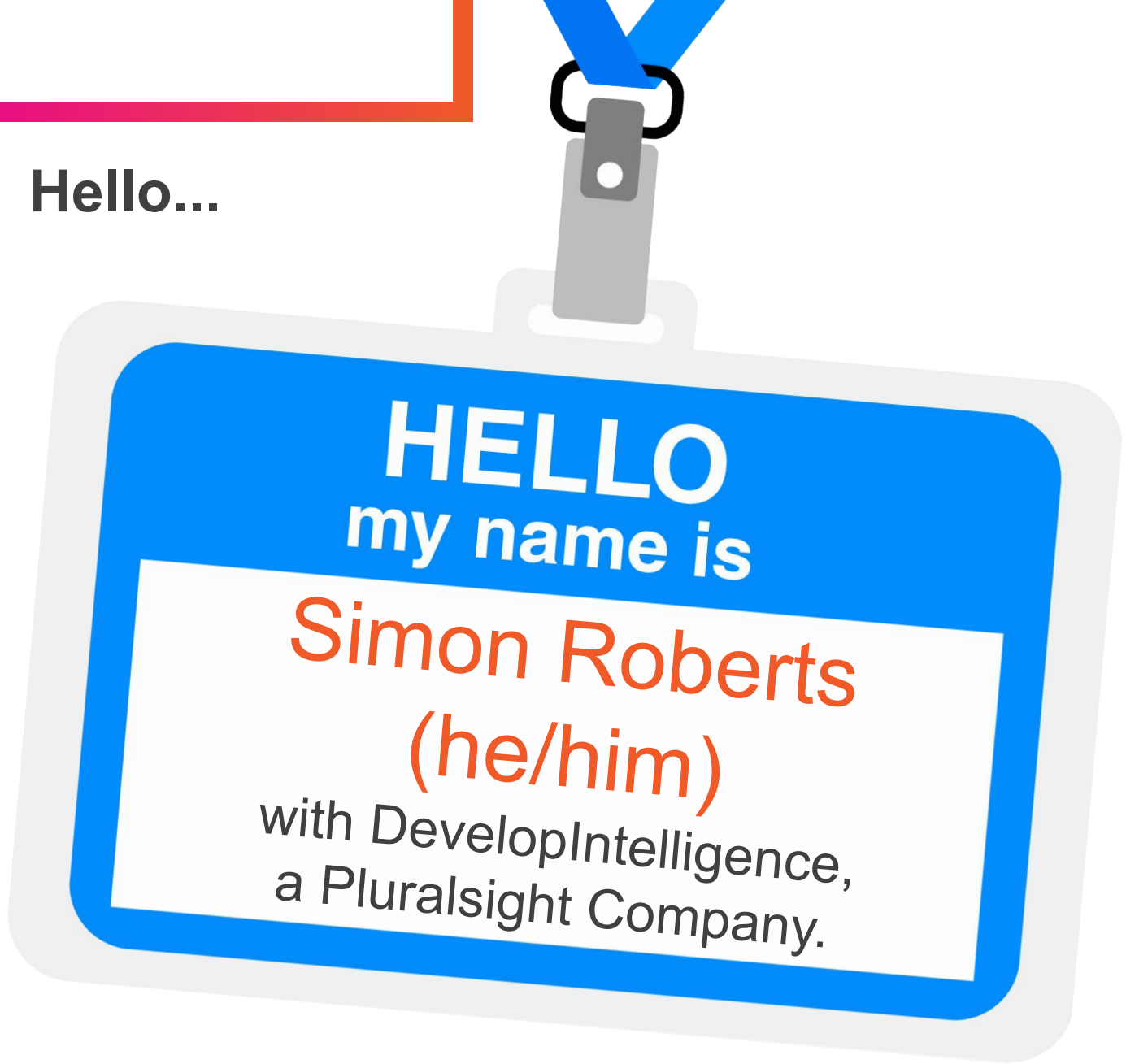
Welcome

Advanced Java TT New Features II

DevelopIntelligence

A PLURALSIGHT COMPANY

Hello...



About me...

We teach over 400 technology topics

You experience our impact on a daily basis!

Prerequisites

This course assumes you

- Solid understanding of the Java programming language to Java 11

Why study this subject?

- Over time Java incorporates new features that improve the expressiveness of the language. Learning to use these features will improve your ability to create robust, maintainable code.

My pledge to you

I will...

- Make this interactive
- Ask you questions
- Ensure everyone can speak
- Create an inclusive learning environment
- Use an on-screen timer for breaks

…also, if you have an accessibility need, please let me know

Objectives

At the end of this course you will be able to:

- Write code using switch expressions
- Create multi-line text blocks
- Write code using the pattern matching feature for instanceof
- Describe the benefits of, and write code using, sealed classes
- Implement records and correlate them with equivalent class declarations
- Explain the significance of value types, and why warnings are needed for certain uses of core Java types

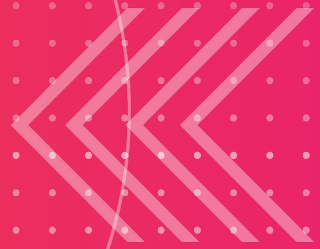
How we’re going to work together

- Discussions, whiteboard diagrams
- Code examples
- You'll have a copy of all the course materials in github
    - Please note, the git repository will be deleted–clone it if you want it!

Student Introductions

- Job title?
- Where are you based?
- Experience with Java
- Fun fact?

Thank you!



DevelopIntelligence
A PLURALSIGHT COMPANY

Improved `switch` syntax

Develop Intelligence

Java 14 introduces a new case label format using an arrow instead of a colon:

```
case 0 ->
```

- The right hand side of this must be an expression, a block, or a `throw`
- This rule ensures that it's only possible to declare variable in a block, which forces a more reasonable scope for such variables
- This form does not "fall through", and does not need or use `break`
- Multiple labels are provided as a comma-separated list

```
case 1, 2, 3 ->
```

`switch` expressions

Java 14 introduces an expression form of `switch` which yields a value rather than simply having side-effects

If `switch` is used where an expression value is needed, then it becomes an expression

- for example:
  - value to be assigned in an assignment
  - operand in another expression
  - actual parameter to method invocation
  - return value from a method

A `switch` expression must produce a value (or throw an exception) for all possible input values, otherwise a compiler error occurs

Producing a value from a `switch` expression

If a `switch` uses the arrow label form:

- an expression to the right of the arrow is the value produced by that case
- a block to the right of the arrow produces the value to the right of a `yield` statement

If a switch uses the colon label form:

- produces the value to the right of a `yield` statement

In the colon label form, fall through still occurs, even with the expression form of `switch`

Text blocks

Develop Intelligence

Java 15 added multi-line string literals, referred to as "text blocks"

- these are not "raw" strings, some processing is applied
- surround text with triple-quotes
  - opening quotes must be at the end of the line that contains them (except for whitespace)
  - closing quotes may be on the end of a line of text
  - if closing quotes on a line by themselves produce a string that ends with a newline character
- consistent leading whitespace is removed

Text block compiler processing

Text blocks span multiple lines, but all newlines are normalized to `\u000A` regardless of the source platform

- the escape literals `\n`, `\r` are not affected by this

Leading and trailing whitespace on each line is cleaned up

- leading whitespace is that which is present in every line (though blank lines other than the last are ignored for this)
- `String.indent(int n)` can insert spaces at the start of every line if desired

Text block compiler processing

Develop Intelligence

Escape sequences (e.g. `\n`) are interpreted

- consequently `\` should be represented as `\\` as with regular string literals
- a single `\` can be used at the end of the line to indicate that no newline is desired (as with Unix shell scripts)
- an escaped double quote does not form part of the closing `"""` delimiter
  - this can be used to assemble three or more double quotes in succession that are intended to be part of the string literal, rather than to terminate it

Pattern matching with `instanceof`

Develop Intelligence

Java 16 provides pattern matching with `instanceof`

- this simplifies the situation of "test variable for type, and then cast" into a single operation

If expressions with instance of can now create a context-sensitive new, initialized, variable

```
if (o instanceof String s) {
   out.println(s.length());
}
```

Extended boolean logic

Develop Intelligence

The scope of a variable created by pattern matching is carefully computed

```
if (!(o instanceof String s)) { /* s NOT in scope */}
else { System.out.println(s.length()); }
```

and short-circuit conditional expressions work properly here too

```
if (o instanceof String s && s.length() > 3) {
 System.out.println("it's a long string!");
}
```

this fails, s is not in scope after the `||` operator

```
if (o instanceof String s || s.length() > 3)
```

Extended boolean logic

Develop Intelligence

The boolean logic approach is not restricted to an `if` condition, but is valid in its own right

```
public boolean equals(Object obj) {
 return obj instanceof Customer c
      && this.name == c.name
      && this.credit == c.credit;
}
```

Pattern matching with `switch`

Develop Intelligence

Java 17 gives a preview of pattern matching with `switch`

```
switch (o) {
   case Integer i -> System.out.println("Integer " + i);
   case String s -> System.out.println("String " + s);
   default -> System.out.println("something else");
}
```

The type for the `switch` in this form *cannot* be primitive, but can be any object or array type

Guard conditions with `switch` patterns

Develop Intelligence

Switch patterns allow a boolean test to form a "guard" condition

```
switch (o) {
   case Integer i  -> System.out.println("Integer " + i);
   case String s && s.length() > 3 ->
        System.out.println("it's a longish String " + s);
   case String s -> System.out.println("String " + s);
   default -> System.out.println("something else");
}
```

Guard conditions with `switch` patterns

More general cases should follow more specific cases

- if two cases with guards both match, only the first will be applied
- if two cases are such that one can be shown to completely include the entirety of the other, the more specific *must* come after the more general
  - this is typically one unguarded match by type and one guarded version
  - the more general is said to "dominate" the less general version

Using null in `switch` patterns

Develop Intelligence

In the `switch` pattern form, a `null` value is allowed

- previously a `null` pointer in the `switch` expression immediately caused a `NullPointerException`

```
switch (o) {
  case null -> System.out.println("it's a null!");
  case String s -> System.out.println("String " + s);
  default -> System.out.println("something else");
}
```

Colon form switch with patterns and guards

Develop Intelligence

The colon label form of `switch` can also use patterns and guards

- in this situation, fall-through *to a pattern* is prohibited and will cause compilation failure

All the `break` statements in this (partial) example are essential

```
switch (o) {
  case null:
    System.out.println("it's a null!"); break;
  case Integer i:
    System.out.println("it's an integer " + i); break;
  case String s && s.length() > 3:
    System.out.println("it's a longish String " + s);
```

Scope of pattern variables

Develop Intelligence

Pattern variables have a scope limited to their particular case

- even in the situation that fall through occurs
  - remember that fall-through cannot happen to another pattern

Record types

Java 16 introduces record types. These represent "data carrier" types

```
public record Customer(String name, long credit) {}
```

This example provides:

- `private final` fields `name` and `credit`
- simple accessor methods for `name()` and `credit()`
  - but no mutators; name and credit are `final` fields
- a constructor with `name` and `credit` arguments
- `equals` and `hashCode` methods
  - `equals` returns true if all fields pass the `equals(Object o)` test
- a `toString` method
- Immutability is intended for these types, but is impossible to enforce in Java as fields themselves can be mutable types

Specializing record types

Develop Intelligence

Records can be specialized with

- `static` fields
- `static` initializers
- user defined methods, instance or `static`
- modifications to the canonical accessor methods
  - and can use `@Override` for these methods

The parent class of a record is always `java.lang.Record`

- they cannot carry an `extends` clause
- however, a record can implement interfaces in the normal way

Constructors for record types

Develop Intelligence

Records permit user-provided code to be executed prior to running the canonical constructor

- This has a special syntax that does not declare arguments

```
public record Customer(String name, long credit) {
 public Customer { ...
```

- the implicit fields of the record cannot be assigned in this constructor, but the *actual parameters* can be modified
- the user-provided code runs before the canonical constructor
- this can be useful for validation and normalization

User-defined constructors can also be added, but they must delegate to the canonical constructor using `this(...)` calls

More rules for record types

Develop Intelligence

Record types

- are implicitly `final` (and cannot be abstract)
- cannot have additional instance fields, nor instance initializers
- can be top-level, nested, local, and generic
- can implement a sealed interface
- can contain nested types
- cannot declare native methods
- can implement `java.io.Serializable`
- but the serialization mechanism cannot be customized
- a nested class is implicitly static

The need for sealed types

Develop Intelligence

Java 17 adds sealed classes and interfaces to the language

- Prior to sealed types, a class could only control its subtypes in limited ways
  - use `final` to prevent subclassing entirely
  - declare only `private` constructors, restricting subtypes to nested types
  - use default access to restrict subtypes to existing in the same package

A **sealed** type has the effect of controlling the possible subtypes in a manner analogous to how an `enum` controls the possible instances of that type

- this improves over package access, since with a sealed class the base type can be `public`, and therefore fully accessible in client code

Sealed types are a good addition to pattern matching in `switch` statements, since they make it possible to know that all possible matches have been listed

Sealed types

Develop Intelligence

A sealed type can control what subtypes can be created directly

- All subtypes must be either in the same module, or in the same package if non-modular
- a `permits` clause enumerates all the types that can be subtypes
  - if all subtypes are declared in the same compilation unit as the parent class, the `permits` clause can be omitted
  - all permitted types must be *immediate* subtypes of the parent
- subtypes *must* be marked `final`, or `sealed`, or `non-sealed`
  - or be enum or record (both of which control their own subclassing)
- a `sealed` type *must* have subclasses
- a non-sealed type can have arbitrary subtypes, so "breaks open" the hierarchy again

Sealed class example

Develop Intelligence

The parent:

```
public sealed class Kingdom
   extends Object implements Serializable
   permits Animal, Plant/*, Fungi, Protist, Monera*/ {
}
```

Note that `permits` follows any `extends` and/or `implements` clauses

An example child:

```
public final class Animal extends Kingdom {
}
```

Additional notes for sealed types

Develop Intelligence

Subtypes of `sealed` types must be exactly *one of* `final`, `sealed`, or `non-sealed` or must be either an enum or record type

A `permits` clause must not be empty (a `final` class is equivalent to a `sealed` class with no permitted children)

`sealed` and `non-sealed` (but not `final`) types may be `abstract`

Subtypes do not have to be equally accessible as their parents, which might prevent exhaustive switching with patterns

`instanceof` tests that represent situations that are impossible within the rules of sealed types will cause compilation errors