PXL IT

W1: Language execution basics (Java)

.NET / Java

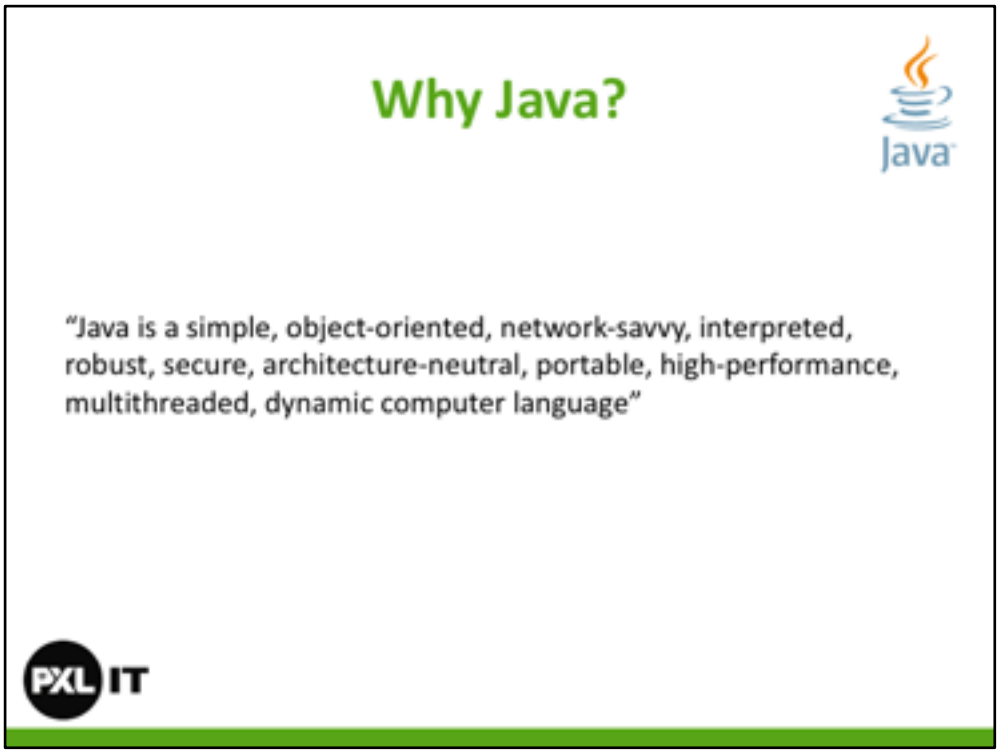
DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Dep. PXL-IT – Elfde-Liniestraat 26 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

What is Java?

- 97% of Enterprise Desktops Run Java
- 89% of Desktops (or Computers) in the U.S. Run Java
- 9 Million Java Developers Worldwide
- #1 Choice for Developers
- #1 Development Platform
- 3 Billion Mobile Phones Run Java
- 100% of Blu-ray Disc Players Ship with Java
- 5 Billion Java Cards in Use
- 125 million TV devices run Java
- 5 of the Top 5 Original Equipment Manufacturers Ship Java ME

PXL IT



What is Java?

You can think of Java as a general-purpose, object-oriented language that looks a lot like C and C++, but which is easier to use and lets you create more robust programs. Unfortunately, this definition doesn't give you much insight into Java. A more detailed definition from Sun Microsystems is as relevant today as it was in 2000:

Java is a simple, object-oriented, network-savvy, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded, dynamic computer language.Let's consider each of these definitions separately:

Java is a simple language.

Java was initially modeled after C and C++, minus some potentially confusing features. Pointers, multiple implementation inheritance, and operator overloading are some C/C++ features that are not part of Java. A feature not mandated in C/C++, but essential to Java, is a garbage-collection facility that automatically reclaims objects and arrays.

Java is an object-oriented language.

Java's object-oriented focus lets developers work on adapting Java to solve a problem, rather than forcing us to manipulate the problem to meet language constraints. This is different from a structured language like C. For example, whereas Java lets you focus on

savings account objects, C requires you to think separately about savings account *state* (such a balance) and *behaviors* (such as deposit and withdrawal).

Java is a network-savvy language.

Java's extensive network library makes it easy to cope with Transmission Control Protocol/Internet Protocol (TCP/IP) network protocols like HTTP (HyperText Transfer Protocol) and FTP (File Transfer Protocol), and simplifies the task of making network connections. Furthermore, Java programs can access objects across a TCP/IP network, via Uniform Resource Locators (URLs), with the same ease as you would have accessing them from the local file system.

Java is an interpreted language.

At runtime, a Java program indirectly executes on the underlying platform (like Windows or Linux) via a virtual machine (which is a software representation of a hypothetical platform) and the associated execution environment. The virtual machine translates the Java program's bytecodes (instructions and associated data) to platform-specific instructions through interpretation. *Interpretation* is the act of figuring out what a bytecode instruction means and then choosing equivalent "canned" platform-specific instructions to execute. The virtual machine then executes those platform-specific instructions.
Interpretation makes it easier to debug faulty Java programs because more compile-time information is available at runtime. Interpretation also makes it possible to delay the link step between the pieces of a Java program until runtime, which speeds up development.

Java is a robust language.

Java programs must be reliable because they are used in both consumer and mission-critical applications, ranging from Blu-ray players to vehicle-navigation or air-control systems. Language features that help make Java robust include declarations, duplicate type checking at compile time and runtime (to prevent version mismatch problems), true arrays with automatic bounds checking, and the omission of pointers. (We will discuss all of these features in detail later in this series.)
Another aspect of Java's robustness is that loops must be controlled by Boolean expressions instead of integer expressions where 0 is false and a nonzero value is true. For example, Java doesn't allow a C-style loop such as while (x) x++; because the loop might not end where expected. Instead, you must explicitly provide a Boolean expression, such as while (x != 10) x++; (which means the loop will run until x equals 10).

Java is a secure language.

Java programs are used in networked/distributed environments. Because Java programs can migrate to and execute on a network's various platforms, it's important to safeguard these platforms from malicious code that might spread viruses, steal credit card information, or perform other malicious acts. Java language features that support robustness (like the omission of pointers) work with security features such as the Java sandbox security model and public-key encryption. Together these features prevent

viruses and other dangerous code from wreaking havoc on an unsuspecting platform. In theory, Java is secure. In practice, various security vulnerabilities have been detected and exploited. As a result, Sun Microsystems then and Oracle now continue to release security updates.

Java is an architecture-neutral language.

Networks connect platforms with different architectures based on various microprocessors and operating systems. You cannot expect Java to generate platform-specific instructions and have these instructions "understood" by all kinds of platforms that are part of a network. Instead, Java generates platform-independent bytecode instructions that are easy for each platform to interpret (via its implementation of the JVM).

Java is a portable language.

Architecture neutrality contributes to portability. However, there is more to Java's portability than platform-independent bytecode instructions. Consider that integer type sizes must not vary. For example, the 32-bit integer type must always be signed and occupy 32 bits, regardless of where the 32-bit integer is processed (e.g., a platform with 16-bit registers, a platform with 32-bit registers, or a platform with 64-bit registers). Java's libraries also contribute to portability. Where necessary, they provide types that connect Java code with platform-specific capabilities in the most portable manner possible.

Java is a high-performance language.

Interpretation yields a level of performance that is usually more than adequate. For very high-performance application scenarios Java uses just-in-time compilation, which analyzes interpreted bytecode instruction sequences and compiles frequently interpreted instruction sequences to platform-specific instructions. Subsequent attempts to interpret these bytecode instruction sequences result in the execution of equivalent platform-specific instructions, resulting in a performance boost.

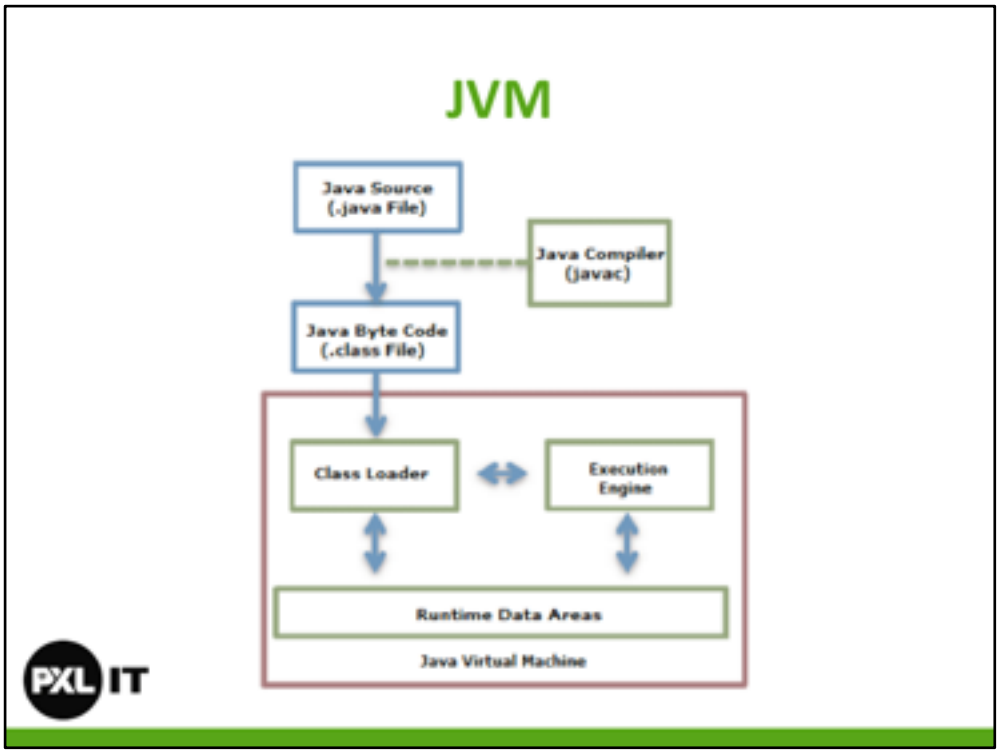
Java is a multithreaded language.

To improve the performance of programs that must accomplish several tasks at once, Java supports the concept of *threaded execution*. For example, a program that manages a Graphical User Interface (GUI) while waiting for input from a network connection uses another thread to perform the wait instead of using the default GUI thread for both tasks. This keeps the GUI responsive. Java's synchronization primitives allow threads to safely communicate data between themselves without corrupting the data. (See threaded programming in Java discussed elsewhere in the Java 101 series.)

Java is a dynamic language.

Because interconnections between program code and libraries happen dynamically at runtime, it isn't necessary to explicitly link them. As a result, when a program or one of its

libraries evolves (for instance, for a bug fix or performance improvement), a developer only needs to distribute the updated program or library. Although dynamic behavior results in less code to distribute when a version change occurs, this distribution policy can also lead to version conflicts. For example, a developer removes a class type from a library, or renames it. When a company distributes the updated library, existing programs that depend on the class type will fail. To greatly reduce this problem, Java supports an *interface type*, which is like a contract between two parties. (See interfaces, types, and other object-oriented language features discussed elsewhere in the Java 101 series.)



The JRE is composed of the Java API and the JVM. The role of the JVM is to read the Java application through the Class Loader and execute it along with the Java API.

A virtual machine (VM) *is a software implementation of a machine (i.e. a computer) that executes programs like a physical machine*. Originally, Java was designed to run based on a virtual machine separated from a physical machine for implementing **WORA** (*Write Once Run Anywhere*), although this goal has been mostly forgotten. Therefore, the JVM runs on all kinds of hardware to execute the **Java Bytecode** without changing the Java execution code.

The features of JVM are as follows:

Stack-based virtual machine: The most popular computer architectures such as Intel x86 Architecture and ARM Architecture run based on a *register*. However, *JVM runs based on a stack*.

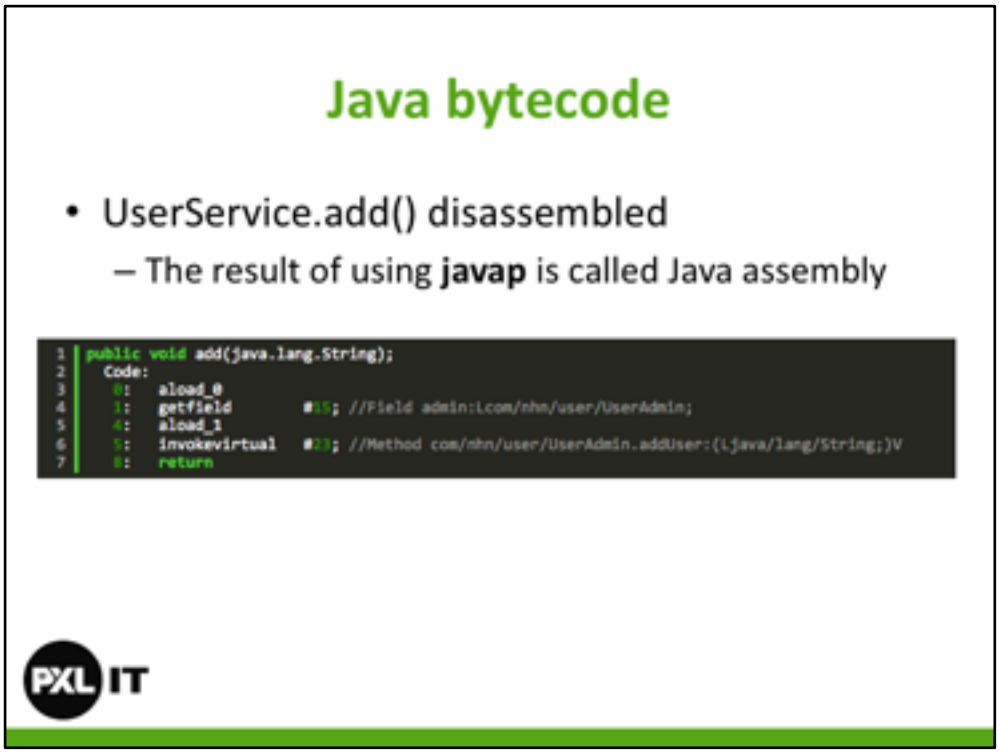
Symbolic reference: All types (class and interface) except for primitive data types are referred to through symbolic reference, instead of through explicit memory address-based reference.

Garbage collection: A class instance is explicitly created by the user code and automatically destroyed by garbage collection.

Guarantees platform independence by clearly defining the primitive data type: A traditional language such as C/C++ has different int type size according to the platform. The JVM clearly defines the primitive data type to maintain its compatibility and guarantee platform independence.

Network byte order: The Java class file uses the network byte order. To maintain platform independence between the little endian used by Intel x86 Architecture and the big endian used by the RISC Series Architecture, a fixed byte order must be kept. Therefore, JVM uses the network byte order, which is used for network transfer. The network byte order is the big endian.

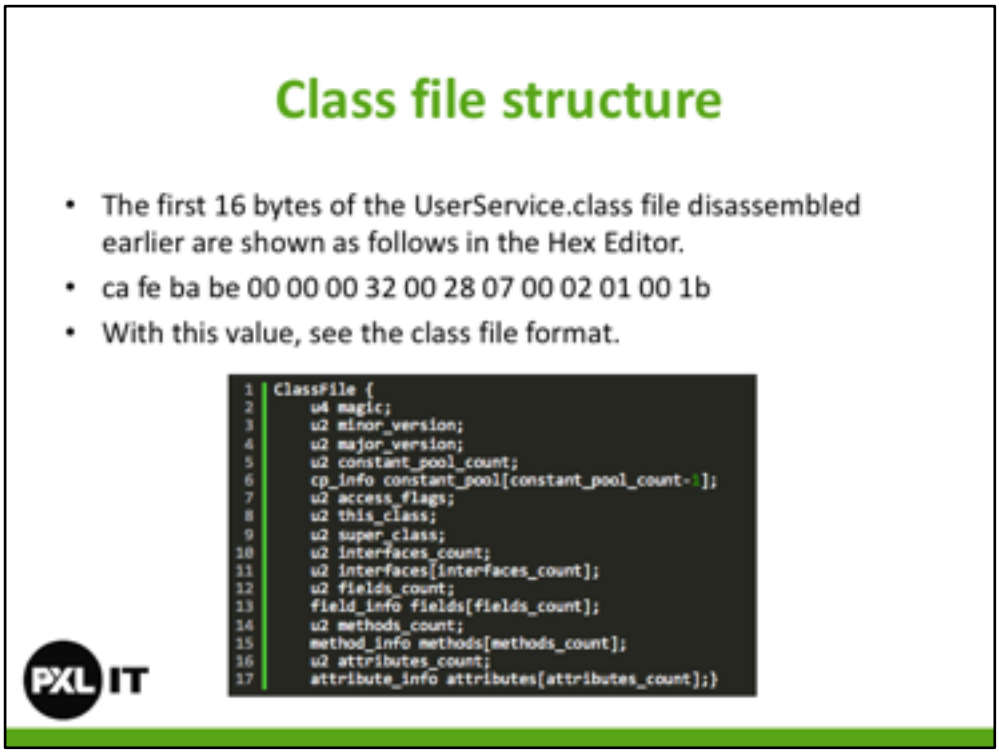
Sun Microsystems developed Java. However, any vendor can develop and provide a JVM by following the Java Virtual Machine Specification. For this reason, there are various JVMs, including Oracle Hotspot JVM and IBM JVM. The Dalvik VM in Google's Android operating system is a kind of JVM, though it does not follow the Java Virtual Machine Specification. Unlike Java VMs, which are stack machines, the Dalvik VM is a register-based architecture. Java bytecode is also converted into an register-based instruction set used by the Dalvik VM.



Java bytecode

To implement WORA, the JVM uses Java bytecode, a middle-language between Java (user language) and the machine language. This Java bytecode is the smallest unit that deploys the Java code.

Java Bytecode is the essential element of JVM. The JVM is an emulator that emulates the Java Bytecode. Java compiler does not directly convert high-level language such as C/C++ to the machine language (direct CPU instruction); it converts the Java language that the developer understands to the Java Bytecode that the JVM understands. Since Java bytecode has no platform-dependent code, it is executable on the hardware where the JVM (accurately, the JRE of the same profile) has been installed, even when the CPU or OS is different (a class file developed and compiled on the Windows PC can be executed on the Linux machine without additional change.) The size of the compiled code is almost identical to the size of the source code, making it easy to transfer and execute the compiled code via the network.



With this value, see the class file format.

magic: The first 4 bytes of the class file are the magic number. This is a pre-specified value to distinguish the Java class file. As shown in the Hex Editor above, the value is always 0xCAFEBABE. In short, when the first 4 bytes of a file is 0xCAFEBABE, it can be regarded as the Java class file. This is a kind of "witty" magic number related to the name "Java".

minor_version, major_version: The next 4 bytes indicate the class version. As the UserService.class file is 0x00000032, the class version is 50.0. The version of a class file compiled by JDK 1.6 is 50.0, and the version of a class file compiled by JDK 1.5 is 49.0. The JVM must maintain backward compatibility with class files compiled in a lower version than itself. On the other hand, when a upper-version class file is executed in the lower-version JVM, java.lang.UnsupportedClassVersionError occurs.

constant_pool_count, constant_pool[]: Next to the version, the class-type constant pool information is described. This is the information included in the Runtime Constant Pool area, which will be explained later. While loading the class file, the JVM includes the constant_pool information in the Runtime Constant Pool area of the method area. As the constant_pool_count of the UserService.class file is 0x0028, you can see that the

constant_pool has (40-1) indexes, 39 indexes.

access_flags: This is the flag that shows the modifier information of a class; in other words, it shows public, final, abstract or whether or not to interface.

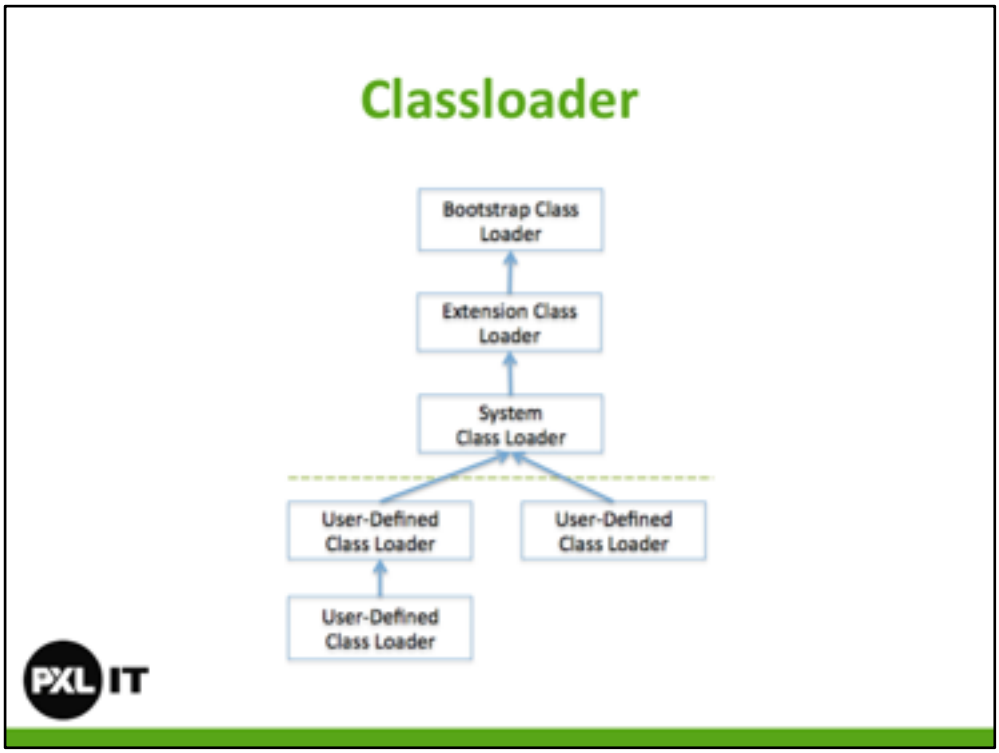
this_class, super_class: The index in the constant_pool for the class corresponding to this and super, respectively.

interfaces_count, interfaces[]: The index in the the constant_pool for the number of interfaces implemented by the class and each interface.

fields_count, fields[]: The number of fields and the field information of the class. The field information includes the field name, type information, modifier, and index in the constant_pool.

methods_count, methods[]: The number of methods in a class and the methods information of the class. The methods information includes the methods name, type and number of the parameters, return type, modifier, index in the constant_pool, execution code of the method, and exception information.

attributes_count, attributes[]: The attribute_info structure has various attributes. For field_info or method_info, attribute_info is used.



Class Loader

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. The features of Java class loader are as follows:

Hierarchical Structure: Class loaders in Java are organized into a hierarchy with a parent-child relationship. 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 of unloading, the current class loader can be deleted, and a new class loader can be

created.

Each class loader has its namespace that stores the loaded classes. When a class loader loads a class, it searches the class based on FQCN (Fully Qualified Class Name) stored in the namespace to check whether or not the class has been already loaded. Even if the class has an identical FQCN but a different namespace, it is regarded as a different class. A different namespace means that the class has been loaded by another class loader.

When a class loader is requested for class load, it checks whether or not the class exists in the class loader cache, the parent class loader, and itself, in the order listed. In short, it checks whether or not the class has been loaded in the class loader cache. If not, it checks the parent class loader. If the class is not found in the bootstrap class loader, the requested class loader searches for the class in the file system.

Bootstrap class loader:

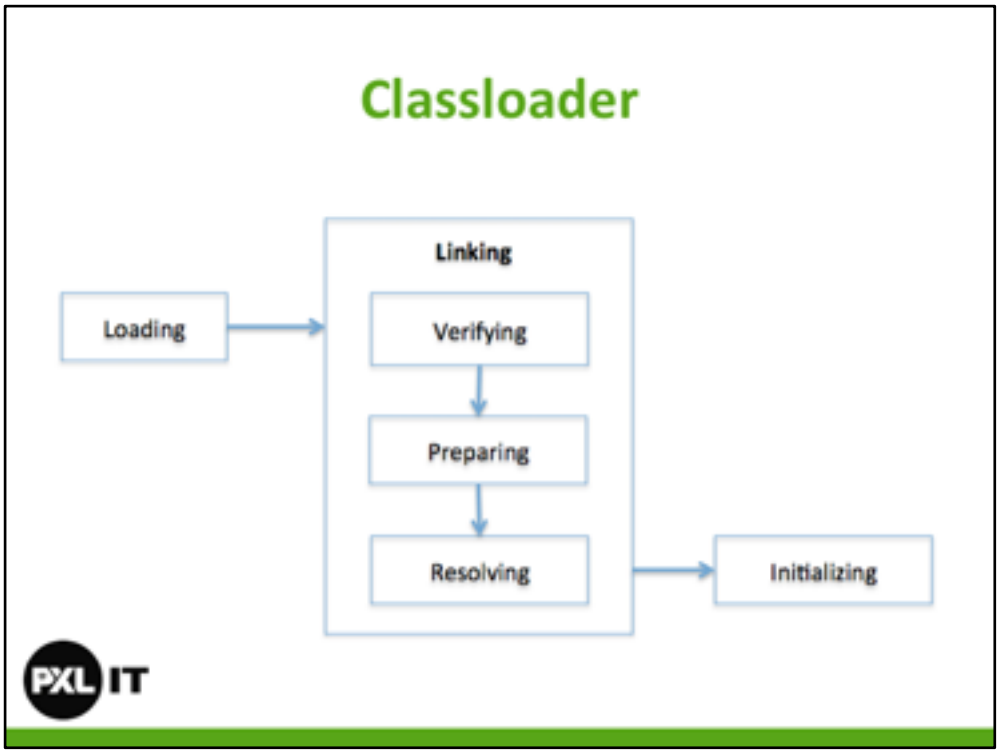
This is created when running the JVM. It loads Java APIs, including object classes. Unlike other class loaders, it is implemented in native code instead of Java.

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

System class loader: If the bootstrap class loader and the extension class loader load the JVM components, the system class loader loads the application classes. It loads the class 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.

Frameworks such as Web application server (WAS) use it to make Web applications and enterprise applications run independently. In other words, this guarantees the independence of applications through class loader delegation model. Such a WAS class loader structure uses a hierarchical structure that is slightly different for each WAS vendor.



Each stage is described as follows.

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

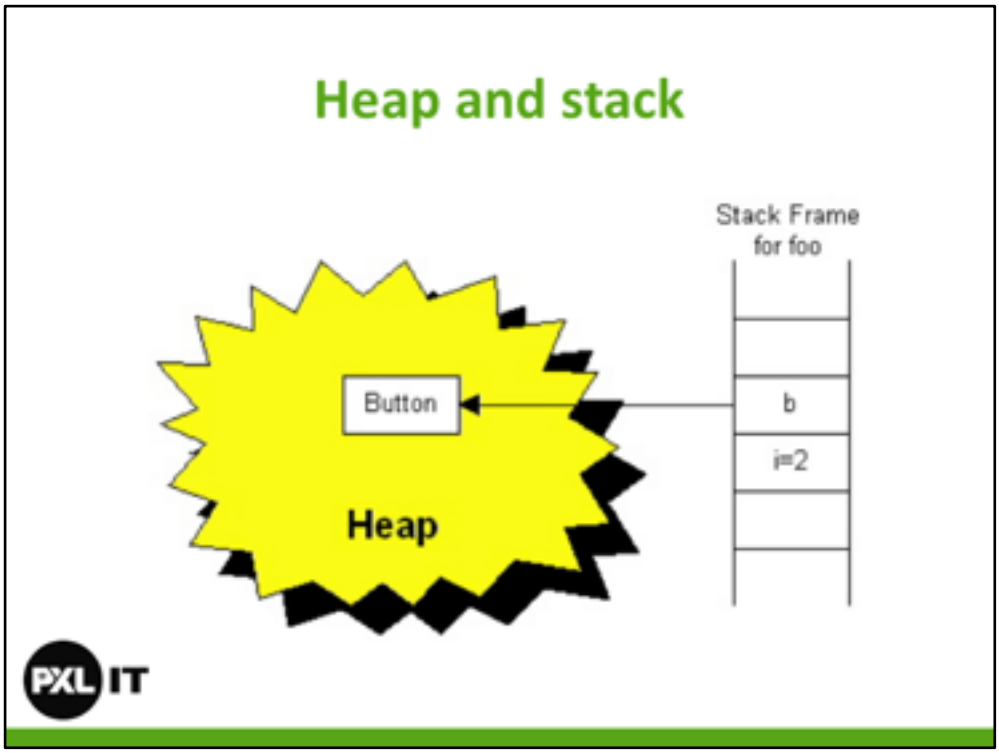
Verifying: Check whether or not the read class is configured as described in the Java Language Specification and JVM specifications. This is the most complicated test process of the class load processes, and takes the longest time. Most cases of the JVM TCK test cases are to test whether or not a verification error occurs by loading wrong classes.

Preparing: Prepare a data structure that assigns the memory required by classes and indicates the fields, 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 fields to the configured values.

The JVM specification defines the tasks. However, it allows flexible application of the execution time.



JVM Stack Configuration.

− 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. The size of local variable array and Operand stack is determined while compiling. Therefore, the size of stack frame is fixed according to the method.

− Local variable array: It has an 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 saved.

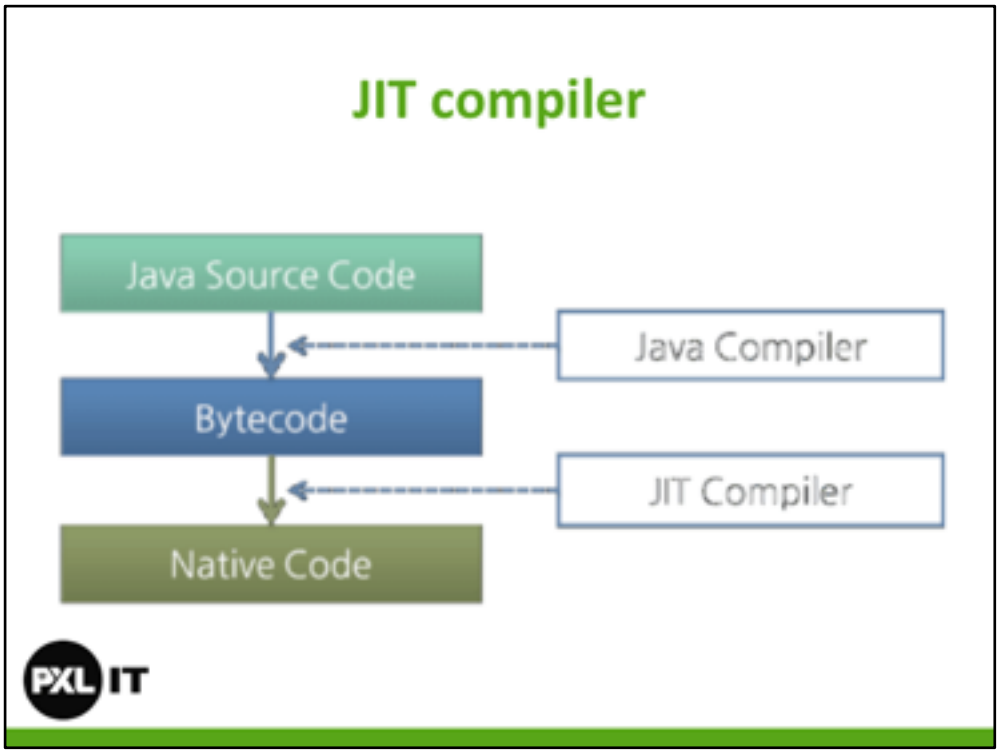
− Operand stack: An actual workspace of a method. Each method exchanges data between the Operand stack and the local variable array, and pushes or pops other method invoke results. The necessary size of the Operand stack space can be determined during compiling. Therefore, the size of the Operand stack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

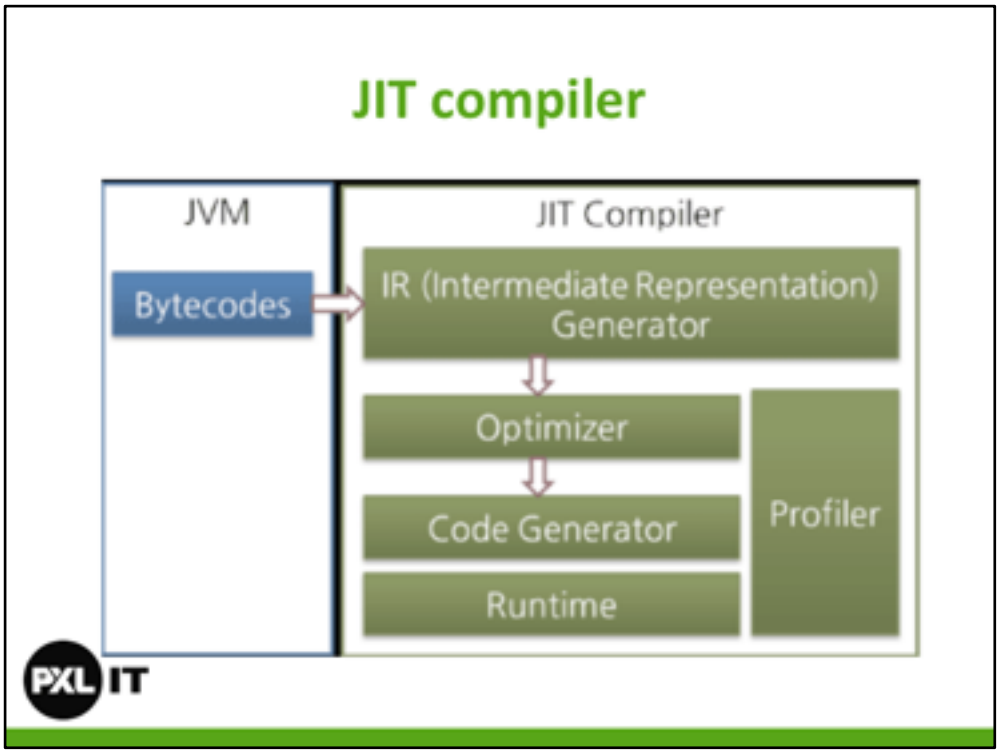
Native method stack: A stack for native code written in a language other than Java. In other words, it is a stack used to execute C/C++ codes invoked through JNI (Java Native Interface). According to the language, a C stack or C++ stack is created.

Method area: The method area is shared by all threads, created when the JVM starts. It stores runtime constant pool, field and method information, static variable, and method bytecode for each of the classes and interfaces read by the JVM. The method area can be implemented in various formats by JVM vendor. Oracle Hotspot JVM calls it Permanent Area or Permanent Generation (PermGen). The garbage collection for the method area is optional for each JVM vendor.

Runtime constant pool: An area that corresponds to the constant_pool table in the class file format. This area is included in the method area; however, it plays the most core role in JVM operation. Therefore, the JVM specification separately describes its importance. As well as the constant of each class and interface, it contains all references for methods and fields. In short, when a method or field is referred to, the JVM searches the actual address of the method or field on the memory by using the runtime constant pool.

Heap: A space that stores instances or objects, and is a target of garbage collection. This space is most frequently mentioned when discussing issues such as JVM performance. JVM vendors can determine how to configure the heap or not to collect garbage.


JIT compiler
Java Source Code
Java Compiler
Bytecode
JIT Compiler
Native Code
PXL IT


JIT compiler
JVM
Bytecodes
JIT Compiler
IR (Intermediate Representation) Generator
Optimizer
Code Generator
Runtime
Profiler
PXL IT

Java is strongly typed

- Java is a strongly typed programming language because every variable must be declared with a data type. A variable cannot start off life without knowing the range of values it can hold, and once it is declared, the data type of the variable cannot change.
- **Example:** boolean hasDataType;

Variable

Definition:

- A variable is a container that holds values that are used in a Java program. Every variable must be declared to use a data type. For example, a variable could be declared to use one of the eight primitive data types: byte, short, int, long, float, double, char or boolean. And, every variable must be given an initial value before it can be used.
- **Examples:**
  - int myAge = 21; The variable "myAge" is declared to be an int data type and initialized to a value of 21.

Declaration statement

Definition:

- A declaration statement is used to declare a variable by specifying its data type and name. int number; boolean isFinished; String welcomeMessage; In addition to the data type and name, a declaration statement can initialize the variable with a value:

```
int number;
boolean isFinished = false;
int number, anotherNumber, yetAnotherNumber;
```

Parameter

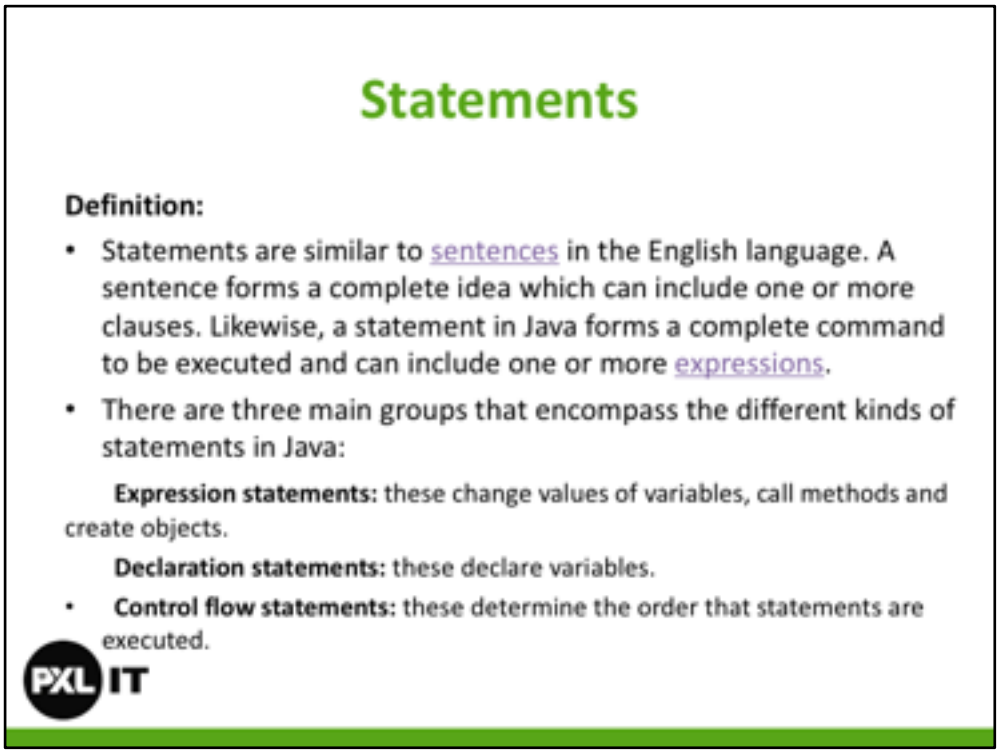
Definition:

- Parameters are the variables that are listed as part of a method declaration. Each parameter must have a unique name and a defined data type.

Scope

Definition:

- Scope refers to the lifetime and accessibility of a variable. How large the scope is depends on where a variable is declared. For example, if a variable is declared at the top of a class then it will accessible to all of the class methods. If it's declared in a method then it can only be used in that method.

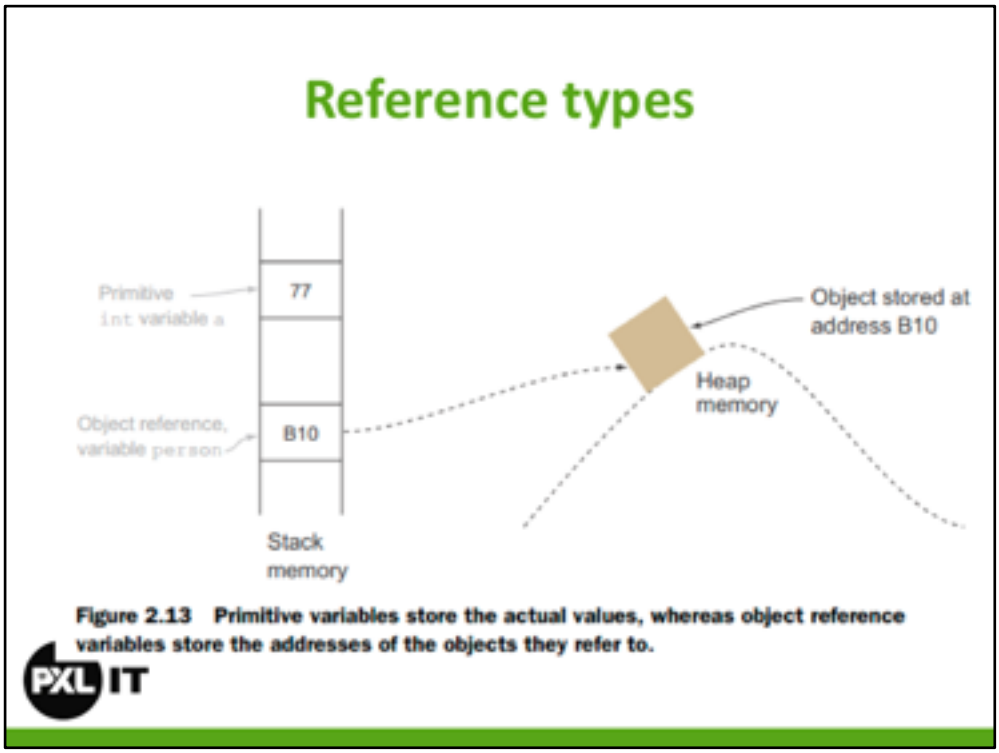


Examples:

```
//declaration statement int number;

//expression statement number = 4;

//control flow statement if (number < 10 ) {
                //expression statement
                System.out.println(number  + " is less than ten");
}
```



A *reference type* is a data type that's based on a class rather than on one of the primitive types that are built in to the Java language. The class can be a class that's provided as part of the Java API class library or a class that you write yourself.

Either way, when you create an object from a class, Java allocates the amount of memory the object requires to store the object. Then, if you assign the object to a variable, the variable is actually assigned a *reference* to the object, not the object itself. This reference is the address of the memory location where the object is stored.

To declare a variable using a reference type, you simply list the class name as the data type. For example, the following statement defines a variable that can reference objects created from a class named Ball:

Ball b;

You must provide an import statement to tell Java where to find the class.

To create a new instance of an object from a class, you use the new keyword along with the class name:

```
Ball b = new Ball();
```

One of the key concepts in working with reference types is the fact that a variable of a particular type doesn't actually contain an object of that type. Instead, it contains a reference to an object of the correct type. An important side effect is that two variables can refer to the same object.

Consider these statements:

```
Ball b1 = new Ball();
Ball b2 = b1;
```

Here, both b1 and b2 refer to the same instance of the Ball class.

Enum type

- In the Java programming language, you define an enum type by using the enum keyword. For example, you would specify a days-of-the-week enum type as:
  - public enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }
- You should use enum types any time you need to represent a fixed set of constants where you know all possible values at compile time—for example, the choices on a menu, command line flags, and so on.

Enum type

- *All* enums implicitly extend java.lang.Enum. Since Java does not support multiple inheritance, an enum cannot extend anything else.
- Enum in Java are type-safe: Enum has there own name-space. It means your enum will have a type for example “Company” in below example and you can not assign any value other than specified in Enum Constants.

Exercise

- Build a standalone console application that creates a number objects during a given time (use a Java Thread).
- Use JConsole to inspect your application
- Use Pluralsight for more information about the topics seen today.

Resources

- http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/