# Language execution basics

## What is Java ?

**Java IS SON**'**S** **RAP**per **MHD** computer language

Java is a **simple**, **object-oriented**, **network-savvy**, **interpreted**, **robust**, **secure**, **architecture-neutral**, **portable**, **high-performance**, **multithreaded**, **dynamic** computer language.

Java is simple: modelled after C & C++. Pointers, multiple implementation inheritance and operator overloading are not part of Java (C/C++ features). Garbage collection is an essential feature of Java.

Java is network-savvy: Java has an extensive network library 🡪 make it easy to cope with TCP/IP network protocols (HTTP & FTP) & simplifies making network connections.

Java is an interpreted language: Java program is indirectly executed on underlying platform via VM. VM translates Java program’s bytecode to platform-specific instructions through interpretation.

Java is robust: Java is used in both consumer and mission-critical applications. Loops must be controlled by Boolean.

Java is secure: Security features such as the Java sandbox security model and public-key encryption. Prevent viruses.

Java is an architecture-neutral language: Java generates platform-independent bytecode instructions.

Java is a portable language: Combined with architecture neutrality, Java’s libraries provide types that connect Java code with platform-specific capabilities (platform with 16bit, 32bit or 64bit registers).

Java is high-performance: Interpretation yields a level of performance that is usually more than adequate. For every high-performance application scenarios Java uses JIT compilation, which analyses interpreted bytecode instruction sequences and compiles frequently interpreted instruction sequences to platform-specific instructions.

Java is multithreaded: To improve the performance of programs that must accomplish several tasks at once, Java support the concept of threaded execution.

Java is dynamic: Because interconnections between program code and libraries happen dynamically at runtime, it isn’t necessary to explicitly link then. Dynamic behaviour results in less code to distribute when a version change occurs.

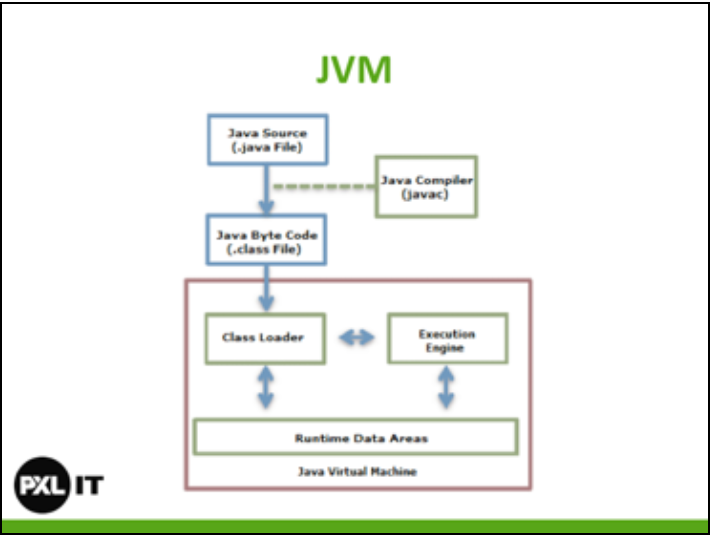
Java is strongly typed : Every variable must be declared with a datatype.

## JVM

JRE (Java Runtime Environment) : composed of Java API and JVM.

Role of JVM is to read the Java application through the Class Loader and execute it along with the Java API.

JVM is an emulator that emulates the Java Bytecode.



Java Source 🡪 Java Compiler 🡪 Java Byte Code 🡪 JVM

Java Source wordt via Java Compiler naar Java Byte Code vertaalt.

Class loader : Bootstrap Class loader en Extension Class Loader

Executions engine: JIT compiler en GC

Runtime Data Areas : Memory allocation

## Java bytecode

Middle-language between Java and machine language. Smallest unit that deploys Java code.

Java bytecode has no platform-dependent code, it is executable on the hardware where the JVM has been installed.

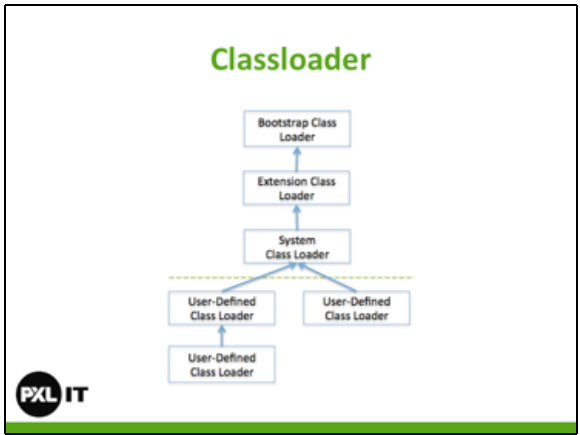
First 4 bytes of Java bytecode file are the magic: ca fe ba be.  
🡪 When the first 4 bytes of a file is 0xCAFEBABE it can be regarded as a Java class file.

## Classloader

Java provides a dynamic load feature. It loads and links the class when it refers to a class for the first time at runtime (NOT COMPILE TIME). JVM’s class loader executes the dynamic load.

Features of Java Class Loader:

* Hierarchical structure: Class loaders in Java are organized into a hierarchy with a parent-child relantionship. The Bootstrap Class Loader is the parent of all class loaders.
* Delegation mode: Based on the hierarchical structure, load is delegated between class loaders. When a class is loaded, the parent class loader is checked to determine whether or not the class is in the parent class loader; if the upper class loader has the class, the class is used; if not the class loader requested for loading loads the class.
* Visibility limit: A child class loader can find the class in the parent class loader, however a parent class loader cannot find the class in the child class loader.
* Unload is not allowed: A class loader can load a class but cannot unload it. Instead the current class loader can be deleted and a new class loader can be created.

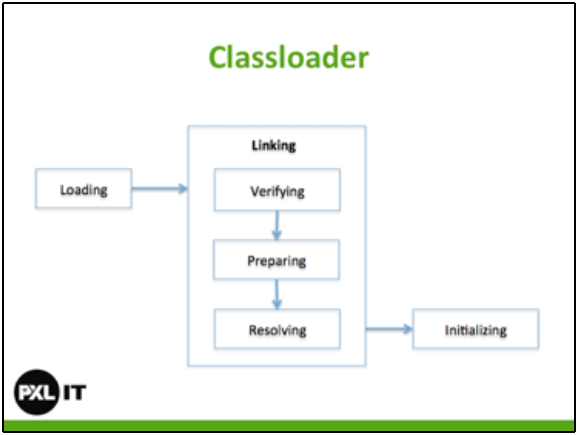


Bootstrap Class Loader: This is created when running the JVM. It load Java APIs, including object classes. Unlike other class loaders, it is implemented in native code.

Extension Class Loader: It loads the extension classes excluding the basic Java APIs. It also loads various security extension functions.

System Class Loader: It loads the application classes in the $CLASSPATH specified by the user.

User-defined Class Loader: This is a class loader that an application user directly creates on the code.  
Framework such as Web Application Server use it to make Web applications and enterprise applications run independently. This guarantees the independence of applications through class loader delegation model.

Stages of a Classloader:

Loading: A class is obtained from a file and loaded to the JVM memory

Verifying : Check whether or not the read class is configured as described in the Java Language Specification and JVM specifications. Most complicated test process, takes the longest time.

Preparing: Prepare a data structure that assigns the memory required by cases and indicates the field, methods, and interfaces defined in the class.

Resolving: Change all symbolic references in the constant pool of the class to direct references.

Initializing: Initialize the class variables to proper values. Execute the static initializers and initialize the static field to the configured values.

## Heap and stack

Stack frame: One stack frame is created whenever a method is executed in the JVM and the stack frame is added to the JVM stack of the thread. When the method is ended, the stack frame is removed. Each stack frame has the reference for local variable array, Operand stack and runtime constant pool of a class where the method being executed belongs. Size of local variable array and Operand stack is determined while compiling. Size of stack frame is fixed according to the method.

Local variable array: Index starting from 0. 0 is the reference of a class instance where the method belongs. From 1, the parameters sent to the method are saved. After the method parameters, the local variables of the method are save.

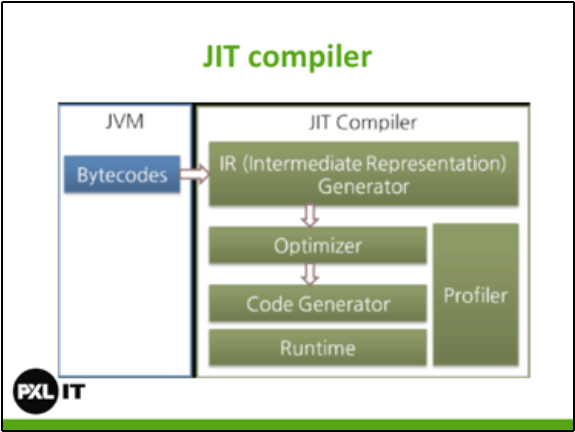
Operand stack: An actual workspace of a method. Each method exchanges data between the Operand Stack and the local variable array, and pushes or pops other method invoke results. Size of Operand stack space can be determined during compiling.

Native method stack: A stack for native code written in another language than Java. This stack is used to execute C/C++ code invoked through JNI (Java native Interface). According to the language, a C stack or C++ stack is created.

Method area: The method area is created when the JVM starts and is shared by all threads. Stores runtime constant pool, field and method information, static variable and method bytecode for each of the classes and interfaces read by the JVM. GC for method area is optional for each JVM vendor.

Heap: A space that stores instances or objects. Is the target of GC.

JIT Compiler  
Java Source Code 🡪 Java Compiler 🡪 Bytecode 🡪 JIT Compiler 🡪 Native code



## Scope

Scope refers to the lifetime and accessibility of a variable. How large the scope is depends on where a variable is declared.

## Statements

* Expression statements: change values of variables, call methods and create objects.
* Declaration statements: declare variables.
* Control flow statements: determine if statements are executed.

## Reference types

Data type that’s based on a class. When assigning object to a variable, a reference is assigned to it instead of the actual object. The reference is the address of the memory location where the object is stored. Variable contains in his stack memory a reference to the heap memory.

# Default methods

A method that is implemented in an interface is called a default method. The method in the interface need the **default** keyword.

If a class implements two interfaces that implements the same default method, then you need to override that method in the class. You can make the class its own implementation or call in the method the default implementation of one or the other interface : InterfaceName.super.methodName()

# Nested classes

Four kinds of nested class:

* Static class: declared as a static member of another class
* Inner class: declared as an instance member of another class
* Local inner class: declared inside an instance method of another class
* Anonymous inner class: like a local inner class, but written as an expression which returns a one-off object

## Static nested class

public class OuterClass {

private static int classField = 1;

public static class NestedClass{

private int nestedField;

public NestedClass(){

nestedField = classField++;

}

...

}

}

Because it is a static class, the class belongs to the OuterClass and not to an object of it.  
Thanks to this you can initialize the class from outside OuterClass:

OuterClass.NestedClass nested = new OuterClass.NestedClass();

The nested class has access to all **static** private variables of the OuterClass.

## Inner class

Is a not-static nested class.

A inner class is not static and belongs to an object of the OuterClass.

A inner class has access to all private variable of the OuterClass.

public class OuterClass {

private int field = 1;

public class InnerClass{

public InnerClass(){

}

public void initialize(){

field = 5;

}

}

public void doSomething(){

InnerClass i = new InnerClass();

i.initialize();

System.out.println(field);

}

}

## Local inner class

Class that is declared in a method. A local inner class cannot modify local variables. Compiler considers them as final.

So the following is wrong:

public class OuterClass {

public Object getInner(){

int val = 7;

class LocalInnerClass{

public String toString(){

val++; // THIS IS WRONG

return "Inner " + val;

}

}

return new LocalInnerClass();

}

}

## Anonymous class

public class OuterClass {

public void method(){

SuperClass object1 = new SuperClass(){

// override methods

}

}

}

As the methods of SupperClass only are defined for object1, they can only be called in the method() method.

## Callback method

A callback is some code that you pass to a given method, so that it can be called at a later time.  
- Tom Schuyten

## Functional interface

An interface that implements only one method is called a functional interface.

* Supplier: No parameter but a return value
* Function: Parameter and return value
* Consumer: Parameter but no return value
* Predicate: Parameter and Boolean return value

## Method reference

Qualifier::identifier

|  |  |
| --- | --- |
| val -> ClassName.staticMethod(val) | ClassName::staticMethod |
| val -> myObject.method(val) | myObject::method |
| (Type val) -> val.method() | Type::method |
| val -> new ClassName(val) | ClassName::new |

# Multithreading

BlockingQueue: Thread waiting for another thread to puts or request something from it.

Threadpool: Set van een aantal Thread dat tegelijkertijd mag uitgevoerd worden.

Latch: Counter voor Threading. Via de wait() methode laat je de Main tread wachten tot deze op 0 staat en pas vanaf dan mag de Main thread verder.

Future: Type die dient als een soort placeholder. Op initialisatie geef je de implementatie die de waarde (de Callable) zal genereren. Bij aanroepen van de get() methode op de Future zal de Callable aangeroepen worden en zal ofwel een random waarde genereren, ofwel de waarde uit de queue halen. Genereren/ophalen van de randoms gebeurt dynamisch, daarom gebruikt men Future objecten.  
Gebeurt via de Threadpool. Je submit telkens de Callable zodat dit telkens op een nieuwe Thread gebeurt maar aangezien de Threadpool en max heeft kunnen er maar x aantal Thread tegelijk lopen.

# Garbage collection

GC: Garbage Collection

Heap: Where instances or objects are stored  
Stack: Reference to object

Responsiveness: How quickly an application or system responds to a request.

Throughput: Hoeveelheid werk dat verricht kan worden door een applicatie.  
Maximizing the amount of work by an application in a specific period of time.  
Example of how throughput might be measured: The number of transactions completed in a given time, the number of jobs that a batch program can complete in an hour or the number of database queries that can be completed in an hour.  
High pause times are acceptable for applications that focus on throughput because high throughput applications focus on benchmarks over longer periods of time.

## Automatic GC

Automatic GC is the process of looking at heap memory, identifying which objects are in use and which are not and deleting the unused objects.

### Step 1: Marking

This is where the GC identifies which pieces of memory are in use and which are not.

### Step 2: Normal Deletion

Normal deletion removes unreferenced objects leaving references objects and pointers to free space.   
The memory allocator holds references to blocks of free space where a new objects can be allocated.

### Step 2a: Deletion with compacting

To further improve performance, in addition to deleting unreferenced objects, you can also compact the remaining referenced objects. By moving referenced objects together, this makes new memory allocation much easier and faster.

## JVM Generations

Young Generation: where all new objects are allocated and aged. When young generation fills up, causes minor GC. Some surviving objects are aged and eventually move to the old generation.

Stop the World Event: All minor GC are “Stop the World” events which means that all application threads are stopped until the operation completes.

Old Generation: used to store long surviving objects. Eventually the old generation needs to be collected, this event is called a Major GC.

Major GC are also Stop the World events. Major GC are often much slower because it involves all live objects. So Major GC should be minimized. The length of the Stop the World event is affected by the kind of garbage collector that is used for the old generation space.

## GC process

First, any new objects are allocated to the eden space. Both survivor spaces start out empty.

Second, when the eden space fills up, a minor GC is triggered.

Third, referenced objects are moved to the first survivor space (S0). Unreferenced objects are deleted when the eden space is cleared.

Fourth, as the next minor GC, the same happens for the eden space. But in this case, referenced objects are moved to the second survivor space (S1) and objects from the last minor GC have their age incremented and get also moved to S1. Once all surviving objects have been moved to S1, both S0 and eden are cleared.

Fifth: As the next minor GC, the same process repeats. However this time the survivor spaces switch. Referenced objects are move to S0, surviving objects are aged, Eden and S1 are cleared.

Sixth: After a certain amount of minor GC, when aged objects reach a certain age threshold, they are promoted from young generation to old generation.

Seventh: As minor GCs continue to occur, objects will continue to be promoted to the old generation space.

Eighth: Eventually, a major GC can be performed on the old generation which cleans up and compacts that space.

# Application logging

same name for appender:

log4j.rootLogger=DEBUG, stdout, file  
log4j.appender.stdout=...  
log4j.appender.file=...

You can give your own name to the properties stdout and file:

log4j.rootLogger=DEBUG, something, somethingE  
log4j.appender.something=...  
log4j.appender.somethingE=...

DEBUG < INFO < WARN < ERROR < FATAL

bij rootlogger config, given type log the type and all higher levels.  
WARN -> log WARN, ERROR and FATAL

Declare logger:

final static Logger logger = Logger.getLogger(classname.class);