The Future of Java: Welke richting gaan we uit? Dit zijn de rauwe notas van de gevonden informatie over dit onderwerp. Neem deze kennis mee op want hiervan is er ook 1 examenvraag.

- 8->9 modularity (Jigsaw)
    - Verdeelt jdk in modules voor combineren run-compile-build
    - JShell? Java + REPL
- Ahead of time compilation? Gebruikt Graal compiler Java 9 extensie voor de JVM om meer programmeertalen en extensies te aanvaarden.
- http2 client support, sneller dan http1.
- Verbeteringen in de stream API, condities toepassen in streams.
- CodeCash? Optimalisatie snelheid verbetering.

Sinds Java 9: Java linker? Om uitvoer bestand maken inclusief JVM.

Java DB verwijderd sinds Java 9, (opensource DB Derby van Apache).

JavaFX included in JDK, gaan splitsen uit JDK als apart opensource project, makkelijker om updates te releasen, niet wachten op JDK.

6 maandelijks update releases van de JDK.

JDK10: Variabelen zonder type declareren.

Optional class vanaf JDK 10 .orElseThrow method.

JDK 10: API calls voor het maken van onaantastbare collections.

Java support voor meerdere stylesheets in JavaDoc

Java 10: Bytecode generation forloops enhanced
Java 10: Garbage collector interface, GC helpen?
Java 10: uitbreiding aan de tijd API.

RELEASES: origineel 2 jaar. Java 9 heel veel delay en besloten elk half jaar te releasen en niet wachten op grote features. En om kwartaal ook al updates, fixes en security issue patches.
Releases september en maart.

Java 11 uit sinds september. Bevat nieuwe Garbage Collector. Noop GC, optie om GC uit te zetten.
GC gebaseerd op Redhat technology. Z GC is de noop GC .

JavaFX is verwijderd, opensource API afgestoten om sneller updates te kunnen doen.

Supporteren niet alles, opletten hier. TLS1.3 Java11.

Klas 2:

8 veel security en bugfixes. Daarna meer nadruk op features en bugfixes.

Snellere releases om sneller bug fixes te hebben en nieuwe features.
LTS niet voor alle versies 1.8 en 1.11

FX en JavaDB en JEE? Loggekoppeld, snellere updates mogelijk. Als onderdeel van jdk lang wachten op updates. 1.11 losgekoppeld.

1.10 var list = new ArrayList<Person> : Type inference.

1.10 GC interface, performantere GC, gebruikt meer threads.
1.10 Experimentele Jit compiler.
1.9 improvements in try with resources.
Enhancements stream API.
1.9 JLink welke modules includen bij runtime.
SOAP, webstart uit Jdk.
Javah ook uit JDK.
1.11 applets eruit.

1.11 Collection.toArray()
1.11 Deel talen gedropt.

Sinds 1.9 root, corba, hotspot, jaxp, jaxws, jdk, lang tools en nashorn verschillende trees samengevoegd om bugfixing makkelijker te maken.

Klas 3:

Sneller releases, minder grote veranderingen en makkelijk updates voor developers. Grotere versies LTS en kleinere versies w support sneller stop gezet (9-10).

Var in Java10 en heap op andere plaats stockeren, als stroom uitvalt bij prog runnen.
Vanaf 11 var in lambdas.

Java 10: Experimentele VM -> meerdere talen runnen hierop. Graal compiler.

Java 11: ZGC barbage collector laagst mogelijke latency en overhead.

Java http client standaard in 11.

Java 11: Flight Recorder, inkijk in je JVM.
Java 11 – JavaEE, FX en corba modules weg.

1.8 Collections.toArray is nieuw.

PXL IT

W1: Language execution basics (Java)

.NET / Java

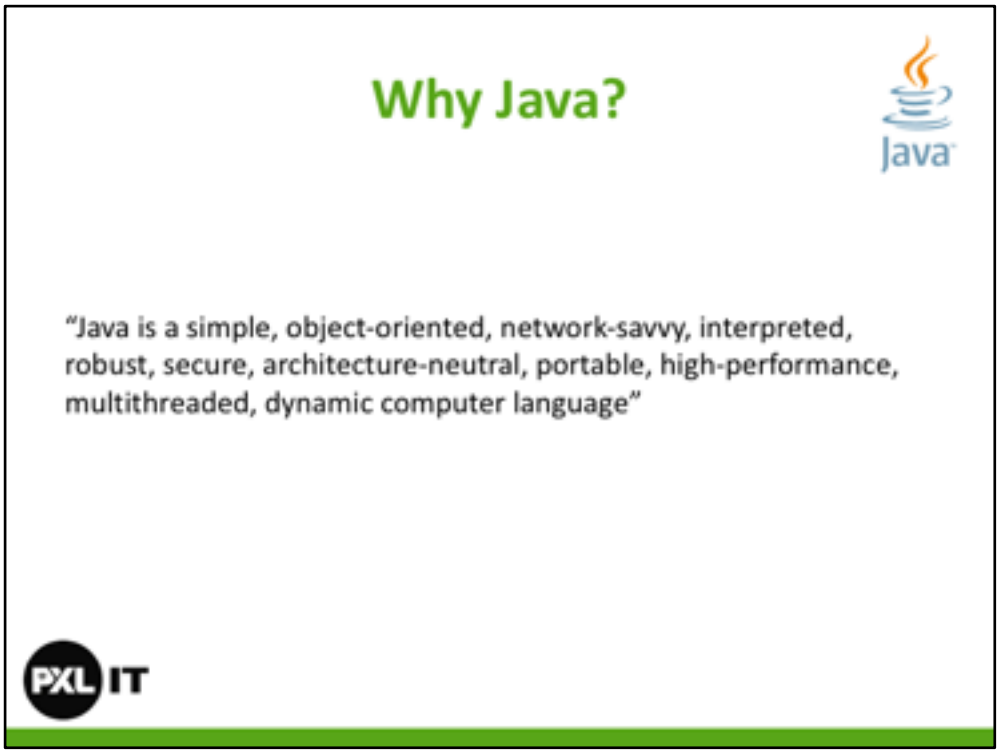
DE HOGESCHOOL
MET HET NETWERK

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.

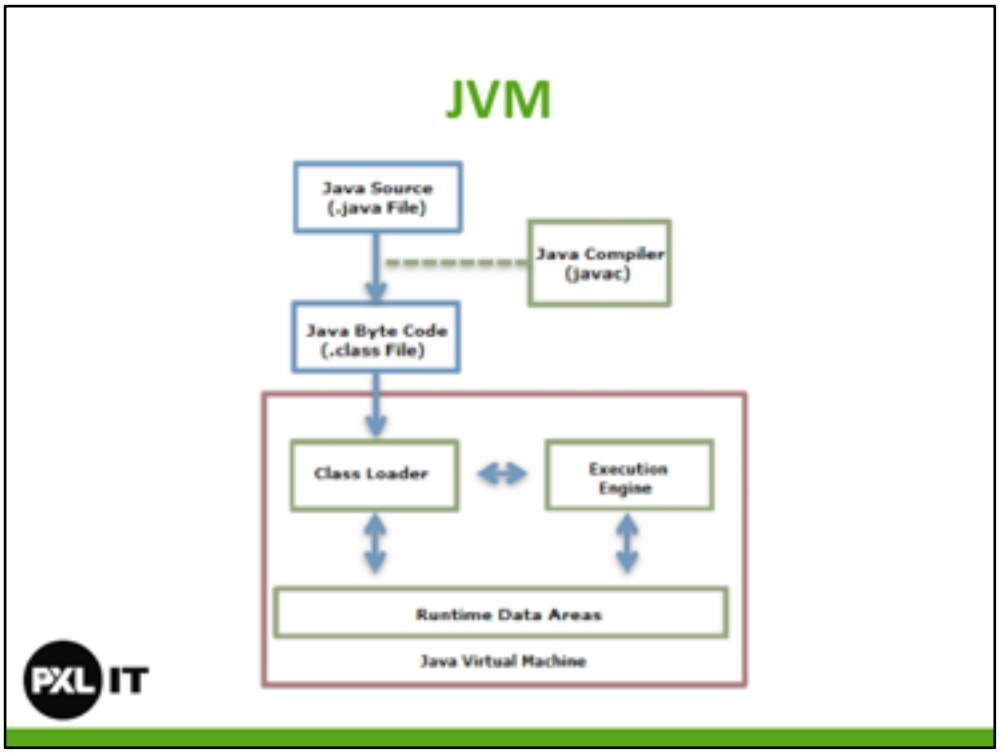
Java is a multithreaded language.

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*.

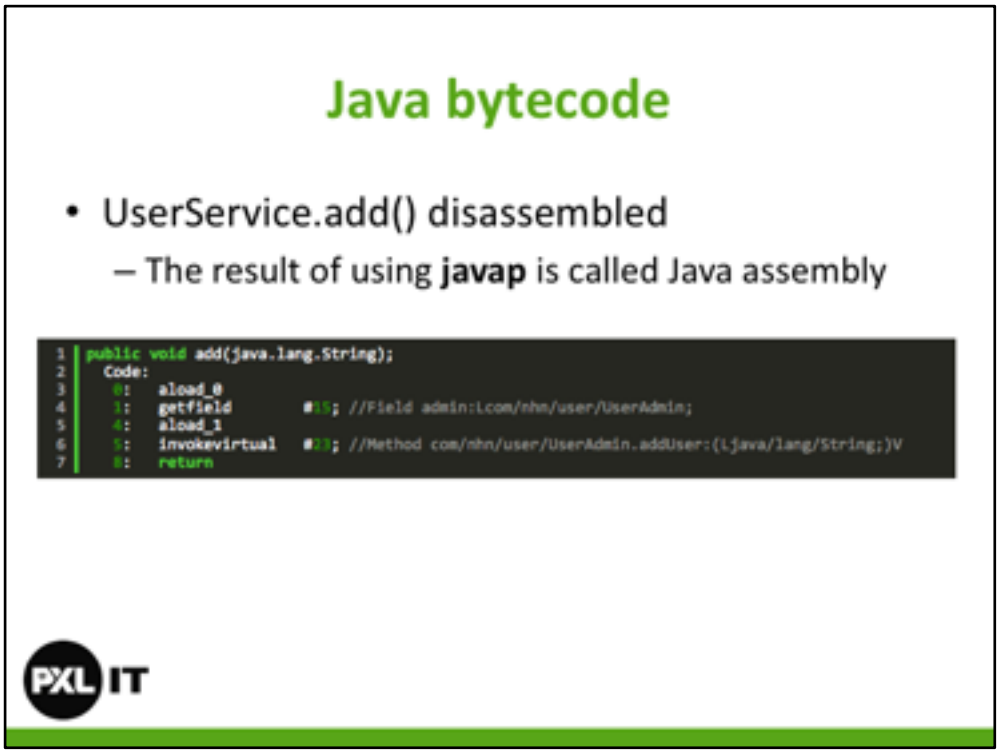
Symbolic reference: All types (class and interface) except for primitive data types are referred to through symbolic reference, instead of through explicit memory address-based reference.

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.

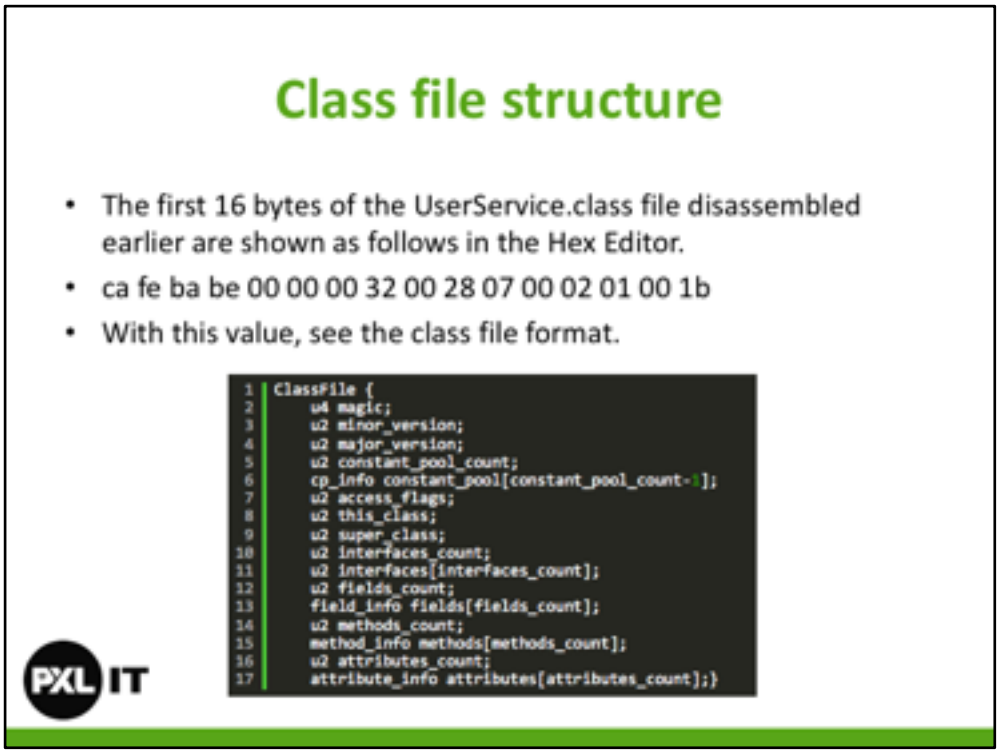
Sun Microsystems developed Java. However, any vendor can develop and provide a JVM by following the Java Virtual Machine Specification. For this reason, there are various JVMs, including Oracle Hotspot JVM and IBM JVM. The Dalvik VM in Google's Android operating system is a kind of JVM, though it does 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.

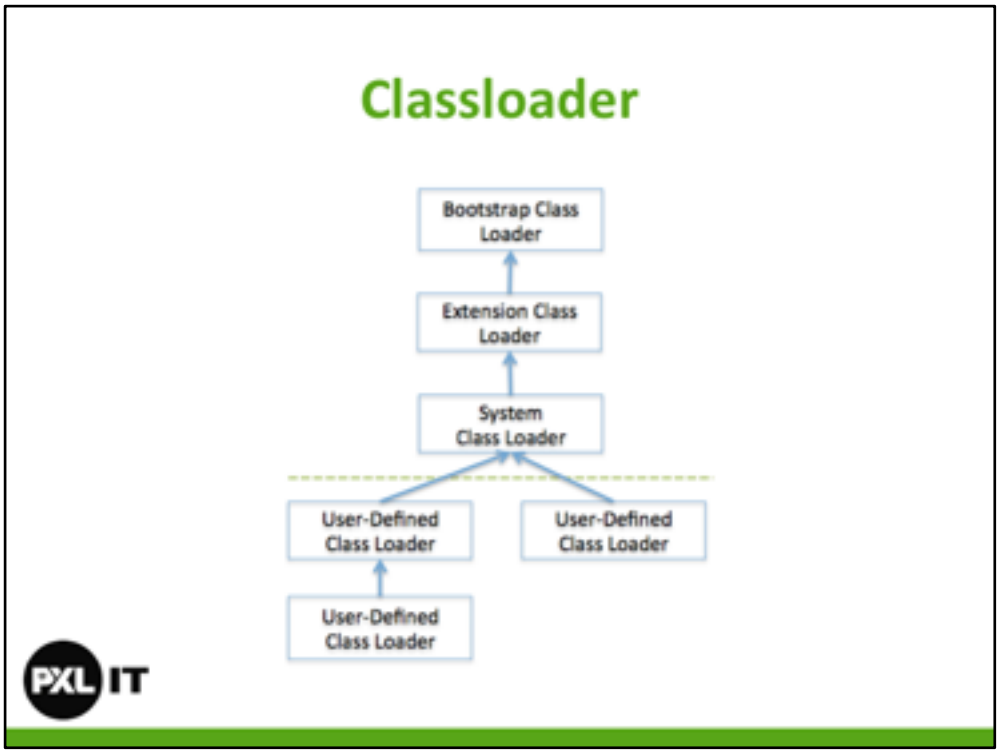
this_class, super_class: The index in the constant_pool for the class corresponding to this and super, respectively.

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:

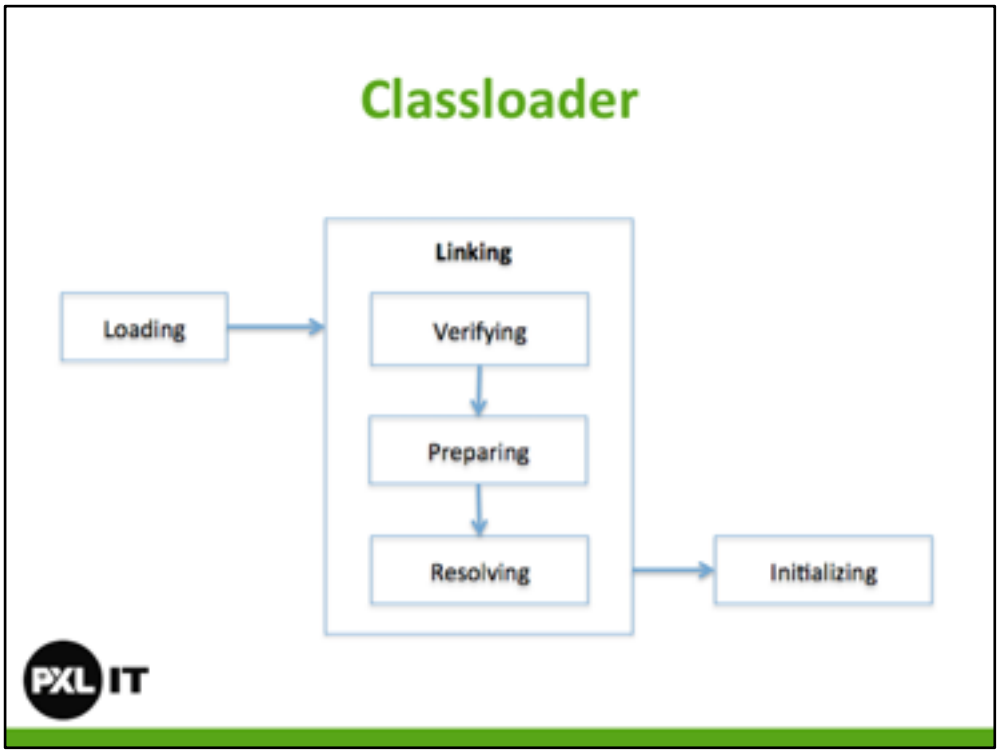
This is created when running the JVM. It loads Java APIs, including object classes. Unlike other class loaders, it is implemented in native code instead of Java.

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.

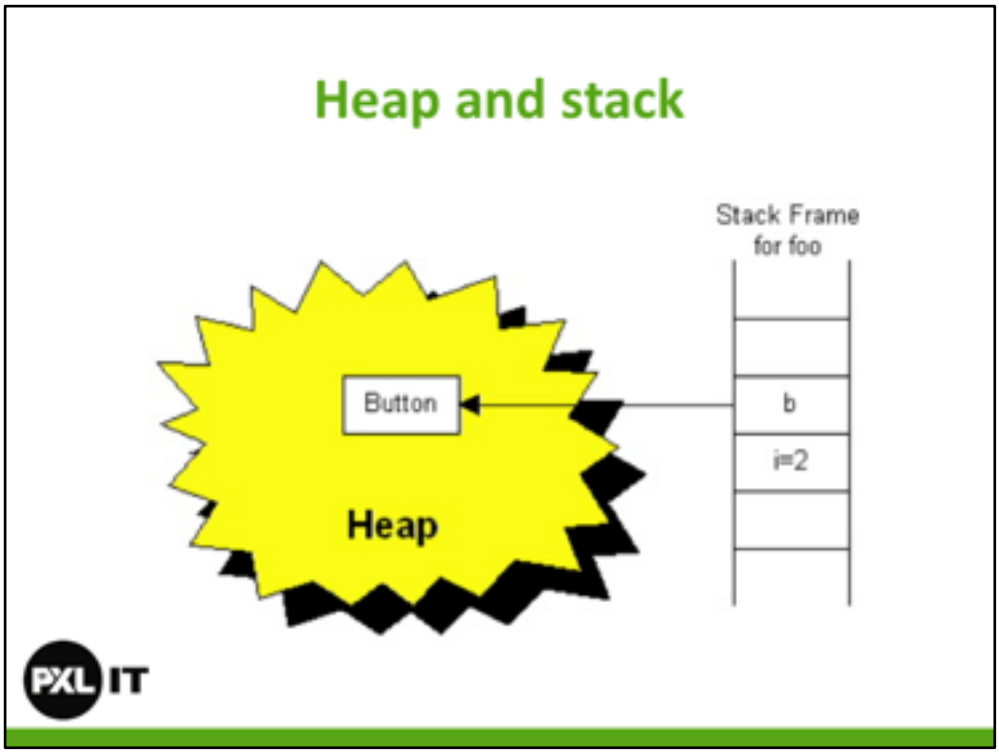
Verifying: Check whether or not the read class is configured as described in the Java Language Specification and JVM specifications. This is the most complicated test process of the class load processes, and takes the longest time. Most cases of the JVM TCK test cases are to test whether or not a verification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

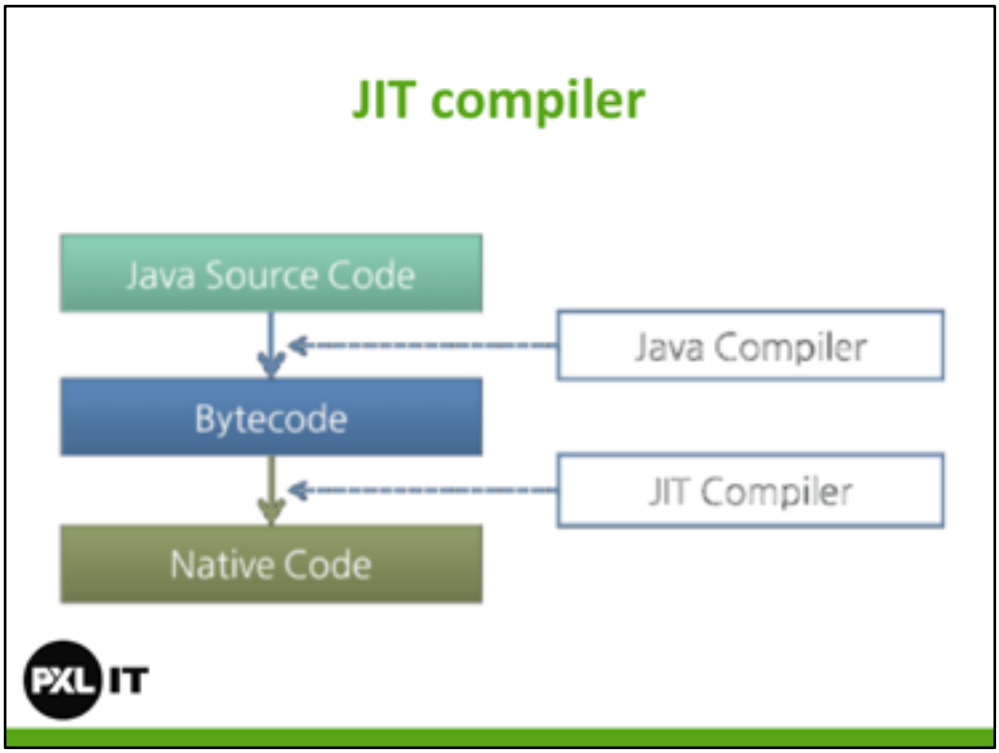
− Operand stack: An actual workspace of a method. Each method exchanges data between the Operand stack and the local variable array, and pushes or pops other method invoke results. The necessary size of the Operand stack space can be determined during compiling. Therefore, the size of the Operand stack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

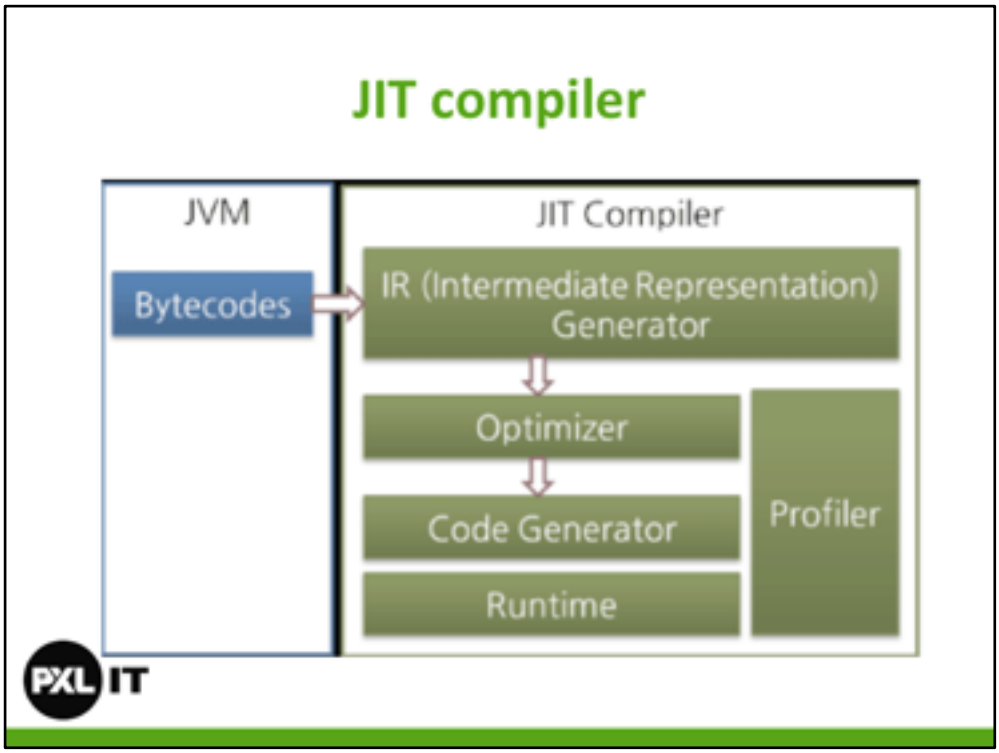
Native method stack: A stack for native code written in a language other than Java. In other words, it is a stack used to execute C/C++ codes invoked through JNI (Java Native Interface). According to the language, a C stack or C++ stack is created.

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

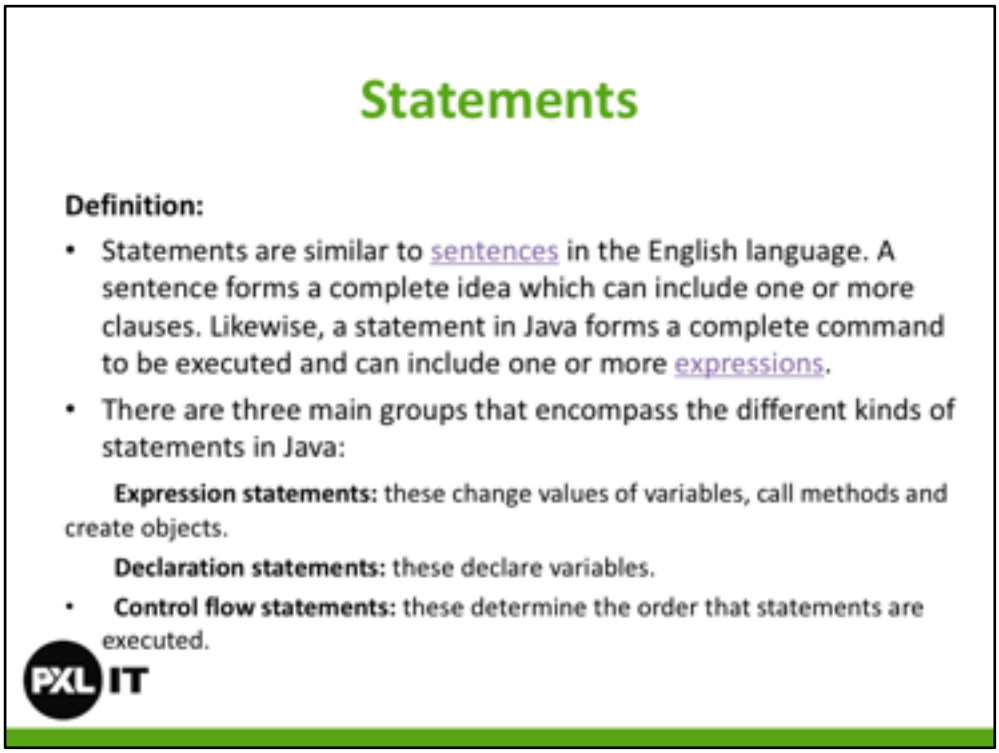
Definition:

- 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

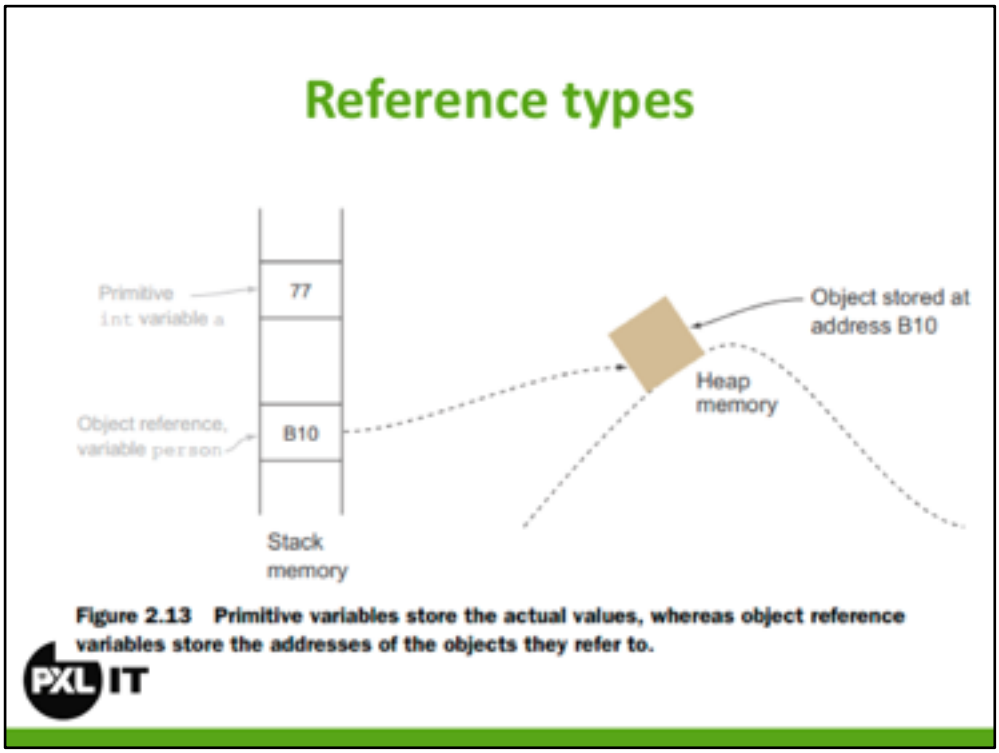
Definition:

- 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

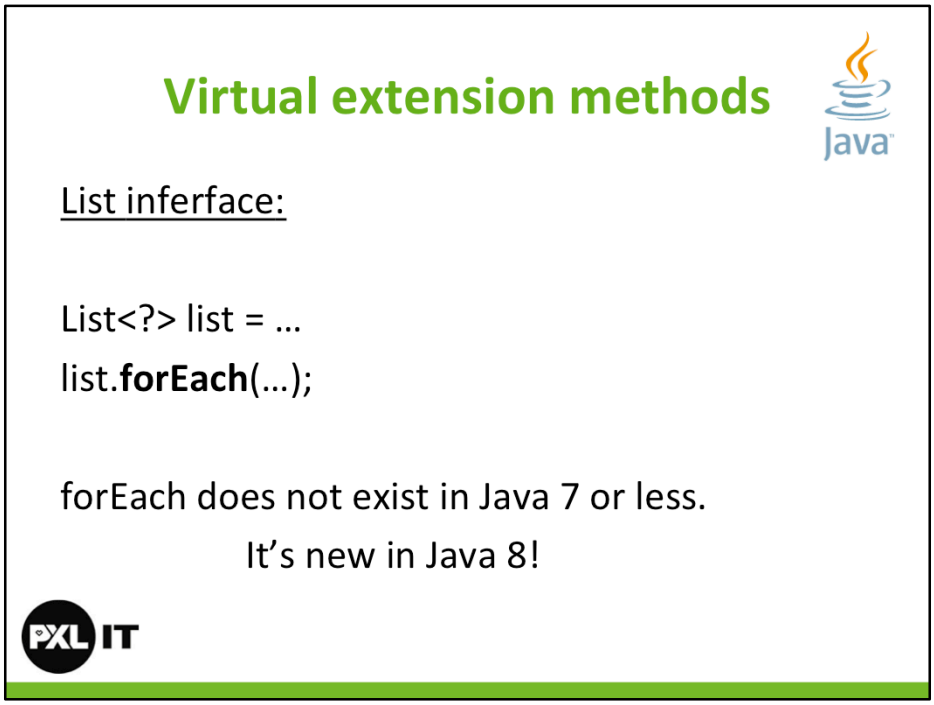
- http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/

W2: Default methods (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



The forEach isn't declared by java.util.List nor the java.util.Collection interface yet. One obvious solution would be to just add the new method to the existing interface and provide the implementation where required in the JDK. However, once published, it is impossible to add methods to an interface without breaking the existing implementation.

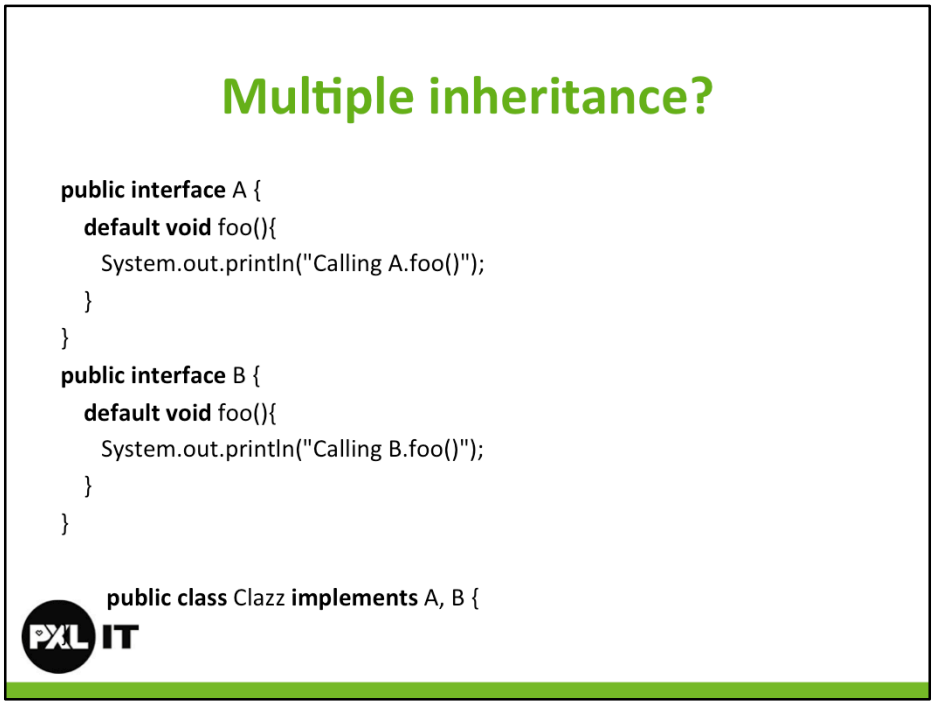
Due to the problem described above a new concept was introduced. Virtual extension methods, or, as they are often called, defender methods, can now be added to interfaces providing a default implementation of the declared behavior.

Simply speaking, interfaces in Java can now implement methods. The benefit that default methods bring is that now it's possible to add a new default method to the interface and it doesn't break the implementations.

A simple example


```
public interface A {
   default void foo(){
      System.out.println("Calling A.foo()");
   }
}
public class Clazz implements A {
}

Clazz clazz = new Clazz();
clazz.foo(); // Calling A.foo()
```

The code compiles even though Clazz does not implement method foo(). Method foo() default implementation is now provided by interface A.



There is one common question that people ask about default methods when they hear about the new feature for the first time: *"What if the class implements two interfaces and both those interfaces define a default method with the same signature?"*.

This code fails…

java: class Clazz inherits unrelated defaults for foo() from types A and B

Fix:

```
public class Clazz implements A, B {
   public void foo(){}
}
```

But what if we would like to call the default implementation of method foo() from interface A instead of implementing our own. It is possible to refer to refer to A#foo() as follows:

```
public class Clazz implements A, B {
   public void foo(){
     A.super.foo();
   }
}
```

Real example

```
@FunctionalInterface
public interface Iterable<T> {
   Iterator<T> iterator();

   default void forEach(Consumer<? super T> action) {
     Objects.requireNonNull(action);
     for (T t : this) {
          action.accept(t);
     }
   }
```

http://stackoverflow.com/questions/4343202/difference-between-super-t-and-extends-t-in-java

Method invocation

One interface!

```
A clazz = new Clazz();
clazz.foo(); // invokeinterface foo()

Clazz clazz = new Clazz();
clazz.foo(); // invokevirtual foo()
```

From the client code perspective, default methods are just ordinary virtual methods. Hence the name – virtual extension methods. So in case of the simple example with one class that implements an interface with a default method, the client code that invokes the default method will generate invokeinterface at the call site.

Method invocation

Multiple interfaces!

```
public class Clazz implements A, B {
   public void foo(){
     A.super.foo(); // invokespecial foo()
   }
}
```

In case of the default methods conflict resolution, when we override the default method and would like to delegate the invocation to one of the interfaces the invokespecial is inferred as we would call the implementation specifically:

javap output

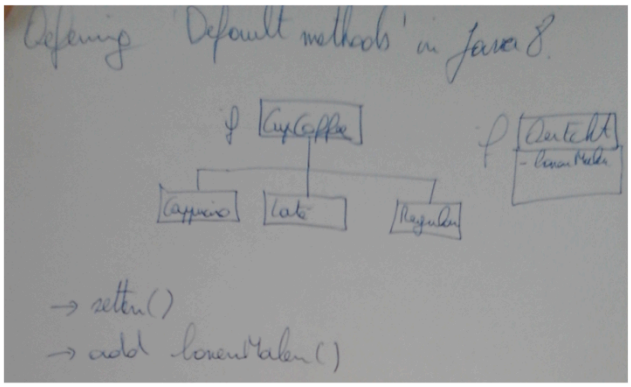
```
public void foo();
Code:
0: aload_0
1: invokespecial #2 // InterfaceMethod A.foo:()V
4: return
```

As you can see, invokespecial instruction is used to invoke the interface method foo(). This is also something new from the bytecode point of view as previously you would only invoke methods via super that points to a class (parent class), and not to an interface.

Finally

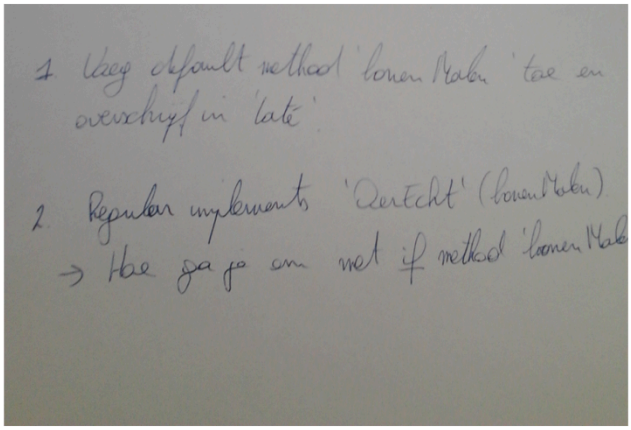
- The primary goal of default methods is to enable an evolution of standard JDK interfaces and provide a smooth experience when we finally start using lambdas in Java 8.

Excercise



PXL IT

Excercise



PXL IT

Todo: paginatitels aanpassen zodat structuur duidelijker wordt
Todo: slides verbergen/schrappen om een vlottere flow in de les te verkrijgen

Inhoud

Nested class
Inner class
Local inner class
Anonymous class
Lambda
Functional Interface
Method Reference

There are **four kinds of nested class in Java**. In brief, they are:
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

Voorkennis

Klassen

Interfaces

static

1 klasse, 1 bestand

new ActionListener() { … }

ActionListener is gezien bij GUI applicaties vorig jaar, maar is een Interface.

Static nested class

```
public class OuterClass {
    public static class NestedClass {
        ....
    }
}
```

Gebruik (buitenaf):

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

NestedClass noemt men een static nested class. Static: de klasse hoort in de outer klasse, niet in een object van de outer klasse. De klasse kan wel geinstantieerd worden.

Static nested class

```
public class OuterClass {
    private static int classField = 1;

    public static class NestedClass {
        private int nestedField;
        public NestedClass() {
            nestedField = classField++;
        }
    }
}
```

NestedClass heeft toegang tot private classField.

Static nested class

```
public class OuterClass {
    private int field = 1;

    public static class NestedClass {
        public void setField(OuterClass outer, int val) {
            outer.field = val;
        }
    }
}
```

NestedClass heeft onrechtstreeks toegang tot de private data van instanties van OuterClass.
Merk op, setField kan hier evengoed static zijn. NestedClass (object) heeft geen rechtstreekse toegang tot field, want het heeft geen standaard instantie van OuterClass object om mee te werken.

Static nested class

```
public class OuterClass {
    private NestedClass nested;

    public OuterClass() {
        nested = new NestedClass();
    }

    public int getValue() {
        return nested.calculate();
    }

    private static class NestedClass {
        private int calculate() { … };
    }
}
```

NestedClass kan private, package, protected of public zijn. Bij private kan alleen OuterClass er aan, bijv. in de constructor. Bij protected enkel subklassen _van OuterClass_ en klassen in hetzelfde package.
Verder kan de OuterClass ook aan de private methodes en velden van de NestedClass

Static nested class: voorbeeld

```
public class ColorEnums {
    public enum Color {
        RED, GREEN, BLUE, YELLOW, …;
    }
}
```

Gebruik:

```
ColorEnums.Color color = ColorEnum.Color.RED;
```

Bij enum mag static worden weggelaten.

Nested class

Samenvatting:

- Static nested class kan geïnstantieerd worden
- Static nested class (instantie) zit in de Outer class
- private ≈ in hetzelfde bestand

9

Verder zijn extends, abstract, final, enz. zijn ook mogelijk op nested classes. “In hetzelfde bestand” is een goede techniek om het te onthouden, mits rekening te houden met static. NestedClass1 kan bijvoorbeeld aan NestedClass2.privateData.

Inner class

```
public class OuterClass {
    private int field = 1;

    public class InnerClass {
        private int nestedField;
        public InnerClass() {
            nestedField = field;
        }
    }
}
```

Inner class is niet static en zit conceptueel in een object van OuterClass i.p.v. in de class zelf. Het directe gevolg is rechtstreekse toegang tot private non-static velden en methodes.
Cursus over inner class: “De geneste klasse kan alleen gebruikt worden binnen de context van de nestende klasse.” De naam van het klassebestand wordt OuterClass $InnerClass.class

Inner class

```
public class OuterClass {
    public void doSomething() {
        InnerClass inner = new InnerClass();
        inner.doMethod();
    }

    public class InnerClass {
        public void doMethod () { … }
    }
}
```

Aanmaak van een object van de inner class binnen de nestende klasse.

Inner class

```
public class OuterClass {
    public void doSomething() {
        InnerClass inner = this.new InnerClass();
        inner.doMethod();
    }

    public class InnerClass {
        public void doMethod () { … }
    }
}
```

Exact hetzelfde, nu met expliciete this-verwijzing. De InnerClass is in de context van een object van OuterClass en vereist dus een referentie. In de vorige slide was het gewoon impliciet.

Inner class

```
public class OuterClass {
    public class InnerClass {  }
}

OuterClass outer = new OuterClass();
OuterClass.InnerClass inner;
inner = outer.new InnerClass();
```

Hetzelfde, van buitenaf. Het datatype is dus OuterClass.InnerClass (volgens cursus. Na import mijnpkg.OuterClass.InnerClass kan InnerClass op zich ook)

Inner class

```
public class OuterClass {
    private int field = 1;

    public class InnerClass {
        private int field;
        public InnerClass(int field) {
            this.field = field;
            OuterClass.this.field = field;
        }
    }
}
```

Shadowing “field”. Met OuterClass.this kom je aan de context, de instantie van OuterClass

Local inner class

```
public class OuterClass {
    public void method() {
        final int CONST = 5;
        int val = 7;

        class LocalInnerClass {
            ...
            System.out.println(CONST); // OK
            System.out.println(val);       // OK (val wijzigt niet).
            ...
        }
        LocalInnerClass local = new LocalInnerClass();
    }
}
```

Een local inner class is een klasse gedefinieerd in een methode. Enkel binnen de methode kan de klasse geinstantieerd worden. De local inner class kan aan lokale variabelen die niet veranderen. De compiler beschouwt ze als final (sinds Java 8).

Local inner class

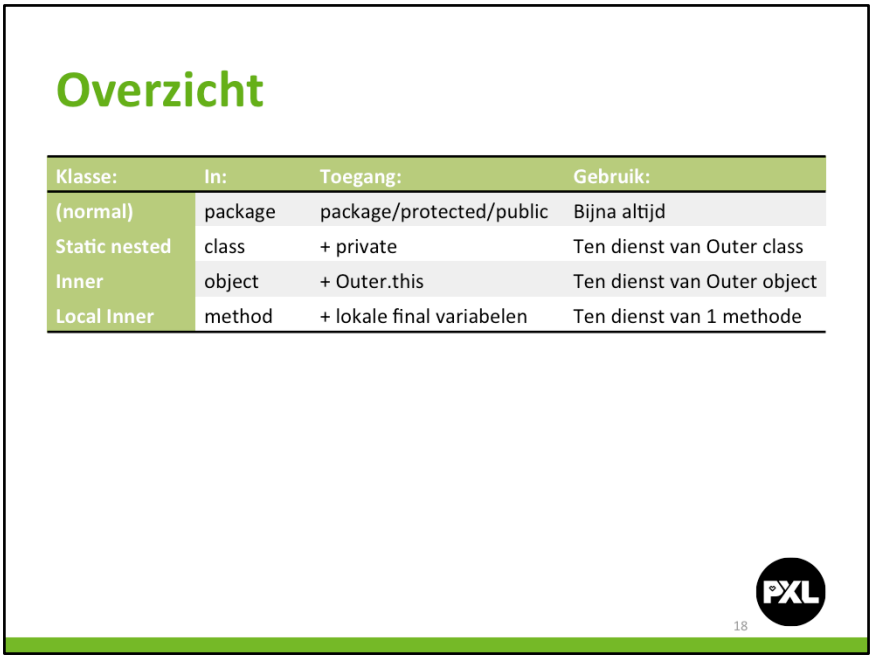
```
public class OuterClass {
	public Object getInner() {
		int val = 7;
		class LocalInnerClass {
			public String toString() {
				val++; // Fout
				return "Inner " + val;
			}
		}
		return new LocalInnerClass();
	}
}

public static void main(String[] args) {
	OuterClass outer = new OuterClass();
	outer.getInner().toString();
}
```

Opgelet: als de lokale variabelen wijzigen en niet onveranderlijk zijn, kan de local inner class niet aan hun waarden. Na het beëindigen van de methode, bestaan de lokale variabelen niet meer, maar het object eventueel nog wel.

Local inner class

```
public class OuterClass {
    private int val = 1;
    public void method() {
        class LocalInnerClass {
            …
            System.out.println(OuterClass.this.val); // OK
            …
        }
        LocalInnerClass local = new LocalInnerClass();
    }
}
```

Een local inner class is een inner klasse en kan dus wel aan de instantievariabelen van haar bijhorende object. Daarnaast, een local inner class kan net als lokale variabelen niet private, public of protected zijn.



Korte samenvatting. In geeft de context aan, in welke ‘parent container’ het conceptueel thuishoort. Toegang is waar de klasse kan aankomen. Het verschil tussen static nested en inner is niet de private instantie velden, het is dat de inner klasse altijd een Outer instantie heeft; een static nested kan aan de private velden als het Outer instantie krijgt om mee te werken. Gebruik geeft een zeer beknopte uitleg wanneer de klasse te gebruiken. Static nested kan nuttig zijn als er een zeer sterk verband is tussen de Outer en de nested. Bij het schrappen van de Outer class is de nested class waardeloos geworden. Bij Inner gaat het verder: het schrappen van 1 Outer object, maakt het inner object waardeloos. Bij local inner wordt na het schrappen van die ene methode de local class waardeloos. (vage leidraad, korrel zout, bron: oracle.com)

Anonymous class

```
public class OuterClass {
    public void method() {
        class SubClass extends SuperClass {
            // Vervangen methodes
        }
        SuperClass object1 = new SubClass();

        class InterfaceClass implements Interface {
            // Implementatie van methodes
        }
        Interface object2 = new InterfaceClass();
    }
}
```

object1 is een object van een subklasse van de klasse SuperClass. In deze subklasse worden alleen methodes van de superklasse vervangen. object2 is een object van een klasse die de interface Interface implementeert. We kunnen dit ook m.b.v. anonieme klassen realiseren.

Anonymous class

```
public class OuterClass {
    public void method() {
        SuperClass object1 = new SuperClass() {
            // Vervangen methodes
        };


        Interface object2 = new Interface() {
            // Implementatie van methodes
        };

    }
}
```

Opgelet, vermits de klasse anoniem is, kan je geen constructor definiëren en kunnen nieuwe methodes niet worden opgeroepen van buitenaf (want niet gedefinieerd in SuperClass of Interface).

Toepassing: iterator

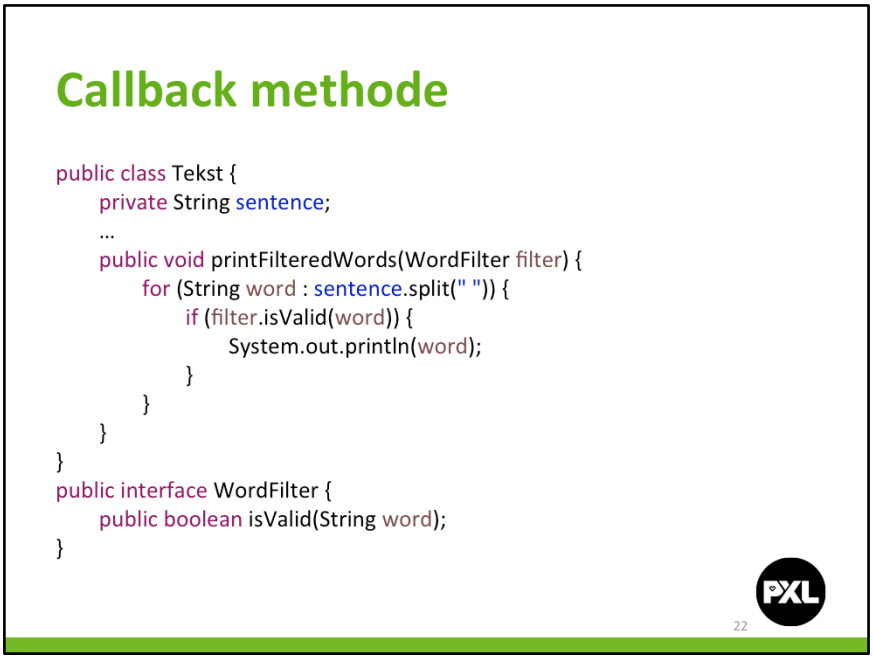
```
Iterator it;
it = collection.iterator();
while (it.hasnext()) {
    Object o = it.next();
    // use o …
}
```

```
for (Object o : collection) {
    // use o …
}
```

De Iterator interface kan geïmplementeerd worden in een nested class in de collection klasse.

21

Iterable collection; Iterable is de interface die de methode iterator aanbiedt die een object van de interface Iterator aanbiedt. (toepassing uit cursus p. 30)



Deze klasse laat toe om woorden af te drukken die aan een bepaald criterium voldoen. De aanroeper kiest het criterium door een object mee te geven dat de interface WordFilter implementeert. De methode in het object meegegeven door de aanroeper wordt telkens opgeroepen om elk woord te testen. Zo een methode noemen we een callback-methode.
Niet getoond: de constructor waarmee sentence wordt ingesteld.

Callback methode

```
public class TekstApp {
    ...
    Text text = new Text("Hello this is an example sentence");
    text.printFilteredWords(new WordFilter() {
        @Override
        public boolean isValid(String word) {
            return word.contains("e");
        }
    });
    ...
}
```

23

Voorbeeld van het gebruik van WordFilter en de implementatie van een callback-methode. Merk op, bij het debuggen springt de code per woord in deze methode om te testen of word een e bevat.

Callback methode

```
public class TekstApp {
    ...
    Text text = new Text("Hello this is an example sentence");
    text.printFilteredWords(new WordFilter() {
        @Override
        public boolean isValid(String word) {
            return word.length() > 4;
        }
    });
    ...
}
```

24

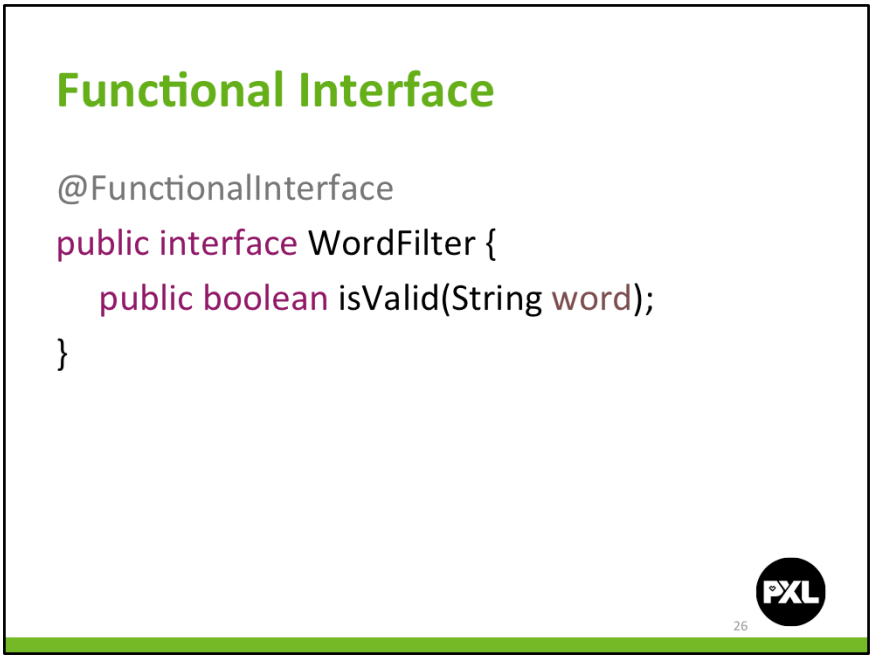
Andere filter. Hier testen we of de lengte groter is dan 4. Merk op, dit hadden we ook direct in printFilteredWords kunnen zetten, maar nu is de conditie los van de klasse Tekst.

Callback methode

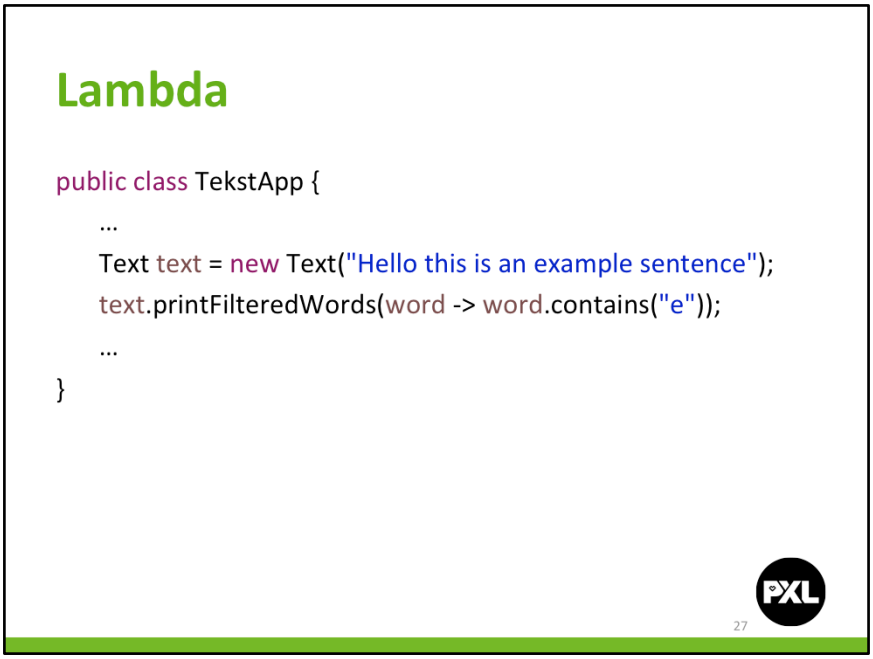
```
public class TekstApp {
    ...
    Text text = new Text("Hello this is an example sentence");
    text.printFilteredWords(new WordFilter() {
        @Override
        public boolean isValid(String word) {
            return word.startsWith("a");
        }
    });
    ...
}
```

25

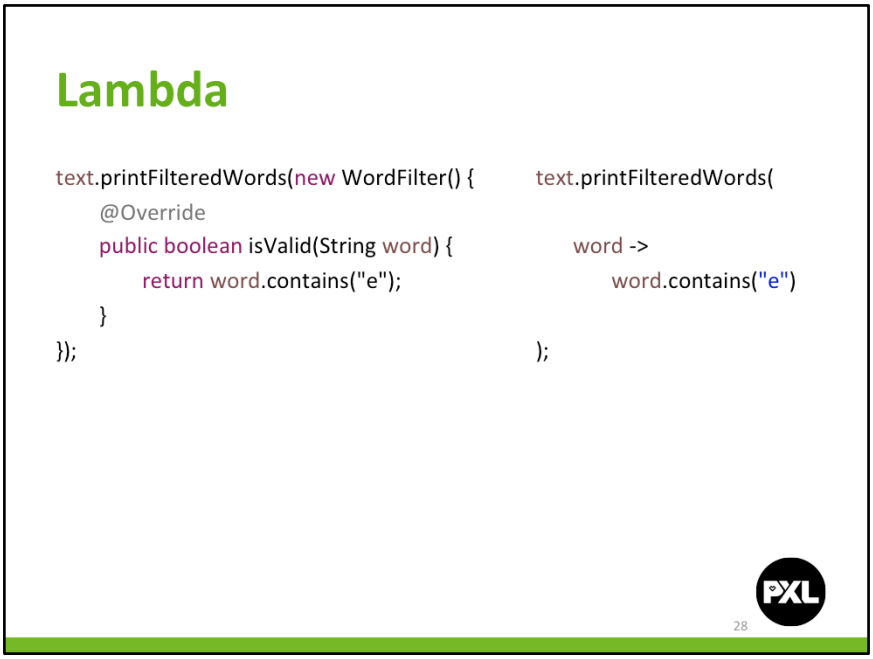
Andere filter. Dit toont de flexibiliteit. We hebben succesvol het filter-criterium losgetrokken van de iteratie en het outputten. In conceptuele termen hebben we de conditie variabel gemaakt; de WordFilter is een variabele die niet het resultaat van een conditie is (een boolean), maar de bepaling ervan, de conditie zelf. Enige nadeel: nogal omslachtige code. De essentie is de code in isValid.



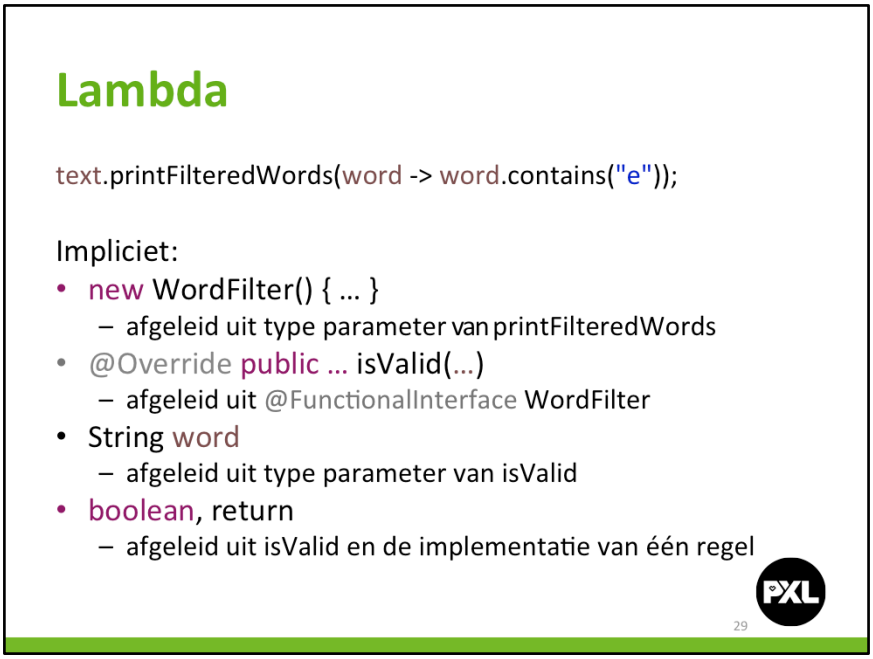
Zo'n interface met slechts één methode, noemen we een functionele interface. De objecten en klassen die deze interface implementeren hebben namelijk enkel een eenvoudige 'functie', ze gebruiken geen extra eigenschappen van het object. In feite is dus geen object met eigen data nodig, enkel een stukje code dat uitgevoerd moet worden, een functie. We kunnen expliciet opleggen door de annotatie FunctionalInterface toe te voegen.



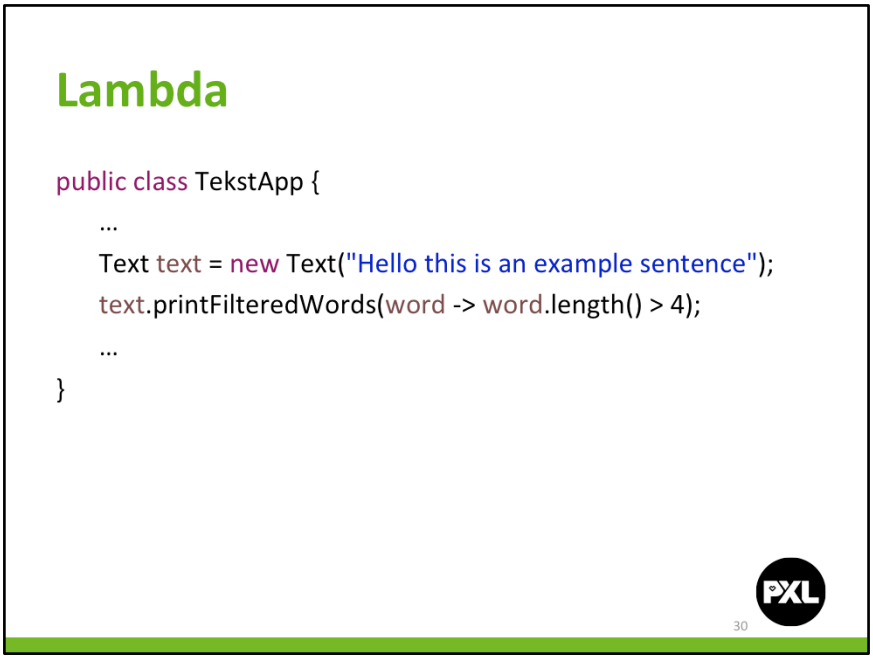
Weg met de omslachtige code. De essentie, de implementatie van isValid is behouden. Het enige wat nog overblijft van de definitie is de naam van de parameter. (volgende slide vergelijking)



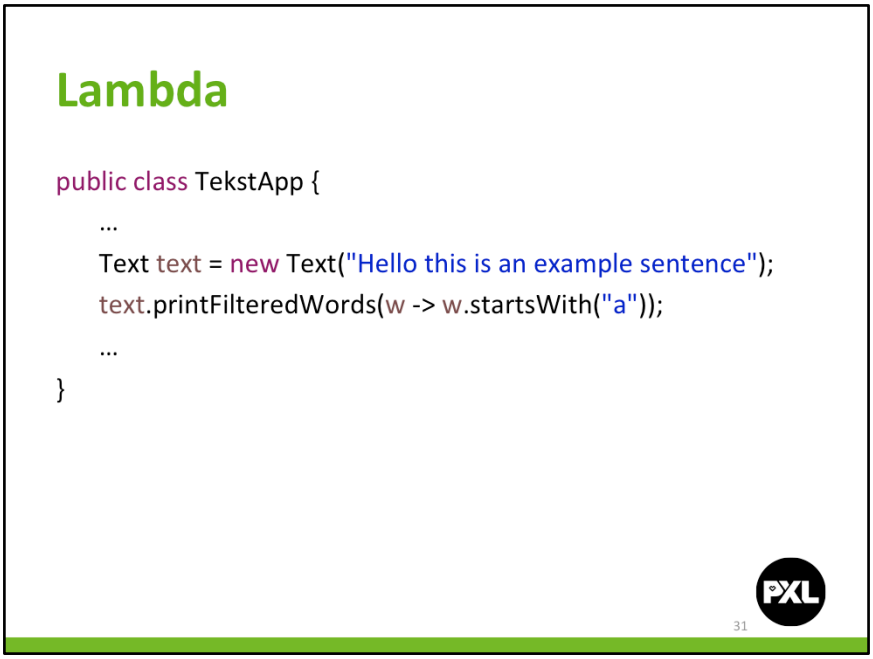
Vergelijking oud/nieuw. Ze doen hetzelfde. Enkel behouden is de naam van de parameter (zodat die niet 'word' hoeft te zijn) en de code. Al de rest is niet meer nodig. Het rechtse is wel degelijk een 'methode' waar de debugger in komt om de conditie te testen. Meer uitleg over wat weg is op de volgende slide.



Herinnering: FunctionalInterface had slechts één methode, dus het kan enkel die zijn die wordt geïmplementeerd. Doordat word.contains("e") de enige implementatie is én isValid een boolean vereist, is return overbodig: enkel die ene regel kan een boolean teruggeven.



Weg met de omslachtige code. De essentie, de implementatie van isValid is behouden. Het enige wat nog overblijft van de definitie is de naam van de parameter. (volgende slide vergelijking)



De naam van de parameter mag wijzigen.

Lambda

Syntax:

- WordFilter filter = word -> word.charAt(2) == 'i';
- WordFilter filter = word -> {
  return word.charAt(2) == 'i';
  };
- WordFilter filter = (word) -> word.charAt(2) == 'i';
- WordFilter filter = (String word) -> word.charAt(2) == 'i';

32

Langere syntax, exact hetzelfde (hier). Langere syntax laat andere mogelijkheden (volgende slides). In block {…} is return wel nodig. Daarnaast, hier leidt Java af dat het WordFilter wordt uit de assignment, i.p.v. parameter tot nog toe.

Lambda

Voorbeeld:

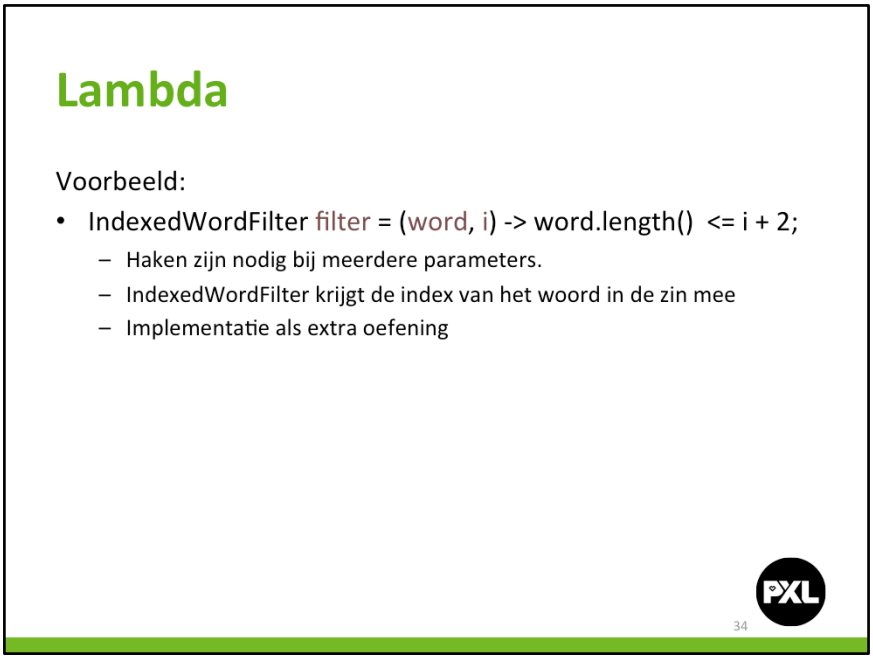
- WordFilter filter = word -> {

```
try {
    int value = Integer.parseInt(word);
    return value > 10;
} catch (NumberFormatException nfe) {
    return false;
}
};
```

  - Accolades en return zijn nodig bij meerdere statements.

33

Voorbeelden met langere syntax.



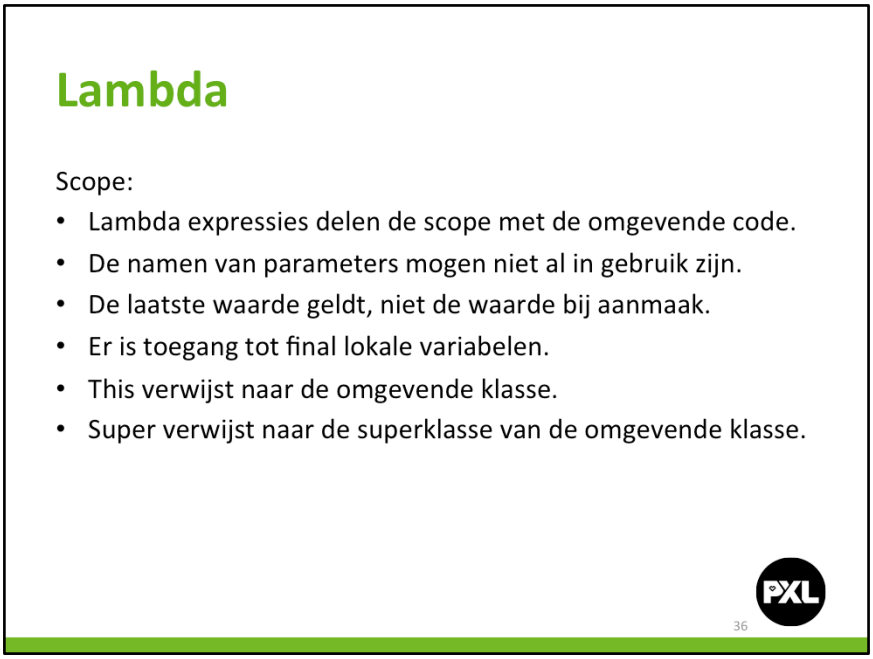
Voorbeelden met langere syntax; Voorbeeld tekst hier: “In het zware boek van gelukkige tante Francine” wordt “In het boek van tante Francine”. Datatype (String word, int i) is nodig als de compiler het niet uit de context kan afleiden (cursus). Vermits er echter altijd een Functionele interface moet zijn, is er zo goed als altijd een context. Uitzondering: als generic functionele interfaces worden gebruikt in een situatie waar een generische parameter niet kan bepaald worden. Uitzonderlijk (details: zie http://stackoverflow.com/questions/31370113/when-does-java-require-explicit-type-parameters).

Lambda

```
public class TekstApp {
    private String start;
    ...
    Text text = new Text("bob blijft achter de blauwe auto");
    start = "a";
    WordFilter filter = w -> w.startsWith(start);
    start = "b";
    text.printFilteredWords(filter);
    ...
}
```

35

Lambdas delen een scope met de omgevende code, ze hebben niet hun eigen scope (uitleg op volgende slide).



Zoals bij local inner classes, lokale variabelen die niet wijzigen zijn final.

Method reference

```
@FunctionalInterface
public interface WordProcessor {
    public String process(String word);
}

// in Text.java
public void printProcessedWords(WordProcessor processor) {
    for (String word : sentence.split(" ")) {
        System.out.println(processor.process(word));
    }
}
```

37

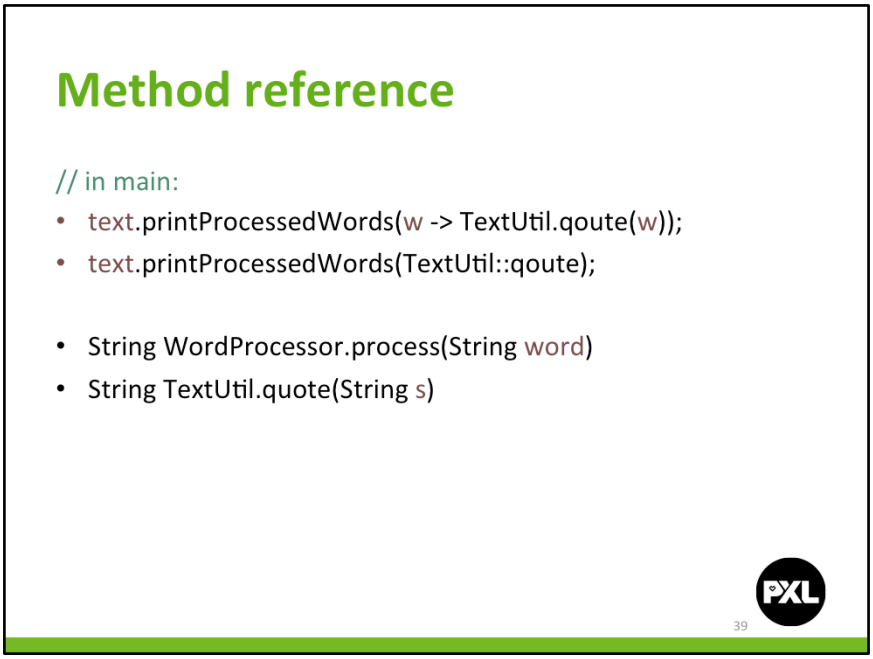
Deze interface geeft String terug i.p.v. boolean: het verwerkt de woorden/input en geeft tekst/output terug.

Method reference

```
public interface TextUtil {
    public static String quote(String str) {
        return String.format("<<%s>>", str);
    }
}

// in main:
text.printProcessedWords(w -> TextUtil.qoute(w));
```

38

We willen een bepaalde quoting techniek meerdere keren gebruiken. We besluiten de code onder te brengen in een utility-interface (cursus)



Deze twee doen juist hetzelfde. De methode quote van TextUtil heeft namelijk hetzelfde aantal en type parameters en hetzelfde return type als WordProcessor.process en kan daarom rechtstreeks ingezet worden in een lambda expression.

Method reference

Syntax:

qualifier::identifier

Mogelijkheden:

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

40

De verschillende vormen van method references. Type is hier expliciet, ook al is dat bij een gewone lambda niet nodig.

Method reference

data -> TextUtil.quote(data)	TextUtil::quote
woord -> vertaler.vertaal(woord)	vertaler::vertaal
tekst -> System.out.println(tekst)	System.out::println
(String w) -> w.toUpperCase()	String::toUpperCase
(naam, leeftijd) -> new Persoon(naam, leeftijd)	Persoon::new

41

Concrete voorbeelden. Merk op dat de parameter telkens verdwijnt bij het omzetten in een method reference

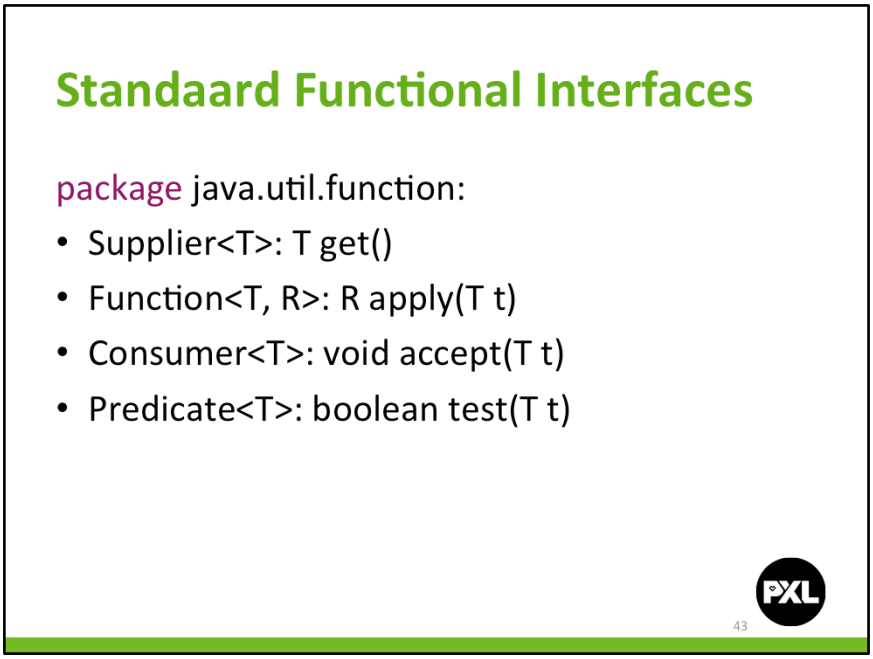
Method reference

Voorbeelden

data -> show(data)	this::show
getal -> bereken(getal, 5)	/
persoon -> super.toon(persoon)	super::toon
tekst -> tekst.indexOf(",")	/
tekst -> tekst.concat(tekst)	/

42

De verschillende vormen van method references. Details: p 36 tot 40.



Supplier levert items, Function berekent er een ander item mee en Consumer doet er iets mee. Dus krijg je vaak: Supplier > Function > Function > Consumer. Predicate is een conditie, typisch gebruikt bij filter.

Standaard Functional Interfaces

```
public class Text {
    private int i = 0;
    public Supplier<String> split(String teken) {
        i = 0;
        String[] data = text.split(teken);
        return () -> i < data.length ? data[i++] : null;
    }
    …
}
```

44

Voorbeeld van hoe een Supplier<String> te verkrijgen

Standaard Functional Interfaces

```
public class TestApp {
    public static void main(String[] args) {
        Text text = new Text("test 1 2 3 test");
        Supplier<String> woorden = text.split(" ");

        String woord = woorden.get();
        while (woord != null) {
            System.out.println(woord);
            woord = woorden.get();
        }
    }
}
```

45

Voorbeeld van hoe een Supplier<String> kan worden gebruikt. Dit soort code zou heel herkenbaar moeten zijn. Voordeel: het verkrijgen van de woorden staat los van het gebruik. En je kan vanaf het eerste woord dat je krijgt het beginnen verwerken.

Standaard Functional Interfaces

```
public class User {
    public Supplier<String> lees() {
        Scanner lezer = new Scanner(System.in);
        return () -> lezer.next();
    }
    ...
}
```

46

Je kan ook woorden vragen aan de gebruiker, over het netwerk, … Supplier<T> laat toe om het verkrijgen van T los te maken van het verwerken van T. De andere Standaard Functional Interfaces komen terug in de oefeningen en in het volgende hoofdstuk bij Streams.

W4: Testing of Multi-threaded applications

.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

Java concurrency

Topics of today:

- Blocking queue
- Test its blocking behavior
- Test its performance under stress test conditions
- Available frameworks for unit testing of multi-threaded classes.

PXL IT

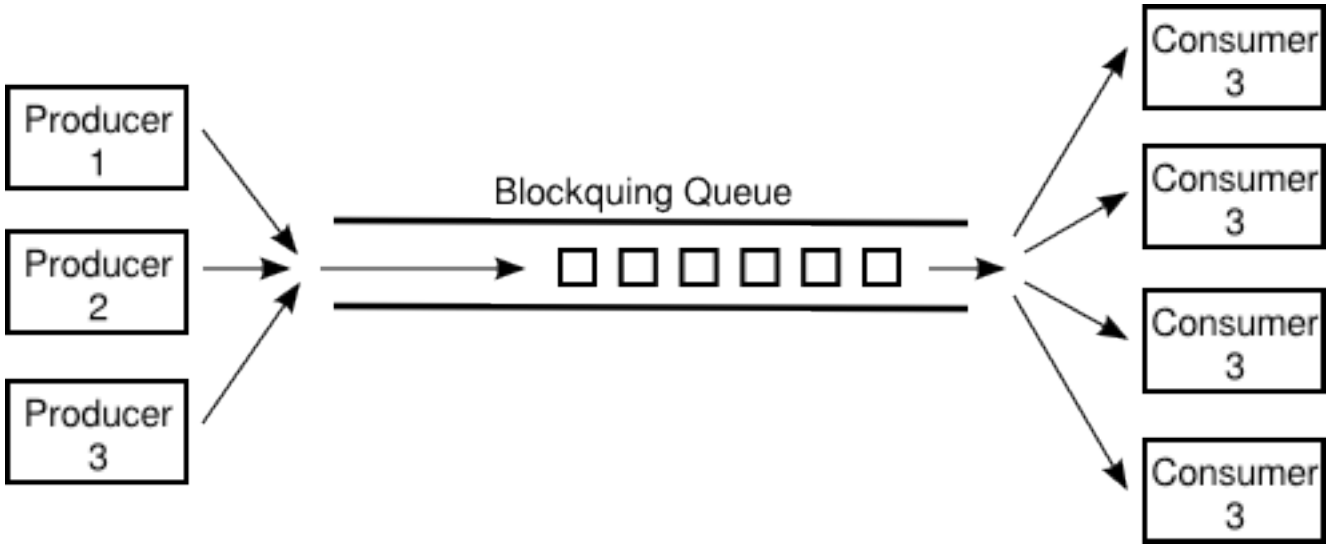
A SimpleBlockingQueue

- Use package *java.util.concurrent*
- As data structure use *java.util.LinkedList* (not synchronized)
- Add synchronization and blocking behavior (wait/notify)
- Requirement: make the *queue generic!*

PXL IT

Testing blocking operations

- wasInterrupted
- reachedAfterGet
- throwableThrown



PXL IT

Testing blocking operations

```
@Test
public void testPutOnEmptyQueueBlocks() throws InterruptedException {
        final SimpleBlockingQueue queue = new SimpleBlockingQueue();
        BlockingThread blockingThread = new BlockingThread(queue);
        blockingThread.start();
        Thread.sleep(5000);
        assertThat(blockingThread.isReachedAfterGet(), is(false));
        assertThat(blockingThread.isWasInterrupted(), is(false));
        assertThat(blockingThread.isThrowableThrown(), is(false));
        queue.put(new Object());
        Thread.sleep(1000);
        assertThat(blockingThread.isReachedAfterGet(), is(true));
        assertThat(blockingThread.isWasInterrupted(), is(false));
        assertThat(blockingThread.isThrowableThrown(), is(false));
        blockingThread.join();
```

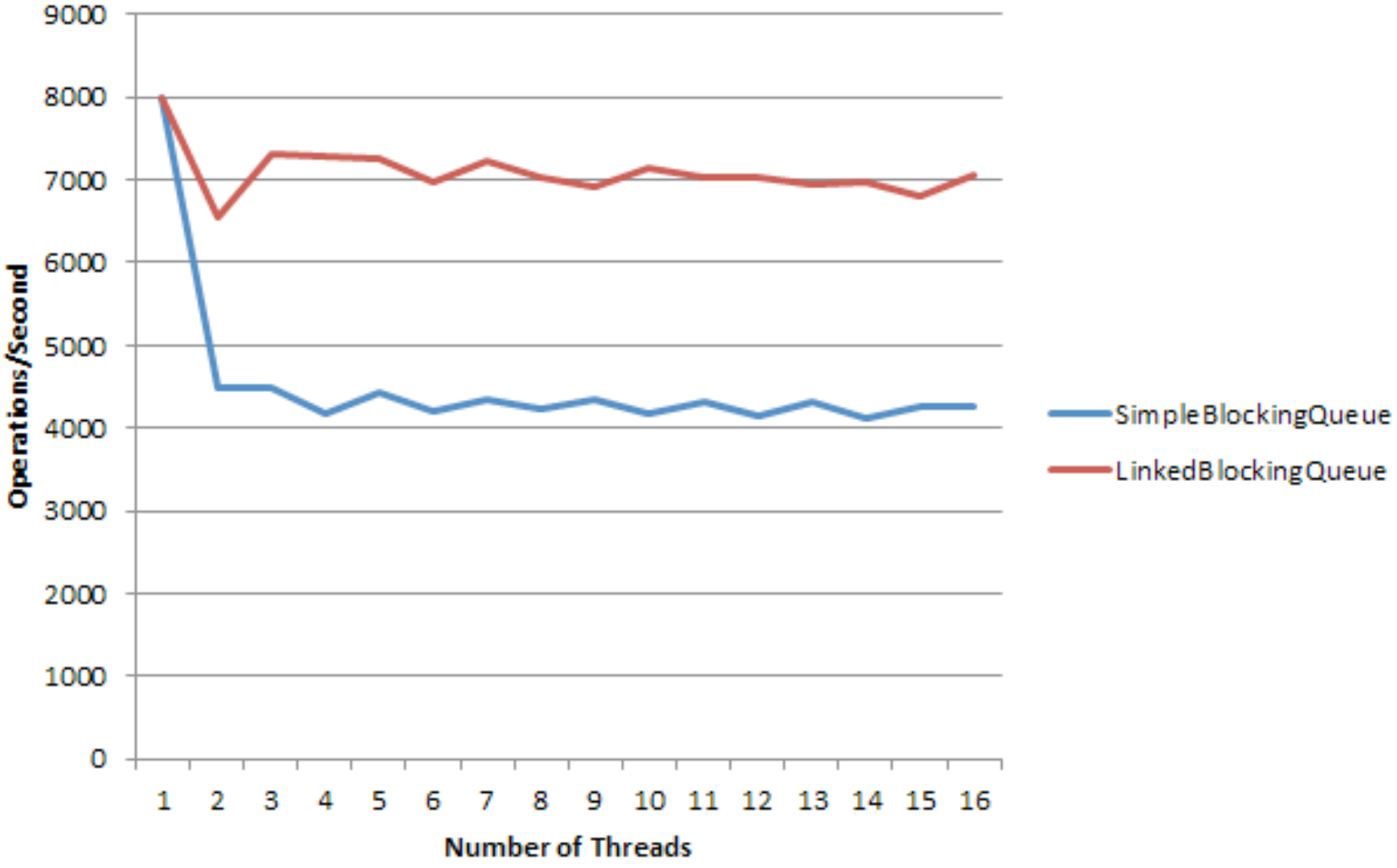
Testing for correctness

Write a new JUnit test to test that Producer out == Consumer in

- Integer values as queue elements
- Integer values from thread local random generator
- Producer thread computes the sum over the elements inserted.
- Sum over all producer threads is compared to the sum of all consumer threads

PXL IT

Testing performance



Testing frameworks

- JMock
- Grobo Utils

JMock Blitzer

```
@Test
public void stressTest() throws InterruptedException {
    final SimpleBlockingQueue<Integer> queue = new SimpleBlockingQueue<Integer>();
    blitzer.blitz(new Runnable() {
        public void run() {
            try {
                queue.put(42);
                queue.get();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    assertThat(queue.getSize(), is(0));
}
```

Grobo Utils

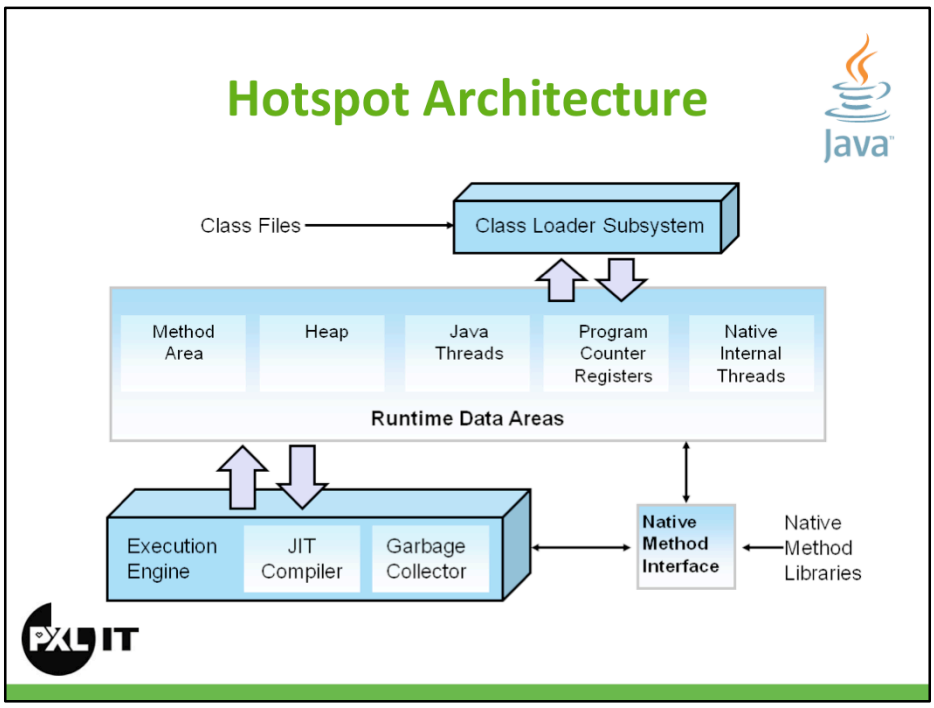
```
public class SimpleBlockingQueueGroboUtilTest {

	private static class MyTestRunnable extends TestRunnable {
		private SimpleBlockingQueue<Integer> queue;

		public MyTestRunnable(SimpleBlockingQueue<Integer> queue) {
			this.queue = queue;
		}
		@Override
		public void runTest() throws Throwable {
			for (int i = 0; i < 1000000; i++) {
				this.queue.put(42);
				this.queue.get();
			}
		}
	}
	@Test
	public void stressTest() throws Throwable {
		SimpleBlockingQueue<Integer> queue = new SimpleBlockingQueue<Integer>();
		TestRunnable[] testRunnables = new TestRunnable[6];
		for (int i = 0; i < testRunnables.length; i++) {
			testRunnables[i] = new MyTestRunnable(queue);
		}
		MultiThreadedTestRunner mttr = new MultiThreadedTestRunner(testRunnables);
		mttr.runTestRunnables(2 * 60 * 1000);
		assertThat(queue.getSize(), is(0));
	}
}
```

W5: Garbage Collection GC

.NET / Java

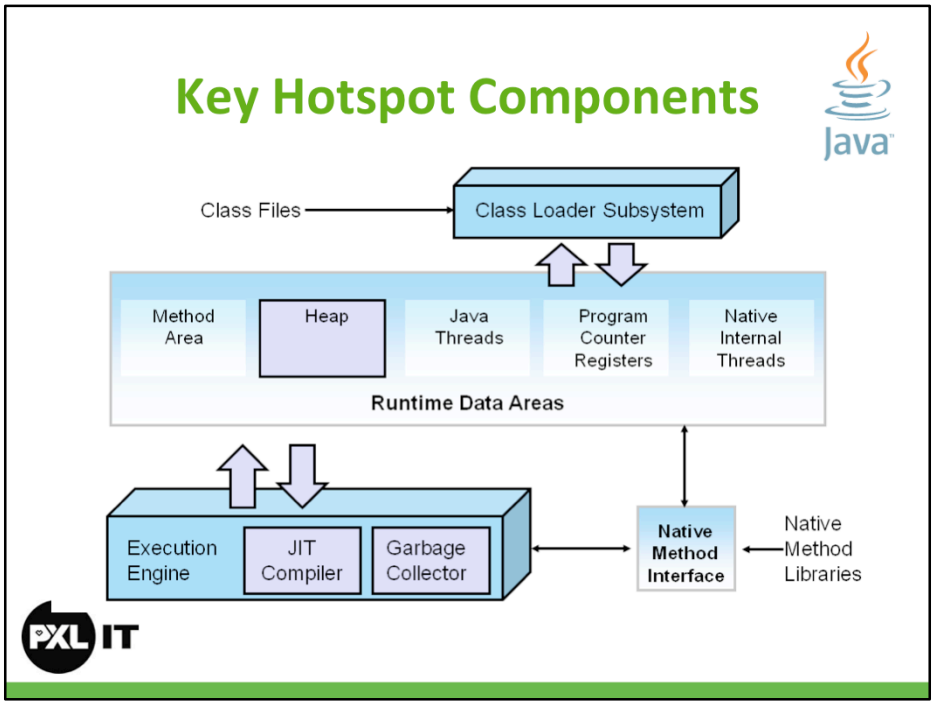
DE HOGESCHOOL
MET HET NETWERK

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

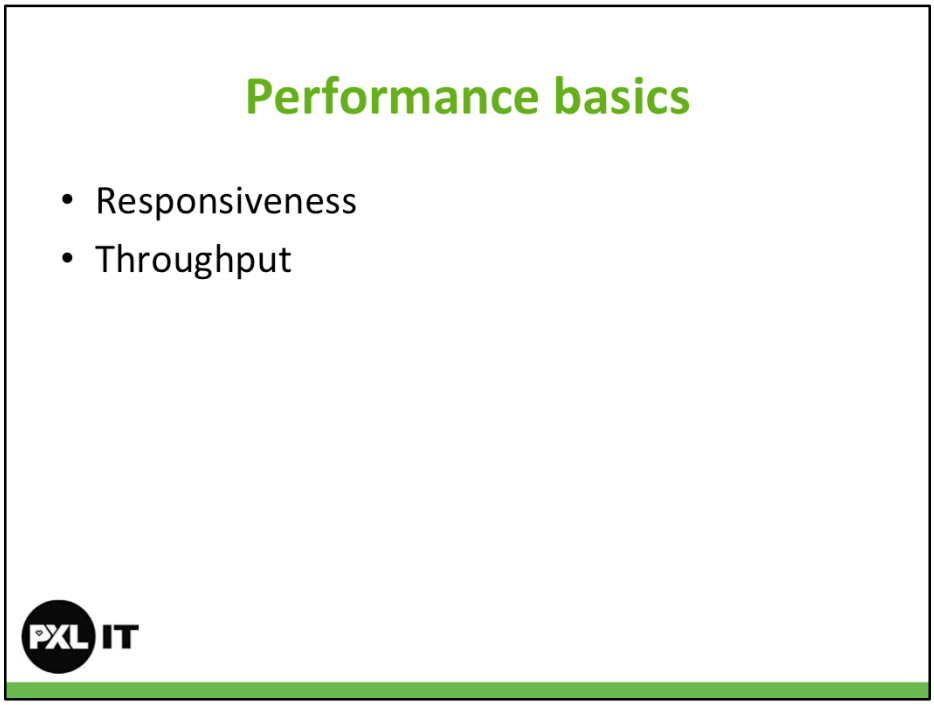


The HotSpot JVM possesses an architecture that supports a strong foundation of features and capabilities and supports the ability to realize high performance and massive scalability. For example, the HotSpot JVM JIT compilers generate dynamic optimizations. In other words, they make optimization decisions while the Java application is running and generate high-performing native machine instructions targeted for the underlying system architecture. In addition, through the maturing evolution and continuous engineering of its runtime environment and multithreaded garbage collector, the HotSpot JVM yields high scalability on even the largest available computer systems.

The main components of the JVM include the classloader, the runtime data areas, and the execution engine.



There are three components of the JVM that are focused on when tuning performance. The *heap* is where your object data is stored. This area is then managed by the garbage collector selected at startup. Most tuning options relate to sizing the heap and choosing the most appropriate garbage collector for your situation. The JIT compiler also has a big impact on performance but rarely requires tuning with the newer versions of the JVM.



Responsiveness

Responsiveness refers to how quickly an application or system responds with a requested piece of data. Examples include:

How quickly a desktop UI responds to an event

How fast a website returns a page

How fast a database query is returned

For applications that focus on respsonsiveness, large pause times are not acceptable. The focus is on responding in short periods of time.

Throughput

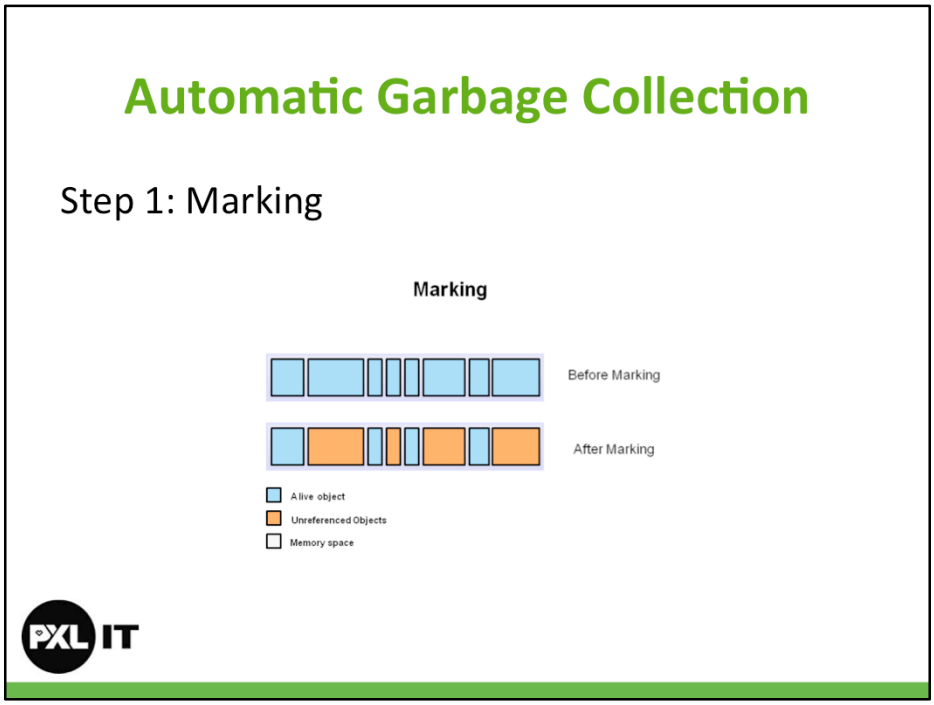
Throughput focuses on maximizing the amount of work by an application in a specific period of time. Examples of how throughput might be measured include:

The number of transactions completed in a given time.

The number of jobs that a batch program can complete in an hour.

The number of database queries that can be completed in an hour.

High pause times are acceptable for applications that focus on throughput. Since high throughput applications focus on benchmarks over longer periods of time, quick



What is Automatic Garbage Collection?

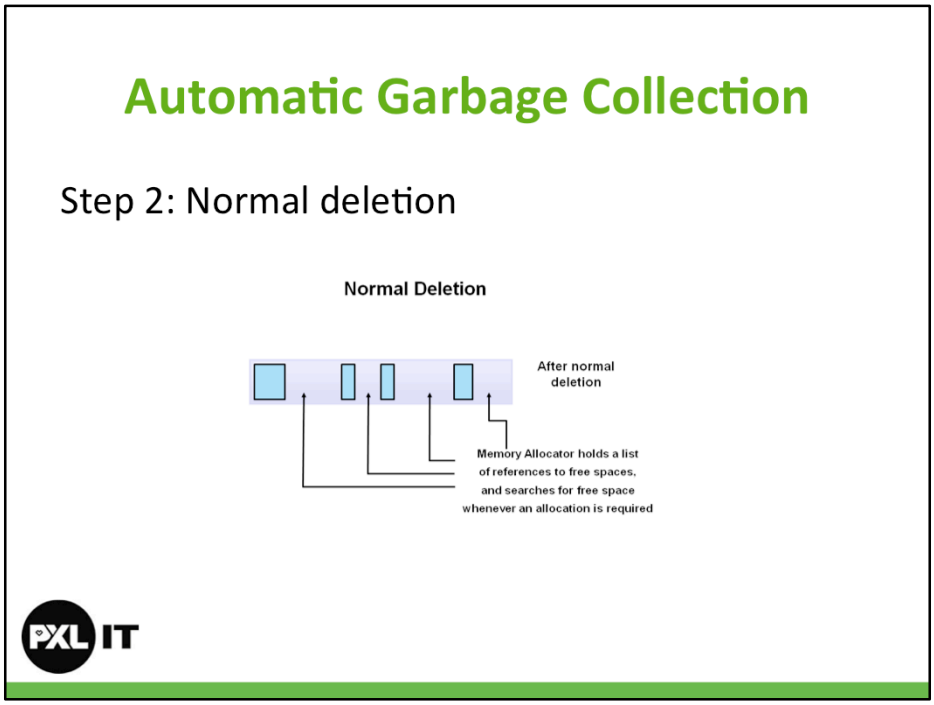
Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.

In a programming language like C, allocating and deallocating memory is a manual process. In Java, process of deallocating memory is handled automatically by the garbage collector. The basic process can be described as follows.

Step 1: Marking

The first step in the process is called marking. This is where the garbage collector identifies which pieces of memory are in use and which are not.

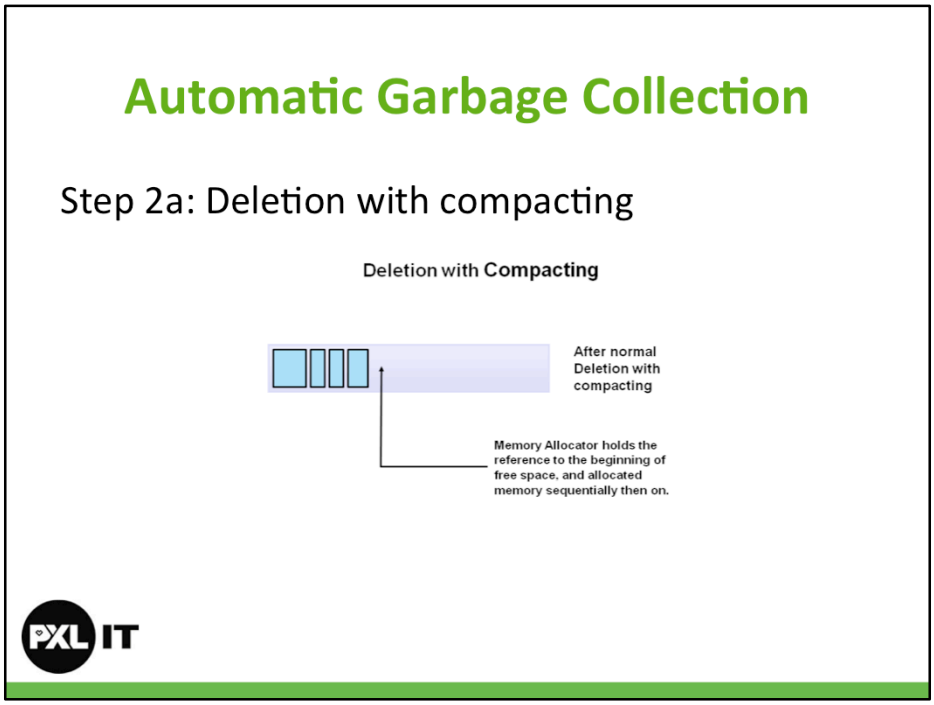
Referenced objects are shown in blue. Unreferenced objects are shown in gold. All objects are scanned in the marking phase to make this determination. This can be a very time consuming process if all objects in a system must be scanned.



Step 2: Normal Deletion

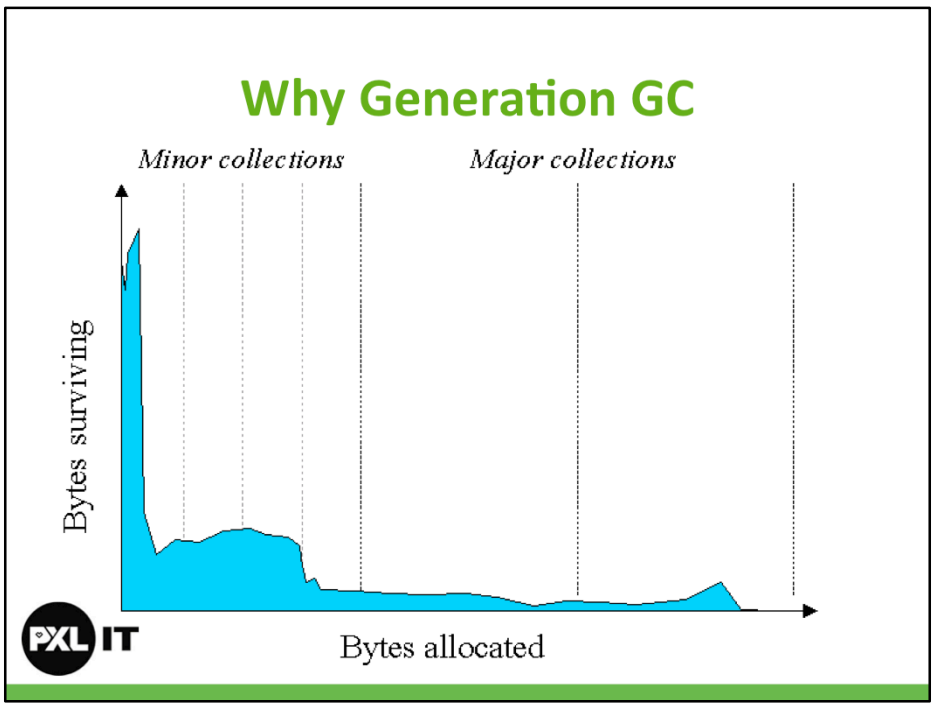
Normal deletion removes unreferenced objects leaving referenced objects and pointers to free space.

The memory allocator holds references to blocks of free space where new object can be allocated.



Step 2a: Deletion with Compacting

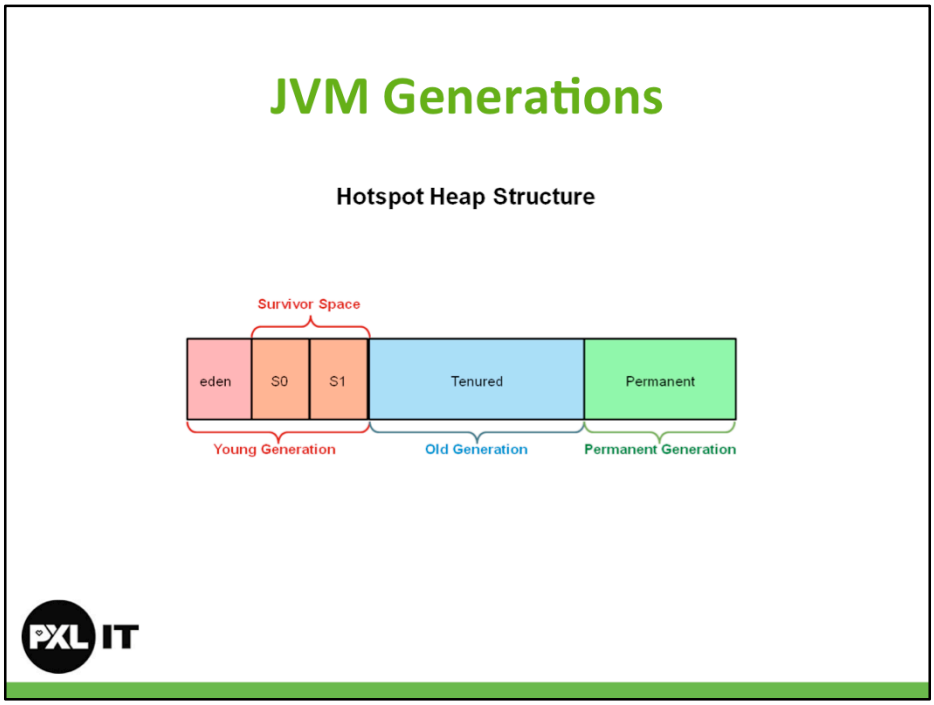
To further improve performance, in addition to deleting unreferenced objects, you can also compact the remaining referenced objects. By moving referenced object together, this makes new memory allocation much easier and faster.



As stated earlier, having to mark and compact all the objects in a JVM is inefficient. As more and more objects are allocated, the list of objects grows and grows leading to longer and longer garbage collection time. However, empirical analysis of applications has shown that most objects are short lived.

Here is an example of such data. The Y axis shows the number of bytes allocated and the X access shows the number of bytes allocated over time.

As you can see, fewer and fewer objects remain allocated over time. In fact most objects have a very short life as shown by the higher values on the left side of the graph.



The **Young Generation** is where all new objects are allocated and aged. When the young generation fills up, this causes a ***minor garbage collection***. Minor collections can be optimized assuming a high object mortality rate. A young generation full of dead objects is collected very quickly. Some surviving objects are aged and eventually move to the old generation.

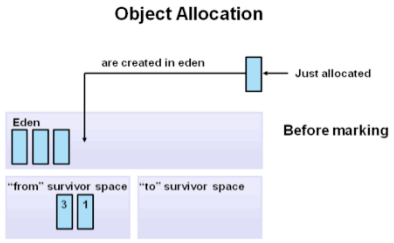
Stop the World Event - All minor garbage collections are "Stop the World" events. This means that all application threads are stopped until the operation completes. Minor garbage collections are *always* Stop the World events.

The **Old Generation** is used to store long surviving objects. Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation. Eventually the old generation needs to be collected. This event is called a ***major garbage collection***.

Major garbage collection are also Stop the World events. Often a major collection is much slower because it involves all live objects. So for Responsive applications, major garbage collections should be minimized. Also note, that the length of the Stop the World event for a major garbage collection is affected by the kind of garbage collector that is used for the old generation space.

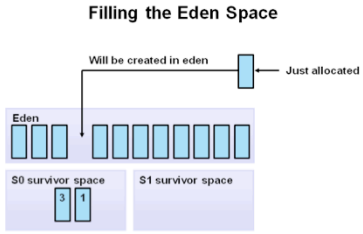
GC process

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



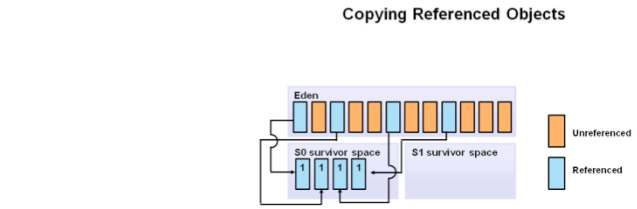
GC process

- Second, when the eden space fills up, a minor garbage collection is triggered.



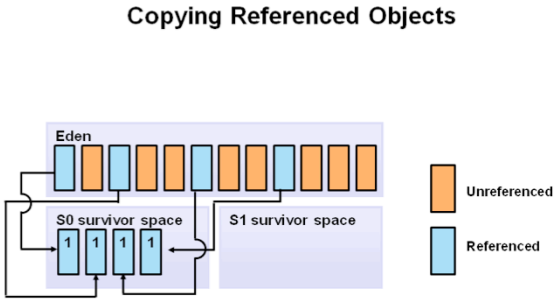
GC process

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



GC process

- Fourth

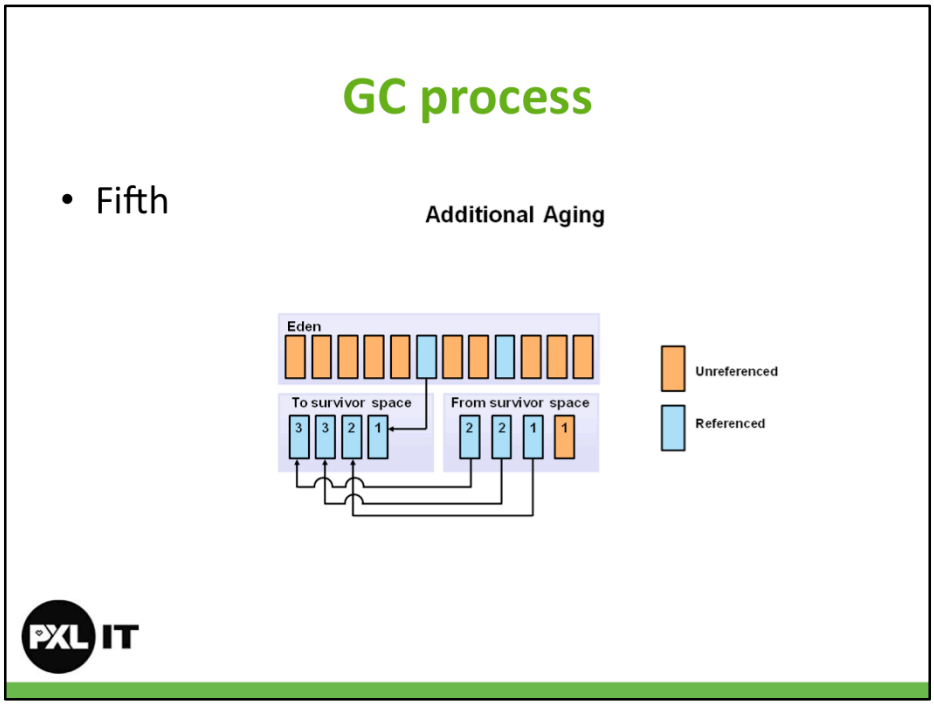


PXL IT

At the next minor GC, the same thing happens for the eden space. Unreferenced objects are deleted and referenced objects are moved to a survivor space. However, in this case, they are moved to the second survivor space (S1). In addition, objects from the last minor GC on the first survivor space (S0) have their age incremented and get moved to S1. Once all surviving objects have been moved to S1, both S0 and eden are cleared. Notice we now have differently aged object in the survivor space.

GC process

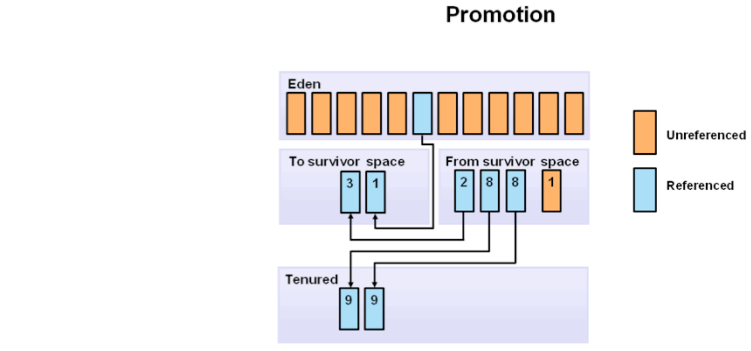
- Fifth



At the next minor GC, the same process repeats. However this time the survivor spaces switch. Referenced objects are moved to S0. Surviving objects are aged. Eden and S1 are cleared.

GC process

- Sixth

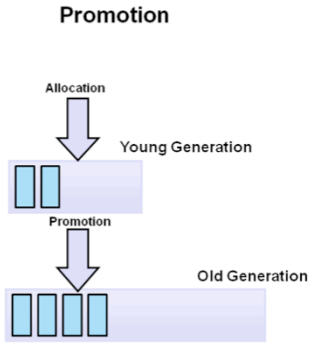


PXL IT

This slide demonstrates promotion. After a minor GC, when aged objects reach a certain age threshold (8 in this example) they are promoted from young generation to old generation.

GC process

- Seventh

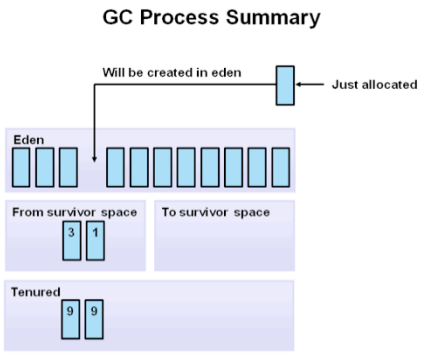


PXL IT

As minor GCs continue to occure objects will continue to be promoted to the old generation space.

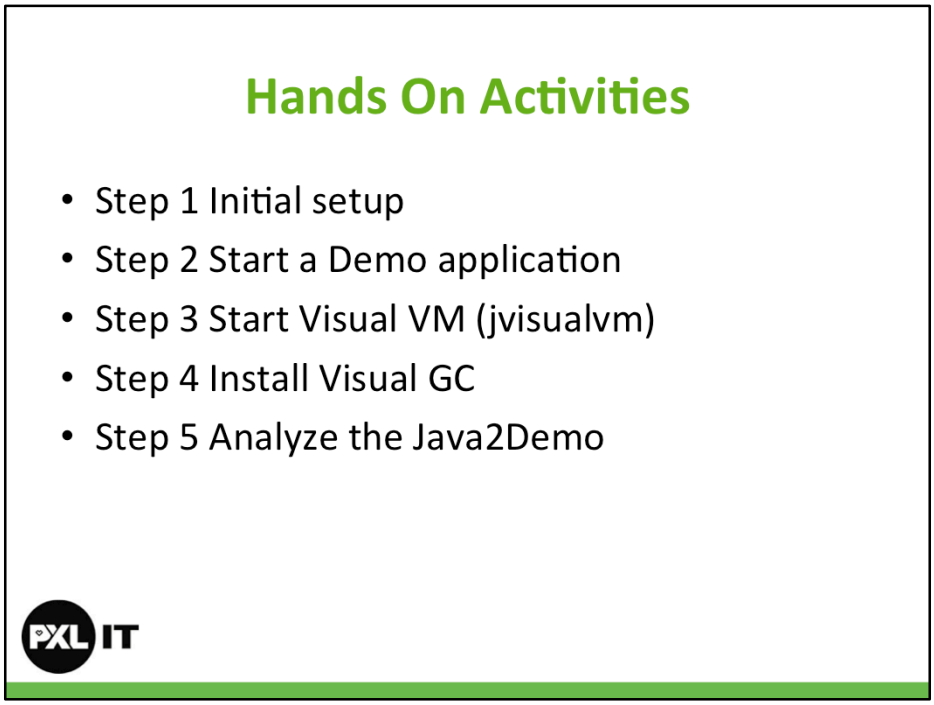
GC process

- Eight



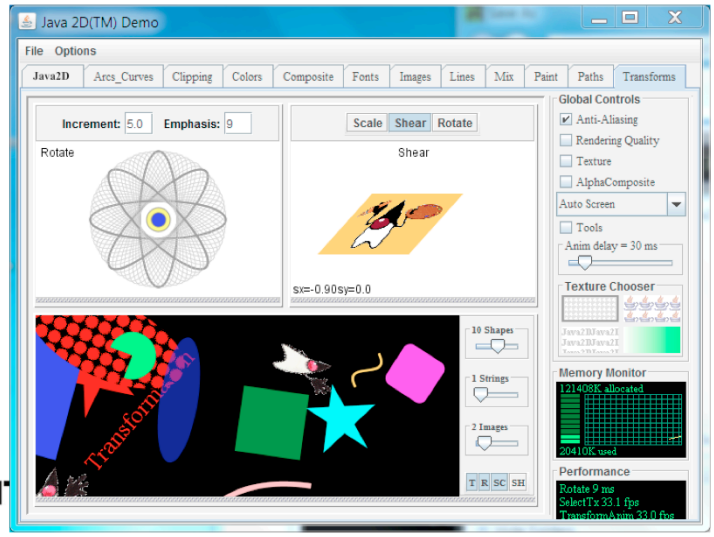
PXL IT

So that pretty much covers the entire process with the young generation. Eventually, a major GC will be performed on the old generation which cleans up and compacts that space.

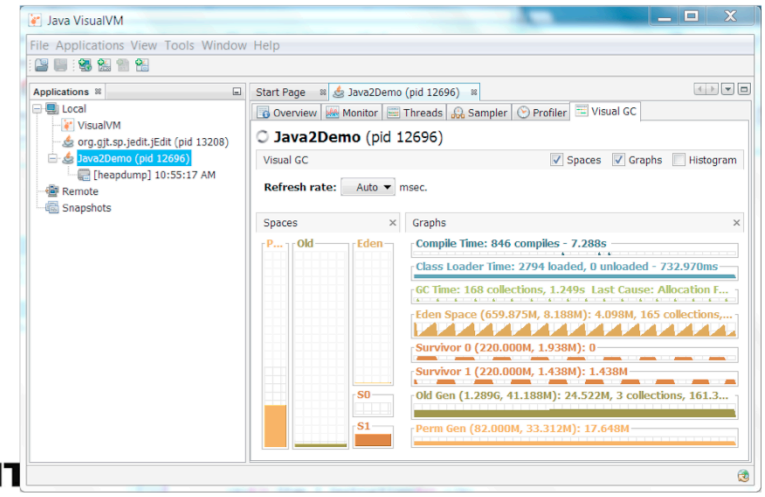


You have seen the garbage collection process using a series of pictures. Now it is time to experience and explore the process live. In this activity, you will run a Java application and analyze the garbage collection process using Visual VM. The Visual VM program is included with the JDK and allows developers to monitor various aspects of a running JVM.

Hands On Activities



Hands On Activities



W6: Application Logging

.NET / Java

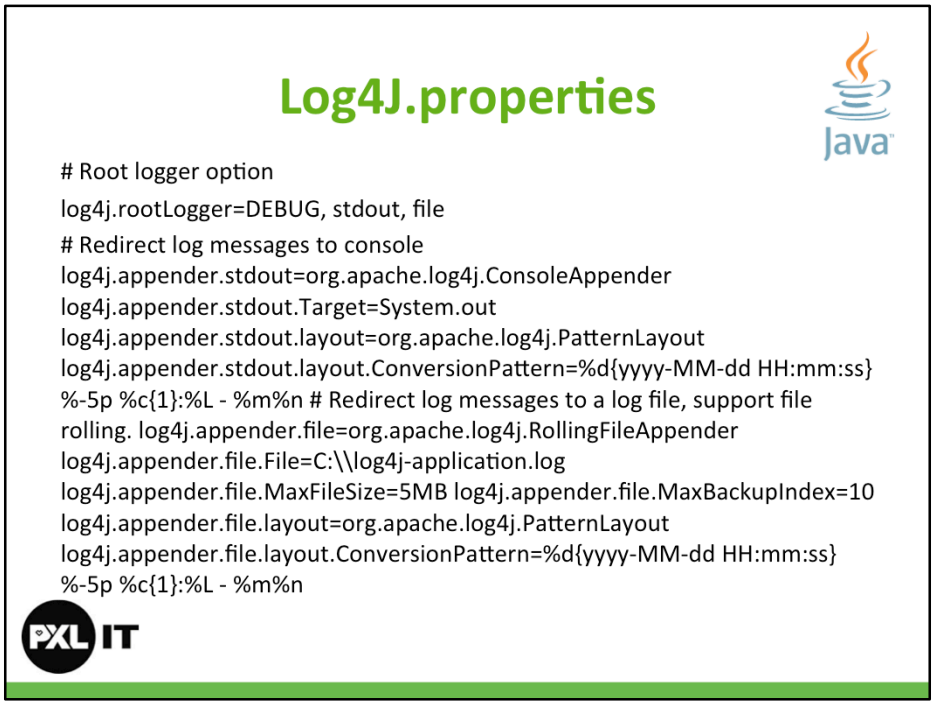
DE HOGESCHOOL
MET HET NETWERK

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

Log4J

- Log4j.properties on classpath
- Get Log4j from MavenRepository

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```



Let break it down :

%d{yyyy-MM-dd HH:mm:ss} = Date and time format, refer to SimpleDateFormat JavaDoc.

%-5p = The logging priority, like DEBUG or ERROR. The -5 is optional, for the pretty print format.

%c{1} = The logging name we set via getLogger(), refer to log4j PatternLayout guide.

%L = The line number from where the logging request.

%m%n = The message to log and line break.

Note

For standalone Java app, make sure the log4j.properties file is under the project/classes directory

For Java web applications, make sure the log4j.properties file is under the WEB-INF/ classes directory

Log message examples

...
2014-07-02 20:52:39 DEBUG className:200 - This is debug message
2014-07-02 20:52:39 DEBUG className:201 - This is debug message2

How to log a Message?

- Declare logger:

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

- Write code:

```
//logs a debug message
if(logger.isDebugEnabled()) {
        logger.debug("This is debug");
}
//logs an error message with parameter
logger.error("This is error : " + parameter);

//logs an exception thrown from somewhere
logger.error("This is error", exception);
```

Logger Priority

log4j.rootLogger=DEBUG, stdout

#...

Output:

2014-07-02 20:52:39 DEBUG HelloExample:19 - This is debug : PXL
2014-07-02 20:52:39 INFO HelloExample:23 - This is info : PXL
2014-07-02 20:52:39 WARN HelloExample:26 - This is warn : PXL
2014-07-02 20:52:39 ERROR HelloExample:27 - This is error : PXL
2014-07-02 20:52:39 FATAL HelloExample:28 - This is fatal : PXL


```
import org.apache.log4j.Logger;

public class HelloExample {

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

        public static void main(String[] args) {

                HelloExample obj = new HelloExample();
                obj.runMe(”PXL");

        }

        private void runMe(String parameter){

                if(logger.isDebugEnabled()){
```

Logger Priority

log4j.rootLogger=ERROR, stdout

#...

Output:

2014-07-02 20:52:39 ERROR HelloExample:27 - This is error : PXL
2014-07-02 20:52:39 FATAL HelloExample:28 - This is fatal : PXL

Logger Priority

If priority is defined in log4j.properties, only the same or above priority message will be logged.

```
package org.apache.log4j;
public class Priority {
  public final static int OFF_INT = Integer.MAX_VALUE;
  public final static int FATAL_INT = 50000;
  public final static int ERROR_INT = 40000;
  public final static int WARN_INT  = 30000;
  public final static int INFO_INT  = 20000;
  public final static int DEBUG_INT = 10000;
   //public final static int FINE_INT = DEBUG_INT;
  public final static int ALL_INT = Integer.MIN_VALUE;
```

How to log an Exception

```
try{
        obj.divide();
} catch(ArithmeticException ex) {
        logger.error("Sorry, something wrong!", ex);
}
```

Output:

2014-07-02 21:03:10 ERROR HelloExample2:16 - Sorry, something wrong! java.lang.ArithmeticException: / by zero at com.mkyong.HelloExample2.divide(HelloExample2.java:24) at com.mkyong.HelloExample2.main(HelloExample2.java:14)

Output to Console

- **log4j.properties:**

```
# Root logger option log4j.rootLogger=INFO, stdout
# Direct log messages to stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-
dd HH:mm:ss} %-5p %c{1}:%L - %m%n
```

Output to File

- **log4j.properties:**

```
# Root logger option
log4j.rootLogger=INFO, file
# Direct log messages to a log file
log4j.appender.file=org.apache.log4j.RollingFileAppender
#Redirect to Tomcat logs folder
#log4j.appender.file.File=${catalina.home}/logs/logging.log
log4j.appender.file.File=C:\\logging.log
log4j.appender.file.MaxFileSize=10MB
log4j.appender.file.MaxBackupIndex=10
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss} %-5p %c{1}:%L - %m%n
```

Colored formatting

- **Zoek op internet hoe je kleuren kan gebruiken voor je logging patroon in Log4J.**

More info

- Manual:
  http://logging.apache.org/log4j/1.2/manual.html
- Pattern layout:
  http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/PatternLayout.html