Nesting classes (or types) within another class (or t

A class can contain other types within the class body, such as other classes, interfaces, enums, and records.

These are called nested types, or nested classes.

You might want to use nested classes when your classes are tightly coupled, meaning their functionality is interwoven.

Nested Classes

The four different types of nested classes you can use in Java are: the static nested class, the inner class, a local class and an anonymous class.

Type	**Description**
static nested class	declared in class body. Much like a static field, access to this class is through the Class name identifier.
instance or inner class	declared in class body. This type of class can only be accessed through an instance of the outer class.
local class	declared within a method body.
anonymous class	unnamed class, declared and instantiated in same statement.

Important Restrictions for nested classes were removed in JDK

Before JDK16, only static nested classes were allowed to have static methods.

As of JDK16, all four types of nested classes can have static members of any type, including static methods.

In the next video, I'll start out talking about static nested classes.

There's a lot to cover, so let's get started.

Static Nested Class

The static nested class is a class enclosed in the structure of another class, declared as static.

This means the class, if accessed externally, requires the outer class name as part of the qualifying name.

This class has the advantage of being able to access private attributes on the outer class.

The enclosing class can access any attributes on the static nested class, including private attributes.

Inner Classes

Inner classes are non-static classes, declared on an enclosing class at the member level.

Inner classes can have any of the four valid access modifiers.

An inner class has access to instance members, including private members of the enclosing class.

Instantiating an inner class from external code is a bit tricky. I'll cover that shortly.

As of JDK16, static members of all types are supported on inner classes.

Inner Classes

To create an instance of an inner class, you first need to have an instance of the Enclosing Class.

From that instance you call .new, followed by the inner class name and the parentheses, taking any constructor arguments.

This definitely looks strange the first time you see it.

```
EnclosingClass outerClass = new EnclosingClass();
EnclosingClass.InnerClass innerClass = outerClass.new InnerClass();
```

Bills Burgers with Inner Classes

Create another inner class, called Burger.

This should be a specialized Item, and should also include a list of toppings, also Items.

Remember Items have a name, type, price, and methods to convert prices.

Allow a user to add toppings using the Meal class, which it should then delegate to its burger class.

Allow toppings to be added with a method that allows for a variable number of Strings to be entered, representing the toppings selected.

Allow toppings to be priced differently, some are free, some have an additional cost.

Print the toppings out along with the burger information.

Local Classes

Local classes are inner classes, but declared directly in a code block, usually a method body.

Because of that, **they don't have access modifiers** and are only accessible in that method body while it's executing.

Like an inner class, they have access to all fields and methods on the enclosing class.

They can also access local variables and method arguments, that are final or effectively final.

Local Class's 'Captured Variables'

When you create an instance of a local class, referenced variables used in the class, from the enclosing code, are 'captured'.

This means a copy is made of them, and the copy is stored with the instance.

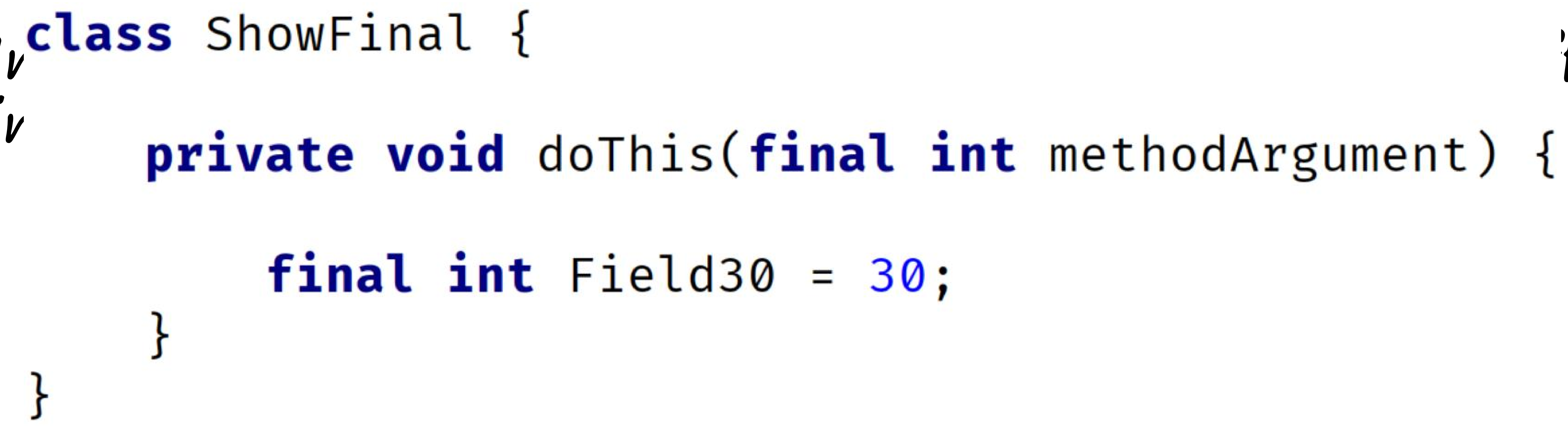
This is done because the instance is stored in a different memory area, than the local variables in the method.

For this reason, if a local class uses local variables, or method arguments, from the enclosing code, these must be final or effectively final.

Final Variables and Effectively Final

The code sample on this slide shows:

- A method parameter, called methodArgument in the doThis method, declared as final.
- And a local variable, in the method block, field30, also declared with the key word final.



```
class ShowFinal {

    private void doThis(final int methodArgument) {

        final int Field30 = 30;
    }
}
```

I ferent value, once these are
i

These are **explicitly final**, and as a result, any of these could be used in a local class.

Effectively Final

In addition to explicitly final variables, you can also use **effectively final** variables in your local class.

A local variable or a method argument are effectively final, if a value is assigned to them, and then never changed after that.

Effectively final variables can be used in a local class.

Additional Local Types

As of JDK 16, you can also create a local record, interface and enum type, in your method block.

These are all implicitly static types, and therefore aren't inner classes, or types, but static nested types.

The record was introduced in JDK16.

Prior to that release, there was no support for a local interface or enum in a method block.

Anonymous Classes

An anonymous class is a local class that doesn't have a name.

All the nested classes we've looked at so far have been created with a class declaration.

The anonymous class is never created with a class declaration, but it's always instantiated as part of an expression.

Anonymous classes are used a lot less, since the introduction of Lambda Expressions in JDK 8.

But there are still some use cases where an anonymous class might be a good solution.

Anonymous class creation

An anonymous class is instantiated and assigned in a single statement.

The new keyword is used followed by any type.

This is **NOT** the type of the class being instantiated.

It's the super class of the anonymous class, or it's the interface this anonymous class will implement as I'm showing here.

```
var c4 = new Comparator<StoreEmployee>() {};
```

Anonymous class creation

In the first example on this slide, the anonymous unnamed class will implement the Comparator interface.

```
var c4 = new Comparator<StoreEmployee>() {};
```

In the second example on this slide, the anonymous class extends the Employee class, meaning it's a subclass of Employee.

```
var e1 = new Employee {};
```

In both cases, it's important to remember the semi-colon after the closing bracket, because this is an expression, not a declaration.

The Local and Anonymous Class Challenge

First, you need to create a record named Employee, that contains First Name, Last Name, and hire date.

Setup a list of Employees with various names and hire dates in the main method.

Setup a new method that takes this list of Employees as a parameter.

Create a local class to wrap this class, (pass Employee to the constructor and include a field for this) and add some calculated fields, such as full name, and years worked.

The Local and Anonymous Class Challenge

Create a list of employees using your local class.

Create an anonymous class to sort your local class employees, by full name, or years worked.

Print the sorted list.

Hint: Here is another review of a date function, which should help you with calculating years worked.

```
int currentYear = LocalDate.now().getYear();
```