

Advanced Java T.T.

New Features II



Simon Roberts





WELCOME



Mark Your Attendance:

- Go to the Course Calendar Invite
- Click on the Course Event Link
- Press Check-In



Join Us in Making Learning Technology Easier



Our mission...

Over 16 years ago, we embarked on a journey to improve the world by making learning technology easy and accessible to everyone.



...impacts everyone daily.

And it's working. Today, we're known for delivering customized tech learning programs that drive innovation and transform organizations.

In fact, when you talk on the phone, watch a movie, connect with friends on social media, drive a car, fly on a plane, shop online, and order a latte with your mobile app, you are experiencing the impact of our solutions.

Over The Past Few Decades, We've Provided

Over
62,300,000
expert-led learning hours

In 2019 Alone, We Provided





Upskilling and Reskilling Offerings



Intimately customized learning experiences just for your teams.



Workshop

2-3 day upskilling experiences



Fast Track

5-day reskilling experiences



Learning Spike

1-day technology overviews



Target Topics

90-minute instructor-led micro-learnings



Hack-a-thon

Learn and build an MVP in 2-3 days

BACK END DEVELOPMENT

BIG DATA

CLOUD COMPUTING

DEVOPS

FRONT END DEVELOPMENT

MACHINE LEARNING

MOBILE APP DEVELOPMENT

SOFTWARE ENGINEERING

SYSTEM ADMINISTRATION



Jenkins



akka



ANGULAR



ANSIBLE



APACHE SPARK



Azure



cassandra



CHEF



docker



GO



Google Cloud



GraphQL

GraphQL



iOS



java



JS



kafka



Kubernetes



mongoDB



MySQL



node



puppet



python



R



React



React Native

React Native



redis



Scala



spring

spring



Swift



Kotlin



TypeScript



TS



Vue.js

AND MANY OTHER TRENDING TECHNOLOGIES



World Class Practitioners





Recording Policy



Recordings are provided to participants who have attended the training, in its entirety. Recordings are provided as a value-add to your training, and should not be utilized as a replacement to the classroom experience.

Participants can expect the following:

- Recordings will be provided the Monday after class is completed
- Recordings will be provided for 14 days
- Recordings will be accessible as “View Only” status and cannot be copied
- Sharing recordings is strictly prohibited

To request recordings, please fill out the form linked in Learn++.

Thank you for your understanding and adhering to the policy.



Note About Virtual Trainings



What we want



...what we've got



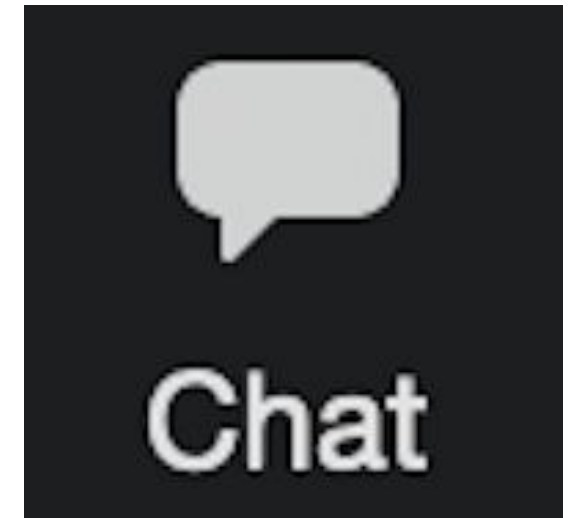
Virtual Training Expectations for You



Arrive on time / return on time



Mute unless speaking



Use chat or ask questions verbally



Virtual Training Expectations for Me



I pledge to:

- Make this as interesting and interactive as possible
- Ask questions in order to stimulate discussion
- Use whatever resources I have at hand to explain the material
- Try my best to manage verbal responses so that everyone who wants to speak can do so
- Use an on-screen timer for breaks so you know when to be back



Prerequisites



- Solid understanding of the Java programming language to Java 11



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



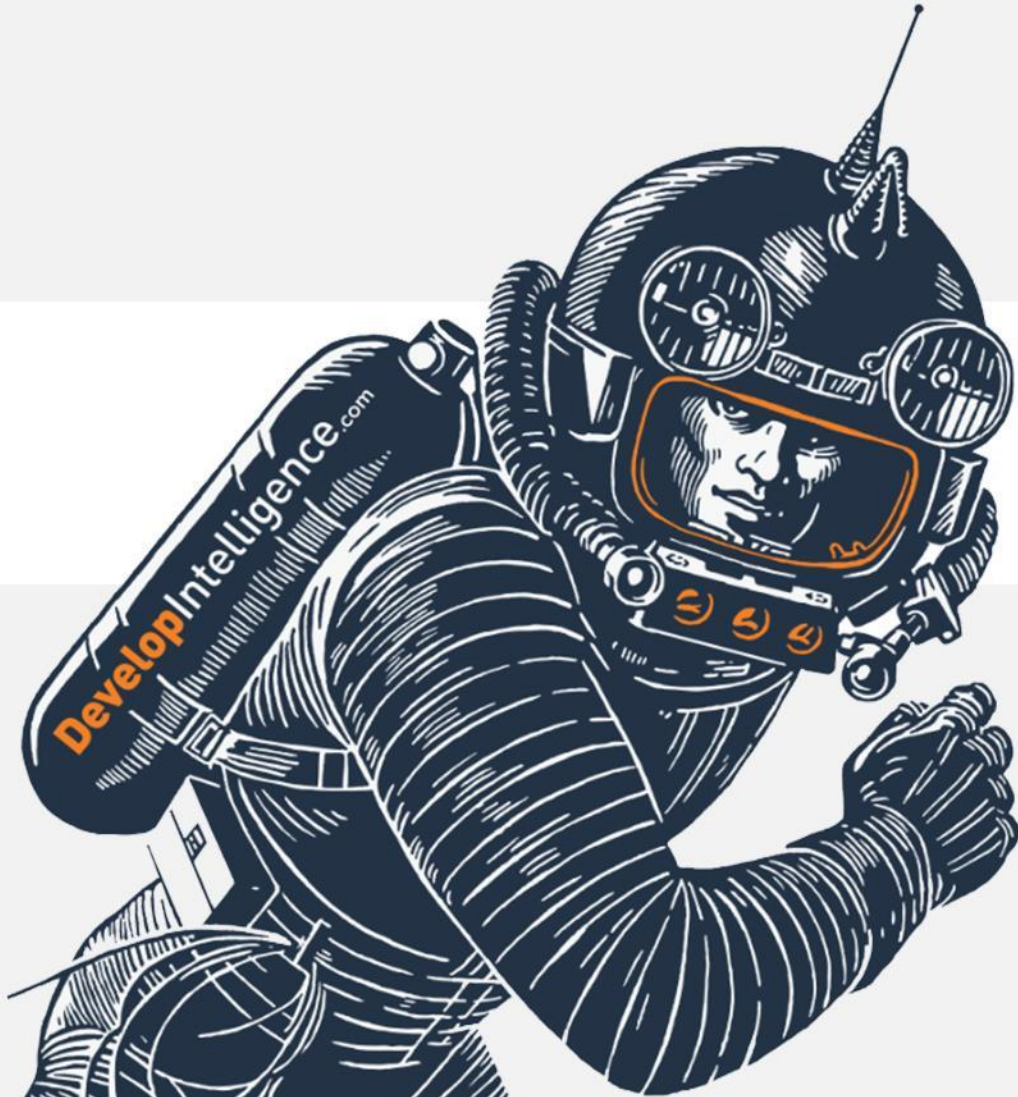
About you



In 90 seconds!

- What you hope to learn
- What your background level is

THANK YOU





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 ->`



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`



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 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



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`

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());  
}
```




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)
```



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;  
}
```



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

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



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

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

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



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



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



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



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

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



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`
- a `sealed` type *must* have subclasses
- a non-sealed type can have arbitrary subtypes, so "breaks open" the hierarchy again



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 {
}
```

Subtypes of sealed types must be exactly *one of* `final`, `sealed`, or `non-sealed`

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