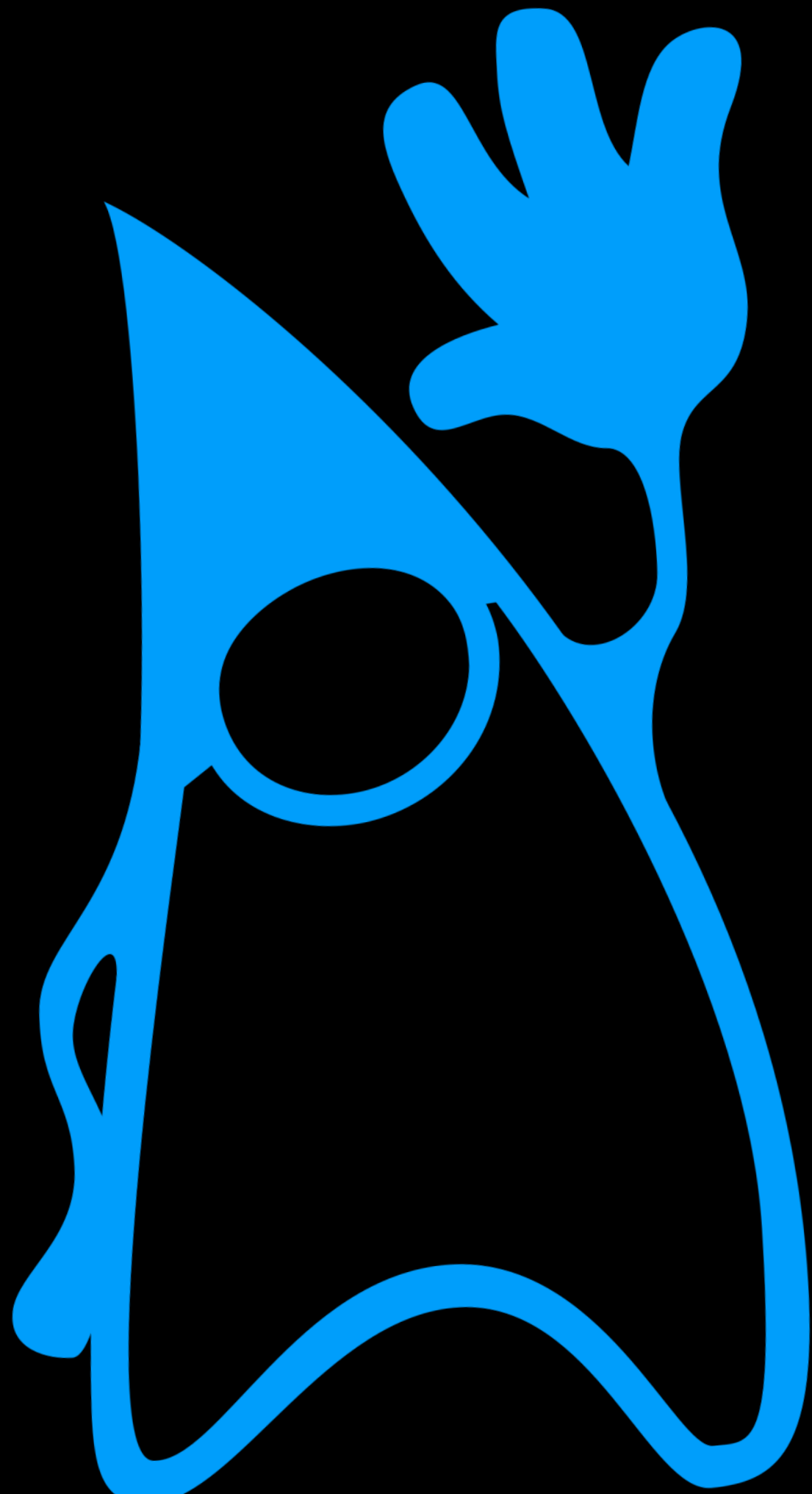


# Java 10, 11 & 12






The Netcetera University

2019-05-15 @ ZRH zyp60.right

2019-06-19 @ SKP Alu1

Philippe Marschall & Michael Pellaton

# Topics

-  General News
-  Breaking Changes
-  Habits to Stop
-  Habits to Start
-  Further Readings

# General News



# NCA Java Strategy

- CTO: [Netcetera Java Strategy](#)
- AO: Themas «Platforms & Runtimes > [Java](#)»
- Netcetera wants projects to use the current **LTS** version of Java for **production** (currently 11)
- Netcetera also supports the most **current non-LTS** version for **testing** (currently 12)
- Netcetera support for Java 8 until March 2020, after that SYS will automatically upgrade you to Java 11
- Current packages and base images use Zulu CA

# Java Strategy & Thomas TL;DR:

## Use Java 11

# Java Licences, LTS, ...

Watch the first 40 minutes of the

[NCAU Talk: What's up Java?](#)

# Breaking Changes



# Use CLDR Locale Data by Default

- Was part of **Java 9**
- Somehow **flew under the radar**
- Techblog: [Unexpected Incompatibilities with Java 11](#)



# CLDR Locale Data

- Common Locale Data Repository is the de-facto standard for locale data
- Introduced in Java 8, made default in Java 9

# So What?

Locale	Java 8	Java 11
de_CH	1'234'567.80	1'234'567.80
de_AT	1.234.567,80	1 234 567,80

Affects **parsing and printing**

Is this an issue for you?

**It depends**



# How to get the old behavior back

9

`DecimalFormatSymbols#setGroupingSeparator(char)`

or

`java.locale.providers` to a value with  
COMPAT ahead of CLDR

or

implement a custom `LocaleServiceProvider`

# DecimalFormatSymbols#setGroupingSeparator(char)

- Pro
  - most future proof
  - benefit from CLDR updates
- Con
  - have to find and change every code place
    - hard for 3rd party code

# -Djava.locale.providers=COMPAT,CLDR

- Pro

- no code changes required

- Con

- unclear how future proof
- do not benefit from CLDR updated
- JVM option
  - hard in “managed environment” / application server

# custom LocaleServiceProvider

- Pro
  - no code changes
  - benefit from CLDR updates
- Con
  - JVM option, JAR on modulepath or classpath
    - hard in “managed environment” / application server

# Removal of javah

- The native header generator **jvah was removed**
- The functionality of javah is **now covered** with new functionality in the Java compiler **javac**
- With Maven builds, just pass the -h parameter and the output path to the maven-compiler-plugin



# Removal of Web Service Stack

- Full web service stack incl common annotations (JSR-250) was included with Java 6 (developed for Java EE)
- Syncing development of SE and EE was a nightmare
- Deprecated in Java 9
- **Removal in Java 11**
  - **JAX-WS**
  - **JAXB**
  - **JAF**
  - **common annotations**
- Mitigation: **add dependencies explicitly**

# Removal of CORBA

- CORBA shares the fate of the web service stack
- Deprecated in Java 9
- **Removal in Java 11**
- Mitigation: **use another CORBA implementation** like the GlassFish-Corba or JacORB

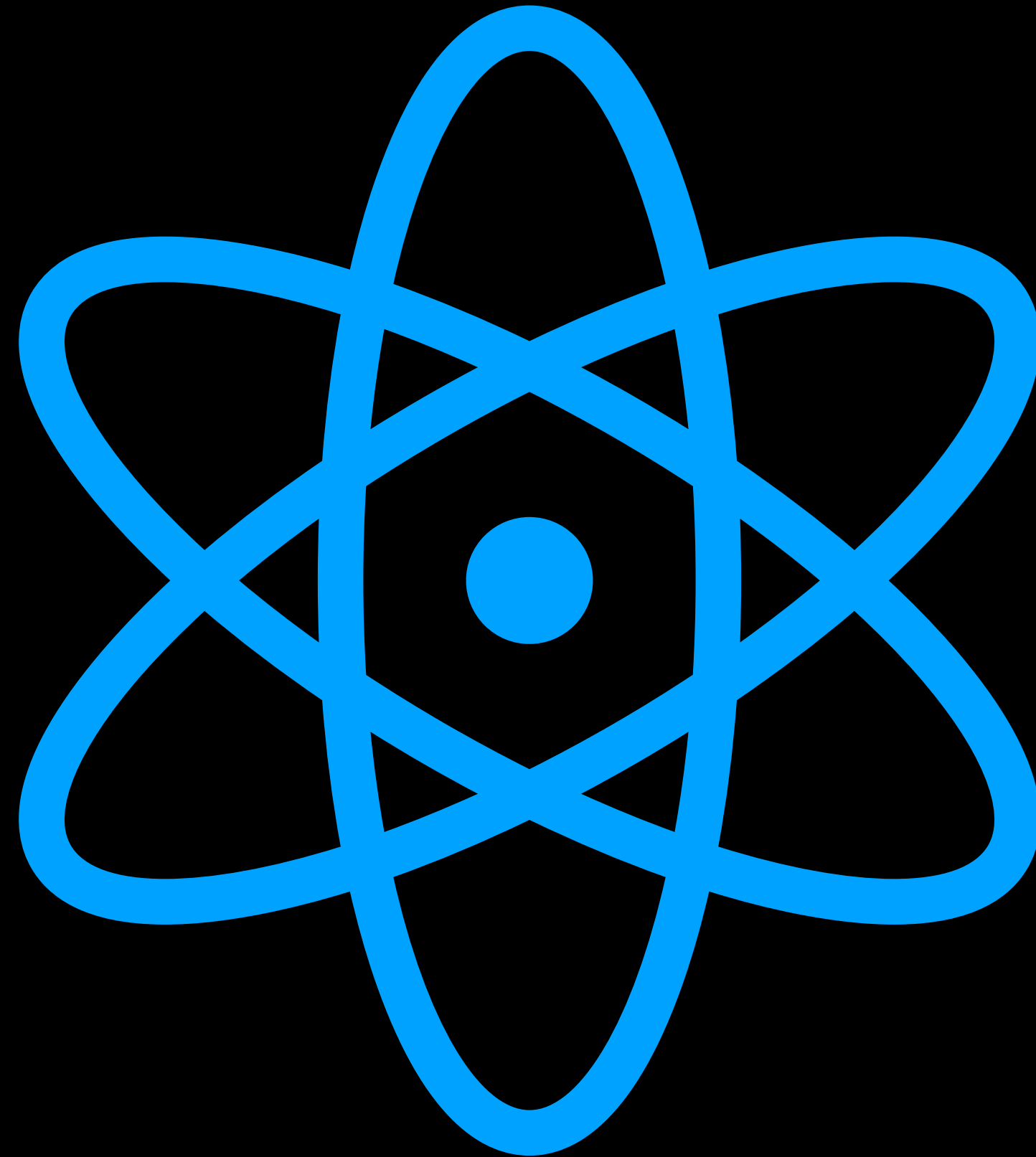
# Habits to Stop



# Nashorn JS Engine Depreacted

- The **Nashorn JavaScript engine** bundled with the JDK was marked as **depreacted for removal**
- Oracle is unwilling to maintain Nashorn and keep the pace of JavaScript
- Mitigation: migrate to **GraalVM**, use **Rhino** engine or the **J2V8** V8 bindings for Java

# Habits to Start



# Local-Variable Type Inference

- Enhanced type inference to reduce boiler plate
- For local variables with initializers including loop variables (Java 10)
- For implicitly typed lambda parameters (Java 11)

# var Demo

```
// var can be used for local variables with initializers  
List<String> names = new ArrayList<String>();  
var names = new ArrayList<String>();
```

```
// var can be used for loop variables  
for (String name : names) { }  
for (var name : names) { }
```

```
for (int i = 0; i < 10; i++) { }  
for (var i = 0; i < 10; i++) { }
```

# So What?

```
Map<String, Map<Class<?>, List<Object>>> entries  
    = new HashMap<String, Map<Class<?>, List<Object>>>();
```

```
var entries = new HashMap<String, Map<Class<?>, List<Object>>>();
```



# With Lambdas

```
// Java <11: implicitly typed lambda param  
IntStream.of(1, 2)  
    .map(i -> i * 42)  
    .forEach(System.out::println);
```

```
// Java <11: explicitly typed lambda var to annotate  
IntStream.of(1, 2)  
    .map(@TypeAnnotation int i) -> i * 42)  
    .forEach(System.out::println);
```

```
// Java >=11: implicitly typed lambda var to annotate  
IntStream.of(1, 2)  
    .map(@TypeAnnotation final var i) -> i * 42)  
    .forEach(System.out::println);
```

# Guidelines

## Style Guidelines for Local Variable Type Inference in Java

- Choose variable names that provide useful information
- Consider var when the initializer provides sufficient information to the reader
- ...
- Take care when using var with literals

# Launch Single-File Source-Code Programs

- **Single source file** Java **programs** can now be **compiled and run in one step** by **java** without employing **javac**
- The same programs also support the **#!** (shebang) mechanism on **Unixoids**.

```
#!/usr/bin/java --source 11
```

- However, files containing the **#!** must not have the **.java** suffix and are **invalid** when placed in ordinary packages

# Launch Single Source Demo

```
$ cat Hello.java
```

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello NCAU!");  
    }  
}
```

```
$ java Hello.java
```

```
Hello NCAU!
```

```
$
```

# Shebang Demo

```
$ cat shebang
#!/usr/bin/java --source 11
public class Shebang {
    public static void main(String[] args) {
        System.out.println("Hello Shebang!");
    }
}
```

```
$ chmod +x hello
```

```
$ ./hello
Hello Shebang!
```

```
$
```

# Java Flight Recorder

- Profiler integrated into JVM
  - Does not rely on byte code transformation
  - Low overhead, can be run in production
- Can be extended with custom events

# User Interface

- Java / Zulu Mission Control 7
- Includes JMX browser like VisualVM / JConsole
- Includes interface for jcmd

# Download

- Azul Zulu Mission Control 7
- Oracle JMC 7 Early-Access Builds



# Demo JFR

- JMX
- Custom chart
- Diagnostic command

# Custom Events

- APIs to generate custom events
  - Declarative API
  - Generic API

# Create Custom Event Recipe



1. Subclass Event
2. Add annotations
3. Add annotated fields

# Supported Data Types

- primitives
- `java.lang.String`
- `java.lang.Class`
- `java.lang.Thread`

# Supported Data Types

- Can be annotated for semantics
  - @BooleanFlag
  - @DataAmount
  - @Frequency
  - @MemoryAddress
  - @Percentage
  - @Timespan
  - @Timestamp
  - @Unsigned
  - @TransitionFrom
  - @TransitionTo

# Generating Events

new

.begin( )

.end( )

.commit( )

# Demo Custom Events

# What did I just witness?

- <https://github.com/marschall/jfr-libraries>



# Creating Recordings

- Command line: `jcmd`
- Mission Control GUI
- JMX

# Java 8 Backports

- Azul Zulu 8u202
- Alibaba Dragonwell 8
- Work on upstream ongoing

# (Application) Class Data Sharing

App CDS

# Caching

- Speeding up application by not doing the same thing over and over again
- Storing the result for the future

# (Application) Class Data Sharing

- Caches result of class loading
  - C++ objects
  - Writes to file
  - Memory maps into application

# Double Benefit

- Faster startup
- Smaller footprint
  - Pages shared between JVM instances
  - Important so use same Docker base image

# CDS Updated in

- JDK 5
- JDK 9
- JDK 10
- JDK 11
- JDK 12
- JDK 13

# CMS

- Not supported
- Habit to stop



# Application Class Data Sharing

- AKA App CDS
- Extends CDS to application classes and classes from custom class loaders

# -Xshare

- -Xshare:dump creates CDS archive
- -Xshare:off disables CDS
- -Xshare:on require CDS, fail if not possible
- -Xshare:auto use CDS if possible

# Create Shared Archive

10

- `java -Xshare:dump`
  - by default uses `lib/classlist`
  - by default stored to `lib/server/classes.jsa`
- Default archive ships with JDK 12

# Create App CDS Archive Recipe

10

1. Run application with `-Xlog:class+load=debug`
2. Run `cl4cds`
3. Run application with `-Xshare:dump`
4. Repeat every time application or dependency or middleware changes

# Conclusion CDS

- On by default in 11
- Default archive in 12 → used by default
- Very easy to use
- Worst case only start up improvements
- Best case additional footprint reduction
- Create default CDS archive on Java 11 installation or Docker image creation

# Conclusion App CDS

- Biggest improvement
- Hardest to use
- Probably too much of a hassle
- JEP 350 may make it easier to use

# TLSv1.3

# TLSv1.3

11+12

- Safer by
  - removing old cruft
  - adding better things
- Faster by
  - shortening handshake



# Enterprise TLS

- aka ETLS / eTLS
- allows MITM because snake oil  
(mostly US banking industry)

# Java Support for TLSv1.3

11+12

- Java 11 only partial support
- Java 12 added support for
  - ChaCha20
  - Poly1305
- Still missing
  - x25519/x448 elliptic curve algorithms, JDK-8171279
  - EdDSA signature algorithms, JDK-8166596

# Can I use it?

11+12

- who knows

# When will I have to use it?

- who knows

# Java 8 backport

- Zulu has one
  - details hard to come by
  - some classes in Zulu namespace

# Further Readings



**See README.md on GitHub**

<https://github.com/netceteragroup/ncan-java10-11-12>

# FYI

- Java 8 → 11 Migration Experience Reports
- 2019-06-26