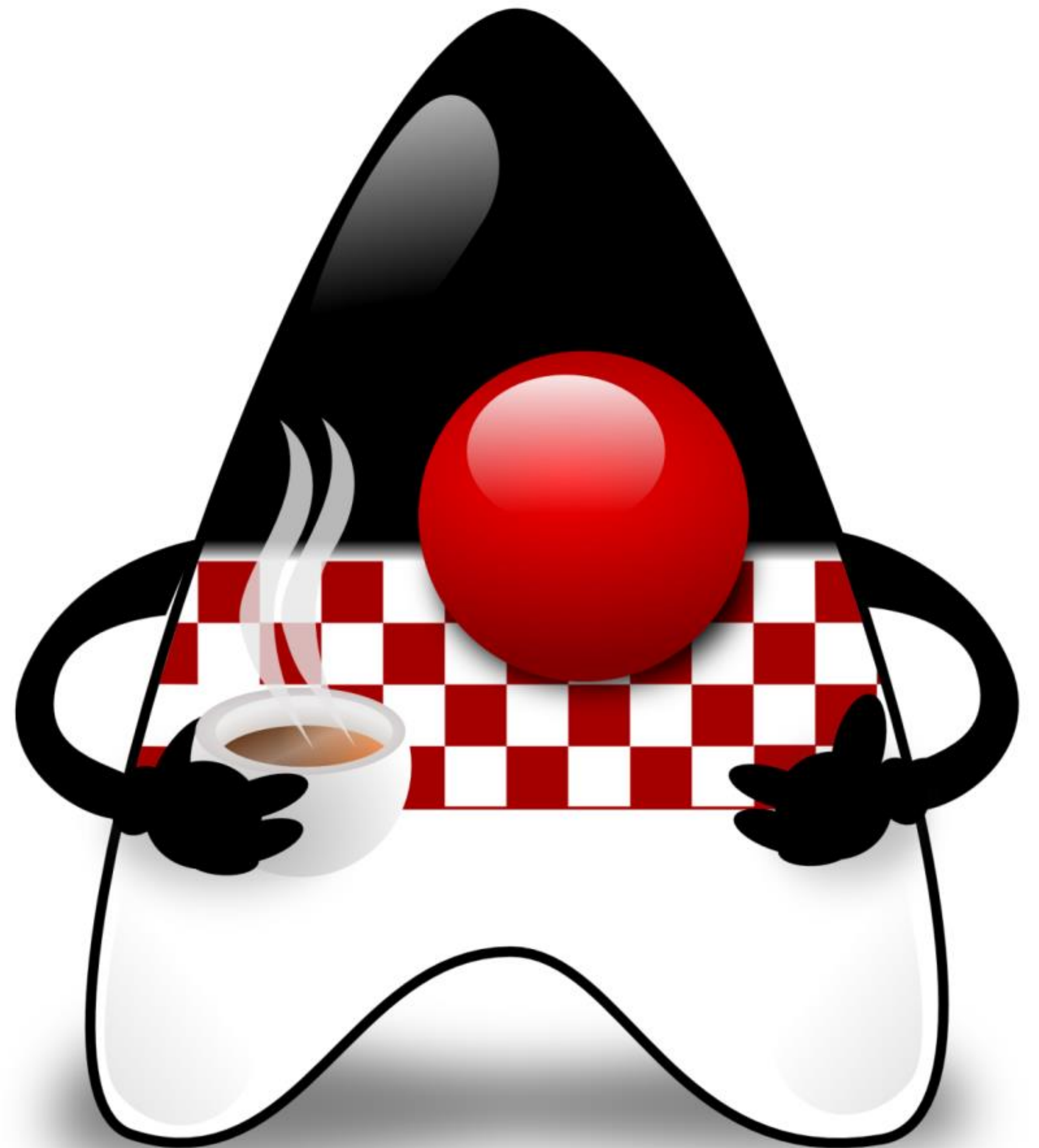
The Best
Java Tools
for Developers
(in 2021)

dr. sc. **Branko Mihaljević**

dr. sc. **Aleksander Radovan**

doc. dr. sc. **Martin Žagar**

HUJAK



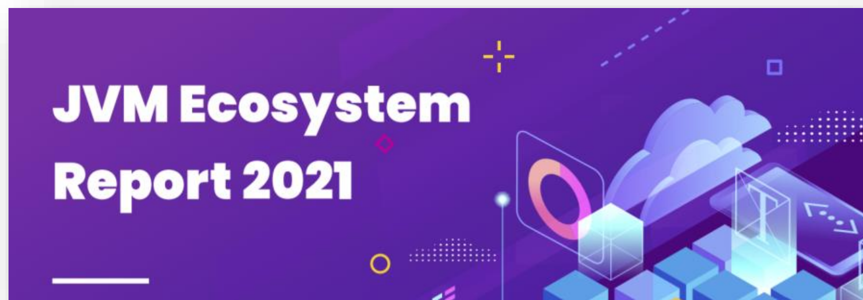
The most used tools in Java?

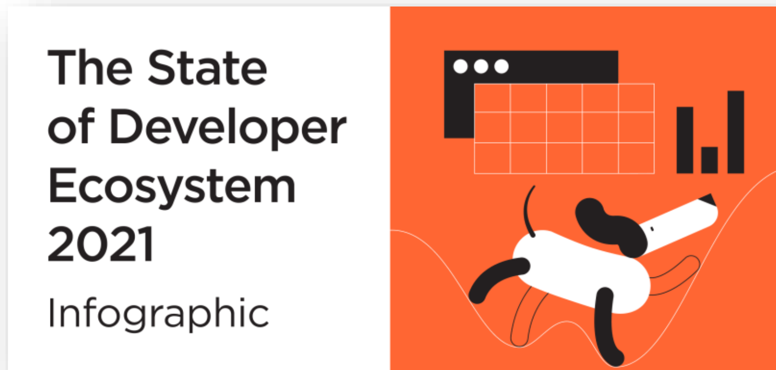
- Are we talking about the "developers tools" or best practices as "tools"?
- Well, both!

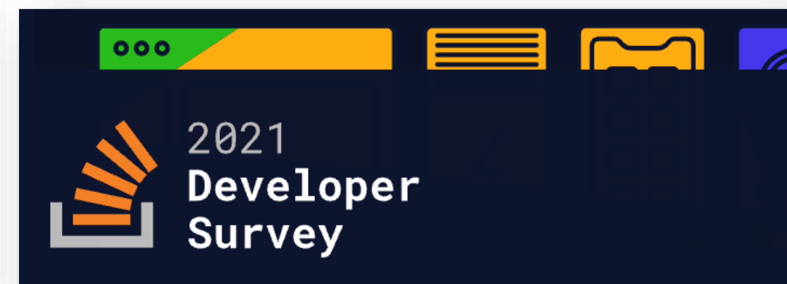
- **Part 1 – Developers Tools for Java** (incl. Platforms and Frameworks)

- **Part 2 – The new "tools" in Java**

What do we most commonly/actually use?







Some Surveys

Several relevant surveys taken into consideration:

- **JVM Ecosystem Report 2021** by **snyk**
  - 2000 responses in February-March 2021
- **2021 Java Developer Productivity Report** by **JRebel**
  - 876 responses August-November 2020
- **The State of Developer Ecosystem 2021** by **JetBrains**
  - 31743 responses
- **2021 Developer Survey** by **StackOverflow**
  - 83052 responses (out of which 58031 professional developers) in May 2021
- **CodingGame Developer Survey 2021**
  - 15000 responses (?)
- **Java Survey 2020** by **HUJAK** ☺
  - 45+ responses in Feb 2020
  - THANK YOU ☺

Versions of Java

- **Java platform versions** used in projects?

Version	HUJAK	snyk prod.	snyk dev.	JRebel	JetBrains
7 or older	32%	5%	3%	15%	8%
8	82%	60%	50%	69%	72%
9	7%	1%	1%		4%
10	7%	7%	1%		2%
11	46%	62%	64%	36%	42%
12	9%	1%	2%		4%
13	7%	3%	3%		4%
14		5%	8%		8%
15		12%	26%		14%
16			4%		
17			3%		

Which JDK?

- Which Java **vendor's JDK** do you currently use in **production**?

	snyk	JRebel
AdoptOpenJDK	44%	22%
Oracle OpenJDK	28%	59%
Oracle JDK	23%	
Azul Zulu	16%	6%
Amazon Corretto	9%	10%
GraalVM	7%	4%

Other JVM languages

- **Other JVM languages** in production?

	snyk 2021
Kotlin	18%
Groovy	13%
Scala	10%
Clojure	8%
JRuby	2%
Other	1%

Application Server

- What **application server** do you use on your main application?

	JRebel	JetBrains
Tomcat	66%	61%
JBoss / Wildfly	19%	9%
WebLogic	18%	5%
Jetty	15%	17%
WebSphere	14%	5%
GlassFish	6%	4%
Other	6%	4%
None		28%

Web Application Frameworks?

- Which **Web** application frameworks do you use?

Framework	JetBrains	JRebel	snyk	HUJAK
Spring Boot	65%	62%	58%	77%
Spring MVC	42%		29%	54%
JSF / Java EE	5%		24%	28%
Dropwizard		9%		
Quarkus	3%	6%	11%	5%
JHipster			3%	2%
Micronaut	2%	4%	5%	
Grails	2%		4%	2%
GWT			4%	5%
Vaadin	2%			9%

Cloud Platforms

- **Cloud platforms and PaaS** used?

	StackOverflow	JRebel
Amazon (AWS)	54%	39%
Google (GCP)	31%	18%
Microsoft (Azure)	31%	24%
Heroku	24%	
Digital Ocean	18%	
IBM (Cloud or Watson)	3%	9%
Oracle (OCI)	1,9%	
SAP Cloud Platform		9%
Pivotal Cloud Foundry		7%

Top IDEs?

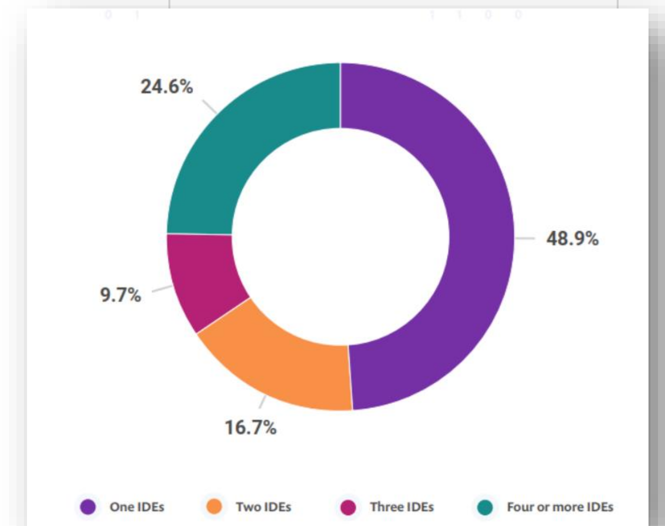
- The **TOP 10 IDE Index** pypl.github.io/IDE.html

Rank	Language	Ratings
1	Visual Studio	31.2%
2	**Eclipse**	**21.9%**
3	**Visual Studio Code**	**10.2%**
4	**IntelliJ**	**9.0%**
…		…
7	**NetBeans**	**3.0%**

IDEs and Editors

- Which **IDEs** or **editors** do you use?

	snyk	JRebel	StackOverflow	JetBrains
IntelliJ IDEA	72%	65%	29%	75%
Eclipse	25%	48%	16%	11%
VS Code	23%	27%	71%	4%
Vi/Vim/Emacs...	14%		24%	
NetBeans	13%	13%	7%	2%
Some Browser-based		4%		
Notepad++			30%	
Sublime			20%	
Some Other	1%	2%		

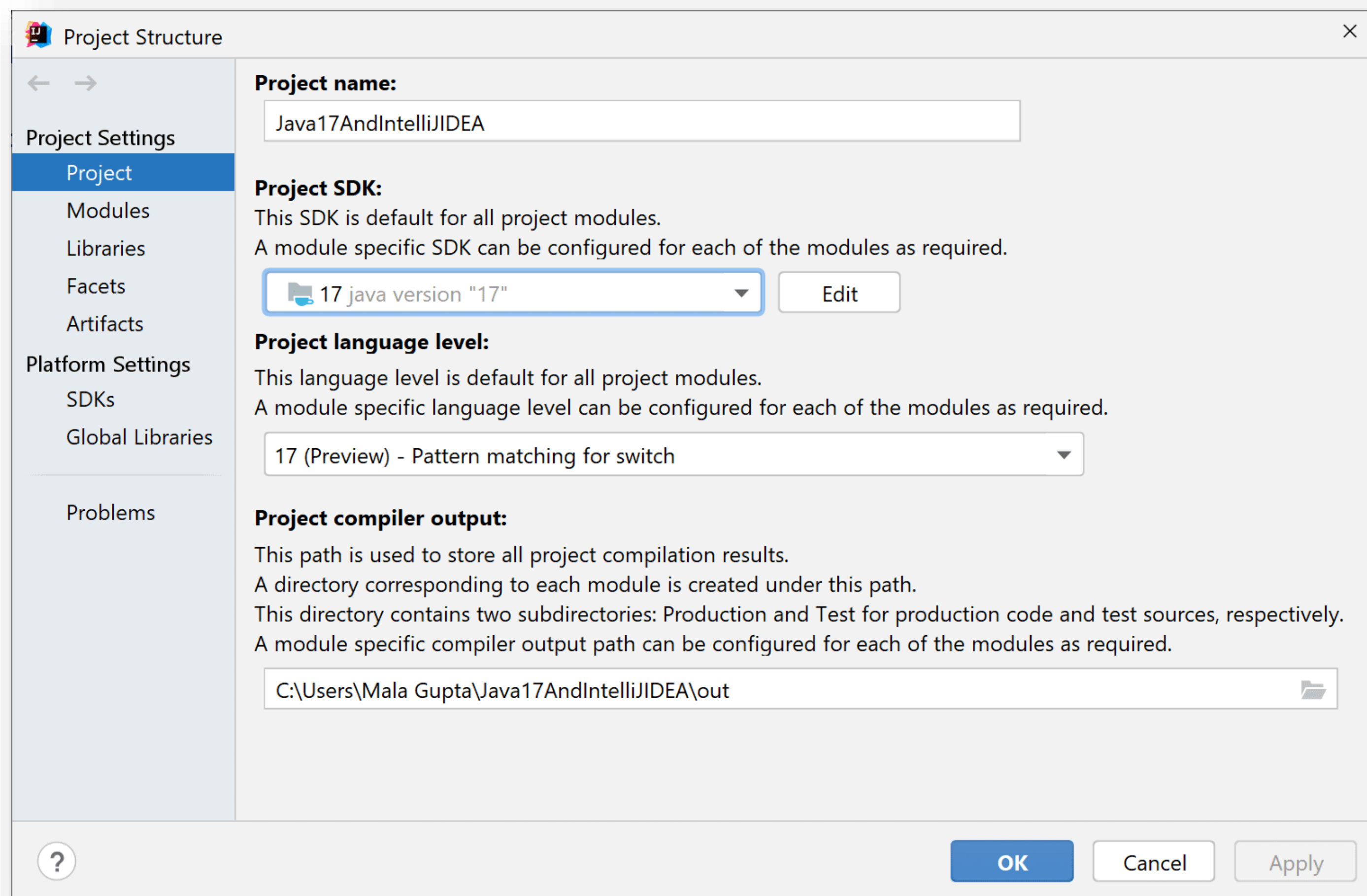


IDEs for JDK 17

- **IntelliJ IDEA 2021.2.1**
  - Supports **Java 1**7 Sealed Types, Always-strict Floating-point Semantics, and Pattern Matching for Switch expressions (Preview)
  - Also supports Java 16 Records, patterns, local enums and interface, and Text Blocks
  - blog.jetbrains.com/idea/2021/09/java-17-and-intellij-idea/
- **Eclipse IDE 2021-09 (4.21)**
  - Supports Java 17, including Pattern Matching for Switch (Preview), Sealed Classes, and more via Eclipse Marketplace
  - Wait for 2021-12 (4.22) for integrated support
- **NetBeans 12.5**
  - With experimental support for JDK 17
  - netbeans.apache.org/download/nb125/index.html
- **Visual Studio Code**

IDEs for JDK 17 – IntelliJ IDE Example

- Test **IntelliJ IDEA 2021.2.1**
- More at blog.jetbrains.com/idea/2021/09/java-17-and-intellij-idea/
- Watch ***Learn Java 17 with IntelliJ IDE*** by Mala Gupta
  - youtu.be/FP0V98S4l9w (31:30)
  - **Pattern matching for instanceof**
  - **Pattern matching for switch**
  - **Sealed classes and interfaces**



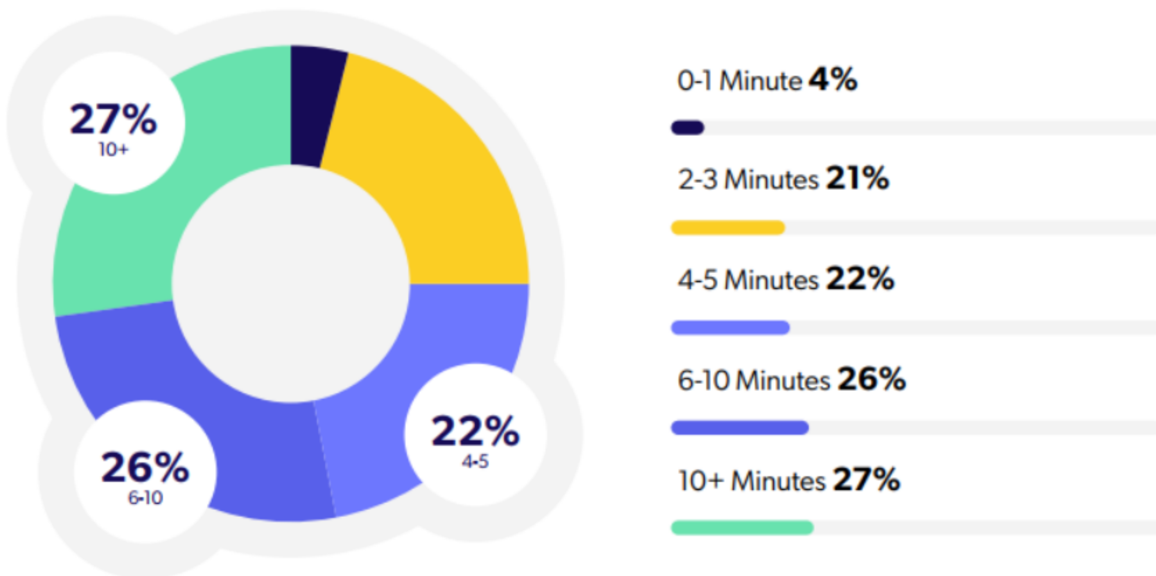
App Building Tools?

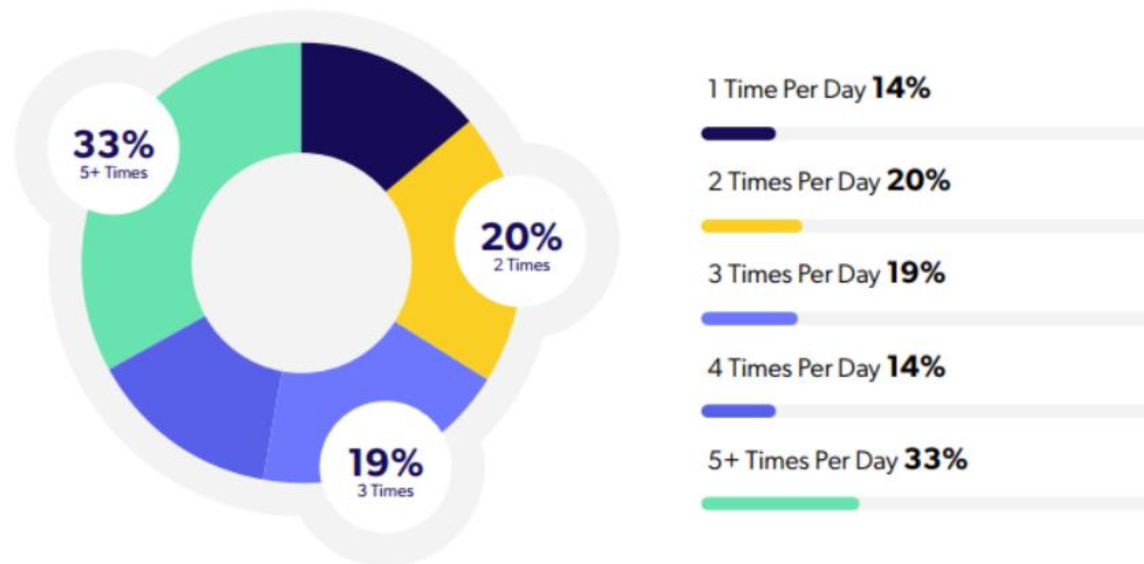
- Tools for **building** applications?

	snyk	JRebel	JetBrains
Maven	76%	67%	72%
Gradle	38%	20%	49%
Ant	12%	11%	8%
Other	6%	2%	1%
None	3%		8%

CI/CD Time and Frequency?

- **CI/CD time** and **commit** frequency?





Unit Testing?

- Which unit-testing frameworks do you use?

Framework	JetBrains
JUnit	85%
Mockito	45%
PowerMock	9%
TestNG	8%
Spock	5%
EasyMock	4%
JMockit	4%
My own tests	4%

JVM Profilers?

- Which **JVM profilers** do you regularly use?

	JetBrains
Integrated IntelliJ	26%
VisualVM	20%
JProfiler	11%
Java Mission Control	10%
NetBeans profiler	4%
YourKit	3%
async-profiler	1%
My own	3%
None	44%

VM platforms?

- Which **VM platform** do you use?

	JRebel
Docker	57%
Kubernetes	42%
VMware	27%
Vagrant	12%
Other	4%
N/A	12%

What if Stuck?

- What do you do when you get **stuck**?

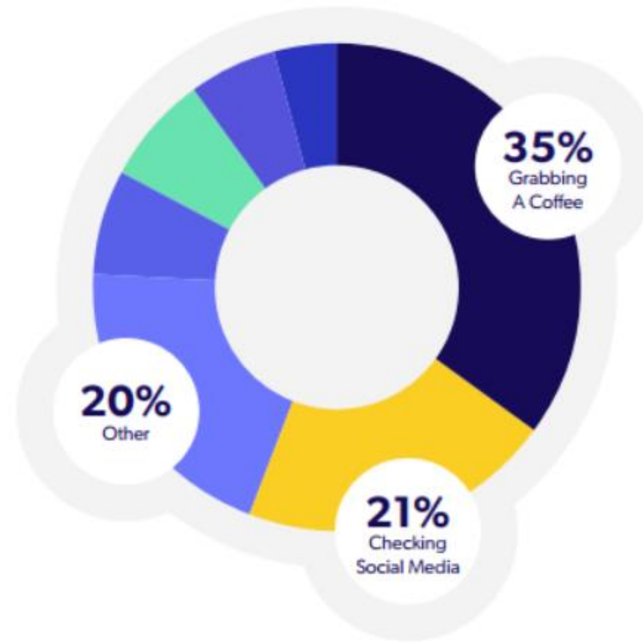
0% 20% 40% 60% 80% 100%

	StackOverflow
Google it	90%
Visit StackOverflow	80%
Do some other work	49%
Read help/manual/tutorial	40%
Call a coworker	46%
Some physical activity	40%
Play a game	12%
Panic	10%
Meditate	10%

While waiting… on redeploy?

- Coffeeeeeeeee!

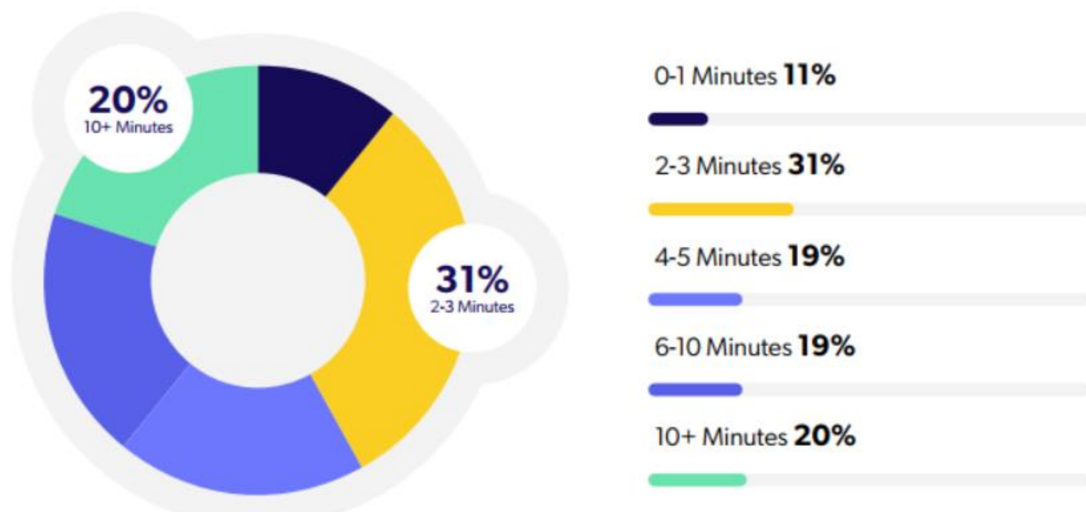
While waiting on a redeploy, what are you typically doing?

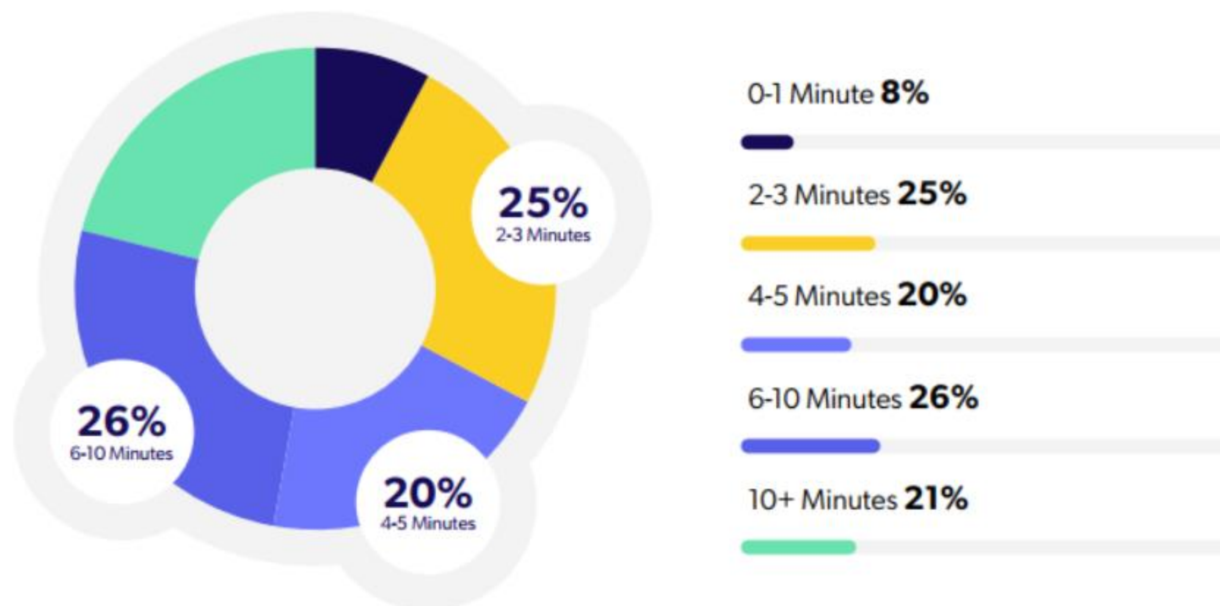


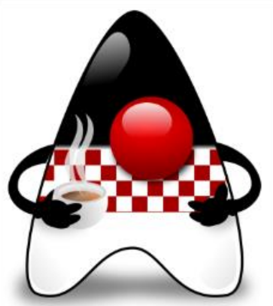
- Grabbing A Coffee **35%**
- Checking Your Twitter/Facebook/Instagram **21%**
- Other **20%**
- Checking In With Family **7%**
- Taking A Nap **7%**
- Walking Dog **6%**
- Playing Video Games **4%**

Time to Redeploy?

- How long to redeploy? (by JRebel)
- Remote redeploy in a container?



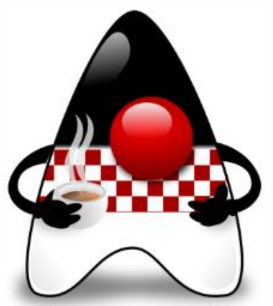




Part 2 – The new "tools" in Java

The Most Important Java Features (11-17)

- **Languages Features** – Records, Switch Expressions, Text Blocks, vars, Pattern Matching for instanceof, Sealed Classes, Pattern Matching for switch…
- **Memory Management** – G1 GC enhancements (Full Parallel), ZGC (with improvements), Shenandoah GC, Elastic Metaspace…
- **Library Enhancements** – Pseudo-Random Generator, Deserealization Filters, Vector API, Foreign Fuction & Memory API…
- **Future Proofing** – Module System (JPMS), Strong Encapsulation for JDK Internals…
- **Easier Debugging** – Flight Recorder, JFT Event Streaming, NullPointerExceptions…
- **Modernizing Infrastructure** – ~~Mercurial~~→Git, GitHub, AArch64 port…
- **Deprecations & Removals** – ~~CMS GC, Nashorn, Biased Locking, RMI Activation, Applet API, Security Manager~~…



Switch Expressions

Switch Expressions

- First Preview as JEP 325 in JDK 12
- **Extend switch** statement so it can be used as a **statement** or an **expression**
  - Both can use either a "traditional" or "simplified" scoping and control flow behavior
- **"Simplified" switch** form with "**case L ->**" to switch labels
  - If a label is matched, then only the expression or statement to the right of an arrow label is executed – there could be no fall through
- Without **fall through**! No need for **break**
- Can also be used as an **expression**?
- All possible values covered (**totality**)?
- Without **default** (**completeness**)?

Switch Expressions (Preview) – Example

```
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        System.out.println(6);
        break;
    case TUESDAY:
        System.out.println(7);
        break;
    case THURSDAY:
    case SATURDAY:
        System.out.println(8);
        break;
    case WEDNESDAY:
        System.out.println(9);
        break;
}
```

```
enum Weekdays { MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
FRIDAY, SATURDAY, SUNDAY }
```

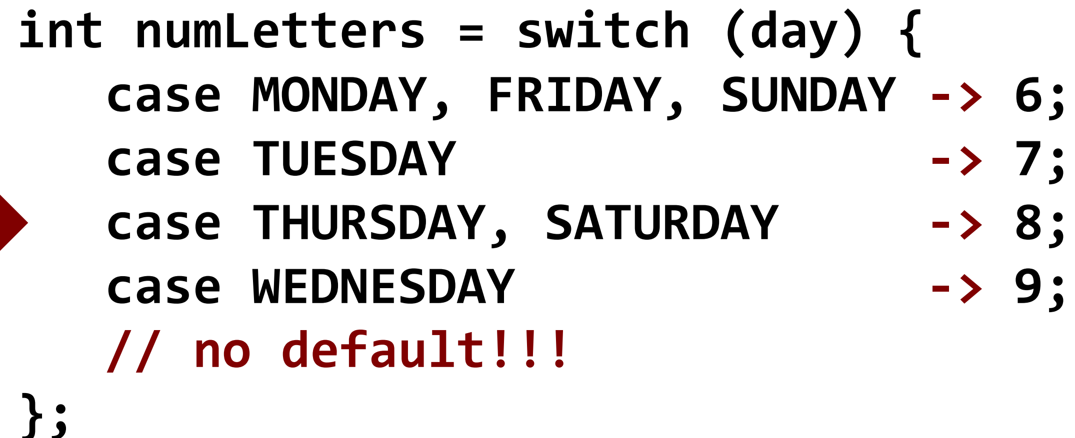
```
switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> System.out.println(6);
    case TUESDAY                -> System.out.println(7);
    case THURSDAY, SATURDAY     -> System.out.println(8);
    case WEDNESDAY              -> System.out.println(9);
}
```

No fall through (no need for break)

Switch Expressions (Preview) – Example #2

```
int numLetters;
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        numLetters = 6;
        break;
    case TUESDAY:
        numLetters = 7;
        break;
    case THURSDAY:
    case SATURDAY:
        numLetters = 8;
        break;
    case WEDNESDAY:
        numLetters = 9;
        break;
    default:
        throw new IllegalStateException("Hmm: " + day);
};
```

```
enum Weekdays { MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
FRIDAY, SATURDAY, SUNDAY }
```



Used as an expression

No fall through

No default needed

Switch Expressions

- Second Preview as JEP 354 in JDK 13
- **Extended switch** so it can be used as either a **statement** or an **expression**
  - Both can use either traditional **case ... :** labels (with fall through) or new **case ... ->** labels (with no fall through)
- Expression must be either:
  - Char, byte, short, int
  - Character, Byte, Short, Integer
  - String
  - enum
- Introduces new statement for **yielding a value** from a switch expression
  - Based on feedback on design and experience in JDK 12 to simplify everyday coding

Switch Expressions (2nd Preview) – Example

- When working switch expression, if a full block is needed, a new **yield statement** is introduced
- It **yields a value** that becomes the value of the enclosing switch expression

```
int j = switch (day) {
    case MONDAY  -> 0;
    case TUESDAY -> 1;
    default      -> {
        int k = day.toString().length();
        int result = f(k);
        yield result;
    }
};
```

Switch Expressions (2nd Preview) – Example

- A switch expression can also use a traditional switch block with **case L:** switch labels (implying fall through semantics)
- In this case, **values are yielded** using the new yield statement

```
int result = switch (s) {
    case "Foo":
        yield 1;
    case "Bar":
        yield 2;
    default:
        System.out.println("Neither Foo nor Bar, hmmm...");
        yield 0;
};
```

Switch Expressions (2nd Preview) – Example

```java
int numLetters;
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        numLetters = 6;
        break;
    case TUESDAY:
        numLetters = 7;
        break;
    case THURSDAY:
    case SATURDAY:
        numLetters = 8;
        break;
    case WEDNESDAY:
        numLetters = 9;
        break;
    default:
        throw new IllegalStateException("Hmm: " + day);
};
```

```java
int numLetters =
switch (day) {
   case MONDAY:
   case FRIDAY:
   case SUNDAY:
      yield 6;
   case TUESDAY
      yield 7;
   case THURSDAY
   case SATURDAY
      yield 8;
   case WEDNESDAY
      yield 9;
   default:
      throw new IllegalStateException(
        "Hmm: " + day);
};
```

Switch Expressions (Standard)

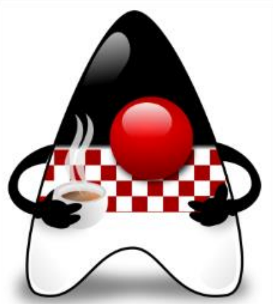
- Standard as JEP 361 in JDK 14
- **Final** version of the extended **switch**

a) Can be used as either a **statement** or an **expression**

b) Can use traditional **case ... :** labels or new **case ... ->** labels (no fall through)

c) Can **yield** **value** from a switch expression

- Additionally, to prepare for the use of **pattern matching** (JEP 305) in switch



Text Blocks

Text Blocks

- *Idea*: Adding **text blocks** to the Java language (using """)
- Relates to JEP 326: Raw String Literals, Preview as JEP 355 (JDK 13), Second Preview as JEP 368 (JDK 14), Standard as JEP 378: Text Blocks (JDK 15) https://openjdk.java.net/jeps/378
- A text block is a **multi-line string literal** that
  - Avoids the need for most escape sequences
  - Automatically formats the string in a predictable way
  - Gives the developer control over format when desired
- Goals
  - Easy to express strings that **span several lines** of source code (avoiding escape seqs)
  - Enhance **readability of non-Java code** embedded in Java programs as strings
  - Support **migration from string literals**

Text Blocks – examples

- **SQL** example using a "two-dimensional" block of text

```
String query = """
               SELECT "EMP_ID", "LAST_NAME" FROM "EMPLOYEE_TB"
               WHERE "CITY" = 'INDIANAPOLIS'
               ORDER BY "EMP_ID", "LAST_NAME";
               """;
```

- **Polyglot language** example using a "two-dimensional" block of text

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("js");
Object obj = engine.eval("""
                         function hello() {
                             print('"Hello, world"');
                         }
                         hello();
                         """);
```

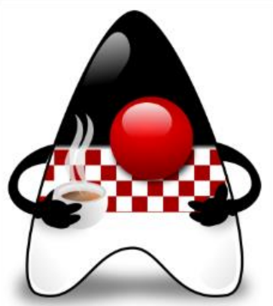
Text Blocks – examples

- **HTML** example using "one-dimensional" string literals

```
String html = "<html>\n" +
       "    <body>\n" +
       "        <p>Hello, world</p>\n" +
       "    </body>\n" +
       "</html>\n";
```

- **HTML** example using a "two-dimensional" block of text

```
String html = """
       <html>
           <body>
               <p>Hello, world</p>
           </body>
       </html>
       """;
```



Records

Records

- Idea: ***Data Classes*** for Java – from project **Amber**
- JEP 395: **Records** (JDK 16) https://openjdk.java.net/jeps/395
  - Preview in JEP 359 (JDK 14), Second Preview in JEP 385 (JDK 15)
- Provide a **compact syntax** for declaring classes which are transparent holder for **shallowly immutable data**
  - **Similar to nominal tuples**
- Eliminate boilerplate code
- Any of the members that are automatically derived from the state description can also be declared explicitly
- Deeper improvement – when deserializing a record, a record **constructor** is called and checks are made (not just forcing values into fields)

Records – Example

- Example: We need a point

```
class Point {

   final double x;
   final double y;

   public Point (double x, double y) {
      this.x = x;
      this.y = y;
   }

   public double x() { return x; }

   public double y() { return y; }
```

```
@Override
public double equals (Object o) {
   if (…)
      …
   return …
}

@Override
Public double hashCode () {
   return …
}

@Override
Public double toString() {
   return …
}
```

Records – Example

- Example: We need a point

```
record Point { }

    final double x;
    final double y;

    public Point (double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double x() { return x; }

    public double y() { return y; }
```

```
@Override
public double equals (Object o) {
    if (…)
        …
    return …
}

@Override
Public double hashCode() {
    return …
}

@Override
Public double toString() {
    return …
}
```

Sometimes data is just … data.

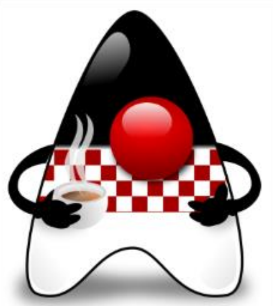
Mark Reinhold

Records – Goals

- An object-oriented construct that expresses a **simple aggregation** of **values**
- Help programmers to focus on **modeling** immutable data rather than extensible **behavior**
- **Automatically implement** data-driven methods (such as *equals* and accessors)
- Preserve principles such as **nominal typing** and **migration compatibility**
- Not declaring a "war on boilerplate" or adding features such as properties or annotation-driven code generation (not streamlining POJOs)
- *Hint:* **Pattern matching for records** comes later

Records – Some Rules

- Some records rules:
  - Records do **not** have an ***extends*** clause – superclass is always java.lang.Record
  - Implicitly ***final*** and **cannot** be ***abstract*** – defined solely by its state description
  - Cannot explicitly declare instance fields and cannot contain instance initializers
  - Implicitly declared fields corresponding to the components of a record class are final and are not modifiable via reflection (immutability)
  - Any explicit declarations of a member (that would otherwise be automatically derived) must match the type of the automatically derived member exactly (disregarding type annotations)
  - A record cannot declare native methods
- More at https://openjdk.java.net/jeps/395
- Records work well with ***sealed types***



Sealed Classes

Sealed Classes

- Idea: ***Sealed Classes and Interfaces*** for Java – from project **Amber**
- JEP 409: **Sealed Classes** (JDK 17) https://openjdk.java.net/jeps/409
  - Preview in JEP 360 (JDK 15), Second Preview in JEP 397 (JDK 16)
- Restrict which classes or interfaces may **extend or implement** this class or interface
- Java already supports ***enum*** *classes* to model the situation where a given class has only a fixed number of instances
- Sometimes we want to model a fixed set of ***kinds* of values** – using a class hierarchy not as a mechanism for code inheritance and reuse but, rather, as a way to list kinds of values

Sealed Classes – Goals

- Allow the author of a class or interface to **control** which code is **responsible** for **implementing** it
- Provide a more **declarative way** than access modifiers to **restrict** the use of a **superclass**
- Support future directions in **pattern matching** by providing a foundation for the exhaustive analysis of patterns
- Not providing new forms of access control (such as "friends") or changing final (in any way)

Sealed Classes – Example

- *Example*: Shape can **only** have subclasses Circle, Rectangle, and Square

```
package com.example.geometry;
public abstract sealed class Shape
    permits Circle, Rectangle, Square {
... }
public final class Circle extends Shape { ... }
```

- The classes specified by permit must be located **near** the superclass
  - In the same **module** (if the superclass is in a named module) or
  - In the same **package** (if the superclass is in the unnamed module)
- *Hint*: When the permitted subclasses are small in size and number, it may be convenient to declare them in the **same source file** as the sealed class

Sealed Classes – Example #2

- *Example*: Sealed class may omit permit if subclasses are defined in same file

```
abstract sealed class Shape { ...
    final class Circle extends Shape { ... }
    final class Rectangle extends Shape { ... }
    final class Square extends Shape { ... }
}
```

- Anonymous classes and local classes cannot be permitted subtypes of a sealed class

Sealed Classes – Constraints

- Every permitted subclass must directly extend the sealed class
- Every permitted subclass must use a modifier to describe how it propagates the sealing initiated by its superclass:
  - **final** – to prevent from being extended further
    - *Note*: Record classes are implicitly declared final
  - **sealed** – to allow to be extended further than envisaged by its sealed superclass, but in a restricted fashion
  - **non-sealed** – being open for extension by unknown subclasses
    - *Note:* The modifier "non-sealed" is the first hyphenated keyword proposed for Java ☺

Sealed Classes – Example #3

- *Example*: Sealed class ma omit permit if subclasses are defined in same file

```
public abstract sealed class Shape
    permits Circle, Rectangle, Square, WeirdShape { ... }

public final class Circle extends Shape { ... }

public sealed class Rectangle extends Shape
    permits TransparentRectangle, FilledRectangle { ... }
public final class TransparentRectangle extends Rectangle { ... }
public final class FilledRectangle extends Rectangle { ... }

public final class Square extends Shape { ... }

public non-sealed class WeirdShape extends Shape { ... }
```

- *Note*: Even though the WeirdShape is open to extension, all instances of those subclasses are also instances of WeirdShape
  - Therefore code written to test whether an instance of Shape is either a Circle, a Rectangle, a Square, or a WeirdShape remains exhaustive.

Sealed Interfaces

- An interface can also be sealed by applying the sealed modifier
- After any extends clause to specify superinterfaces, the implementing classes and subinterfaces are specified with a permits clause
- *Example*: Modeling mathematical expressions as a known set of subclasses

```
package com.example.expression;

public sealed interface Expr
    permits ConstantExpr, PlusExpr, TimesExpr, NegExpr { ... }

public final class ConstantExpr implements Expr { ... }
public final class PlusExpr     implements Expr { ... }
public final class TimesExpr    implements Expr { ... }
public final class NegExpr      implements Expr { ... }
```

Sealing and Records

- Sealed classes work well with record classes and record classes are implicitly final
- *Example*:

```
package com.example.expression;

public sealed interface Expr
    permits ConstantExpr, PlusExpr, TimesExpr, NegExpr { ... }

public record ConstantExpr(int i)       implements Expr { ... }
public record PlusExpr(Expr a, Expr b)  implements Expr { ... }
public record TimesExpr(Expr a, Expr b) implements Expr { ... }
public record NegExpr(Expr e)           implements Expr { ... }
```

- *Note*: This combination of sealed and record classes is known as **algebraic data types**
  - Record classes allow to express product types, and sealed classes allow to express sum types

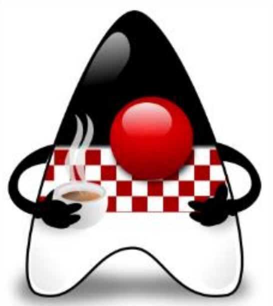
Sealed Classes and Pattern Matching

- A significant benefit of sealed classes in JEP 406: **Pattern Matching for switch (Preview)**, which proposes to extend switch with pattern matching
- Instead of inspecting an instance of a sealed class with if-else chains, we can use a switch enhanced with patterns, and sealed classes will allow the compiler to check that the patterns are **exhaustive**
- Example:

```
Shape rotate(Shape shape, double angle) {
    return switch (shape) {   // pattern matching switch
        case Circle c    -> c;
        case Rectangle r -> shape.rotate(angle);
        case Square s    -> shape.rotate(angle);
    }   // no default needed!
}
```

Sealed Classes

- Making Use of Sealed Classes in Java, Dr. Venkat Subramaniam, at https://www.youtube.com/watch?v=Xkh5sa3vjTE (30:22)



Pattern Matching

Patterns

- A ***pattern*** is a combination of
  1. a ***predicate*** that can be applied to a target, and
  2. a set of ***binding variables*** that are extracted from the target only if the predicate successfully applies to it
- A ***type test pattern*** consists of a **predicate** that specifies a **type**, along with a single binding **variable**

Type Patterns

- Type Pattern is a **test** with **initialization**
- *Example*:

```
if (o instanceof String s) {
    System.out.println("String length: ", s.length());
}
```

- What? – test if the value is a String; if so, initialize s with the String
- Where? – in an *instanceof* expression
- There could be *else* block

Operator instanceof

- Need to enhance the Java programming language with the new version of **instanceof** type comparison operator
- Pattern matching allows common logic in a program, namely the **conditional extraction** of components from objects, to be expressed more concisely and safely
- Motivation:

```
if (obj instanceof String) {
      String s = (String) obj;
      // use s
}
```

Involves:

a) **test** – is *obj* a String?
b) **conversion** – casting *obj* to String
c) **declaration** – of a new local variable *s*

Extending instanceof

- The **instanceof** operator was extended to take a **type test pattern** instead of just a type
- *Example*: the phrase String s is the type test pattern:

```
if (obj instanceof String s) {
     // can use s here
} else {
     // can't use s here
}
```

- The instanceof operator "matches" *obj* to the type test pattern:
- If *obj* is an instance of String, then it is cast to String and assigned to the binding variable *s*
- The binding variable is in scope in the true block of the if statement, and not in the false block of the if statement

Pattern Matching for instanceof

- First preview as JEP 305 in JDK 14, Standard as JEP 375 in JDK 16
- *Example*:

```
if (obj instanceof String s && s.length() > 5) {..
s.contains(..) ..}
```

- Another example:

```
@Override public boolean equals(Object o) {
    return (o instanceof CaseInsensitiveString cis) &&
        cis.s.equalsIgnoreCase(s);
}
```

Pattern Matching for switch (Preview)

- *Idea*: Extend ***pattern matching*** to ***switch*** – from project **Amber**
- Preview as JEP 406: **Pattern Matching for switch** (JDK 17) https://openjdk.java.net/jeps/406
  - Relates to **JEP 405: Record Patterns & Array Patterns** (Preview) (JDK 18)
- Enhance Java with pattern matching for **switch expressions** and **statements**
- Pattern matching on switch allows an expression to be tested against a number of patterns, each with a specific **action**, so that complex data-oriented queries can be expressed concisely and safely
- Allows patterns to appear in **case** labels
- Introduces case **null** statement

Pattern Matching for switch – Example

- *Example*: Testing with else-ifs

```
static String formatter(Object o) {
    String s = "unknown";
    if (o instanceof Integer i) {
        s = String.format("int %d", i);
    } else if (o instanceof Long l) {
        s = String.format("long %d", l);
    } else if (o instanceof Double d) {
        s = String.format("double %f", d);
    } else if (o instanceof String s) {
        s = String.format("String %s", s);
    }
    return s;
}
```

Pattern Matching for switch – Example

- *Example*: Testing with switch

```
static String formatterPatternSwitch(Object o) {
    return switch (o) {
        case Integer i -> String.format("int %d", i);
        case Long l    -> String.format("long %d", l);
        case Double d  -> String.format("double %f", d);
        case String s  -> String.format("String %s", s);
        default        -> o.toString();
    };
}
```

- *Note:* Switch can be of any type – no type restriction anymore

Pattern Matching for switch – Example #2

- *Example*: Testing for ***null*** to avoid NullPointerException

```
static String formatterPatternSwitch(Object o) {
    if (o == null) {
        return "oops";
    }
    return switch (o) {
        case Integer i -> String.format("int %d", i);
        case Long l    -> String.format("long %d", l);
        case Double d  -> String.format("double %f", d);
        case String s  -> String.format("String %s", s);
        default        -> o.toString();
    };
}
```

- This could be resolved in a better way

Pattern Matching for switch – Example #2

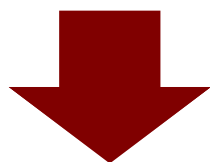
- *Example*: Testing for ***null*** in the switch

```
static String formatterPatternSwitch(Object o) {
    return switch (o) {
        case null      -> "null";
        case Integer i -> String.format("int %d", i);
        case Long l    -> String.format("long %d", l);
        case Double d  -> String.format("double %f", d);
        case String s  -> String.format("String %s", s);
        default        -> o.toString();
    };
}
```

Pattern Matching for switch – Example #2

- Various ways to use **null**
- *Example*: Testing for ***null*** or String in the switch

```
switch (o) {
    case null       -> ...
    case String s   -> ...
    ...
}
```



- is the same as

```
switch (o) {
    case null, String s -> ... // null or String
    ...
}
```

Guarded Patterns

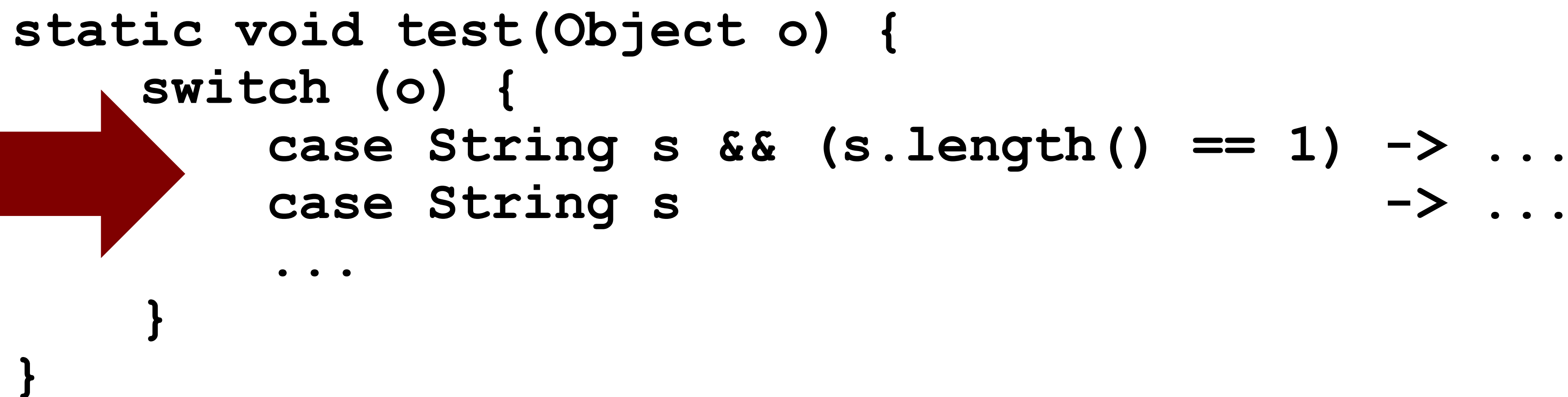
- Rather than add another special case label, we enhance the pattern language by adding **guarded patterns**
- *Example*: Test that *o* is a String of length 1 (or not)

```
static void test(Object o) {
    switch (o) {
        case String s:
            if (s.length() == 1) { ... }
            else { ... }
            break;
        ...
    }
}
```

- The test that *o* is a String of length 1 is split between **case** label and **if** statement

Guarded Patterns – Solution

- After a successful pattern match we often further test the result of the match
- *Example*:



```
static void test(Object o) {
    switch (o) {
        case String s && (s.length() == 1) -> ...
        case String s                      -> ...
        ...
    }
}
```

- The first case matches if *o* is both a String and of length 1
- The second case matches if *o* is a String of some other length

Guarded Patterns – Issue

- Possible issue – **Dominance of pattern labels** where **order matters**
- *Example*:

```
static void test(Object o) {
    switch (o) {
        case String s                            -> ...
        case String s && (s.length() == 1) -> ...
        ...
    }
}
```

- The second label is **never reached** – second pattern is **dominated** by the first pattern

Pattern Matching for switch – Issues

- Possible issue – **Dominance of pattern labels** where **order matters**
- *Example*:

```
static void test(Object o) {
    switch (o) {
        case CharSequence cs -> ...
        case String s -> ...
        ...
    }
}
```

- The string *s* is **never reached** – pattern is dominated by *cs* pattern

Mixing patterns and constants

- We can **mix** **constants** and **patterns**
- *Example*:

```
static void test(Object o) {
    switch (o) {
        case "Hello" -> ...
        case String s -> ...
        ...
    }
}
```

- However, if the order is s first, "Hello" second, the string "Hello" is **never reached** – pattern is **dominated** by *s* pattern

Be Careful with Totality

- *Example*:

```
static int test(Object o) {
    return switch (o) {
        case String s -> s.length();
        case Integer i -> i;
        default -> 0; // we need totality!
    }
}
```

Be Careful with Totality

- *Example*:

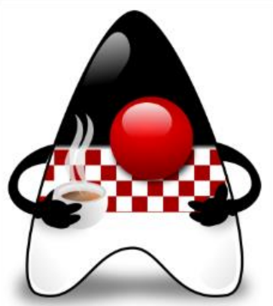
```
Object o = ...
switch (o) {
    case String s: System.out.println(s); break;
    case Integer i: System.out.println("Integer"); break;
    default: break; // we need totality!
     }
}
```

Record Patterns

- Take a look at JEP 405 : Record Patterns & Array Patterns (Preview) in JDK 18 https://openjdk.java.net/jeps/405
- Record pattern tests if the target is of certain record type, and if so, pull apart the structure

Pattern Matching in Java 17

- Pattern Matching in Java 17 and Beyond, Nicola Parlog, https://www.youtube.com/watch?v=UlFFKkq6fyU (27:01)
- Java Language Futures: Late 2021 Edition, Gavin Bierman, https://www.youtube.com/watch?v=hDV6G1MbUH8 (31:11)



Hidden Classes

Hidden Classes

- *Idea*: Classes that **cannot be used directly** by the bytecode of other classes
- Standard as JEP 371: Hidden Classes (JDK 15)
- Intended for use by **frameworks** that generate classes at run time and use them indirectly, **only via reflection**
  - May be defined as a member of an access control nest
  - May be unloaded independently of other classes
- Whereas a normal class is created by invoking ClassLoader::defineClass, a hidden class is created by invoking Lookup::defineHiddenClass
  - Causes the JVM to derive a hidden class from the supplied bytes, link the hidden class, and return a lookup object that provides reflective access to the hidden class

Thank you & greetings from HUJAK!

- Web page **hujak.hr**
  - www.hujak.hr
- LinkedIn group **HUJAK**

  - www.linkedin.com/groups?gid=4320174
- Facebook group page **HUJAK.hr**

  - www.facebook.com/HUJAK.hr
- Twitter profile **@HUJAK_hr**

  - twitter.com/HUJAK_hr