstore_1
9 dload_1
10 ldc2_w #4
13 dcmpg
14 ifLt 5
17 return
```

```
// Push double constant 0.0
// Store into local variables 1 and 2
// First time through don't increment
// Push local variables 1 and 2
// Push double constant 1.0
// Add; there is no dinc instruction
// Store result in local variables 1 and 2
// Push local variables 1 and 2
// Push double constant 100.0
// There is no if_dcmplt instruction
// Compare and loop if less than(i < 100.0)
// Return void when done
```

Accessing literals in the Constant Pool

Sample Code

Can compile to

```
void useManyNumeric() {
        int i = 100;
        int j = 1000000;
        long l1 = 1;
        long l2 = 0xffffffff;
        double d = 2.2;
        ...do some calculations...  }
```

```
0 bipush 100
2 istore_1
3 ldc #1
5 istore_2
6 lconst_1
7 lstore_3
8 ldc2_w #6
11 lstore 5
13 ldc2_w #8
16 dstore 7
...
```

```
// Push small int constant with bipush
// Push large int (1000000) with ldc
// A tiny long value uses fast lconst_1
// Push long 0xffffffff (that is, int -1)
// Any long can be pushed with ldc2_w
// Push double constant 2.200000
...do those calculations…
```

Parameter passing: Receiving Arguments

- Sample Code
- Can compile to

```
int addTwo(int i, int j) {
        return i + j;
}
```

```
0 iload_1
1 iload_2
2 iadd
3 ireturn
```

```
// Push value of local variable 1 (i)
// Push value of local variable 2 (j)
// Add; leave int result on operand stack
// Return int result
```

- Local variable 0 used for **this** in instance methods
- Sample Code
- Can compile to

```
static int addTwo(int i, int j) {
        return i + j;
}
```

```
0 iload_0
1 iload_1
2 iadd
3 ireturn
```

Invoking Methods

- Sample Code
- Can compile to

```
int add12and13() {
        return addTwo(12, 13);
}
```

```
0 aload_0          // Push local variable 0 (this)
1 bipush 12        // Push int constant 12
3 bipush 13        // Push int constant 13
5 invokevirtual #4 // Method Example.addtwo(II)I
8 ireturn          // Return int on top of operand stack;
                   // it is the int result of addTwo()
```

invokevirtual causes the allocation of a new frame, pops the arguments from the stack into the local variables of the caller (putting **this** in 0) and passes the control to it by changing the **ps.** 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. **return** just passes the control to the caller.
invokestatic – for calling methods with "static" modifiers – **this** is not passed
invokespecial – for calling constructors (not dynamically dispatched), private/superclass methods – **this** is always passed
invokeinterface – used when the called method is declared in an interface
invokedynamic –support dynamic typing

Working with objects

- Sample Code
- Can compile to

```
Object create() {
        return new Object();
}
```

```
0 new #1           // Class java.lang.Object
3 dup
4 invokespecial #4 // Method java.lang.Object.<init>()V
7 areturn
```

Objects are manipulated like primitive types, but through references using the corresponding instructions (e.g. areturn)

Accessing fields (instance variables)

- Sample Code
- Can compile to

```
void setIt(int value) {
    i = value;
}
int getIt() {
    return i;
}
```

```
Method void setIt(int)
0 aload_0
1 iload_1
2 putfield #4 // Field Example.i I
5 return
Method int getIt()
0 aload_0
1 getfield #4 // Field Example.i I
4 ireturn
```

Requires resolution of the symbolic reference in the constant pool. Similar for **static variables**, using **putstatic** and **getstatic.**

Compiling switches

Compiling switches (1)

- Sample Code
- Can compile to

```
int chooseNear(int i) {
   switch (i) {
      case 0:  return  0;
      case 1:  return  1;
      case 2:  return  2;
      default: return -1;
} }
```

```
0 iload_1
1 tableswitch 0 to 2:
      0: 28
      1: 30
      2: 32
      default:34
28 iconst_0
29 ireturn
30 iconst_1
31 ireturn
32 iconst_2
33 ireturn
34 iconst_m1
35 ireturn
```

```
// Push local variable 1 (argument i)
// Valid indices are 0 through 2
// If i is 0, continue at 28
// If i is 1, continue at 30
// If i is 2, continue at 32
// Otherwise, continue at 34
// i was 0; push int constant 0...
// ...and return it
// i was 1; push int constant 1...
// ...and return it
// i was 2; push int constant 2...
// ...and return it
// otherwise push int constant -1...
// ...and return it
```

Operation Access jump table by index and jump

Format

tableswitch
<0-3 byte pad>
defaultbyte1
defaultbyte2
defaultbyte3
defaultbyte4
lowbyte1
lowbyte2
lowbyte3
lowbyte4
highbyte1
highbyte2
highbyte3
highbyte4
jump offsets...

Forms *tableswitch* = 170 (0xaa)

Operand Stack ..., *index* →

...

A tableswitch is a variable-length instruction. The index must be int and is popped from the operand stack. If the index is less than low or greater than high, then a target address is calculated by adding default to the address of the opcode of this tableswitch instruction. Otherwise, the offset at position index - low of the jump table is extracted. The target address is calculated by adding that offset to the address of the opcode of this tableswitch instruction.
lookupswitch is used when the cases of the switch are sparse: each case is a pair <value: address>, instead of an offset in the table of addresses. Cases are sorted, so a binary search can be used. Only support for int, other types need a translation.

Throwing Exceptions

- Sample Code
- Can compile to

```
void cantBeZero(int i) throws TestExc
{
        if (i == 0) {
            throw new TestExc();
        }}
```

```
0 iload_1          // Push argument 1 (i)
1 ifne 12          // If i==0, allocate instance and throw
4 new #1           // Create instance of TestExc
7 dup              // One reference goes to its constructor
8 invokespecial #7 // Method TestExc.<init>()V
11 athrow          // Second reference is thrown
12 return          // Never get here if we threw TestExc
```

try-catch

- Sample Code
- Can compile to

```
void catchOne() {
        try {
            tryItOut();
        } catch (TestExc e) {
            handleExc(e);
        }}
```

```
0 aload_0          // Beginning of try block
1 invokevirtual #6 // Method Example.tryItOut()V
4 return           // End of try block; normal return
5 astore_1         // Store thrown value in local var 1
6 aload_0          // Push this
7 aload_1          // Push thrown value
8 invokevirtual #5 // Invoke handler method:
                   // Example.handleExc(LTestExc;)V
11 return          // Return after handling TestExc

Exception table:
From To Target  Type
0    4  5       Class TestExc
```

- Compilation of **finally** more tricky

athrow looks in the method for a catch block using the exception table and, if it exists, the stack is cleared and control passed to the first instruction, otherwise, the current frame is discarded and the same exception is thrown on the caller. Compiles a catch clause like another method. The table records boundaries of **try** and is used by **athrow** to dispatch the control.

JVM limits = Max number of entries in the constant pool, number of fields/methods/direct superinterfaces/local variables in the local variables array of a frame, operand stack size, length of field and method names (in char): 65535, Max number of parameters of a method, dimensions in an array: 255

\\ Software Components 07…12
a. An introduction to Software Components
Software Components = a composition unit with specified interfaces and explicit context dependencies only. It can be deployed independently and is subject to composition by third parties. With composition unit would mean: black boxes, not source code, no (externally) observable state, indistinguishable from copies.

Why component-based software? = is a system made of components, gaining more reuse instead to create. Why? Thanks to the cost of software development, but system requirements can force the use of certified components, and raising of a component marketplace and of distributed and concurrent systems.

Software components features = **Modular**: compatible (chips or boards that plug in easily), reusable (same processor IC can serve various purposes), extendible (IC technology can be improved: inheritance)
Reliable (an IC works most of the time): robust (it functions in abnormal conditions)
Efficient (ICs are getting faster and faster!)
Portable (ease of transferring to different platforms)
Timely (released when/before users want it)

Contract = A contract is a specification attached to an interface that mutually binds the clients and providers of the components, with pre/post-conditions for the operations specified by API. It requires an Interface Definition Language (IDL) specifying how the component can be deployed/instantiated and how the instances behave through the advertised interfaces.

Explicit context-dependency = is a specification of the deployment/run-time environment, e.g.which tools/platforms/resources/other components are required.

What does it mean deployed independently = The component can be plugged into a system or composed of other components by third parties, not aware of the internals of the component.

Basic concepts of a Component Model = are
Component interface describes the operations (method calls, messages, . . .) that a component may use; Composition mechanism, the manner in which different components can be composed to work, e.g. using message passing; Component platform for the development/execution of components.

What about Modules? (before components) = Modules are the main feature of programming languages for supporting the development of large applications with encapsulation of information, reducing risks of name conflicts and support the integrity of data abstraction. Teams of programmers can work on separate modules in a project. No language support for modules in C and Pascal. A module interface specifies exported variables, data types, and subroutines. The module implementation is compiled separately and implementation details are hidden from the user of the module. Various mechanisms to get module instances are when Modules act as a manager/type. Many OO languages support a notion of Module (packages) independent from classes.

Scoping Rules for Modules = Modules encapsulate variables, data types, and subroutines in a package, which are visible to each other and not visible outside

unless exported: outside are visible [open scopes], or are not visible inside unless imported [closed scopes], or are visible with "qualified name" [selectively open scopes].

Modules vs. Components = The main difference is that modules as part of a program, component as part of a system. Components can include static resources. Modules may expose observable state and include several concepts about components.

What are Components? = Component can be/contain (collections of) classes, objects, functions/algorithms, data structures, supporting unification of data and function, encapsulation, software entity has a unique identity, use of interfaces to represent specification dependencies. Components can be assembled according to the rules specified by the component model/through their interfaces. A Composition is a process of assembling components to form an assembly, a larger component/an application.

OOP vs COP = Object orientation is not primarily concerned with reuse, while component-based software engineering provides methods and tools for: building systems from components; building components as reusable units;
replacement/introducing of components into the system; system architecture detailed in terms of components

Component Forms = **Component Specification**, unit of software that describes the behavior of a set of Component Objects as a set of Interfaces and defines a unit of implementation. 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 does not mean that it is independent of other components or that it is a single physical item, such as a single file. **Installed Component, a**n installed/deployed copy of a Component Implementation 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, a**n instance of an Installed Component, an object with its own data and a unique identity. An Installed Component may have multiple Component Objects (which require explicit identification).

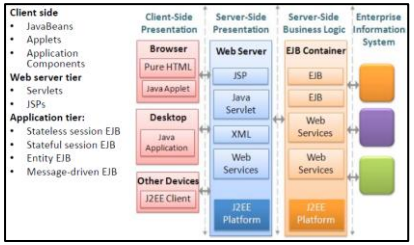
In all cases, there is an infrastructure providing rich foundational functionality.

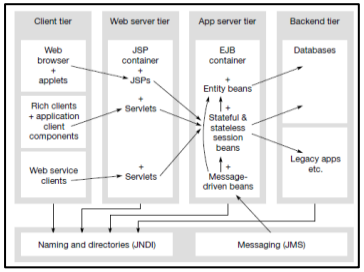
Some successful components = Even bigger components: Unix, a general-purpose OS, commands used as components in a shell-process and low-level (but well-defined) interfaces (file sharing, pipes, and filter), but it is not a real component behavior (difficult to replace or update). More recent components are Microsoft's Visual Basic and COM+, Java Beans, Enterprise JavaBeans, … In all cases, there is an infrastructure providing rich foundational functionality. Components can be purchased from independent providers and deployed by clients. Multiple components from different sources can coexist in the same installation.

b. Software Components: The Sun approach, JavaBeans

Components in Java EE

Java EE is a suite of specifications for application servers. Components in Java EE:





JavaBeans and JavaBeans API = A Java Bean is a reusable software component that has a GUI representation, but not necessarily (Invisible beans). Any Java class can be recognized as a bean in a tool provided that has a public default constructor (no arguments), implements the interface *java.io.Serializable* and is in a jar file with the manifest file containing the flag Java-Bean: True. As a Software Components, beans can be assembled to build a new bean/applet, writing glue code to wire beans together. Beans on a server are not visible. The goal was to define a software component model for Java, allowing vendors to create/ship Java components that can be composed together into applications. All with **granularity**: from small (eg. a button in a GUI) to medium (eg. a spreadsheet as part of al larger document); **portability**: in Java bridges defined to other component models (like OpenDoc, OLE/COM/ActiveX); **uniformity and simplicity** of API to be supported on different platforms.

JavaBeans common features, What kind of properties a bean can have? = Support for **properties**, both for customization and for programmatic use; Support for **events**: communication metaphor used to connect several beans; Support for **customization**: of appearance and behavior; Support for **persistence**: with its customized state saved away and reloaded later; Support for **introspection**: a builder tool can analyze how the bean works. Emphasis on GUI, but textual programming.

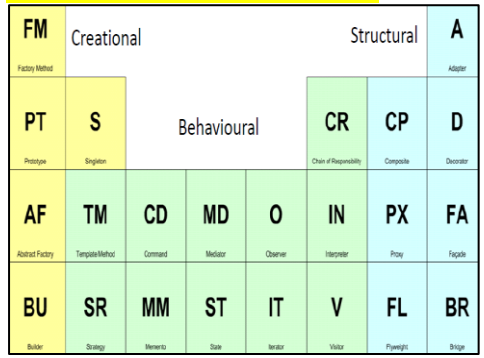
Design time vs. run-time = A bean must be able to run in the design environment of a builder tool providing means to the user to customize aspects and behavior. At run-time, there is less need for customization. A Possible solution: design-time information for customization is separated form run-time information, and not loaded at run-time.

Java Beans introspection = is the process of analyzing a bean to determine the capability, allowing the builder tool to present info about a component to software designers. Implicit method: based on reflection, naming conventions, and design patterns.

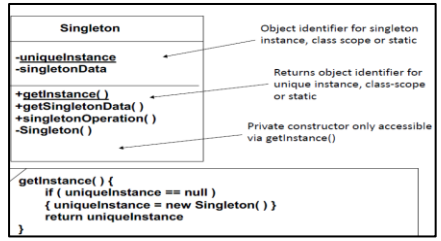
Design Patterns = concept for design a solution to a problem, A (software) design pattern is a general, reusable solution to a commonly occurring problem within a different abstract context in software design like abstraction, encapsulation, information hiding,… Also solve **non-functional** problems like changeability, interoperability, efficiency, reliability,…

Show a Pattern Template =
Name: meaningful text that reflects the problem, e.g. Bridge, Mediator, Flyweight
Problem addressed: the intent of the pattern, objectives achieved within certain constraints
Context: circumstances under which it can occur, to determine the applicability
Forces: constraints or issues that solution must address, forces may conflict!
Solution: the static and dynamic relationships among the pattern components.

The 23 Design Patterns of the Gang of Four



A Sample Pattern: Singleton (Creational) = In some applications, it is important to have exactly one instance of a class. It Can make an object globally accessible as a global variable and use class (static) operations and attributes without violating encapsulation and making polymorphic redefinition. The solution is to create a class with a class operation *getInstance*(). When class is first accessed, this creates relevant object instance and returns object identity to the client. On subsequent calls of *getInstance*(), no new instance is created, but the identity of the existing object is returned. Singleton Structure:



Pro and cons of Singleton = To specify a class has only one instance, we make it inherit from Singleton. The advantages are controlled access to single object instance through Singleton encapsulation. **Cons**: access requires additional message passing, pattern limits flexibility, plus significant redesign if singleton class later gets many instances.

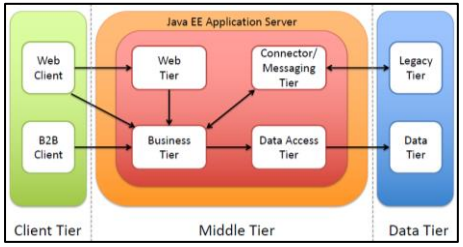
Connection-oriented programming and Observer pattern = is a paradigm for gluing together components in a builder tool, based on the Observer design pattern, adequate for GUIs. The Observer pattern (aka Publish-

Subscribe) define a one-to-many dependency among objects so that when one object changes state, all of its dependents are notified and updated automatically.

Event Adaptors = They are placed between the event source and a listener working as both by implementing an event queuing mechanism between sources and listeners, acting as a filter, demultiplexing multiple event sources onto a single event listener or acting as a generic "wiring manager" between sources and listeners.

What is a Bound/Constrained Property? = It can generate an event of type PropertyChangeEvent when the property is changed and sent to objects that previously registered for receiving such notifications. When implementing bound property on Bean, Helper classes simplify implementation. Constrained Property generates an event of type PropertyChangeEvent when an attempt is made to change its value, then the event is sent to objects that previously registered for receiving such notification. Those objects have the ability to veto the proposed change by raising an exception, allowing a bean to operate differently according to the runtime environment.

What is Java EE and its architecture?, How can we interact with java beans? = a "standard" platform for the development/execution/management of enterprise applications which are Multi-tier, Web-enabled, executed in a server-centric environment and component-based. Build on Java SE. Functionalities of the application are divided into 3 tiers: client/data/middle. The latter, which manages requests from clients and processes application data, is of interest for Java EE (Application Server).



The **Client Tier** includes the client applications that "use" the enterprise application by communicating with the Java EE Application Server. There are 2 types of client applications: Web Client, a web browser that makes requests via HTTP to the Web Tier, and B2B Client, 1+ applications that make requests to the Business Tier. The **Web Tier** consists of components that manage the interactions between the Web clients and Business Tier, serving dynamic generation of content for different clients, collection of input data that users send via the Web client interface, maintaining status for a user session. The **Business Tier** consists of components that provide the business logic of the application. The **Data Tier** refers to the various "data sources" from which the application can draw (MySQL, Oracle, SAP), all located on hosts other than the one on which the Java EE Application Server is running.

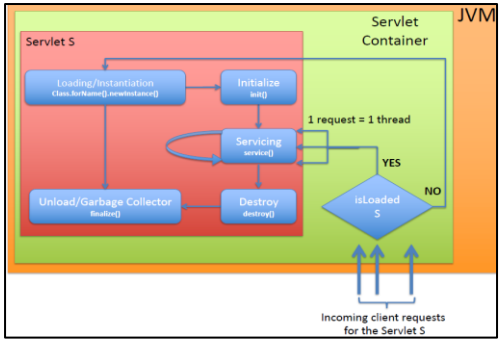
Java EE Application Servers = A Java EE Application Servers hosts the Middle Tier and provides the standard services in the form of a container, which means concurrency management, scalability, security, persistence. A "Famous" Java EE server is GlassFish.

Java EE Containers = They are an interface between an application component and the low-level features provided by the platform to support that component. There is one type of container for each type of component: **Web Container** is an interface between web components (Servlet/JSF/JSP page) and the web server, which manages the component's life cycle and dispatches requests to the various components of the application; **Application Client Container is an i**nterface between Java EE client applications and the Java EE server, running on client machines; **EJB Container** is a runtime environment that controls an EJB component instance and provides transactions/persistence/life cycle management services. They provide an interface for the outside world, isolating the EJB component from direct access.

Life Cycle of Java EE Application (RIVEDO)
Static content design (HTML, CSS, etc.),
Dynamic content development (Servlets, JSPs, EJtis, etc.), according to the parameters provided by the client's request.
Deployment descriptors (web.xml, application.xml, ejb-jar.xml, etc.), customizable files (XML-based) that contain the instruction for a given container on how to use/manage the Java EE components, guarantee the portability of the components.
Deployment packages (JAR, WAR, EAR, etc.) on Java EE server (e.g., Jtioss AS), Management of Java EE applications running on the server.

Dynamic web contents = It may vary according to the parameters provided by the client's request. Server-side scripting manages user sessions and controls the application flow with HTML form, parameters in the request URL,… Main server-side languages are Perl, PHP, Java, ASP

Java Servlet/Java Server Pages (JSP) = A servlet is a Java Component designed to handle a web request. It implements the interface *javax.servlet.Servlet* for life cycle methods. A Server request is handled by independent threads in the JVM. A Java Server Page (JSP) is an HTML page with Java scripts, which causes the execution of a servlet. The Servlet Container provides "low level" services needed for the Servlets and JSPs lifecycle such as HTTP connection management, sessions, threading, security. In response to a request from the client, the container check that the Servlet has already been loaded otherwise it will load the corresponding class and generate an instance by invoking the init method and invoke the service method corresponding to the Servlet instance, passing the objects representing the request and the response as arguments.



Enterprise Java Beans 3.2 (2013) = It is server-side components that implement the "business logic" of an application cooperating within a Java EE server. The relative "clients" are Web Tier Components, Remote client and Web Service Client. Two main components:
– Session Beans, perform the application logic, manage transactions/access control
– Message-Driven Beans, perform actions (asynchronously) in response to events

Pro and cons of EJB = It simplifies the development of large distributed enterprise applications. Developers have to focus only on the logic, simplifying the UI development and leaving the rest to the container, guarantying portability. The main **cons** were difficult to develop, deploy and test due to many file descriptors for configurations, automatic unit testing impossible. This has been changed since version 3 with annotations instead of interface implementation, adoption of Dependency Injection and asynchronous communication.

EJB: Session Beans, How can we interact with java beans? = are the reusable components that contain the application logic. The clients interact either locally or remotely via public methods. There are 2 types: Stateless, the most common EJB, performs a task on behalf of a client, maintains its status ONLY for the duration of the invocation. They can be "shared"; Stateful, despite leaving the state once the requests/bean ends, it may keep the status on secondary memory.

EJB: Message Driven Beans = Similar to Stateless Session Beans, the MDB process application messages in asynchronous mode without maintaining client status information. A single MDB can serve messages from multiple clients without exposing interfaces.

different interfaces of Session Beans = There are 3 possible interfaces: Remote, where clients may be running on different JVMs. "Position" of the EJB is transparent to the client; Local, where clients running on the same JVM. "Position" of the EJB must be that of the client (strong coupling); web service, only for "Stateless" Session Beans where clients are running on a different JVM and can be implemented in any programming language.

Life Cycle of Stateless/Stateful Session and Message-Driven Beans = **Stateless Session Beans**, the instances were created by the EJB container and kept in a "pool" of "ready" instances. Given a method call from a client, the container assigns one of the ready bean instances until the exit and returns to the pool. **Stateful Session Beans**, the client starts a session, the default bean constructor is invoked and the method annotated with the @PostConstruct tag is executed. From this point, it remains in the cache to perform other client requests. **Message-Driven Beans, t**he bean receives a message while the EJB container searches for an instance of the bean available in the pool. Once the execution of the onMessage() method is completed, the instance returns to the pool.

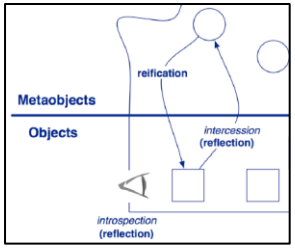
Java Persistence API (JPA) = They allow to automatically store data contained in Java objects on relational DB with Object-Relational Mapping (ORM). Applications can manage DB tables as "normal" Java objects. It has its own SQL-like syntax for static/dynamic queries and prevents the developer from writing "low level"

queries, providing caching services and performance optimization transparently.

c. Reflection in Java

What is 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. Another dimension of reflection is if a program is allowed to read-only, or also to change itself.

What is Introspection/Intercession? = 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 require a mechanism (reification) for encoding execution state as data.



Structural and behavioral reflection = 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.

Which Uses of Reflection? = Class Browsers, need to be able to enumerate the members of classes. Visual Development Environments, can exploit type information available in reflection to aid correct code. Debuggers, need to be able to examine private members on classes. Test Tools, can make use of reflection to ensure a high level of code coverage in a test suite.

Drawbacks of Reflection = If it is possible to perform an operation without using reflection, then it is preferable to avoid using it because Reflection brings: Performance Overhead, involving types that are dynamically resolved, thus JVM optimizations cannot be performed, and reflective operations have slower performance than their non-reflective counterparts; Security Restrictions, requires runtime permission which may not be present when running under a security manager; 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.

Reflection in Java = Java supports introspection and reflexive invocation, but not code modification. For every type the JVM maintains an associated object of class java.lang.Class, which "reflects" all relevant information such as class name & modifiers, superclass & Interfaces implemented, …

How could you retrieve a Class Objects? = Thanks to method Object.getClass() or field *.class* of a type (also primitive).

Class object = Instances of the class Class represent classes and interfaces in a running Java application. Class objects are constructed automatically by the JVM as classes are loaded, providing access to the information read from the class file

What can you obtain inspecting a Class? = After we obtain a Class object we can get the class name/class modifiers/the interfaces implemented by/the superclass

How to Discover Class members? = Class Methods for locating **Fields**:

- getDeclaredField(String name): Returns a Field object representing the field called name. Must belong to the class this and can be private.
- getField(String name): Returns a Field object representing the field called name. Must be public and can belong to a superinterface or superclass.
- getDeclaredFields(): Returns an array of Field objects reflecting all the fields declared by the class or interface represented by this Class object. This includes public, protected, private fields,… but excludes inherited fields.
- getFields(): Returns an array containing Field objects reflecting all the accessible public fields of the class or interface represented by this Class object.

Class Methods for locating **Methods**:

- getDeclaredMethod(String name, Class<?>... parameterTypes): Returns a Method object corresponding to the specified method, declared in this class
- getMethod(String name, Class<?>... parameterTypes): Returns a Method object corresponding to the public specified method
- getDeclaredMethods(): Returns an array of Method objects reflecting all (public and private) the methods declared by the class or interface represented by this Class object.
- getMethods(): Returns an array containing Method objects reflecting all the accessible public methods of the class or interface represented by this Class object.

Class Methods for locating **Constructors**

- getDeclaredConstructor (Class<?>... parameterTypes): Returns a Constructor object that reflects the specified constructor of the class or interface represented by this Class object. The parameterTypes parameter is an array of Class objects that identify the constructor's formal parameter types, in declared order.
- getConstructor(Class<?>... parameterTypes): Returns a Constructor object that reflects the specified public constructor of the class represented by this Class object. The parameterTypes parameter is an array of Class objects that identify the constructor's formal parameter types, in declared order.
- getDeclaredConstructors(): Returns an array of Constructor objects reflecting all the constructors declared by the class represented by this Class object. These are public, protected, default (package) access, and private constructors. The elements in the array returned are not sorted and are not in any particular order.
- getConstructors(): Returns an array containing Constructor objects reflecting all the accessible public constructors

How can you Work with Class members? = For each member (fields, methods, and constructor), the reflection API provides support to retrieve declaration and type information, and operations unique to the member. Field class has a type and a value. The java.lang.reflect.Field class provides methods for accessing type information and setting and getting values of a field on a given object. Method class has return values, parameters and may throw exceptions. The java.lang.reflect.Method class provides methods for accessing type information for return type and parameters and invoking the method on a given object. The Reflection APIs for constructors are defined in java.lang.reflect.Constructor and are similar to those for methods, but constructors have no return values and the invocation of a constructor creates a new instance of an object for a given class.

How to use Reflection for Program Manipulation? = Reflection is a powerful tool for creating new objects of a type or accessing members that were not known at compile time.

Accessible Objects = Certain operations such as changing a final field, writing/invoking a private field, fail also if invoked through reflection. We can request objects to be "accessible": request granted if no security manager, or if the existing security manager allows it. AccessibleObject Class provides the methods: boolean isAccessible( ), gets the value of the accessible flag for this object; void setAccessible(boolean flag), sets the accessible flag for this object to the indicated boolean value;
static void setAccessible(AccessibleObject[] array, boolean flag), sets the accessible flag for an array of objects with a single security check

d. Annotations in Java

Annotations = Modifiers in Java (static, final, public, …) are meta-data describing properties of program elements. Need for additional mechanisms for providing meta-data, without changing the language move to Annotations, (user-) definable modifiers.

Structure of Annotations = Annotations are made of a name and (optional) a finite number of attributes, i.e. "name = value". @annName{name_1 = constExp_1, ..., name_k = constExp_k} with constExp's are expressions that can be evaluated at compile time.

Which elements can be annotated? = Annotations can be applied to a syntactic element such as package declarations, classes, interfaces,…

Predefined annotations = The Java compiler defines and recognizes 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 element is not necessarily included in future releases of the Java API; @SuppressWarnings. Instruct the compiler to avoid issuing warnings for the specified situations; @FunctionalInterface. Declares an interface to be functional.

Define and use your own annotations = Programmers can define new annotations, e.g. to implement tools that process the content of the .class files generated by the compiler. They have a declaration syntax similar to

interfaces (but starting with @interface). Each method determines the name of an attribute and its type (the return type). A default value can be specified for each attribute. 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, enumerations, but rarely used.

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

Recovering annotations through the Reflection API = Annotations can be exploited by appropriate tools for program analysis. Package *javax.annotation.processing* provides a Java API for writing such tools. If at runtime, it must occur through the Reflection API (E.g. Annotation[] getAnnotations() returns an array of Annotation instances).

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

Inversion of control = Unlike text-based interaction, in the GUI-based interaction, the GUI loop decides when to invoke the methods (listeners), based on the order of events. With Frameworks, IoC becomes dominant where one just implements a few callback functions or specializes a few classes, and then invokes a single method or procedure. The framework does the rest of the work for you, invoking any necessary client callbacks or methods at the appropriate time and place, e.g. Java's Swing and AWT classes. Not only control flow but also control over dependencies, coupling, configuration. Framework calls component in well-defined ways (setters, template methods, interface).

Frameworks vs Libraries = Frameworks consist of large sets of classes/interfaces. Not much different from libraries, but wide use of IoC. This describes as a "well-designed library".

Components, Containers, and IoC = Often Frameworks provide containers for deploying components. It may provide at runtime functionalities needed by the components to execute.

Loosely coupled systems: advantages and techniques
OO Systems should be organized as a network of interacting objects gaining high cohesion and low coupling (which means extensibility, testability, reusability).

Dependency injection (RIVEDO) = With IoC and dependencies, we obtain a component orientation, removing the coupling of configuration and dependencies to the point of use. These are handled outside. Dependency injection allows avoiding hard-coded dependencies (strong coupling) and changing them. It allows selection among multiple implementations of a given dependency interface at run time such load plugins dynamically. Three forms:
– Setter injection, used in Spring, add a setter, leaving creation and resolution to others. Often already available leverages existing JavaBean reflective patterns. Cons: possible to create partially constructed objects; advertises that dependency can be changed at runtime.
– Constructor injection, used in PicoContainer, object can't be partially constructed. Cons: Bidirectional dependencies between objects can be tricky; constructors can easily get big and parameters confusing; can make class evolution more complicated (an added dependency affects all users of the class).
– (Interface injection),

Data Access Object (DAO)
The current exposure and the exposure limit are stored in some persistent storage and are accessed using a DAO (Data Access Object) component.

ServiceLocator = A pattern acts as a (static) registry for the LimitDao you need. This gives us extensibility, testability, reusability allowing new components to be dynamically created and used by others. Every component that needs a dependency must have a reference/registration to the service locator. If bound by name, a service can't be type-checked and, if many components share an instance but later you want to specify different instances for some, this becomes difficult. While, if bound by type, it can only bind one instance of a type in a container. The code needs to handle lookup problems.

ServiceLocator vs Dependency Injection = Both provide the desired decoupling. With ServiceLocator we don't have IoC, the application still depends on the locator, while with dependency injection there is no explicit request. It is easier to find dependencies of a component if Dependency Injection is used checking constructors and setters.

Who creates the dependencies? What if we need some initialization code that must be run after dependencies have been set? What happens when we don't have all the components? = Reflection solves these issues, which can be used to determine dependencies, reducing the need for config files. Another solution is IoC Containers which support auto-wiring between properties of a bean providing less typing, static type checking, more intuitive for a developer.

Four levels for understanding frameworks
1. Frameworks are normally implemented in an OOL, understanding the applicable language concepts
2. Understanding the framework concepts and techniques sufficiently well to use frameworks to build a custom application
3. Being able to do detailed design and implementation of frameworks for which the common and variable aspects are already known.
4. Learning to analyze a potential software family, identifying its possible common and variable aspects, and evaluating alternative framework architectures.

Some terminology… = **Frozen Spot**: common (shared) aspect of the software family; **Hot Spot**: variable aspect of the family, is represented by a group of abstract hook methods, realized in the framework as a hot spot; **Template method**: concrete method of a base (abstract) class implementing common behavior, which calls a hook method to invoke a function that is specific to one family member

Which are the principles for Framework Construction?
The unification principle [Template Method DP], which uses inheritance to implement the hot spot subsystem. Both the template methods and hook methods are defined in the same abstract base class, with the latest implemented in subclasses of the base class. The separation principle [Strategy DP], which uses delegation to implement the hot spot subsystem. The template methods are implemented in a concrete context class; the hook methods are defined in a separate abstract class and implemented in its subclasses.

Template Method DP = belongs to an abstract class and it defines the skeleton of an algorithm in terms of abstract operations that subclasses override to provide concrete behavior. Using Java visibility modifiers, it can be declared a public final method (itself should not be overridden). The concrete operations can be declared private ensuring that they are only called by the template method, Primitive operations that must be overridden are declared protected abstract, hook operations that may be overridden are declared protected.

Strategy DP (behavioral) = allows selecting (part of) an algorithm at runtime. The client uses an object implementing the interface and invokes methods of the interface for the hot spots of the algorithm.

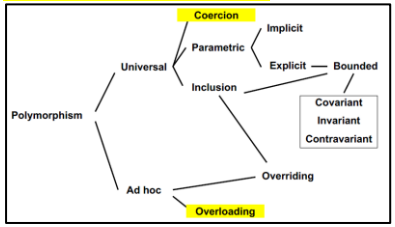
Unification vs. separation principle/Template method vs. Strategy DP = The two approaches differ in the coupling between the client and the chosen algorithm. With Strategy, the coupling is determined by dependency (setter) injection and could change at runtime.

\\ Polymorphism 13…16
a. A classification of Polymorphism

Universal vs. ad hoc polymorphism = 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. They can coexist.

Binding time = 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).

Classification of Polymorphism



Ad hoc polymorphism: overloading = Present in all languages, sometimes supported for user-defined functions (C++, Haskell allow overloading of primitive operators). The code to execute is determined by the type of the arguments.

Universal polymorphism: Coercion = is the automatic conversion of an object to a different type, as opposed to casting, which is explicit. Uninteresting case of polymorphism.

Inclusion polymorphism = Also known as subtyping polymorphism, or just inheritance. Substitution principle: an object of a subtype (subclass) can be used in any context where an object of the supertype (superclass) is expected. Methods/virtual functions

declared in a class can be invoked on objects of subclasses, if not redefined.

Overriding = A method of class A can be redefined in a subclass of A (Dynamic binding). Overriding introduce ad hoc polymorphism in the universal polymorphism of inheritance. Resolved at runtime by the lookup done by the invokevirtual operation of the JVM.

b. Polymorphism in C++: inclusion polymorphism and templates

Parametric polymorphism/generic programming, What happens with generic at runtime/after compilation?
[C++] Templates with function and class. Each concrete instantiation produces a copy of the generic code, specialized for that type.
[Java] Generics methods and classes, strongly type-checked by the compiler. Type erasure: type variables are Object at runtime.

Function Templates in C++ = They support parametric polymorphism where type parameters can also be primitive types (unlike Java generics). Example of polymorphic square function:
template <class T> // or <typename T>
T sqr(T x) { return x * x; }

Compiler/linker automatically generates one version for each parameter type used by a program. These are inferred or indicated explicitly (necessary in case of ambiguity).

Templates vs Macros in C++ = Macros can be used for polymorphism in simple cases, executed by the preprocessor, templates by the compiler. Macro expansion visible compiling with option –E.
Preprocessor makes only (possibly parametric) textual substitution. No parsing, no static analysis check.

Template (partial) specialization = A function/class template can be specialized by defining a template with the same name and more specific parameters (partial specialization) or no parameter (full specialization). Thanks to this we could use better implementation for specific kinds of types. The compiler chooses the most specific applicable template.

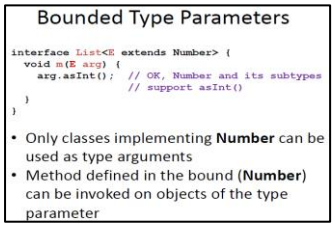
C++ Template implementation and instantiation
Compile-time instantiation (Static binding). The compiler chooses a template that is the best match based on partial (specialization) order of matching templates (1+). Template instance is created similarly to a syntactic substitution of parameters. Overloading resolution after substitution, which will fail if some operator is not defined for the type instance.
On instantiation, the compiler needs both the declaration/definition of the template function. Cannot compile the definition of template and code instantiating the template separately. If the same template function definition is included in different source files, separately compiled and linked, there will be only one instantiation per type of template function. Explicit instantiation is possible.

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

Java Generics: Explicit Parametric Polymorphism
Classes, Interfaces, Methods can have type parameters which will be used arbitrarily in the definition and instantiated by providing arbitrary (reference) type arguments.

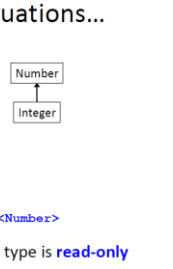
Generic methods = Methods can use the type parameters of the class where they are defined, also introduce their own type parameters
public static <T> T getFirst(List<T> list)

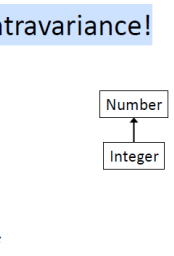
Invocations of generic methods must instantiate all type parameters, either explicitly or implicitly like inference.



How many Type Bounds?
<TypeVar extends SuperType>, **upper bound**, SuperType and any of its subtype are ok.
<TypeVar extends ClassA & InterfaceB & InterfaceC & …> , **multiple upper bounds**
<TypeVar super SubType>, **lower bound**, SubType and any of its supertype are ok
Type bounds for methods guarantee that the type of argument supports the operations used in the method body. Unlike C++, type checking ensures that overloading will succeed.

Explain the concept (with written example) of covariance and contravariance in a language with universal polymorphism and explain in what cases their use is safe = Start by Java rules = 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. Subtyping in Java is invariant for generic classes. The common parent of MyClass<A> and MyClass<B> is MyClass<?> . On the other hand, if A extends B and they are generic classes, for each type C we have that A<C> extends B<C>, e.g.
ArrayList<Integer> is a subtype of List<Integer>.





Limitations of Java Generics, What are the problems between arrays and generics in Java? = 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, Cannot Create Arrays of Parameterized Types, Cannot Create, Catch, or Throw Objects of Parameterized Types, Cannot Overload a Method Where the Formal Parameter Types of Each.

A digression: Java arrays = Arrays are like built-in containers. If Type1 is a subtype of Type2, then Type1[] is a subtype of Type2[]. Thus Java arrays are covariant. Without covariance, a new sort method is needed for each reference type different from Object.

Recalling "Type erasure" = All type parameters of generic types are transformed to Object or to their first bound after compilation due to backward compatibility with legacy code thus at run-time, all the instances of the same generic type have the same type.

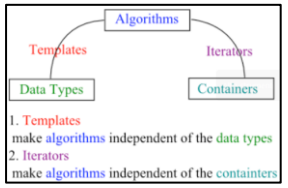
Wildcards for covariance = Wildcards can alleviate the problem of restrictiveness invariance of generic classes. Wildcard = anonymous variable "?" used when a type is used exactly once, and the name is unknown.

The "PECS principle" = Wildcards will be used when you want to get values (from a producer): supports covariance (? extends T) or when you want to insert values (in a consumer): supports contravariance (? super T). Do not use ? (T is enough) when you both obtain and produce values.

d. The Standard Template Library: an overview

The Standard Template Library = It represents algorithms in a general form as possible without compromising efficiency with widely use of templates and overloading. Only uses static binding (and inlining): not OO/dynamic binding. Use of iterators seen as an abstraction of pointers for decoupling algorithms from containers. There are polymorphic abstract types/operations and code is efficient.

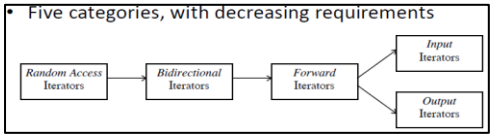
Main entities in STL = Container: Collection of typed objects, array, vector, deque, list, set, map;
Iterator: Generalization of pointer or address. used to step through the elements of collections;
Algorithm: initialization, sorting, searching, and transforming the contents of containers;
Adaptor: Convert from one form to another, e.g. produce stack from list;
Allocator: encapsulation of a memory pool, e.g. GC memory



Iterators in Java = Iterators are supported in the Java Collection Framework: interface Iterator<T>. Usually defined as nested classes (non-static private member classes): each iterator instance is associated with an instance of the collection class, They exploit generics.

C++ namespaces = Containers expose a type named iterator in the container's namespace, with each class implicitly introduces a new namespace.

Categories/Classifying iterators = Inserting/erasing at the end of the vector takes constant time whereas in the middle takes linear time. To control the complexity of algorithms and guarantee that code behaves as expected, the solution proposed by STL is assuming that iterators implement all operations in constant time. Different iterators depending on their structure are: Forward iterators, only dereference (operator*), and pre/post-increment operators (operator++); Input and Output iterators, like forward iterators but with possible issues in dereferencing the iterator (due to I/O operations); Bidirectional iterators, like forward iterators with pre/post-decrement (operator--); Random-access iterators, like bidirectional iterators but with integer sum (p + n) and difference (p – q)



Iterator validity = When a container is modified, iterators to it can become invalid depending 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

Iterator Limits = Iterators provide a linear view of a container thus, we can define only algorithms operating on single dimension containers. If it is needed to access the organization of the container the only way is to define a new iterator.

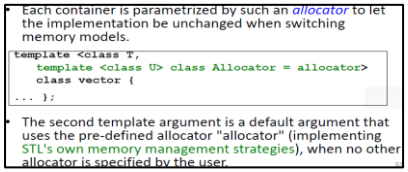
Which language mechanisms it is based STL upon? = Type aliases (typedefs), Template functions and classes, Operator overloading, Namespaces

Iterators: small struct = Iterators are implemented by containers as a struct (classes with only public members). It implements a visit of the container to retains inside information about the state of the visit. The state may be complex in the case of non-linear structures such as graphs

STL Inheritance = STL relies on typedefs combined with namespaces to implement genericity. We refer to *container::iterator* to know the type of the iterator. There is no relation among iterators for different containers due to performance: without inheritance, types are resolved at compile-time, which produces better code but sacrificing inheritance to lower expressivity and lack of type-checking.

Inlining = STL relies also on the compiler. The notion of inlining 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 labeled inline or defined within a class definition. This tends with small bodies and without iteration.

Memory management = STL abstracts from the specific memory model encapsulating all the information about in the Allocator class.



STL problem = error checking: facilities of the compiler fail with STL resulting in lengthy error messages that end with error within the library. The generative approach taken by the C++ compiler also leads to possible code bloat, which can be a problem if the working set of a process becomes too large.

\\ Functional Programming 17-18

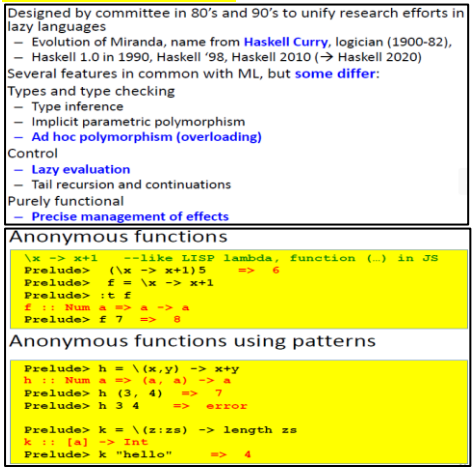
a. Introduction to functional programming

Functional Programming: History = The imperative and functional models grew upon different formalizations of the notion of an algorithm based on automata, symbolic manipulation, recursive function definitions, and combinatorics. The lambda calculus (Church's model of computing) is based on the notion of parameterized expressions (parameters introduced by letter λ), allowing one to define mathematical functions in a constructive/effective way, where computation proceeds by substituting parameters into expressions, just as one computes in a high-level functional program by passing arguments to functions.

Functional Programming Concepts = Functional languages such as LISP/Scheme/ML/Haskell realize Church's lambda calculus as a programming language. The key idea: do everything by composing functions. (no mutable state and no side effects). Features are 1st class and high-order functions, recursion (no "control variables"), powerful list facilities, polymorphism (Containers/Collections), wide use of tuples and records (data structures cannot be modified, have to be re-created), structured function returns, GC.

The LISP family of languages = LISP (LISt Processor, 1958) is the main programming language for AI, including a (few line code) LISP interpreter. Variants of LISP is Scheme, which is statically scoped, elegant, used for teaching with an essential syntax.

Haskell, Lazyness in haskell



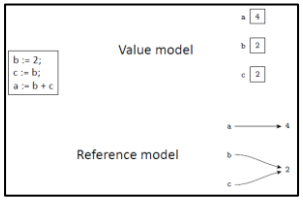
Interactive Interpreter (ghci): read-eval-print. Variables (names) are bound to expressions, without evaluating them (because of lazy evaluation). Patterns can be used in place of variables: *<pat> ::= <var> | <tuple> | <cons> | <record>* … Haskell is a lazy language. This would mean that functions and data constructors don't evaluate their arguments until they need them.

Applicative and Normal Order evaluation = In Applicative Order evaluation, arguments are evaluated before applying the function (Eager evaluation, parameter passing by value). With Normal Order evaluation we evaluated Function first, arguments if and when needed (parameter passing by name, evaluation can be repeated). Church-Rosser :
– If evaluation terminates, the result (normal form) is unique
– If some evaluation terminates, normal order evaluation terminates

Describe the different kind of parameter passing strategies

d. Call by sharing, by name, and by need

L-Values vs. R-Values and Value Model vs. Reference Model = Consider the assignment of the form *a = b* where *a* is an l-value expression denoting a location like an array/variable; *b* is an r-value: any syntactically valid expression with a type compatible to that of a. Languages that adopt the value model of variables copy the value of b into the location of a (imperative programming). Languages that adopt the reference model of variables copy references, resulting in shared data values via multiple references (Lisp/Scheme, ML, Haskell). Java uses the value model for built-in types and the reference model for class instances.



References and pointers = Most PLs have as target architecture a Von Neumann one, where memory is made of cells with addresses. A reference to X is the address of the (base) cell where X is stored, and a pointer to X is a location containing the address of X, then Value model-based implementations can mimic the reference model using pointers and standard assignment where each variable is associated with a location: y refer to data X, the address of (reference to) X is written in the location of y, which becomes a pointer.

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

Call by name & Lazy evaluation = In **call by name** parameter passing arguments (like expressions) are passed as a closure ("thunk") to the subroutine. The argument is (re)evaluated each time it is used in the body. Haskell realizes lazy evaluation by using **call by need** parameter passing, where an expression passed as argument is evaluated only if its value is needed, but the argument is evaluated only the first time, using memoization: the result is saved. Combined with lazy data constructors, this allows to construct/call potentially infinite data structures/recursive functions without necessarily causing non-termination.

\\Haskell 19..21

a. Introduction to Haskell, Laziness, basic and compounds types, Patterns and declarations, Function

declarations, List comprehension, Algebraic Data Types, Higher-order functions, Recursion

Laziness = Haskell is a lazy language: functions and data constructors don't evaluate their arguments until they need it.

List Comprehensions

Notation for constructing new lists from old ones:

```
myData = [1,2,3,4,5,6,7]

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

Datatype Declarations

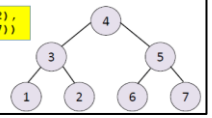
```
data Color = Red | Yellow | Blue
  elements are Red, Yellow, Blue
data Atom = Atom String | Number Int
  elements are Atom "A", Atom "B", ..., Number 0, ...
```

Datatypes and Pattern Matching

```
data Tree = Leaf Int | Node (Int, Tree, Tree)

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

Constructors can be used in Pattern Matching



Case Expression

Datatype

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

Case expression

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

From loops to recursion = In functional programming, for and while loops are replaced by using recursion, subroutines call themselves directly/indirectly.

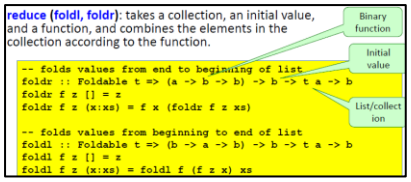
```
length' [] = 0
length' (x:s) = 1 + length'(s)

// definition using guards and pattern matching
take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
    | n <= 0   = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

Higher-Order Functions = Functions that take other functions as arguments or return a function, as a result, are higher-order functions.

```
applyTwice :: (a -> a) -> a -> a  -- function as arg and res
applyTwice f x = f (f x)
> applyTwice (+3) 10     => 16
> applyTwice (++ " HAHA") "HEY" => "HEY HAHA HAHA"
> applyTwice (3:) [1]            =>  [3,3,1]
```

the map/filter/reduce combinator = map applies argument function to each element in a collection.
map (+3) [1,5,3,1,6] => [4,8,6,4,9]
filter takes a collection and a boolean predicate, and returns the collection of the elements satisfying the predicate.
filter (>3) [1,5,3,2,1,6,4,3,2,1] => [5,6,4]

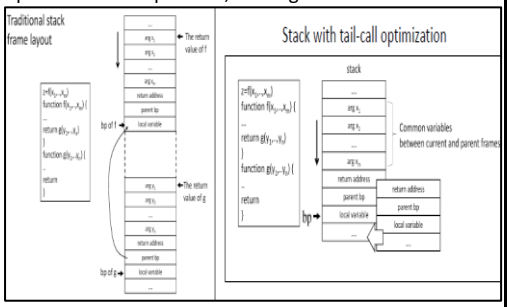


On efficiency = Iteration and recursion are equally powerful in a theoretical sense: Iteration can be expressed by recursion and vice versa. Recursion is defined in terms of simpler versions of the same problem. A procedure call is more expensive than a conditional branch, thus recursion is, in general, less efficient, but good compilers for functional languages can perform good code optimization thanks to using of combinators such as map/reduce/filter,…

Tail-Recursive Functions = Tail-recursive functions are functions in which no operations follow the recursive call(s) in the function, thus the function returns immediately. A tail-recursive call could reuse the subroutine's frame on the run-time stack simply eliminating the push/pop since the current subroutine state is no longer needed. An optimistic way is when the compiler replaces calls by jumps to the beginning of the function.

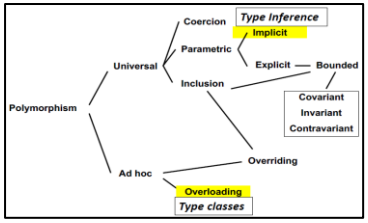
```
int gcd(int a, int b) // possible optimizatior
   { start:
        if (a==b) return a;
        else if (a>b) { a = a-b; goto start; }
        else { b = b-a; goto start; }
```

Tail-Call Optimization = A Tail-call is when a function returns another function, not necessarily itself. Optimization still possible, reusing the stack frame.



d. Type classes in Haskell

Polymorphism in Haskell



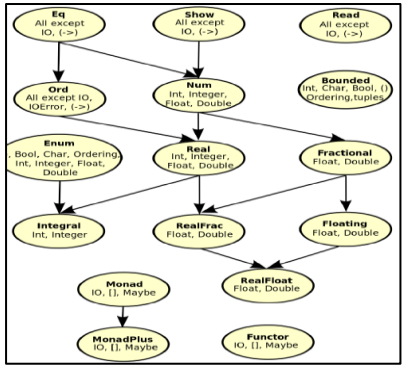
Ad hoc polymorphism: overloading = Present in all languages, at least for built-in arithmetic operators: +, *, -, … (Haskell allow for primitive operators), sometimes supported for user-defined functions, the code to execute is determined by the type of the arguments, thus early/late binding in statically/dynamically typed languages. Which type can be inferred by ML/Haskell? Can list membership work for any type? Only for types w that supports equality.
sort :: [w] -> [w]
Can list sort work for any type? Only for types w that support ordering.

Type Classes = solve the problem of overloading, providing concise types to describe overloaded functions, without an exponential blow-up. This fits within the type inference framework.

Intuition = Consider the "overloaded" parabola function *parabola x = (x * x) + x* . We can rewrite the function to take the operators it contains as an argument *parabola' (plus, times) x = plus (times x x) x* . The extra parameter is a "dictionary" that provides implementations for the overloaded ops. We have to rewrite all calls to pass appropriate implementations for plus and times.

Extra parameter dictionary = The compiler translates each function that uses an overloaded symbol into a function with an extra parameter: the dictionary. References to overloaded symbols are rewritten by the compiler to lookup the symbol in the dictionary. The compiler converts each type of class declaration into a dictionary type declaration and a set of selector functions and converts each instance declaration into a dictionary of the appropriate type. The compiler rewrites call to overloaded functions to pass a dictionary, using static, qualified type of the function to select the dictionary.

Subclasses



e. The Maybe constructor and composition of partial functions

Type Constructor Classes = While Type Classes are predicates over types, [Type] Constructor Classes are predicates over type constructors, allowing to define overloaded functions common to several type constructors, like map.

What is "Functor" in haskell?

Constructor Classes

```
fmap:: (a -> b) -> g a -> g b
– where g is:
  [-] for lists, Tree for trees, and Maybe for options
Note that g is a function from types to types, i.e.
a type constructor
```

This pattern can be captured in a constructor class Functor, a type class where the predicate is over a type constructor rather than on a type. We can then use the overloaded symbol fmap to map over all three kinds of data structures.

What relationships there are between functor and maybe type class?

The Maybe type constructor = Often type constructors can be thought of as defining "boxes" for values (inside boxes with Functors with fmap). Monads are constructor classes introducing operations for putting a value into a "box" (return) and compose functions that return "boxed" values (bind).
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.

Composing partial function = We introduce a higher-order operator to compose partial functions in order to "propagate" undefinedness automatically. The bind operator will be part of the definition of a monad.

Monads as containers = The monadic constructor can be seen as a container. A value of type *m a* is a computation returning a value of type a, a computation which "does nothing". This is given by function return. *return*, *bind* and *then* define basic ways to compose computations. They are used in Haskell to define complex operators and control structures.

```
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
```

f. Monads in Haskell

Pros of Functional Programming = is concise and powerful abstractions which mean higher-order functions, algebraic data types, parametric polymorphism,…, with close correspondence with mathematics and independence of order-of-evaluation, but to be useful it must be able to manage impure features like I/O, error recovery, foreign-language interfaces,… The Direct Approach just add imperative construct, but it works if language determines evaluation order. Since Haskell is lazy, the order of evaluation is undefined. To add imperative features without changing the meaning of pure Haskell expressions, the IO monad defines monadic values which are called actions, and prescribes how to compose them sequentially.

The IO Monad = a functional program defines a pure function with no side effects, but the point is to have some side effect. Before Monads Haskell adopted Stream model to move "side effects" outside of functional program, enriching argument and return type of main to include all input and output events.

```
main :: [Response] -> [Request]
data Request =  ReadFile Filename
       |  WriteFile FileName String
       | …
data Response =  RequestFailed
       |  ReadOK String
       |  WriteOk
       |  Success  | …
```

The problem is:

- laziness, which allows program to generate requests before processing any responses.
- Hard to extend , I/O operations require adding new constructors to Request and Response types, modifying wrapper
- no easy way to combine two "main" programs

IO is a type constructor, instance of Monad. A value of type (IO t) is an action that, when performed, may do some I/O before delivering a result of type t. Without think on return, then and bind, the only way is to call it from Main.main.

Implementation of the IO monad = GHC uses "world-passing semantics" for the IO monad (*type IO t = World -> (t, World)*) . It represents the "world" by an un-forgeable token of type World, and implements bind and return as:

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

Using this form, the compiler can do its optimizations. The dependence on the world ensures the resulting code will still be single-threaded and the code generator then converts the code to modify the world "in-place.".

The Bind Combinator (>>=)

(>>=) :: IO a -> (a -> IO b) -> IO b

This operator is called bind because it binds the result of the left-hand action in the action on the right, performing compound action a >>= \x->b :

– performs action a, to yield value r
– applies function \x->b to r
– performs the resulting action b{x <- r}
– returns the resulting value v

The then (>>) Combinator = The "then" combinator (>>) does sequencing when there is no value to pass.

The "do" Notation = adds syntactic sugar to make monadic code easier to read and designing syntax to look imperative.

```
-- Plain Syntax
getTwoChars :: IO (Char,Char)
getTwoChars = getChar   >>= \c1 ->
              getChar   >>= \c2 ->
              return (c1,c2)
```

```
-- Do Notation
getTwoCharsDo :: IO(Char,Char)
getTwoCharsDo = do { c1 <- getChar ;
                     c2 <- getChar ;
                     return (c1,c2) }
```

References = The IO operations let us write programs that do I/O in a strictly sequential, imperative fashion, obtaining leverage the sequential nature of the IO monad to do other imperative things.

```
data IORef a   -- Abstract type
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

A value of type IORef a is a reference to a mutable cell holding a value of type a.

Monad (I/O) Restriction = In pure Haskell, there is no way to transform a value of type IO a into a value of type a. If we wanted to read a configuration file, then the problem is that readFile returns an IO String, not a String. There are 2 options: write entire program in IO monad, losing the simplicity of pure code or escape from the IO Monad using a function from IO String -> String. This is disallowed! Haskell offers unsafe I/O primitive: unsafePerformIO.

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

\\ Functional programming in Java 8 22-23

a. Lambdas in Java 8

Lambdas in Java = A big challenge was to introduce lambdas without requiring recompilation of existing binaries. The benefits of lambdas in Java 8 are: enabling functional programming introduction of lazy evaluation with stream processing; writing cleaner/compact code; facilitating parallel programming; developing generic/flexible/reusable APIs.

```
List<Integer> intSeq = Arrays.asList(1,2,3);

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

• `x -> System.out.println(x)`

is a lambda expression that defines an *anonymous function (method)* with one parameter named x of type Integer

Implementation of Java 8 Lambdas, How are lambda expressions implemented?, How the java compiler manges lambdas? = compiler first converts a lambda expression into a function, compiling its code. Then it generates code to call the compiled function where needed. *x -> System.out.println(x)* could be converted into a generated static function
public static void genName(Integer x) { System.out.println(x);}
..but what type should be generated for this function? How should it be called? What class should it go in?

What are Functional Interfaces ?= Lambdas are instances of functional interfaces, a Java interface with exactly one abstract method. They can be used as target type of lambda expressions, i.e.
– As type of variable to which the lambda is assigned
– As type of formal parameter to which the 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. The lambda is invoked by calling the only abstract method of the functional interface and can be interpreted as instances of anonymous inner classes implementing the functional interface.

Default Methods = Problem: Adding new abstract methods to an interface breaks existing implementations of the interface. Java 8 allows interface to include:

- Abstract (instance) methods, as usual
- Static methods
- Default methods, defined in terms of other possibly abstract methods

Java 8 uses lambda expressions and default methods in conjunction with the Java collections framework to achieve backward compatibility with existing published interfaces.

From Lambdas to Bytecode, How the java compiler manges lambdas? = Lambdas can be compiled as instances of anonymous inner classes. The strategy for lambdas is left to the designer of the compiler, which can exploit this freedom on behalf of efficiency.

b. The Stream API in Java 8

Streams 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. Don't stores elements, but 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.

Pipelines = 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
```

Anatomy of the Stream Pipeline = A Stream is processed through a pipeline of operations, starting with a source. Intermediate methods are performed on the Stream elements, producing Streams and are not processed until the terminal method is called. 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 sources =

- 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 tiufferedReader.lines();
- Streams of file paths can be obtained from methods in Files;
- Streams of random numbers can be obtained from Random.ints();
- Generators, like generate or iterate;

…

Intermediate Operations

- An intermediate operation keeps a stream open for further operations. Intermediate operations are lazy.
- Several intermediate operations have arguments of *functional interfaces*, thus *lambdas* can be used

```
Stream<T> filter(Predicate<? super T> predicate)  // filter

IntStream mapToInt(ToIntFunction<? super T> mapper) // map f:T -> int

<R> Stream<R> map(Function<? super T,? extends R> mapper) // map f:T->R

Stream<T> peek(Consumer<? super T> action) //performs action on elements

Stream<T> distinct() // remove duplicates - stateful

Stream<T> sorted() // sort elements of the stream - stateful

Stream<T> limit(long maxSize) // truncate

Stream<T> skip(long n) // skips first n elements
```

Terminal Operations

- A **terminal operation** must be the final operation on a stream. Once a terminal operation is invoked, the stream is consumed and is no longer usable.
- Typical: collect values in a data structure, reduce to a value, print or other side effects.

```
void forEach(Consumer<? super T> action)

Object[] toArray()

T reduce(T identity, BinaryOperator<T> accumulator)  // fold

Optional<T> reduce(BinaryOperator<T> accumulator) // fold

Optional<T> min(Comparator<? super T> comparator)

boolean allMatch(Predicate<? super T> predicate) // short-circuiting

boolean anyMatch(Predicate<? super T> predicate) // short-circuiting

Optional<T> findAny() // short-circuiting
```

Types of Streams

Streams only for reference types, int, long and double

Differences between reduce in functional programming and collect in the Java Stream API

From Reduce to Collect: Mutable Reduction

- Suppose we want to concatenate a stream of strings.
- The following works but is highly inefficient (it builds one new string for each element):

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

- Better to "accumulate" the elements in a mutable object (a StringBuilder, a collection, …)
- The ***mutable reduction operation*** is called **collect()**. It requires three functions:
  - a supplier function to construct new instances of the result container,
  - an accumulator function to incorporate an input element into a result container,
  - a combining function to merge the contents of one result container into another.

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

12

Infinite Streams = Streams wrapping collections are finite. Infinite streams can be generated with: iterate or generate.

```
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)

// Example: summing first 10 elements of an infinite stream
int sum = Stream.iterate(0,x -> x+1).limit(10).reduce(0,(x,s) -> x+s);

static <T> Stream<T> generate(Supplier<T> s)

// Example: printing 10 random numbers
Stream.generate(Math::random).limit(10).forEach(System.out::println);
```

Parallelism = Stream operations can execute either in serial (default) or in parallel. The runtime support takes care of using multithreading for parallel execution, in a transparent way. If operations don't have side-effects, thread-safety is guaranteed even if non-thread-safe collections are used (e.g.: ArrayList).

When to use a parallel stream? = When operations are independent, and either/both operations are computationally expensive and applied to many elements of efficiently splittable data structures. A stream wrapping a collection uses a Splititerator over the collection, the parallel analogue of an Iterator: it describes a (possibly infinite) collection of elements with support for sequentially advancing, applying an action to the next or to all remaining elements or splitting off some portion of the input into another spliterator which can be processed in parallel.

\\ Scripting Languages and Python 25…28

a. Overview of Scripting Languages

Characteristics of Scripting Languages = Modern scripting languages have two principal sets of ancestors: command interpreters or "shells" of traditional batch and "terminal" (command-line) computing (Unix sh) and various tools for text processing and report generation (Unix's sed and awk). Then => Perl, Python, Ruby,… Common Characteristics: Both batch and interactive use, compiled/interpreted line by line; Economy of expression, print "Hello, world!\n" ; Simple default scoping rules, which can be overruled via explicit declarations; Dynamic typing, due to lack of declarations; Flexible typing, a variable is interpreted differently depending on the context (kind of coercion); Easy access to system facilities, Perl has 100+ built-in commands for I/O, file/directory manipulation,…; Sophisticated pattern matching and string manipulation, based on extended regular expressions; High level data types, storage is garbage collected; Quicker development cycle

Problem Domains = Scripting languages have features to support modules, separate compilation, reflection,... Most with well-defined problem domains:
1. Shell languages, with features designed for interactive use, providing many mechanisms to manipulate file names, arguments, and commands, and to glue together other programs: Filename and Variable Expansion, Tests, Queries, and Conditions, Pipes and Redirection,…
2. Text Processing and Report Generation, from bash/sed/awk to Perl, Unix-only tool, fast enough for modularization, and dynamic library mechanisms appropriate for large-scale projects.
3. Mathematics and Statistics, Maple, Mathematica and Matlab (Octave), R and S, languages for statistical computing
4. "Glue" Languages and General-Purpose Scripting, Perl, Python
5. Extension Languages, feature of sophisticated tools, Adobe's graphics suite (Illustrator, Photoshop, InDesign, etc.) can be extended (scripted) using JavaScript, Visual Basic (on Windows),… To admit extension, a tool must incorporate, or communicate with, an interpreter for a scripting language.

b. Introduction to Python: Basic and Sequence Datatypes, Dictionaries, Control Structures, List Comprehension

Language features

§ Indentation instead of braces
§ Several sequence types, Strings '…': made of characters, immutable; Lists […]: made of anything, mutable; Tuples (…) : made of anything, immutable
§ Powerful subscripting (slicing)
§ Functions are independent entities (not all functions are methods)
Exceptions as in Java Simple object system Iterators and generators

Why Python?, Is Python more OO or more functional, according to your opinion? = here Style is very concise. Powerful but unobtrusive object system where every value is an object. Powerful collection and iteration abstractions, Dynamic typing makes generics easy.

Dynamic typing = Java: statically typed. Variables are declared to refer to objects of a given type. Methods use type signatures to enforce contracts.
Python: Variables come into existence when first assigned to and can refer to an object of any type. Main drawback: type errors are only caught at runtime.

Useful commands

§ help(), Python interactive help utility
§ help(arg), Prints documentation about arg
§ type(arg), Prints the type of arg
§ _ : in the interpreter is the value of the last expression
§ Since "everything is an object", try "dot-completion" to see what are the options, "hello". <tab><tab>

The dir() Function = The built-in function dir() returns a sorted list of strings containing all names defined in a module, a class, or an object

Import and Modules = Programs will often use classes & functions defined in another file. A Python module is a single file with the same name (plus the .py extension), which can contain many classes and functions, accessing using import. The list of directories where Python looks: sys.path. When Python starts up, this variable is initialized from the PYTHONPATH environment variable. To add a directory of your own to this list, append it to this list sys.path.append('/my/new/path'). Modules are files containing definitions and statements. A module defines a new namespace and can be organized

hierarchically in packages. A module can be invoked as a script from the shell as

```
>           python fibo.py 60
```

Whitespace = especially indentation and placement of newlines. Use \ when must go to next line prematurely. No braces { } to mark blocks of code in Python, but will use indentation instead.

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. Python determines the type of the reference automatically based on what data is assigned to it.

Sequence Types = Tuples(), a simple immutable ordered sequence of (mixed types) items; Strings"", immutable; Lists[], mutable ordered sequence of items of mixed types

Negative indices = Negative lookup: count from right, starting with –1.

Slicing: Return Copy of a Subset (1)

```
>>>         t[1:4]
>>>         t[:2]
>>>         t[2:]
```

To make a copy of an entire sequence, you can use [:].

The 'in'/+ Operator = Boolean test whether a value is inside a collection (container in Python)

```
>>>         3 in t
```

The + operator produces a new tuple, list, or string whose value is the concatenation of its arguments.

Operations on Lists Only

```
>>>         li.append('a')  # Note the method syntax
>>>         li.insert(2, 'i')
```

+ creates a fresh list (with a new memory reference)
Extend is just like add in Java; it operates on list in place.

```
>>> li.index('b')       # index of first occurrence*
>>> li.count('b')       # number of occurrences
>>> li.remove('b')      # remove first occurrence
>>> li.reverse()        # reverse the list *in place*
>>> li.sort()           # sort the list *in place*
```

Tuples vs. Lists = Lists are slower but more powerful than tuples. They can be modified and have lots of handy operations we can perform on them, while tuples are immutable and have fewer features. To convert between tuples and lists use the list() and tuple() functions.

Dictionaries: Like maps in Java = store a mapping between a set of keys of any immutable hashable type, and a set of values of any type. Values and keys can be of different types in a single dictionary. the key-value pairs in the dictionary can be defined/modified/deleted

Creating and accessing dictionaries

```
>>>         d = {'user':'bozo', 'pswd':1234}
>>>         d['user']
'bozo'
```

Updating Dictionaries = Dictionaries are unordered and work by hashing. Assigning to a non-existing key adds a new pair. Assigning to an existing key changes the value.

```
>>>         d['user'] = 'clown'
```

Removing dictionary entries

```
>>> del d['user']       # Remove one.
>>> d.clear()           # Remove all.
```

Assert = check if something is true during the course of a program.

The range() function = We often want to write a loop where the variables ranges over some sequence of numbers. The range() function returns a list of n numbers from 0 up to but not including the number we pass to it. range(5) returns [0,1,2,3,4]
Don't use range() to iterate over a sequence solely to have the index and elements available at the same time.

List Comprehensions = They are a powerful feature of the Python language, generating a new list by applying a function to every member.

```
>>>         [elem*2 for elem in li]
```

Filtered List Comprehension = Filter determines whether expression is performed on each member of the list. First check if each element satisfies the filter condition then, if the filter condition returns False, that element is omitted from the list before the list comprehension is evaluated.

```
>>>   [elem * 2 for elem in li if elem > 4] [12, 14, 18]
```

Functions in Python = Functions are first-class objects with parameters passed by object reference and returning some value (possibly None). A function call creates a new namespace. Functions can take a variable number of args and kwargs. Higher-order functions are supported. The comment after the functions header is bound to the __doc__ special attribute (print(my_function.__doc__))

Decorators = A decorator is any callable Python object that is used to modify a function, method or class definition. It is passed the original object being defined and returns a modified object, which is then bound to the name in the definition.

```
def do_twice(func):
    def wrapper_do_twice():
        func()       # the wrapper calls the
        func()       #      argument twice
    return wrapper_do_twice

@do_twice           # decorate the following
def say_hello():    # a sample function
    print("Hello!")
>>> say_hello()     # the wrapper is called
Hello!
Hello!

@do_twice           # does not work with parameters!!
def echo(str):      # a function with one parameter
    print(str)
>>> echo("Hi...") # the wrapper is called
TypErr: wrapper_do_twice() takes 0 pos args but 1 was given
>>> echo()
TypErr: echo() missing 1 required positional argument: 'str'
```

Other 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
- In a web application, can wrap some code to check that the user is logged in
- @staticmethod and @classmethod make a function invocable on the class name or on an object of the class

Namespaces and Scopes = A namespace is a mapping from names to objects: typically implemented as a dictionary. Examples: global names of a module, created when the module definition is read; local names of a function invocation, created when function is called, deleted when it completes. Name x of a module m is an attribute of m accessible (read/write) with "qualified name" m.x . 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. They are determined statically, but are used dynamically. Assignments to names go in the local scope.

OOP in Python = Python classes provide all the standard features of OOP: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data.

Defining a class (object) = A class is a blueprint for a new data type with internal attributes and functions (methods). To declare a class in Python the syntax is the following:

```
class className:
        <statement-1>
        …
        <statement-n>
```

A new namespace is created, where all names introduced in the statements will go. When the class definition is left, a class object is created, bound to className, on which two operations are defined: class instantiation and attribute reference. The latest allows to access the names in the namespace.

Creating a class instance = 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/functions. The first argument, self, represents the implicit parameter (this in Java).
A constructor is a special instance method with name __init__. At most ONE constructor (no overloading in Python!).

String representation = It is often useful to have a textual representation of an object with the values of its attributes. This is possible with:

```
def __str__(self) :
        return <string>
```

It is invoked automatically when str or print is called.

(Multiple) Inheritance = A class can be defined as a derived class and 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. Python supports multiple inheritance

```
class derived(base1,..., basen):
```

The diamond problem is resolved by the Method resolution order (MRO).

Encapsulation (and "name mangling"/storpiatura) =
Private instance variables don't exist in Python. A name prefixed with underscore (e.g. _spam) is treated as non-public part of the API. Sometimes class-private members are needed to avoid clashes with names defined by subclasses. Limited support for such a mechanism, called name mangling, where any name with at least two leading underscores and at most one trailing underscore like e.g. __spam is textually replaced with _classname__spam, where classname is the current class name. This is helpful for letting

subclasses override methods without breaking intraclass method calls.

Static and Class methods, What does the @staticmethod decorator do? = Sometimes we need to process data associated with classes instead of instances. However, the code is not well associated with class, cannot be inherited and the name of the method is not localized. Hence, Python offers static and class methods. Static methods are simple functions with no self argument, preceded by the @staticmethod decorator. They are defined inside a class but they cannot access instance attributes and methods, and can be called through both the class and any instance of that class! They allow subclasses to customize the static methods with inheritance, without redefining them. They have a first parameter which is the class name and the definition must be preceded by the @classmethod decorator. Can be invoked on the class or on an instance.

What are iterators? = In Python an iterator (an object which allows a programmer to traverse through all the elements of a collection) is used implicitly by the FOR loop construct. They need to support two methods: __iter__ returns the iterator object itself, used in FOR and IN statements; the next method returns the next value from the iterator until there is no more and raising a StopIteration exception.

Generators and coroutines = 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 as described in the previous section (not vice-versa). The local variables and execution state are automatically saved between calls. In addition to automatic method creation and saving program state, when generators terminate, they automatically raise StopIteration.

e. The Global Interpreter Lock (GIL).

Garbage collection in Python = CPython manages memory with a reference counting + a mark&sweep cycle collector scheme. With 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)

Handling reference counters = 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. Synchronization primitives are quite expensive on contemporary hardware and, since almost every operation in CPython can cause a refcount to change somewhere, handling refcounts with some kind of synchronization would cause spending almost all the time on synchronization.

The Global Interpreter Lock (GIL) = The CPython interpreter assures that only one thread, which must hold the GIL before it can safely access python object, executes Python bytecode at a time, thanks to the Global Interpreter Lock. 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 are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing.

Alternatives to the GIL? = Past efforts to create a “free-threaded” interpreter (locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case, make the implementation much more complicated and therefore costlier to maintain. Jython and IronPython (on .NET) have no GIL and can fully exploit multiprocessor systems.

Builtins & Libraries = The Python ecosystem is extremely rich and in fast evolution. There are dozens of other libraries, mainly for scientific computing, machine learning, computational biology, data manipulation and analysis, natural language processing, statistics, symbolic computation, etc.

Differences between component, packages and classes
Scoping Rules for Modules
What are Components?