Про компанію

Intellias – це 1500+ спеціалістів:

- Львів – 800+ спеціалістів
- Київ – 500+ спеціалістів
- Одеса – 70+ спеціалістів
- Харків – 20+ спеціалістів
- відкриття нових офісів у Кракові та Івано-Франківську

#12 за кількістю спеціалістів

У рейтингу ТОП-50 IT-компаній України

Найкращий роботодавець за версією DOU

Багато років поспіль Intellias є одним з найкращих IT-роботодавців року за версією dou.ua

intellias

Java 9/10/11 Features

Part 2

What's new in java 9, except modules?

Presentation Overview

1. See the full list of implemented JEP
2. Java Core Improvements
2. Key Changes
3. Tools
4. Security
4. Deployment
5. Java Language
6. Javadoc
7. JVM
8. JVM Tuning
9. Nashorn
10. Client Technologies
11. Internationalization

Implemented JEPs

Key Changes	Java Platform Module System. JSR 376, JEP 261, JEP 200, JEP 220, JEP 260, JEP 223
Tools	JEP 222, JEP 228, JEP 231, JEP 238, JEP 240, JEP 241, JEP 245, JEP 247, JEP 282
Security	JEP 219, JEP 244, JEP 249, JEP 246, JEP 273, JEP 288, JEP 229, JEP 287
Deployment	JEP 275, JEP 289
Java Language	JEP 213
Javadoc	JEP 221, JEP 224, JEP 225, JEP 261
JVM	JEP 165, JEP 197, JEP 276
JVM Tuning	JEP 158, JEP 214, JEP 248, JEP 271, JEP 291

Implemented JEPs part 2

Java Core	JEP 102, JEP 193, JEP 254, JEP 264,JEP 266, JEP 268, JEP 269, JEP 274, JEP 277, JEP 285, JEP 290, JEP 259, JEP 255
Nashorn	JEP 236, JEP 292
Client Technologies	JEP 251, JEP 253, JEP 256, JEP 262, JEP 263, JEP 272, JEP 283
Internationalization	JEP 267, JEP 252, JEP 226

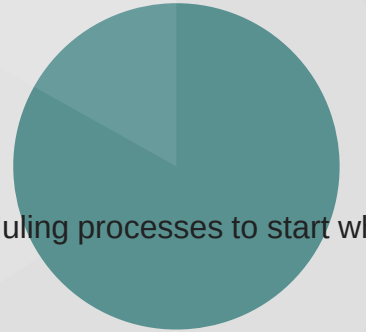
Java Core

JEP 102	Process API Updates
JEP 193	Variable Handles
JEP 254	Compact Strings
JEP 264	Platform Logging API and Service
JEP 266	More Concurrency Updates
JEP 268	XML Catalogs
JEP 269	Convenience Factory Methods for Collections
JEP 274	Enhanced Method Handles
JEP 285	Spin-Wait Hints
JEP 290	Filter Incoming Serialization Data

Java Core part 2

JEP 259	Stack-Walking API
JEP 255	Merge Selected Xerces 2.11.0 Updates into JAXP

JEP 102 Process API Updates



- **Run arbitrary commands**:
    - Filter running processes.
    - Redirect output.
    - Connect heterogeneous commands and shells by scheduling processes to start when another ends.
- **Test the execution of commands**:
    - Run a series of tests.
    - Log output.
    - Cleanup leftover processes.
- **Monitor commands**:
    - Monitor long-running processes and restart them if they terminate
    - Collect usage statistics

JEP 102 Process API Updates

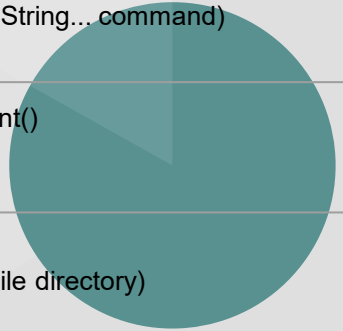
Class	Description
ProcessBuilder	The ProcessBuilder class lets you create and start operating system processes. Since Java 1.7
Process	The methods in the Process class let you to control processes started by the methods ProcessBuilder.start and Runtime.exec. Since Java 1.0
ProcessHandle	The ProcessHandle interface lets you identify and control native processes. It lets you control processes started only by the methods ProcessBuilder.start and Runtime.exec. The Process class lets you access process input, output, and error streams. Since Java 9
ProcessHandle.Info	The ProcessHandle.Info interface lets you retrieve information about a process, including processes created by the ProcessBuilder.start method and native processes. Since Java 9

JEP 102 Process API Updates: ProcessBuilder

Command	Strings that specify the external program file to call and its arguments, if any.	ProcessBuilder constructor command(String... command)
Environment	The environment variables (and their values). This is initially a copy of the system environment of the current process.	environment()
Working directory	By default, the current working directory of the current process.	directory() directory(File directory)
Standard input source	By default, a process reads standard input from a pipe; access this through the output stream returned by the Process.getOutputStream method.	redirectInput(ProcessBuilder.Redirect source)
Standard output and standard error destinations	By default, a process writes standard output and standard error to pipes; access these through the input streams returned by the Process.getInputStream and Process.getErrorStream methods.	redirectOutput(ProcessBuilder.Redirect destination) redirectError(ProcessBuilder.Redirect destination)
redirectErrorStream property	Specifies whether to send standard output and error output as two separate streams (with a value of false) or merge any error output with standard output.	redirectErrorStream() redirectErrorStream(boolean redirectErrorStream)

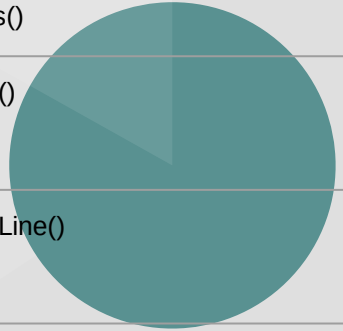
JEP 102 Process API Updates: Process

Strings that specify the external program file to call and its arguments, if any.	ProcessBuilder constructor command(String... command)
The environment variables (and their values). This is initially a copy of the system environment of the current process.	environment()
By default, the current working directory of the current process.	directory() directory(File directory)
By default, a process reads standard input from a pipe; access this through the output stream returned by the Process.getOutputStream method.	redirectInput(ProcessBuilder.Redirect source)
By default, a process writes standard output and standard error to pipes; access these through the input streams returned by the Process.getInputStream and Process.getErrorStream methods.	redirectOutput(ProcessBuilder.Redirect destination) redirectError(ProcessBuilder.Redirect destination)
Specifies whether to send standard output and error output as two separate streams (with a value of false) or merge any error output with standard output.	redirectErrorStream() redirectErrorStream(boolean redirectErrorStream)



JEP 102 Process API Updates:
ProcessHandle.Info Interface

Returns the arguments of the process as a String array.	arguments()
Returns the executable path name of the process.	command()
Returns the command line of the process.	commandLine()
Returns the start time of the process.	startInstant()
Returns the total CPU time accumulated of the process.	totalCpuDuration()
Returns the user of the process	user()



JEP 193 Variable Handles

Motivation

```
public final void lazySet(V newValue) {
    unsafe.putOrderedObject(this, valueOffset, newValue);
}

public final boolean compareAndSet(V expect, V update) {
    return unsafe.compareAndSwapObject(this, valueOffset, expect, update);
}
```

Here the this pointer is used together with a field offset to access the field.
But this is unsafe, since this field offset could be any long, and you might actually be accessing something completely different.
There are however performance benefits in doing it this way (it tells the VM to use specialized CPU instructions for instance), and because of that other people have used sun.misc.Unsafe even though it's an internal, and unsafe API.

JEP 193 Variable Handles

Goals:

- Safety. It must not be possible to place the Java Virtual Machine in a corrupt memory state. For example, a field of an object can only be updated with instances that are castable to the field type, or an array element can only be accessed within an array if the array index is within the array bounds.

- Integrity. Access to a field of an object follows the same access rules as with getfield and putfield byte codes in addition to the constraint that a final field of an object cannot be updated. (Note: such safety and integrity rules also apply to MethodHandles giving read or write access to a field.)

- Performance. The performance characteristics must be the same as or similar to equivalent sun.misc.Unsafe operations (specifically, generated assembler code should be almost identical modulo certain safety checks that cannot be folded away).

- Usability. The API must be better than the sun.misc.Unsafe API.

JEP 254/284 Compact Strings and Indify String Concatenation

Motivation

Adopts a more space-efficient internal representation for strings. Previously, the String class stored characters in a char array, using two bytes (16 bits) for each character. The new internal representation of the String class is a byte array plus an encoding-flag field.

The current implementation of the String class stores characters in a char array, using two bytes (sixteen bits) for each character. Data gathered from many different applications indicates that strings are a major component of heap usage and, moreover, that most String objects contain only Latin-1 characters. Such characters require only one byte of storage, hence half of the space in the internal char arrays of such String objects is going unused.

Replace the entire StringBuilder append dance with a simple invokedynamic call to java.lang.invoke.StringConcatFactory that will accept the values in the need of concatenation." Potential performance improvement can be gained from not needing to wrap primitive types and not needing to instantiate a bunch of extra objects. One of the primary motivations for this change was to "lay the groundwork for building optimized String concatenation handlers, implementable without the need to change the Java-to-bytecode compiler" and to "enable future optimizations of String concatenation without requiring further changes to the bytecode emitted by javac."

https://dzone.com/articles/jdk-9jep-280-string-concatenations-will-never-be-t

JEP 254/284 Compact Strings and Indify String Concatenation

```java
public class HelloWorldStringConcatComplex
{
   public static void main(final String[] arguments)
   {
      String message = "Hello";
      for (int i=0; i<25; i++)
      {
         message += i;
      }
      out.println(message);
   }
}
```

```
Classfile /C:/java/examples/helloWorld/classes/dustin/examples/HelloWorldStringConcatComplex.class
  Last modified Jan 30, 2019; size 766 bytes
  MD5 checksum 772c4a283c812d49451b5b756aef55f1
  Compiled from "HelloWorldStringConcatComplex.java"
public class dustin.examples.HelloWorldStringConcatComplex
  minor version: 0
  major version: 52
     . . .
  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=3, args_size=1
         0: ldc           #2                  // String Hello
         2: astore_1
         3: iconst_0
         4: istore_2
         5: iload_2
         6: bipush        25
         8: if_icmpge     36
        11: new           #3                  // class java/lang/StringBuilder
        14: dup
        15: invokespecial #4                  // Method java/lang/StringBuilder."19: invokevirtual #5
        22: iload_2
        23: invokevirtual #6                  // Method java/lang/StringBuilder.append:(I)Ljava/lang/Stri
        26: invokevirtual #7                  // Method java/lang/StringBuilder.toString:()Ljava/lang/Str
        29: astore_1
        30: iinc          2, 1
":()V 18: aload_1 33: goto 5 36: getstatic #8 // Field java/lang/System.out:Ljava/io/PrintStream; 39: alo
```

```
1
2  Classfile /C:/java/examples/helloWorld/classes/dustin/examples/HelloWorldStringConcatComplex.class
3    Last modified Jan 30, 2019; size 1018 bytes
4    MD5 checksum 967fef3e7625965ef060a831edb2a874
5    Compiled from "HelloWorldStringConcatComplex.java"
6  public class dustin.examples.HelloWorldStringConcatComplex
7    minor version: 0
8    major version: 55
9       . . .
10   public static void main(java.lang.String[]);
11     descriptor: ([Ljava/lang/String;)V
12     flags: (0x0009) ACC_PUBLIC, ACC_STATIC
13     Code:
14       stack=2, locals=3, args_size=1
15          0: ldc           #2                  // String Hello
16          2: astore_1
17          3: iconst_0
18          4: istore_2
19          5: iload_2
20          6: bipush        25
21          8: if_icmpge     25
22         11: aload_1
23         12: iload_2
24         13: invokedynamic #3,  0              // InvokeDynamic #0:makeConcatWithConstants:(Ljava/lang/Str
25         18: astore_1
26         19: iinc          2, 1
27         22: goto          5
28         25: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
29         28: aload_1
30         29: invokevirtual #5                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
31         32: return
```

JEP 264 Platform Logging and Service

Motivation

A minimal logging API which platform classes can use to log messages, together with a service interface for consumers of those messages. A library or application can provide an implementation of this service in order to route platform log messages to the logging framework of its choice. If no implementation is provided then a default implementation based upon the java.util.logging API is used.

Compared to the java.util.logging API, most modern logging frameworks (e.g., Log4J 2.0, Logback) are separated into a facade and an implementation. An application that logs through such an external framework should create loggers and perform logging through the facade provided, or supported, by that framework.

The proposed service enables applications to configure the JDK to use the same logging framework as the application: It would only need to provide an implementation of the service that returns platform loggers that wrap the loggers of the preferred logging framework.

The application would continue to use the facade provided by the logging framework it is using. The LoggerFinder service makes it possible to configure the JDK to use that same framework.

JEP 264 Platform Logging and Service

1. java.util.ServiceLoader API, LoggerFinder implementation is located and loaded using the system class loader.
2. If no concrete implementation is found, the JDK internal default implementation of the LoggerFinder service is used.
3. The default implementation of the service uses java.util.logging as a backend when the java.logging module, by default, log messages are routed to java.util.logging.Logger as before.
4. LoggerFinder service makes it possible for an application/framework to plug in its own external logging backend, without needing to configure both java.util.logging and that backend.

```
package java.lang;

...

public class System {

    System.Logger getLogger(String name) { ... }

    System.Logger getLogger(String name, ResourceBundle bundle) { ... }

}
```

JEP 266 More Concurrency Updates

Adds further concurrency updates to those introduced in JDK 8 in JEP 155: Concurrency Updates, including an interoperable publish-subscribe framework and enhancements to the CompletableFuture API.

1. New async mechanizm Flow with Publisher, Subscriber, Processor and Subscription
2. New methods in CompletableFuture
3. A few methods added in Atomic, new constructor in ForkJoinPool and TimeUnit

JEP 268: XML Catalogs

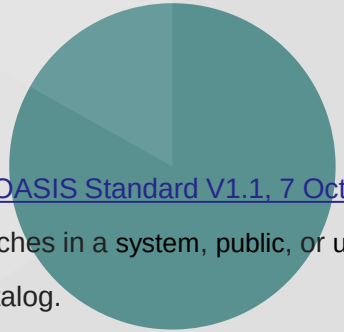
Motivation

XML, XSD and XSL documents may contain references to external resources that the Java XML processors need to retrieve to process the documents. External resources can cause a problem for the applications or the system. The Catalog API and the Java XML processors provide an option for developers and system administrators to better manage these external resources.
External resources can cause a problem for the applications or the system in these areas:

- Availability. When the resources are remote, the XML processors must be able to connect to the remote server. Even though connectivity is rarely an issue, it's still a factor in the stability of an application. Too many connections can be a hazard to servers that hold the resources (such as the well-documented case involving excessive DTD traffic directed to the W3C's servers), and this in turn could affect your applications. See Use Catalog with Schema Validation) for an example that solves this issue using the XML Catalog API.
- Performance. Although in most cases connectivity isn't an issue, a remote fetch can still cause a performance issue for an application. Furthermore, there may be multiple applications on the same system attempting to resolve the same source, and this would be a waste of system resources.
- Security. Allowing remote connections can pose a security risk if the application processes untrusted XML sources.
- Manageability. If a system processes a large number of XML documents, then externally referenced documents, whether local or remote, can become a maintenance hassle.

JEP 268: XML Catalogs

XML Catalog API Interfaces



The XML Catalog API defines the following interfaces:

- The Catalog interface represents an entity catalog as defined by XML Catalogs, OASIS Standard V1.1, 7 October 2005. A Catalog object is immutable. After it's created, the Catalog object can be used to find matches in a system, public, or uri entry. A custom resolver implementation may find it useful to locate local resources through a catalog.
- The CatalogFeatures class holds all of the features and properties the Catalog API supports, including javax.xml.catalog.files, javax.xml.catalog.defer, javax.xml.catalog.prefer, and javax.xml.catalog.resolve.
- The CatalogManager class manages the creation of XML catalogs and catalog resolvers.
- The CatalogResolver interface is a catalog resolver that implements SAX EntityResolver, StAX XMLResolver, DOM LS LSResourceResolver used by schema validation, and transform URIResolver. This interface resolves external references using catalogs.

JEP 269: Convenience Factory Methods for Collections

Motivation

Makes it easier to create instances of collections and maps with small numbers of elements. New static factory methods on the List, Set, and Map interfaces make it simpler to create immutable instances of those collections.

```
Set<String> alphabet = Set.of("a", "b", "c");
```

1. They are structurally immutable. Elements cannot be added or removed. Calling any mutator method will always cause UnsupportedOperationException to be thrown. However, if the contained elements are themselves mutable, this may cause the Set to behave inconsistently or its contents to appear to change.
2. They disallow null elements. Attempts to create them with null elements result in NullPointerException.
3. They are serializable if all elements are serializable.
4. They reject duplicate elements at creation time. Duplicate elements passed to a static factory method result in IllegalArgumentException.
5. The iteration order of set elements is unspecified and is subject to change.
6. They are value-based. Callers should make no assumptions about the identity of the returned instances. Factories are free to create new instances or reuse existing ones. Therefore, identity-sensitive operations on these instances (reference equality (==), identity hash code, and synchronization) are unreliable and should be avoided.

JEP 274: Enhanced Method Handles

Motivation

A method handle is a typed, directly executable reference to an underlying method, constructor, field, or similar low-level operation, with optional transformations of arguments or return values.

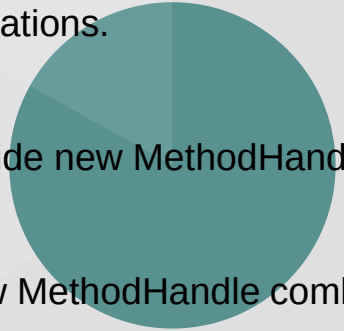
- Low-level mechanism for finding, adapting and invoking methods.
- Immutable and have no visible state.

For creating and using a MethodHandle, 4 steps are required:

1. Creating the lookup
2. Creating the method type
3. Finding the method handle
4. Invoking the method handle

JEP 274: Enhanced Method Handles

Enhances the MethodHandle, MethodHandles, and MethodHandles.Lookup classes of the java.lang.invoke package to ease common use cases and enable better compiler optimizations.



Additions include:

- In the MethodHandles class in the java.lang.invoke package, provide new MethodHandle combinators for loops and try/finally blocks.

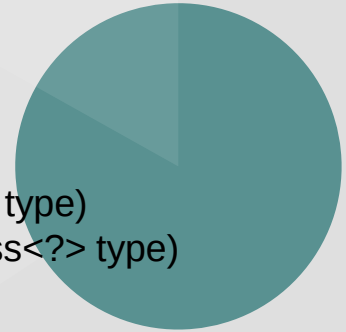
- Enhance the MethodHandle and MethodHandles classes with new MethodHandle combinators for argument handling.

- Implement new lookups for interface methods and, optionally, super constructors in the MethodHandles.Lookup class.

JEP 274: Enhanced Method Handles

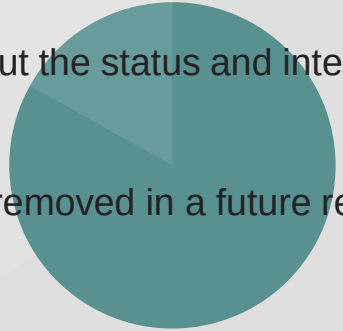
1. privateLookupIn(Class<?> targetClass, Lookup lookup)
2. Lookup dropLookupMode(int modeToDrop)
3. Class<?> defineClass(byte[] bytes)
4. Class<?> findClass(String targetName)
5. Class<?> accessClass(Class<?> targetClass)
6. VarHandle findVarHandle(Class<?> recv, String name, Class<?> type)
7. VarHandle findStaticVarHandle(Class<?> decl, String name, Class<?> type)
8. VarHandle unreflectVarHandle(Field f)
9. hasPrivateAccess()
10. MethodHandle arrayConstructor(Class<?> arrayClass)
11. MethodHandle arrayLength(Class<?> arrayClass)
12. VarHandle arrayElementVarHandle(Class<?> arrayClass)
13. VarHandle byteArrayViewVarHandle(Class<?> viewArrayClass, ByteOrder byteOrder)
14. MethodHandle varHandleExactInvoker(VarHandle.AccessMode accessMode, MethodType type)
15. MethodHandle varHandleInvoker(VarHandle.AccessMode accessMode, MethodType type)
16. MethodHandle zero(Class<?> type)
17. MethodHandle empty(MethodType type)
18. MethodHandle dropArgumentsToMatch(MethodHandle target, int skip, List<Class<?>> newTypes, int pos)

And 10 more….



JEP 277: Enhanced Deprecation

Motivation



Revamps the @Deprecated annotation to provide better information about the status and intended disposition of an API in the specification. Two new elements have been added:

- @Deprecated(forRemoval=true) indicates that the API will be removed in a future release of the Java SE platform.
- @Deprecated(since="version") contains the Java SE version string that indicates when the API element was deprecated, for those deprecated in Java SE 9 and beyond.

For example: @Deprecated(since="9", forRemoval=true)

@Deprecated annotations in the core platform have been updated.

You can use a new tool, jdeprscan, to scan a class library (JAR file) for uses of deprecated JDK API elements.

```
jdeprscan --class-path classes --release 7 dir/jar/class
jdeprscan --class-path classes --release 8 dir/jar/class
jdeprscan --class-path classes dir/jar/class
```

JEP 285: Spin-Wait Hints

Motivation

Defines an API that enables Java code to hint that a spin loop is executing. A spin loop repeatedly checks to see if a condition is true, such as when a lock can be acquired, after which some computation can be safely performed followed by the release of the lock. This API is purely a hint, and carries no semantic behavior requirements

```
Benchmark                    (iterations)  Mode  Cnt      Score     Error  Units
OnSpinWaitJmh.onSpinWait               10  avgt   25      0.099 ±   0.010  us/op
OnSpinWaitJmh.sleep                    10  avgt   25  13119.241 ± 202.076  us/op
OnSpinWaitJmh.yield                    10  avgt   25      7.859 ±   0.080  us/op
```

JEP 285: Spin-Wait Hints

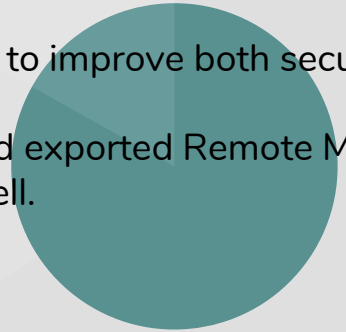
1. It's the same (and probably compiles to) the x86 opcode PAUSE and equivilant the Win32 macro YieldProcessor, GCC's __mm_pause() and the C# method Thread.SpinWait
2. It's a very weakened form of yielding: it's about telling your CPU that you are in a loop that may burn some quite CPU-cycles waiting for something to happen (busy-waiting).
3. This way, The CPU can assign more resources to other threads, without actually loading the OS scheduler and dequeuing a sleeping thread (which may be expensive).
4. A common use for that is spin-locking, when you know the contention on a shared memory is very infrequent or finishes very quickly, a spinlock may perform better than an ordinary lock.
5. yield can be implemented in terms of Thread.onSpinWait(), hinting that while trying to lock the lock, the CPU can give more resources to other threads.

PS. this technique of yielding is extremely common and popular when implementing a lock-free algorithms, since most of them depends on busy-waiting (which is implemented almost always as an atomic CAS loop). this has every real-world use you can imagine.

JEP 290: Filter Incoming Serialization Data

Motivation

- Allows incoming streams of object-serialization data to be filtered to improve both security and robustness.
- Object-serialization clients can validate their input more easily, and exported Remote Method Invocation (RMI) objects can validate invocation arguments more easily as well.



Goal

- Provide a flexible mechanism to narrow the classes that can be deserialized from any class available to an application down to a context-appropriate set of classes.
- Provide metrics to the filter for graph size and complexity during deserialization to validate normal graph behaviors.
- Provide a mechanism for RMI-exported objects to validate the classes expected in invocations.
- The filter mechanism must not require subclassing or modification to existing subclasses of ObjectInputStream.
- Define a global filter that can be configured by properties or a configuration file.

JEP 259: Stack-Walking API

Motivation

Provides a stack-walking API that allows easy filtering and lazy access to the information in stack traces.
There is no standard API to traverse selected frames on the execution stack efficiently and access the Class instance of each frame.

There are existing APIs that provide access to a thread's stack:

1. Throwable::getStackTrace and Thread::getStackTrace return an array of StackTraceElement objects, which contain the class name and method name of each stack-trace element.
2. SecurityManager::getClassContext is a protected method, which allows a SecurityManager subclass to access the class context.

These APIs require the VM to eagerly capture a snapshot of the entire stack, and they return information representing the entire stack. There is no way to avoid the cost of examining all the frames if the caller is only interested in the top few frames on the stack. Both the Throwable::getStackTrace and Thread::getStackTrace methods return an array of StackTraceElement objects, which contain class names and method names but not the actual Class instances. For applications interested in the entire stack, the specification allows the VM implementation to omit some frames in the stack for performance. In other words, Thread::getStackTrace may return a partial stack trace.

JEP 259: Stack-Walking API

1. A StackWalker is easily accessible with the static **getInstance** methods.The different calls allow you to specify one option or a set of them as well as the estimated size of the number of frames to capture
2. The **forEach** method will forward all the unfiltered frames to the specified Consumer<StackFrame> callback
3. The **walk** method takes a function that gets a stream of stack frames and returns the desired result. One of the big advantages of using the walk method is that because the stack-walking API lazily evaluates frames, the use of the limit operator actually reduces the number of frames that are recovered
4. **getCallerClass()** make the common case fast and simple, the StackWalker provides an optimized way to get the caller class. This call is faster than doing the equivalent call through the Stream and is faster than using the SecurityManager
5. **StackFrame** the methods forEach and walk will pass StackFrame instances in the stream or to the consumer callback. It also gives lazy access to file name and line number but this will create a StackTraceElement to which it will delegate the call. The creation of the StackTraceElement is costly and deferred until it is needed for the first time. The toString method also delegates to StackTraceElement.
6. Frame Visibility: By default the stack walker will skip hidden and reflective frames. With the **SHOW_REFLECT_FRAMES** option we see the reflection frames but the hidden frame is still skipped. With **SHOW_HIDDEN_FRAMES** it outputs all the reflection and hidden frames.

JEP 255: Merge Selected Xerces 2.11.0 Updates into JAXP

Motivation

The JDK contains the older Xerces 2.7.1 parser. During the development of JDK 7, it was updated with all critical and many major changes from Xerces 2.10.0. Since then, Xerces 2.11.0 was released. Upgrading to the latest release will help improve the quality of the JDK implementation.

- XML 1.0 (4th Edition)
- Namespaces in XML 1.0 (2nd Edition)
- XML 1.1 (2nd Edition)
- Namespaces in XML 1.1 (2nd Edition)
- W3C XML Schema 1.0 (2nd Edition)
- W3C XML Schema 1.1
- W3C XML Schema Definition Language (XSD): Component Designators (Candidate Recommendation, January 2010)
- XInclude 1.0 (2nd Edition)
- OASIS XML Catalogs 1.1
- SAX 2.0.2
- DOM Level 3 Core, Load and Save(
- DOM Level 2 Core, Events, Traversal and Range
- Element Traversal (org.w3c.dom.ElementTraversal)
- JAXP 1.4
- StAX 1.0 Event API (javax.xml.stream.events)

https://xerces.apache.org/

Key Changes

JSR 376	Java Platform Module System
JEP 261	Module System
JEP 200	The Modular JDK
JEP 220	Modular Run-Time Images
JEP 260	Encapsulate Most Internal APIs
JEP 223	New Version-String Scheme

JEP 223

JEP 223: New Version-String Scheme

Provides a simplified version-string format that helps to clearly distinguish major, minor, security, and patch update releases.

The new version-string format is as follows:

```
$MAJOR.$MINOR.$SECURITY.$PATCH
```

- **`$MAJOR`** is the version number that is incremented for a major release, for example JDK 9, which contains significant new features as specified by the Java SE platform specification. A major release contains new features and changes to existing features, which are planned and announced well in advance.
- **`$MINOR`** is the version number that is incremented for each minor update, such as bug fixes, revisions to standard APIs, or implementation of features outside the scope of the relevant platform specifications.
- **`$SECURITY`** is the version number that is incremented for a security-update release, which contains critical fixes, including those necessary to improve security.
- **`$PATCH`** is the version number that is incremented for a release containing security and high-priority customer fixes that have been tested together.

See New Version String Format in *Java Platform, Standard Edition Installation Guide*.

Tools

JEP 222	jshell: The Java Shell (Read-Eval-Print Loop)
JEP 228	Add More Diagnostic Commands
JEP 231	Remove Launch-Time JRE Version Selection
JEP 238	Multi-Release JAR Files
JEP 240	Remove the JVM TI hprof Agent
JEP 241	Remove the jhat Tool
JEP 245	Validate JVM Command-Line Flag Arguments
JEP 247	Compile for Older Platform Versions
JEP 282	jlink: The Java Linker

JEP 222: jshell: The Java Shell (Read-Eval-Print Loop)

Adds Read-Eval-Print Loop (REPL) functionality to the Java platform.

The jshell tool provides an interactive command-line interface for evaluating declarations, statements, and expressions of the Java programming language. It facilitates prototyping and exploration of coding options with immediate results and feedback. The immediate feedback combined with the ability to start with expressions is useful for education—whether learning the Java language or just learning a new API or language feature.

https://docs.oracle.com/javase/9/tools/jshell.htm#JSWOR-GUID-C337353B-074A-431C-993F-60C226163F00

JEP 228: Add More Diagnostic Commands

The jcmd utility is used to send diagnostic command requests to the JVM. It must be used on the same machine on which the JVM is running, and have the same effective user and group identifiers that were used to launch the JVM. Each diagnostic command has its own set of arguments. To display the description, syntax, and a list of available arguments for a diagnostic command, use the name of the command as the argument. For example

jcmd pid help command
If arguments contain spaces, then you must surround them with single or double quotation marks (' or "). In addition, you must escape single or double quotation marks with a backslash (\) to prevent the operating system shell from processing quotation marks. Alternatively, you can surround these arguments with single quotation marks and then with double quotation marks (or with double quotation marks and then with single quotation marks).

If you specify the process identifier (pid) or the main class (main-class) as the first argument, then the jcmd utility sends the diagnostic command request to the Java process with the specified identifier or to all Java processes with the specified name of the main class. You can also send the diagnostic command request to all available Java processes by specifying 0 as the process identifier.

If you run jcmd without arguments or with the -l option, it prints the list of running Java process identifiers with the main class and command-line arguments that were used to launch the process. Running jcmd with the -h or -help option prints the tool's help message.

JEP 231: Remove Launch-Time JRE Version Selection

The "Multiple JRE" ("mJRE") feature allows a developer to specify what JRE version, or range of versions, can be used to launch an application. The version-selection criteria can be specified in a manifest entry of the application's jar file (JRE-Version) or as a command-line option (-version:) to the java launcher. If the version of the JRE that is launched does not satisfy the criteria then the launcher searches for a version that does, and if it finds one then it launches that version.

Deploying an application, in practice, requires doing more than just selecting a particular JRE. Modern applications are typically deployed via Java Web Start (JNLP), native OS packaging systems, or active installers, and all of these technologies have their own ways of finding, and even sometimes installing and later updating, an appropriate JRE for the application.

The mJRE feature addresses only one part of the overall deployment problem. It was, moreover, never fully documented when it was introduced in JDK 5: The -version: option is mentioned in the documentation of the java command but the JRE-Version manifest entry is not mentioned in any of the usual JDK documentation, nor in the Java SE Platform Specification. So far as we know, this feature was very rarely used. It needlessly complicates the implementation of the Java launcher, making it burdensome to maintain and enhance.

JEP 238: Multi-Release JAR Files

Extend the JAR file format to allow multiple, Java-release-specific versions of class files to coexist in a single archive.

Goals

1. Enhance the Java Archive Tool (jar) so that it can create multi-release JAR files.
2. Implement multi-release JAR files in the JRE, including support in the standard class loaders and JarFile API.
3. Enhance other critical tools (e.g., javac, javap, jdeps, etc.) to interpret multi-release JAR files.
4. Support multi-release **modular** JAR files for goals 1 to 3.
5. Preserve performance: The performance of tools and components that use multi-release JAR files must not be significantly impacted. In particular, performance when accessing ordinary (i.e., not multi-release) JAR files must not be degraded.

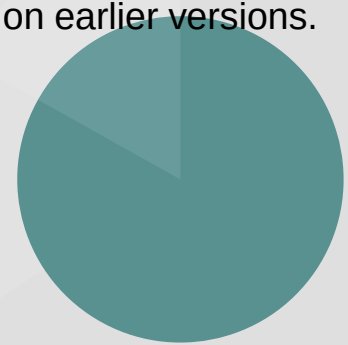
Third party libraries and frameworks typically support a range of Java platform versions, generally going several versions back. As a consequence they often do not take advantage of language or API features available in newer releases since it is difficult to express conditional platform dependencies, which generally involves reflection, or to distribute different library artifacts for different platform versions.

This creates a disincentive for libraries and frameworks to use new features, that in turn creates a disincentive for users to upgrade to new JDK versions---a vicious circle that impedes adoption, to everyone's detriment.

Some libraries and frameworks, furthermore, use internal APIs of the JDK that will be made inaccessible in Java 9 when module boundaries are strictly enforced. This also creates a disincentive to support new platform versions when there are public, supported API replacements for such internal APIs.

JEP 238: Multi-Release JAR Files

Cross-compilation is the feature in Java that can compile files for running on earlier versions.
This means there is no need for us to install separate JDK versions.



1. Firstly, compile the old code for the Java 7 platform:
   javac --release 7 -d classes src\main\java\com*.java
2. Secondly, compile the new code for the Java 9 platform:
   javac --release 9 -d classes-9 src\main\java9\com*.java

The release option is used to indicate the version of Java compiler and target JRE.

Finally, create the MRJAR file using version 9+:

jar --create --file target/mrjar.jar --main-class com.App -C classes . --release 9 -C classes-9

The release option followed by a folder name makes the contents of that folder to be packaged inside the jar file under the version number value.
The MANIFEST.MF file has the property set to let the JVM know that this is an MRJAR file:

Multi-Release: true

JEP 238: Multi-Release JAR Files

```
Root of JAR
        -   A1.class
        -   B1.class
        -   C1.class
        -   D1.class
        -   E1.class
-META-INF
        - MANIFEST.MF
        - versions
                - 8
                        - A1.class
                        - B1.class
                        - F1.class
                - 9
                        - A1.class
                        - C1.class
                        - D1.class
                        - F1.class
                        - G1.class
```

JEP 240: Remove the JVM TI hprof Agent

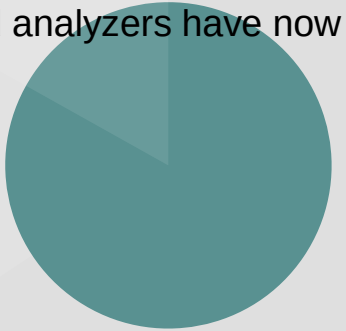
Motivation

Removes the hprof agent from the JDK. The hprof agent was written as demonstration code for the JVM Tool Interface and not intended to be a production tool.

This functionality has been superseded by the same functionality in the JVM. Using the Diagnostic Command GC.heap_dump (jcmd <pid> GC.heap_dump) it is possible to ask the JVM to dump the heap in the hprof file format (this is also available via jmap -dump).

JEP 241: Remove the jhat Tool

jhat was added in JDK 6, based upon the java.net HAT project. jhat is an experimental, unsupported, and out-of-date tool. Superior heap visualizers and analyzers have now been available for many years.



JEP 245: Validate JVM Command-Line Flag Arguments

You use values provided to all Java Virtual Machine (JVM) command-line flags for validation and, if the input value is invalid or out-of-range, then an appropriate error message is displayed.

Whether they're set ergonomically, in a command line, by an input tool, or through the APIs (classes contained in the package java.lang.management) the values provided to all Java Virtual Machine (JVM) command-line flags are validated. Ergonomics are described in Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide.
Range and constraints are validated either when all flags have their values set during JVM initialization or a flag's value is changed during runtime (for example using the jcmd tool). The JVM is terminated if a value violates either the range or constraint check and an appropriate error message is printed on the error stream.

-XX:AllocatePrefetchStyle=5

-XX:+PrintFlagsRanges

JEP 247: Compile for Older Platform Versions

Motivation

Enhances javac so that it can compile Java programs to run on selected earlier versions of the platform.

- javac provides two command line options, -source and -target, which can be used to select the version of the Java language accepted by the compiler and the version of the class files it produces, respectively.
- By default, however, javac compiles against the most-recent version of the platform APIs. The compiled program can therefore accidentally use APIs only available in the current version of the platform. Such programs cannot run on older versions of the platform, regardless of the values passed to the -source and -target options. This is a long-term usability pain point, since users expect that by using these options they'll get class files that can run on the the platform version specified by -target.

javac -source 1.8 -target 1.8 {MyClass}.java

JEP 282: jlink: The Java Linker

Motivation

Create a tool that can assemble and optimize a set of modules and their dependencies into a custom run-time image as defined in JEP 220.

The jlink tool defines a plug-in mechanism for transformation and optimization during the assembly process, and for the generation of alternative image formats. It can create a custom runtime optimized for a single program.

A basic invocation of the linker tool, jlink, is:

```
$ jlink --module-path <modulepath> --add-modules <modules> --limit-modules <modules> --output <path>
```

where:

- --module-path is the path where observable modules will be discovered by the linker; these can be modular JAR files, JMOD files, or exploded modules
- --add-modules names the modules to add to the run-time image; these modules can, via transitive dependencies, cause additional modules to be added
- --limit-modules limits the universe of observable modules
- --output is the directory that will contain the resulting run-time image

The --module-path, --add-modules, and --limit-modules options are described in further detail in JEP 261.

Other options that jlink will support include:

- --help to print a usage/help message
- --version to print version information

JEP 282: jlink: The Java Linker

A basic invocation of the linker tool, `jlink`, is:

```
$ jlink --module-path <modulepath> --add-modules <modules> --limit-modules
<modules> --output <path>
```

where:

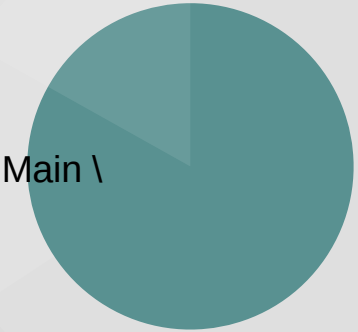
- `--module-path` is the path where observable modules will be discovered by the linker; these can be modular JAR files, JMOD files, or exploded modules
- `--add-modules` names the modules to add to the run-time image; these modules can, via transitive dependencies, cause additional modules to be added
- `--limit-modules` limits the universe of observable modules
- `--output` is the directory that will contain the resulting run-time image

The `--module-path`, `--add-modules`, and `--limit-modules` options are described in further detail in JEP 261.

Other options that `jlink` will support include:

- `--help` to print a usage/help message
- `--version` to print version information

JEP 282: jlink: The Java Linker



```
$JAVA_HOME/bin/jlink \
        --module-path $JAVA_HOME/jmods:out \
        --add-modules com.ra.jokeapp.ui \
        --launcher JOKER=com.ra.jokeapp.ui/com.ra.jokeapp.ui.Main \
        --compress 2        \
        --no-header-files   \
        --no-man-pages      \
        --strip-debug       \
        --output out/jre

$JAVA_HOME/bin/jlink \
        --module-path $JAVA_HOME/jmods:out \
        --add-modules com.ra.jokeapp.ui \
        --launcher JOKER=com.ra.jokeapp.ui/com.ra.jokeapp.ui.Main \
        --output out/jre
```

Java Language

JEP 213	Milling Project Coin

JEP 213: Milling Project Coin

1. Allow @SafeVargs on private instance methods. The @SafeVarargs annotation can only be applied to methods which cannot be overridden, including static methods and final instance methods. Private instance methods are another use case that @SafeVargs could accommodate.

1. Allow effectively-final variables to be used as resources in the try-with-resources statement. If the resource is referenced by a final or effectively final variable, a try-with-resources statement can manage the resource without a new variable being declared.

1. Allow diamond with anonymous classes if the argument type of the inferred type is denotable. Because the inferred type using diamond with an anonymous class constructor could be outside of the set of types supported by the signature attribute, using diamond with anonymous classes was disallowed in Java SE 7. As noted in the JSR 334 proposed final draft, it would be possible to ease this restriction if the inferred type was denotable.

1. Complete the removal, begun in Java SE 8, of underscore from the set of legal identifier names.

1. Support for private methods in interfaces was briefly in consideration for inclusion in Java SE 8 as part of the effort to add support for Lambda Expressions, but was withdrawn to enable better focus on higher priority tasks for Java SE 8.

JVM Tuning

JEP 158	Unified JVM Logging
JEP 214	Remove GC Combinations Deprecated in JDK 8
JEP 248	Make G1 the Default Garbage Collector
JEP 271	Unified GC Logging
JEP 291	Deprecate the Concurrent Mark Sweep (CMS) Garbage Collector

JEP 158: Unified JVM Logging

Introduces a common logging system for all components of the JVM.

The Java Virtual Machine (JVM) unified logging framework provides a common logging system for all components of the JVM. GC logging for the JVM has been changed to use the new logging framework. The mapping of old GC flags to the corresponding new Xlog configuration is described in Convert GC Logging Flags to Xlog. In addition, runtime logging has also been changed to use the JVM unified logging framework. The mapping of legacy runtime logging flags to the corresponding new Xlog configuration is described in Convert Runtime Logging Flags to Xlog.

Default Configuration:

-Xlog:all=trace:stdout:uptime,level,tags

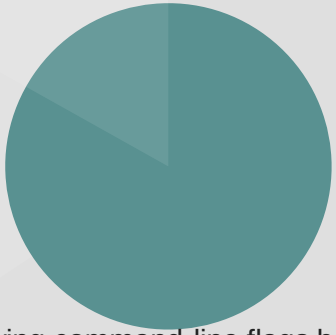
1. Using default logging levels
2. A lot logging tags
3. Output to file or console
4. Convert runtime logging flags to Xlog

https://docs.oracle.com/javase/9/tools/java.htm#JSWOR-GUID-BE93ABDC-999C-4CB5-A88B-1994AAAC74D5

JEP 214: Remove GC Combinations Deprecated in JDK 8

Removes garbage collector (GC) combinations that were deprecated in JDK 8.
This means that the following GC combinations no longer exist:

- DefNew + CMS
- ParNew + SerialOld
- Incremental CMS



The "foreground" mode for Concurrent Mark Sweep (CMS) has also been removed. The following command-line flags have been removed:

- -Xincgc
- -XX:+CMSIncrementalMode
- -XX:+UseCMSCompactAtFullCollection
- -XX:+CMSFullGCsBeforeCompaction
- -XX:+UseCMSCollectionPassing

The command line flag -XX:+UseParNewGC no longer has an effect. ParNew can only be used with CMS and CMS requires ParNew. Thus, the -XX:+UseParNewGC flag has been deprecated and will likely be removed in a future release.

JEP 248: Make G1 the Default Garbage Collector

Makes Garbage-First (G1) the default garbage collector (GC) on 32- and 64-bit server configurations. Using a low-pause collector such as G1 provides a better overall experience, for most users, than a throughput-oriented collector such as the Parallel GC, which was previously the default.

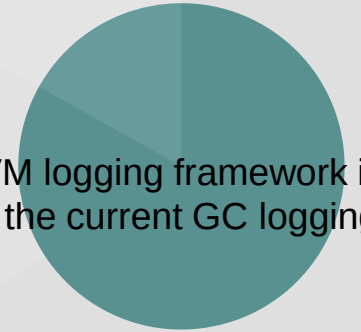
The Garbage-First (G1) garbage collector is targeted for multiprocessor machines with a large amount of memory. It attempts to meet garbage collection pause-time goals with high probability while achieving high throughput with little need for configuration. G1 aims to provide the best balance between latency and throughput using current target applications and environments whose features include:

- Heap sizes up to ten of GBs or larger, with more than 50% of the Java heap occupied with live data.
- Rates of object allocation and promotion that can vary significantly over time.
- A significant amount of fragmentation in the heap.
- Predictable pause-time target goals that aren’t longer than a few hundred milliseconds, avoiding long garbage collection pauses.

G1 replaces the Concurrent Mark-Sweep (CMS) collector. It is also the default collector.

The G1 collector achieves high performance and tries to meet pause-time goals in several ways described in the following sections.

JEP 271: Unified GC Logging



Reimplements Garbage Collection (GC) logging using the unified JVM logging framework introduced in JEP 158. GC logging is re-implemented in a manner consistent with the current GC logging format; however, some differences exist between the new and old formats.

-Xlog:gc,safepoint

JEP 291: Deprecate the Concurrent Mark Sweep (CMS) Garbage Collector

Deprecates the Concurrent Mark Sweep (CMS) garbage collector. A warning message is issued when it is requested on the command line, using the -XX:+UseConcMarkSweepGC option. The Garbage-First (G1) garbage collector is intended to be a replacement for most uses of CMS.

Internationalization

JEP 267	Unicode 8.0
JEP 252	CLDR Locale Data Enabled by Default
JEP 226	UTF-8 Properties Files

JEP 267: Unicode 8.0

- This is a follow-on to JEP 227, which introduced Unicode 7.0 in JDK 9. Unicode 8.0 adds an additional ~8,000 characters, 10 blocks, and 6 scripts.
- Upgrade existing platform APIs to support version 8.0 of the Unicode Standard.
- Support the latest version of Unicode, with changes to the following classes:
  a. Character and String in the java.lang package,
  b. NumericShaper in the java.awt.font package, and
  c. Bidi, BreakIterator, and Normalizer in the java.text package.

JEP 226: UTF-8 Properties Files

In Java SE 9, properties files are loaded in UTF-8 encoding. In previous releases, ISO-8859-1 encoding was used for loading property resource bundles. UTF-8 is a much more convenient way to represent non-Latin characters.

Most existing properties files should not be affected: UTF-8 and ISO-8859-1 have the same encoding for ASCII characters, and human-readable non-ASCII ISO-8859-1 encoding is not valid UTF-8. If an invalid UTF-8 byte sequence is detected, the Java runtime automatically rereads the file in ISO-8859-1.

If there is an issue, consider the following options:

- Convert the properties file into UTF-8 encoding.
- Specify the runtime system property for the properties file's encoding, as:

  `java.util.PropertyResourceBundle.encoding=ISO-8859-1`

Deployment

JEP 275	Modular Java Application Packaging
JEP 289	Deprecate the Applet API

Security

JEP 219	Datagram Transport Layer Security (DTLS)
JEP 244	TLS Application-Layer Protocol Negotiation Extension
JEP 249	OCSP Stapling for TLS
JEP 246	Leverage CPU Instructions for GHASH and RSA
JEP 273	DRBG-Based SecureRandom Implementations
JEP 288	Disable SHA-1 Certificates
JEP 229	Create PKCS12 Keystores by Default
JEP 287	SHA-3 Hash Algorithms

JVM

JEP 165	Compiler Control
JEP 197	Segmented Code Cache
JEP 276	Dynamic Linking of Language-Defined Object Models

Client Technologies

JEP 251	Multi-Resolution Images
JEP 253	Prepare JavaFX UI Controls and CSS APIs for Modularization
JEP 256	BeanInfo Annotations
JEP 262	TIFF Image I/O
JEP 263	HiDPI Graphics on Windows and Linux
JEP 272	Platform-Specific Desktop Features
JEP 283	Enable GTK 3 on Linux

Nashorn

JEP 236	Parser API for Nashorn
JEP 292	Implement Selected ECMAScript 6 Features in Nashorn

Javadoc

JEP 221	Simplified Doclet API
JEP 224	HTML5 Javadoc
JEP 225	Javadoc Search
JEP 261	Module System