Java SE 8 Language Features

This chapter takes a quick look at what's new in Java 8. We'll concentrate on lambda expressions, which are perhaps the headline act in Java 8. In a nutshell, a lambda expression is like an inline function. For example, you can pass a lambda expression into a "sort" method, to specify the sort algorithm to use when sorting items in a collection.

A large portion of the Java API has been refreshed and rejuvenated to support lambda expressions, and we'll see evidence for this in many of the subsequent chapters.

This chapter describes the syntax of lambda expressions, and shows plenty of examples to help you understand the importance.

Contents

Demo project: **DemoJavaSE8**

2

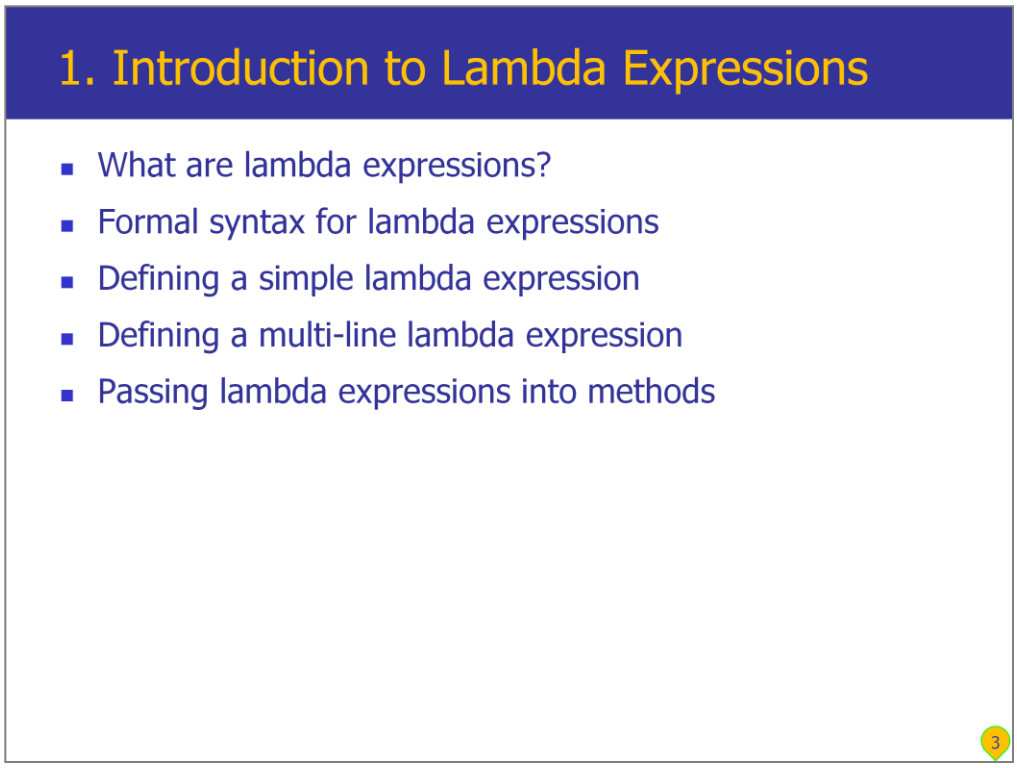
Section 1 describes the key principles behind lambda expressions and introduces the basic syntax.

Section 2 shows how to pass parameters into a lambda expression and how to return a value.

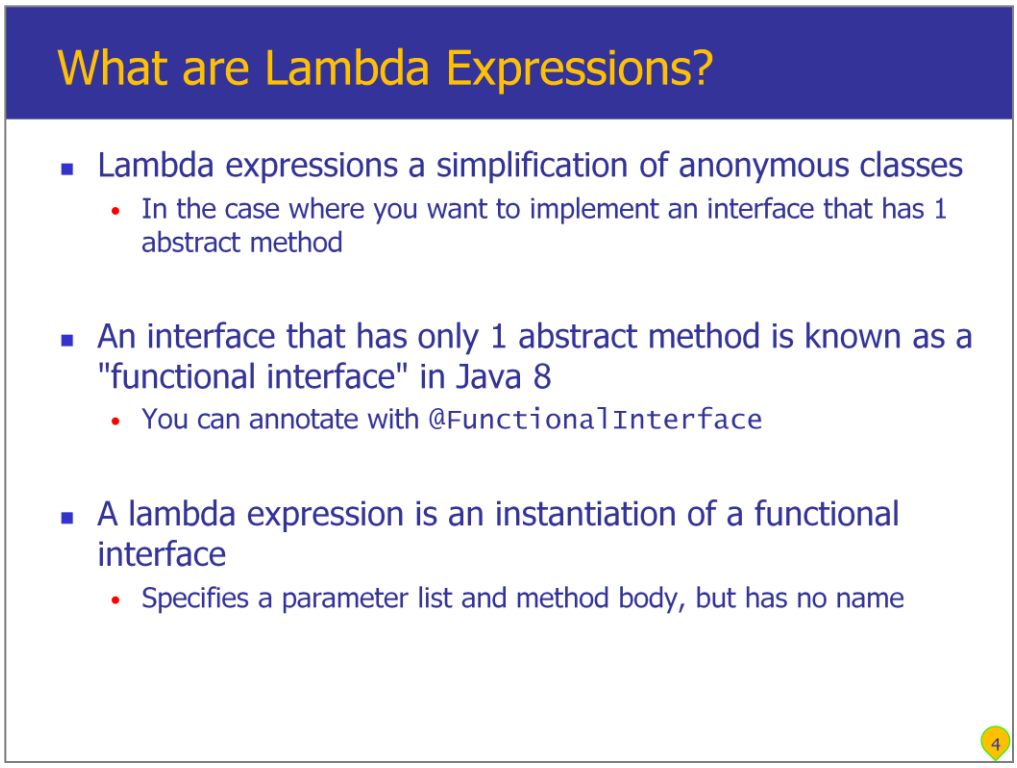
Section 3 explores various additional issues and presents a complete example of lambda usage.

Sections 4 and 5 look at some new concepts relating to methods and constructors - the ability to refer to existing methods and constructors in a class.

Section 6 looks at some new rules for how interfaces work in Java 8.



In this section we'll explain the purpose of lambda expressions, discuss some of the key concepts, and look at the basic syntax for defining and using lambda expressions.

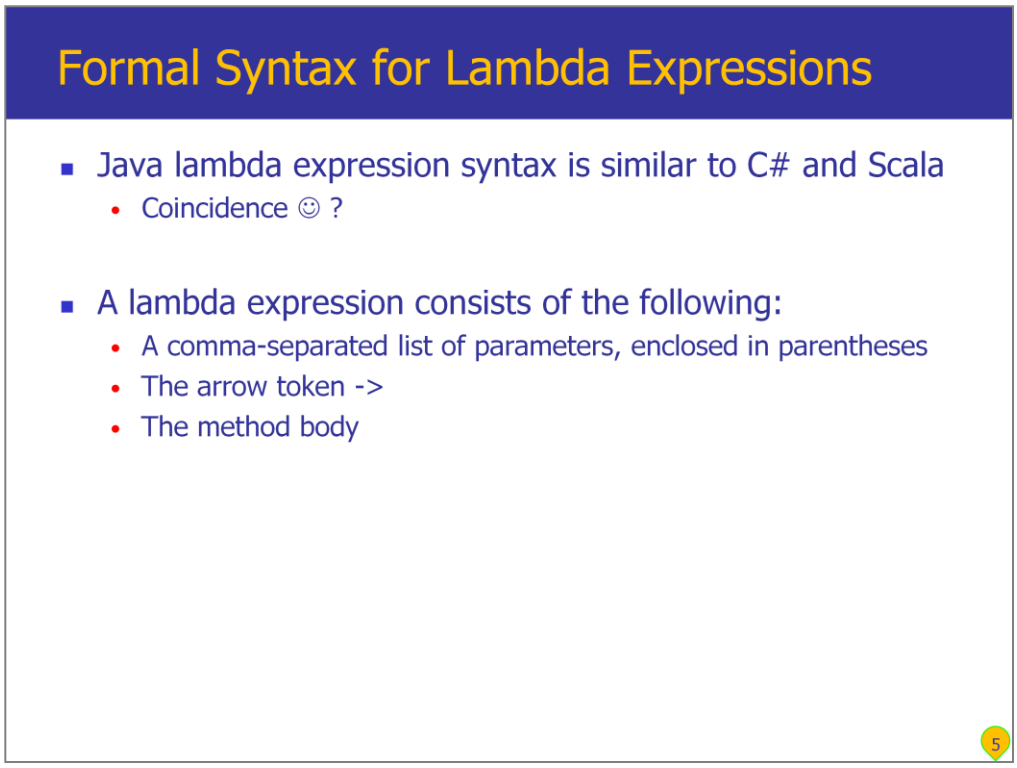


Lambda expressions are the biggest new feature in Java 8, specified by JSR-335. The full technical specification is available here:

- https://jcp.org/en/jsr/detail?id=335

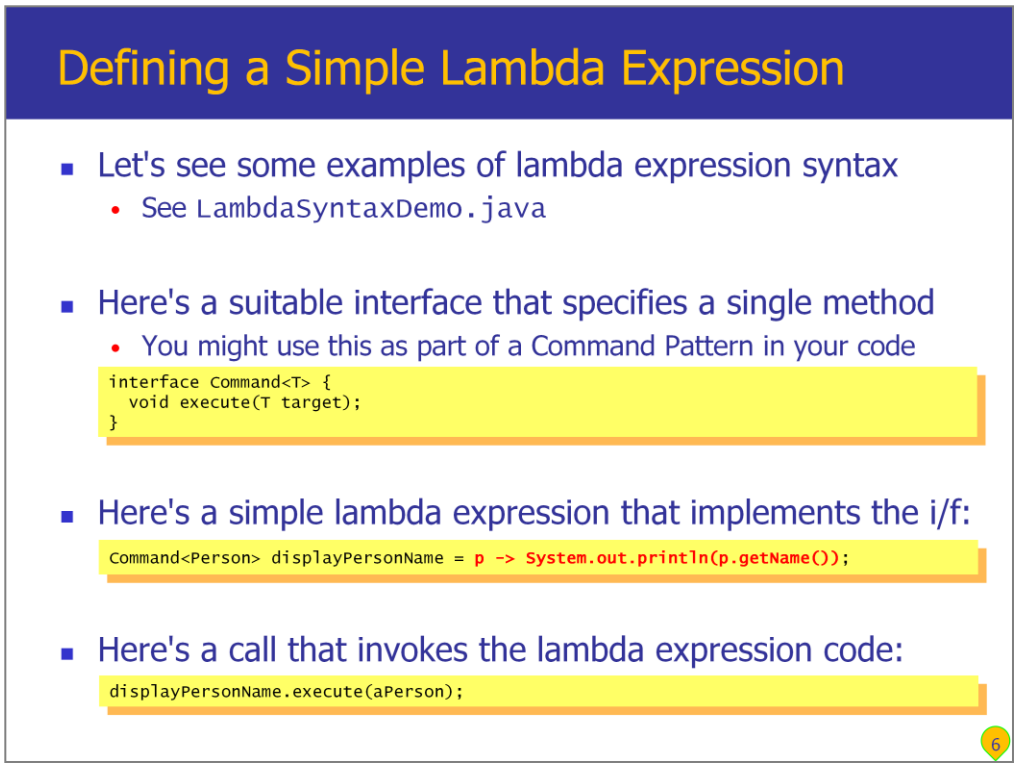
Essentially, lambda expressions are a simplification of anonymous classes, in the case where you want to implement an interface that has a single abstract method. In Java 8, an interface that has only one abstract method is known as a "functional interface" and can be annotated with `@FunctionalInterface`. We'll discuss functional interfaces in detail later in this course.

A lambda expression is an instantiation of a functional interface. A lambda expression specifies a parameter list and a method body, but it has no name. It's similar to the idea of anonymous functions in JavaScript.



A lambda expression consists of the following:

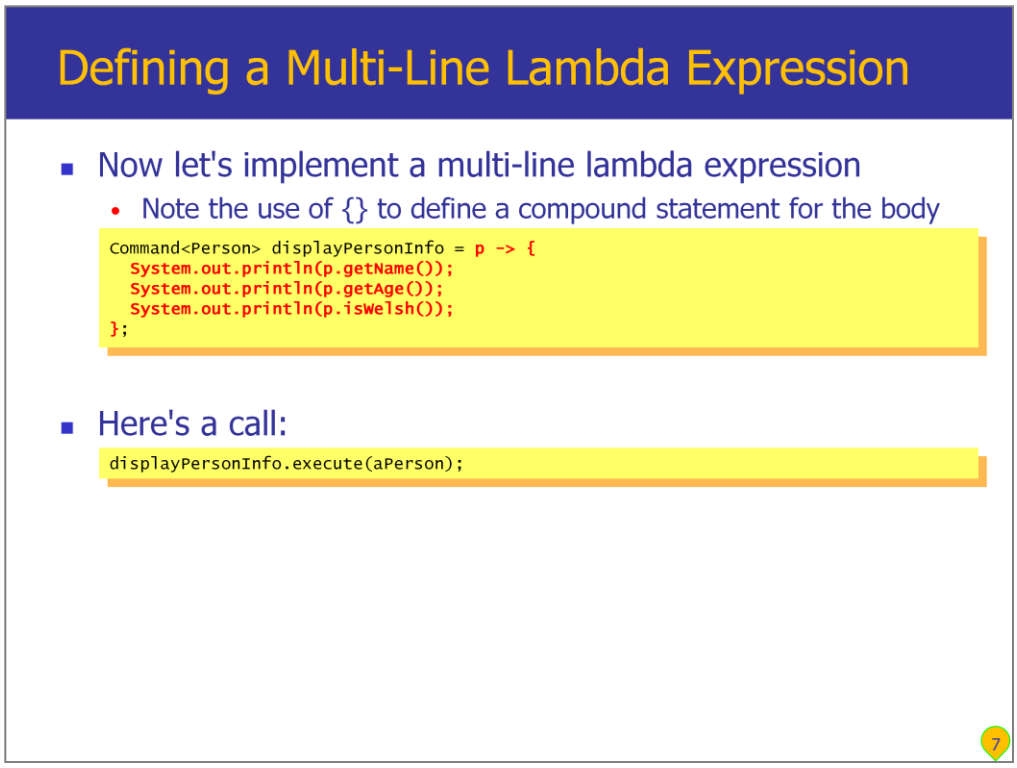
- A comma-separated list of parameters, enclosed in parentheses
  - You can omit the data types of the parameters
  - If there's only one parameter, you can omit the parentheses
- The arrow token ->
  - Note, this is different to C#, which uses =>
- The method body
  - A single expression (this is evaluated as the method return value)
  - Or a statement block enclosed in { }



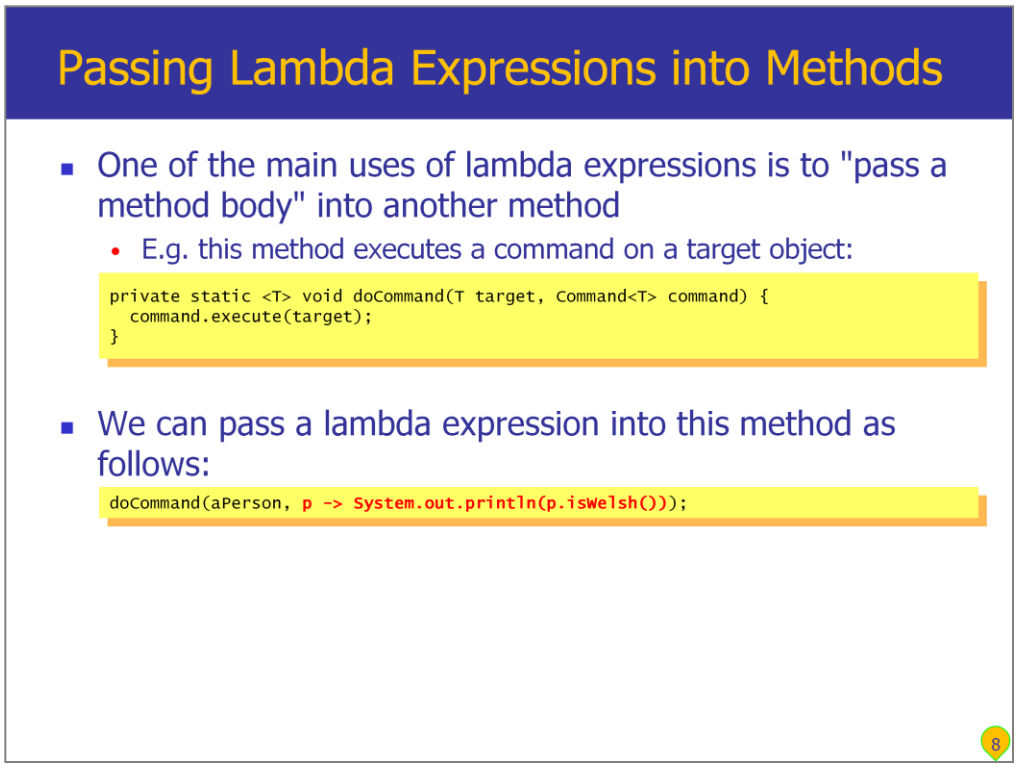
On this and the following slides, we'll show various examples of how to define and use lambda expressions. These examples are in `LambdaSyntaxDemo.java` in the demo project.

Consider the code in the slide:

- The upper code box defines a simple interface that has a single method. The fact it has a single method is critical. As we mentioned on the previous slide, this is an example of a functional interface in Java 8.
- The middle code box defines a lambda expression. The Java 8 compiler understands lambda expressions and it knows what we're trying to do:
  - When the compiler sees the declaration of the `displayPersonName` variable on the left-hand side of the assignment, it knows the variable must point to an object of type `Command<Person>`.
  - Therefore when the compiler sees the lambda expression on the right-hand side of the assignment, the compiler creates an anonymous class that implements the `Command<Person>` interface, and it treats the lambda expression as the implementation of the `execute()` method. The compiler then creates an instance of the anonymous class and assigns it to the `displayPersonName` variable.
- The lower code box calls `execute()` method on the `displayPersonName` object, i.e. it calls the lambda expression on the instance of the anonymous class that implements the `Command<Person>` interface.



This example is similar to the previous slide, except that the lambda expression now spans several lines. In this case, you must use braces {} to mark the beginning and end of the lambda expression body.



One of the main uses of lambda expressions is to "pass a method body" into another method. For example, the standard Java method `Arrays.sort()` allows you to pass in an object that implements the `Comparator<T>` interface, to specify how to compare two elements in the array.

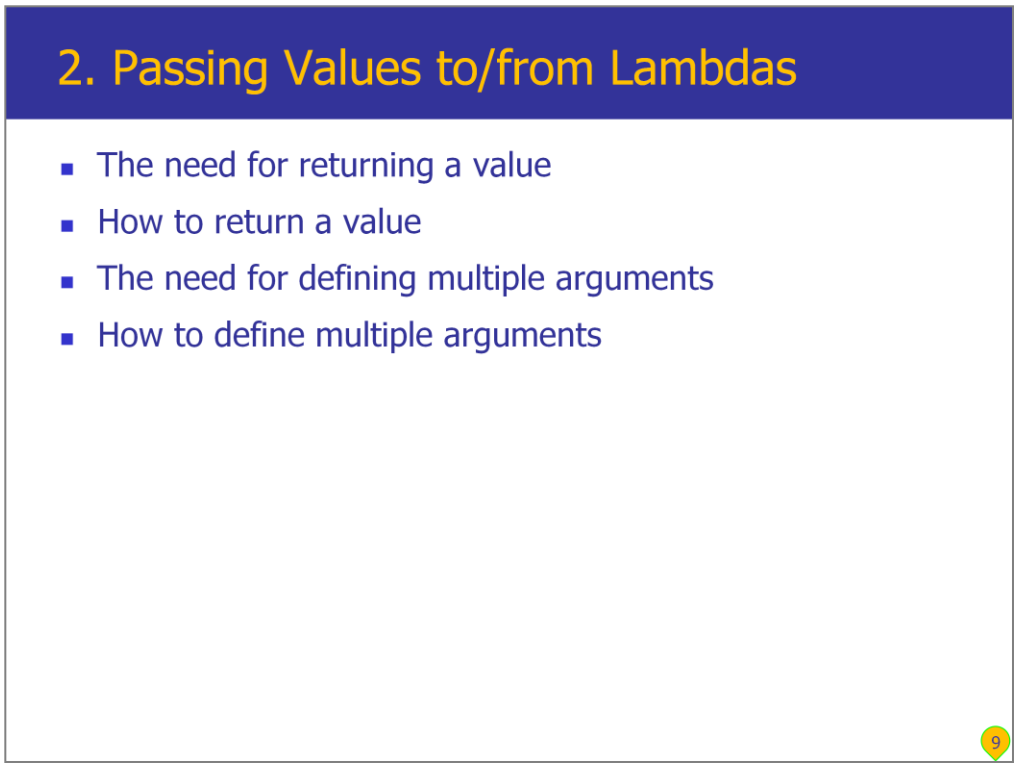
Consider the upper code box in the slide. The `doCommand()` method has two parameters:

- An object of any type `T`. You can pass in any kind of object for this parameter, e.g. a `Person` object.
- An object that implements the `Command<T>` interface (e.g. an object that implements `Command<Person>`).

Inside `doCommand()`, we invoke the `execute()` method to execute some task on the target object . This is an example of the "Command Pattern" in OO design, because it allows you to execute any task on any target object.

Now consider the lower code box in the slide. We call `doCommand()` and pass in two parameters:

- A `Person` object.
- A lambda expression that effectively implements `Command<Person>`. Note that the compiler checks your lambda expression is consistent with the target type.



In this section we're going to dig a bit deeper into the syntax for lambda expressions. Specifically we'll show how to pass multiple parameters to a lambda expression and how to return a value from a lambda expression.

The Need for Returning a Value

- Let's see how to return a value from a lambda expression
  - See `LambdaReturnValueDemo.java`
- Here's an interface with a method that returns a value:

```
interface Predicate<T> {
  boolean test(T obj);
}
```

- Here's a method that uses the interface:

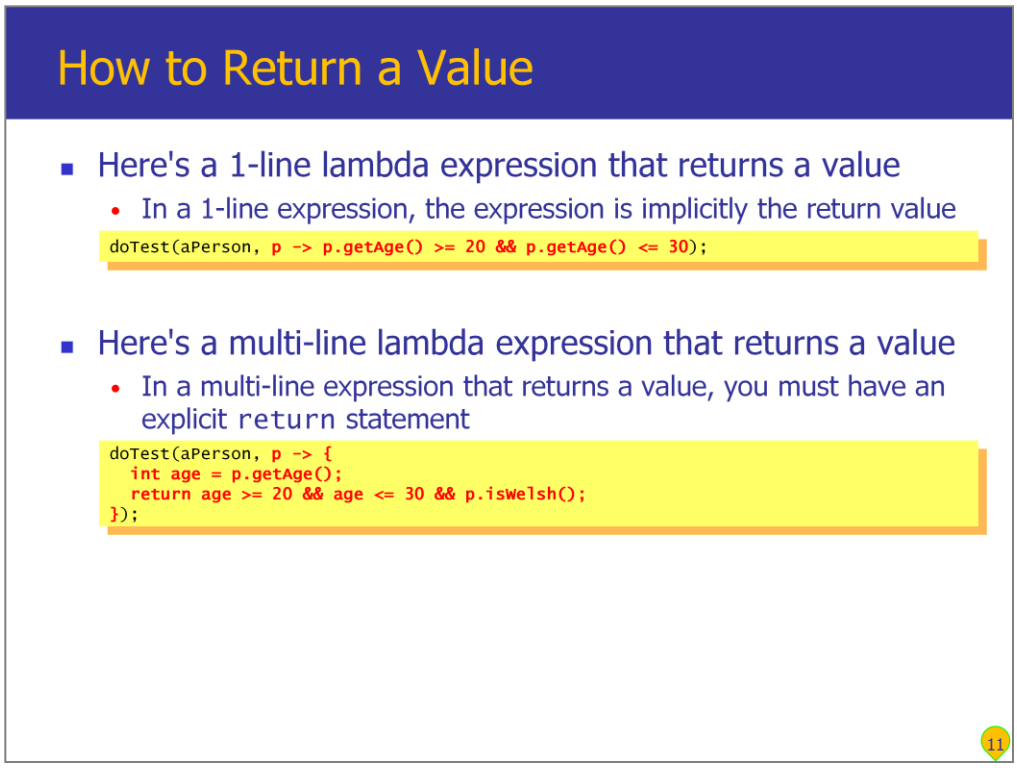
```
private static <T> void doTest(T obj, Predicate<T> predicate) {
  if (predicate.test(obj)) {
    System.out.println("True!");
  } else {
    System.out.println("False!");
  }
}
```

10

On this and the next slide, we show how to return a value from a lambda expression. You can find the sample code in `LambdaReturnValueDemo.java`.

Consider the upper code box on the slide. This shows a functional interface named `Predicate<T>`, whose `test()` method returns a `boolean` result. The idea is that client code can provide various implementations of the interface, to perform different kinds of test on a target object.

Now consider the lower code box on the slide. This shows how the interface might be used in client code. The `doTest()` method receives a target object (`obj`) plus an object that implements the `Predicate<T>` interface. We invoke the `test()` method on the `Predicate<T>` object, to perform whatever test it implements on the target object.



The syntax for returning a value from a lambda expression depends on whether the lambda expression is a one-line expression or a multi-line statement:

- The upper code box shows how to return a value from a one-line lambda expression. In this case, all you need is an expression that yields a value. This is implicitly assumed to be the return value for the lambda expression. You do not need a `return` statement. In fact you must NOT have a `return` statement; one-line lambda expressions really are just "expressions" (i.e. not statements).
- The lower code box shows how to return a value from a multi-line lambda expression enclosed in `{}`. This is more like regular programming, i.e. the `{}` are similar to a method body. In this case, you MUST have a `return` statement to return a value from the lambda expression.

The Need for Defining Multiple Arguments

- You can specify multiple arguments in a lambda expression
  - See `LambdaMultipleArgsDemo.java`
- Here's an interface with a method with multiple arguments:

```
interface Calculation<T> {
  T perform(T op1, T op2);
}
```

- Here's a method that uses the interface:

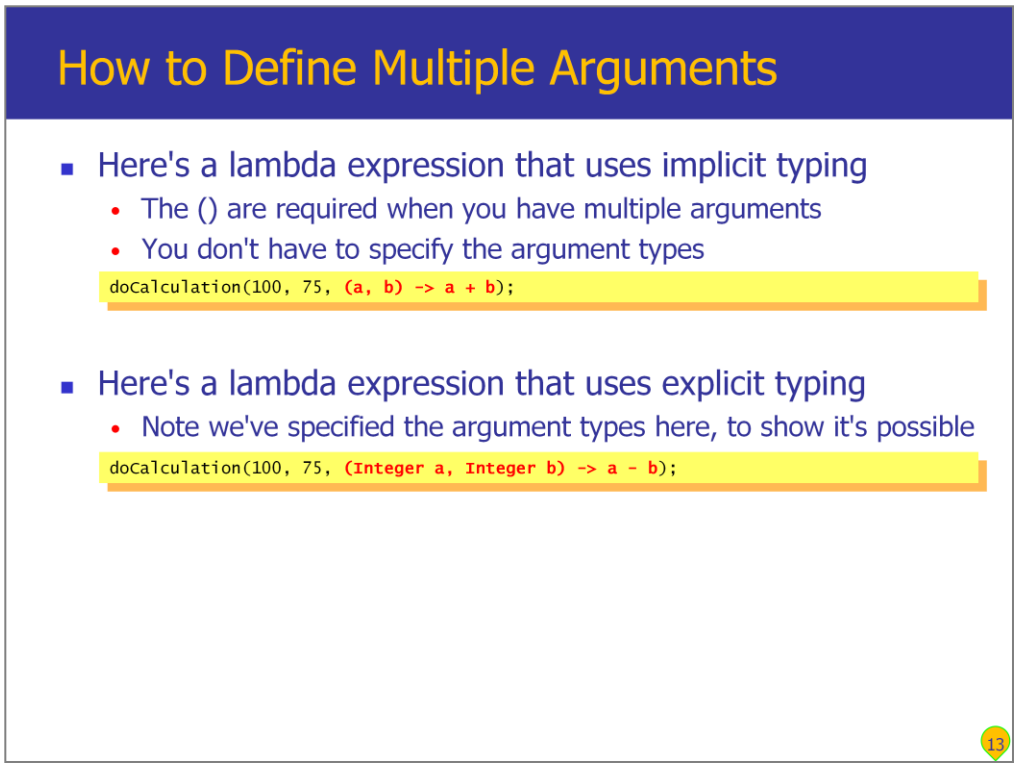
```
private static <T> void doCalculation(T val1, T val2, Calculation<T> calculation) {
  T result = calculation.perform(val1, val2);
  System.out.println("Calculation result: " + result);
}
```

12

On this and the next slide, we show how to pass multiple arguments into a lambda expression. You can find this sample code in `LambdaMultipleArgsDemo.java`.

Consider the upper code box on the slide. This shows a functional interface named `Calculation<T>`, whose `perform()` method takes a couple of parameters. The idea is that client code can provide various implementations of the interface, to perform different kinds of operation using the supplied values.

Now consider the lower code box on the slide. This shows how the interface might be used in client code. The `doCalculation()` method receives a couple of values (`val1` and `val2`), plus an object that implements the `Calculation<T>` interface. We invoke the `perform()` method on the `Calculation<T>` object, to perform whatever operation it implements on the two values.

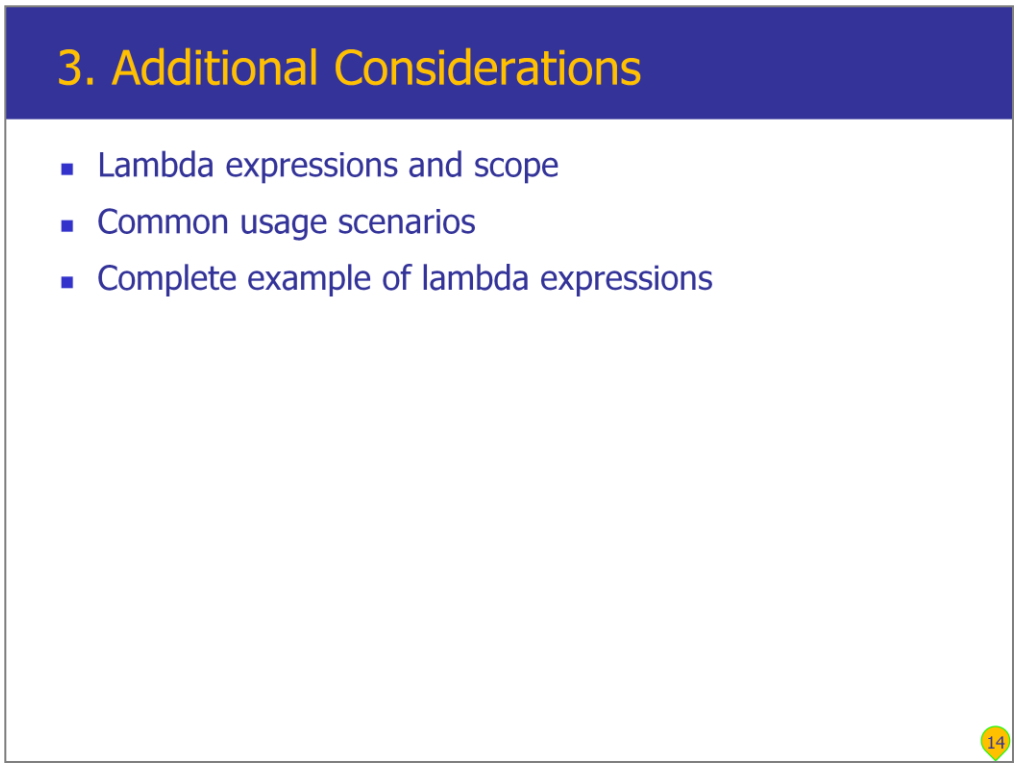


When a lambda expression receives more than one parameter, you must enclose the parameters in parentheses `()`. (You can also use parentheses if the lambda expression receives just one parameter, but the parentheses are optional in this case).

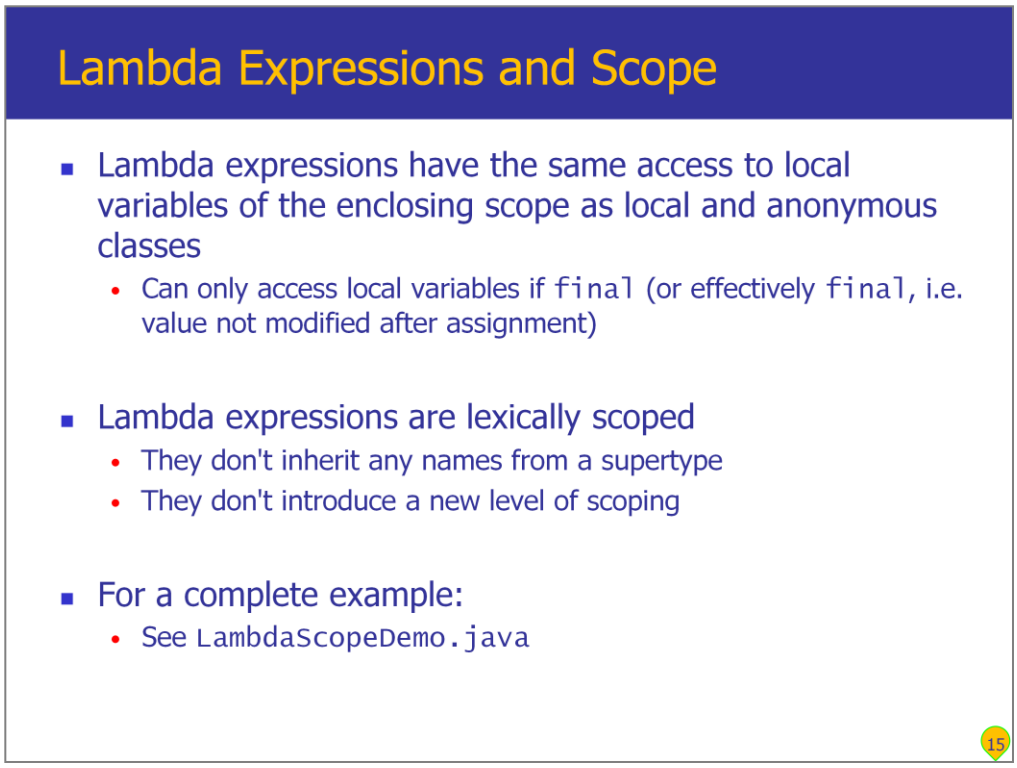
You don't have to specify the data types for parameters in a lambda expression. The compiler can figure out the parameter types for itself, based on context. For example, consider the upper code box on the slide, where we call the generic `doCalculation()` method. Note the following points:

- The first two arguments we pass into `doCalculation()` are integers, so the compiler substitutes `Integer` for the generic method's type parameter `T`.
- The third argument we pass into `doCalculation()` is a lambda expression. The compiler treats this lambda expression as an implementation of the `Calculation<Integer>` interface. Specifically, the compiler maps our lambda expression to the `perform()` method in that interface. The `perform()` method receives two `T` parameters (`T` represents `Integer` here), so the compiler can deduce the lambda expression parameters a and b are both `Integer` type as per the perform() `method` signature.

If you really want to, you are allowed to define the data types for parameters in a lambda expression - see the lower code box in the slide. This is useful if you want to force the compiler to treat the parameters as a certain type. Most times you don't need to do this.



Once you're happy with the core syntax of lambda expressions, you quickly turn your attention to trying to figure out how to use them in real large-scale applications. This section aims to give you some pointers.



The best way to understand how lambda expressions work is to think about how local classes and anonymous classes work in classic Java, because the compiler converts lambda expressions into these constructs internally.

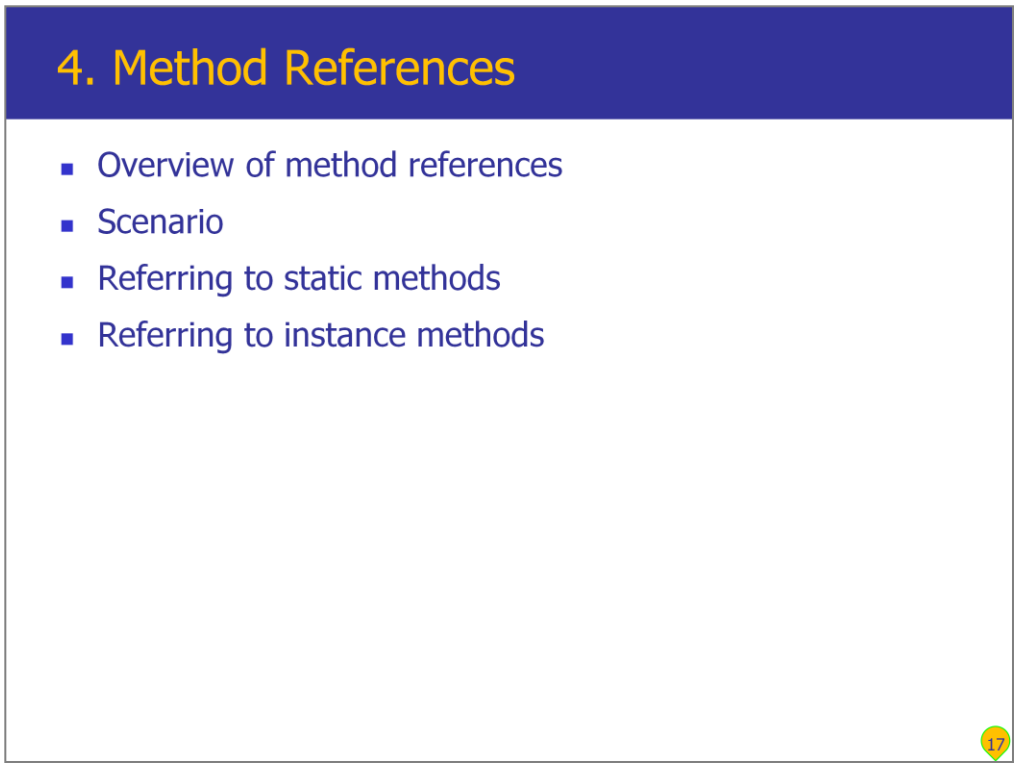
To illustrate lambda expressions and scope, we've put together several examples in `LambdaScopeDemo.java`. The demo code contains copious comments to explain what's happening.

Common Usage Scenarios

- Here are some common usage scenarios for lambda expressions
  - Implement `Runnable` to run a thread
  - Implement `Callable<T>` to run a thread and get a result
  - Implement `Comparator<T>` to specify a comparison rule
  - Handle events in a GUI application
- You'll tackle these tasks in the lab ☺

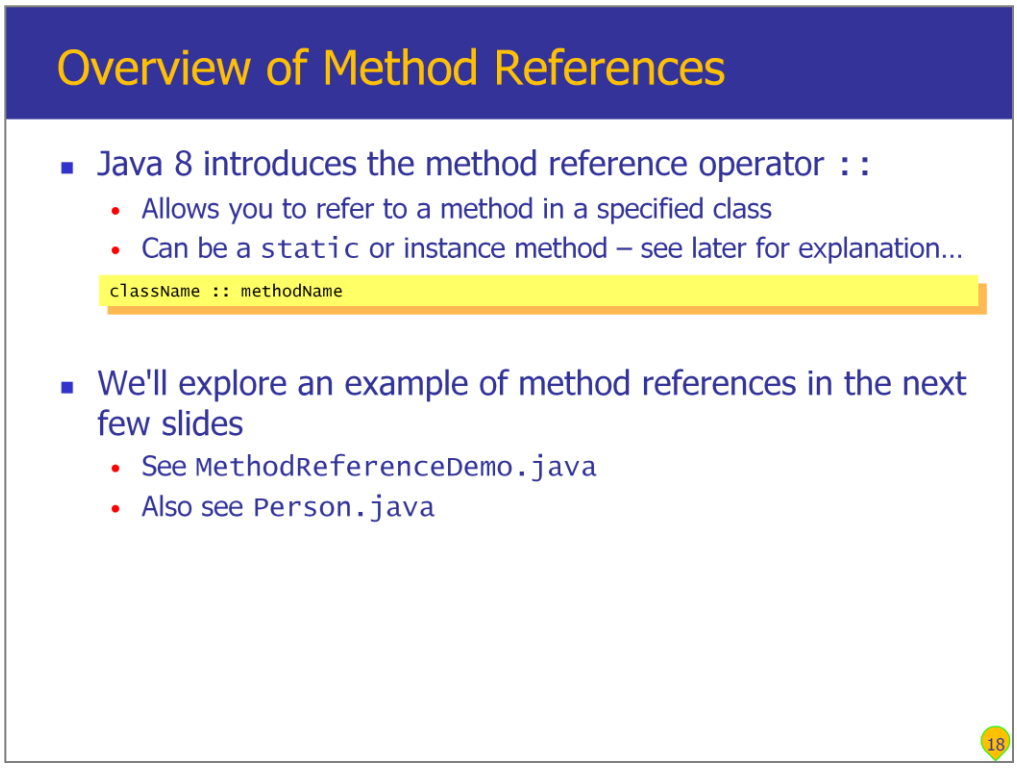
16

This slide lists some common usage scenarios for lambda expressions. You'll implement these scenarios in the lab ☺.



This section introduces new syntax in Java 8 that allows you to refer to `static` methods and instance methods in an existing class.

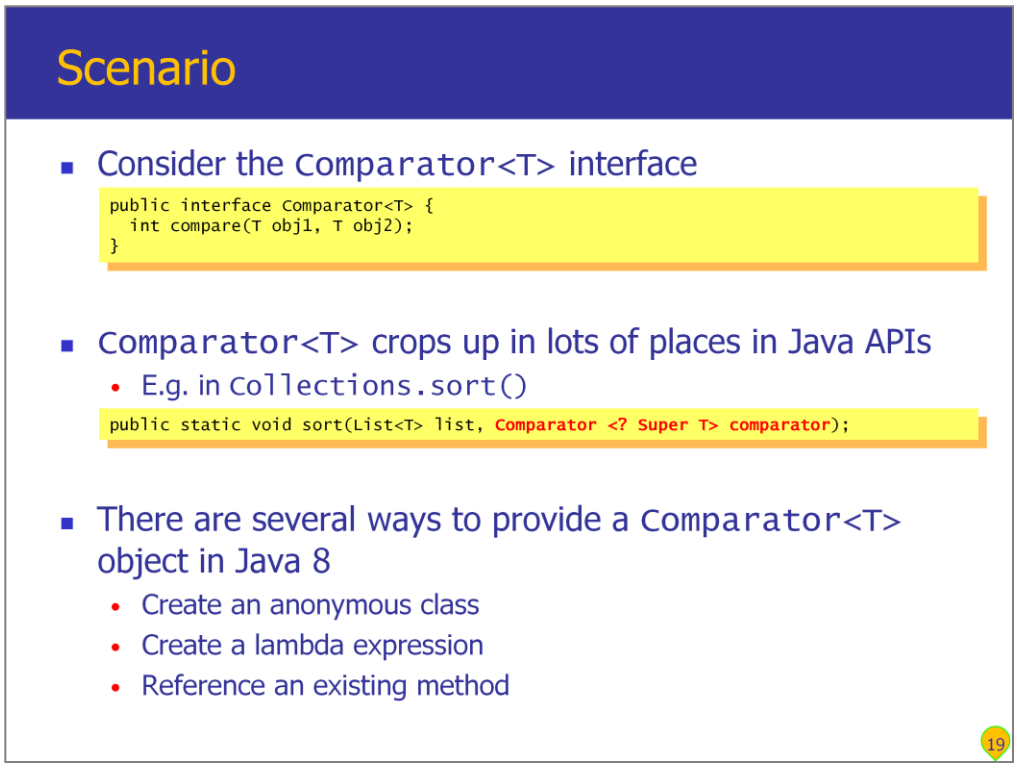
The most common reason for doing this is to refer to a method that can be treated as an effective implementation for an abstract method in a functional interface. All will be revealed as we proceed through this section.



Java 8 introduces a new operator called the "method reference operator", which you can use to reference a method in a class. The syntax is as follows:

```
className :: methodName
```

You can use this syntax to refer to a `static` method or an instance method in the specified class. We'll show some examples of how to use this operator in the next few slides.



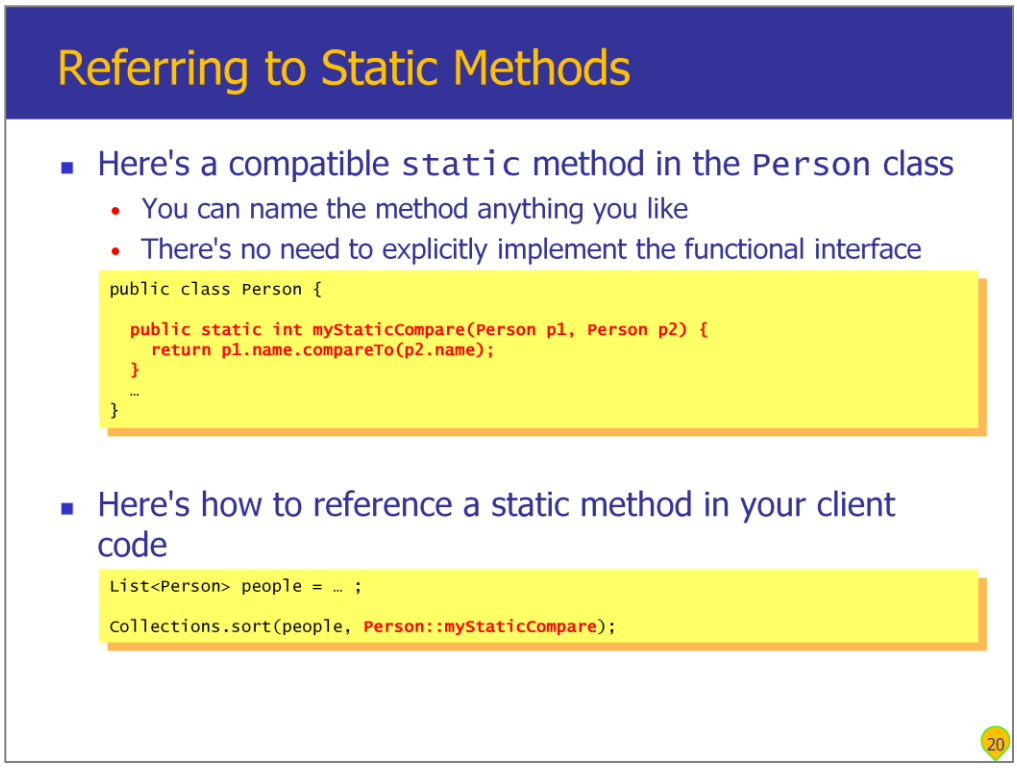
In order to understand the purpose of method references in Java 8, we'll take a look at a specific example of how to implement the `Comparator<T>` interface.

The upper code box in the slide shows the `Comparator<T>` interface (in fact, the Java 8 version of this interface is much more complicated than we've shown here - we'll discuss the details later in the course). The important point is that `Comparator<T>` is a functional interface, i.e. it has one abstract method remaining to be implemented.

The `Comparator<T>` interface appears in lots of places in Java APIs. For example, the lower code in the slide shows the `Collections.sort()` method. The second parameter to this method is a `Comparator<T>` object, which tells the sort algorithm how to compare elements for the purposes of sorting.

Java 8 gives us several ways to provide a `Comparator<T>` object:

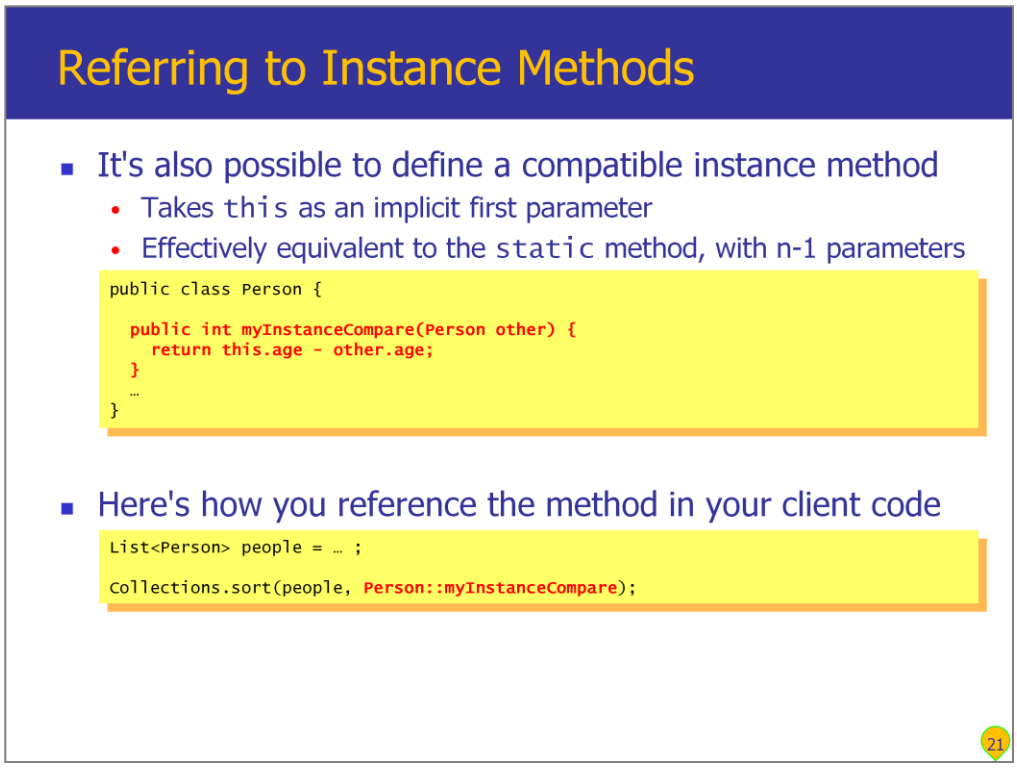
- Create an anonymous class
  This is the traditional approach to providing implementations for interfaces, i.e. you can use it in pre-Java 8 as well.
- Create a lambda expression (Java 8 only)
  This is handy if you want to provide a one-off algorithm, e.g. a comparison algorithm you only want to use in this particular call to `sort()`. We discussed lambda expressions earlier in the course.
- Reference an existing method (Java 8 only)
  This is handy if you want to define a reusable algorithm, e.g. a comparison algorithm you want to use in several separate calls to `sort()`.



This slide describes how to use method references to refer to a `static` method in a class. Consider the following points about the code in the slide:

- In the upper code box, the `Person` class defines a `static` method named `myStaticCompare`. The method takes two `Person` objects and returns an `int`. The important point to note here is that the signature of this method is the same as the `compare()` method in the `Comparator<Person>` interface.
- The lower code box calls the `Collections.sort()` method, passing in a collection of `Person` objects to be sorted. The second parameter is a reference to the `myStaticCompare` method. The compiler was expecting us to pass a `Comparator<Person>` object here, with a suitable `compare()` method. We haven't done that, but we have provided a method with a compatible signature, and that's enough to keep the compiler happy in Java 8. When the `sort()` method executes, it invokes `myStaticCompare` using this kind of syntax internally:

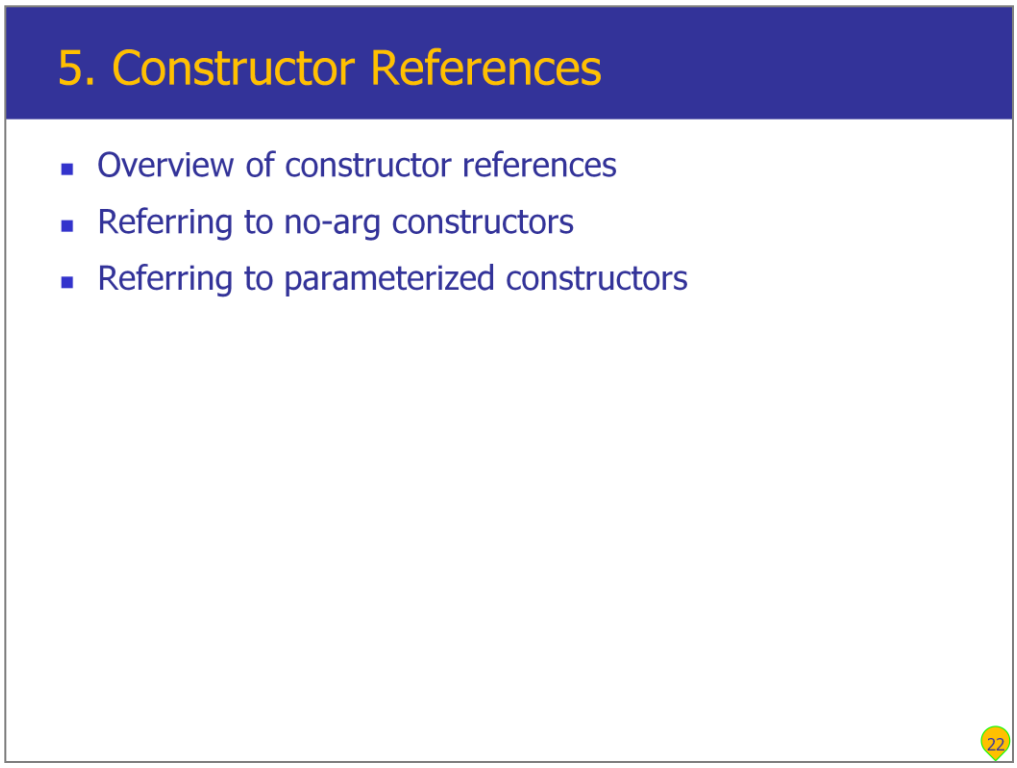
```
Person.myStaticCompare(person1, person2);
```



This slide is similar to the previous slide, but this time we're referring to an instance method rather than a `static` method. Consider the following points about the code in the slide:

- In the upper code box, the `Person` class defines an instance method named `myInstanceCompare`. If you go down the line of defining an instance method rather than a `static` method, the instance method will take one fewer parameter than the `static` methods (the instance method receives `this` as an implicit first parameter to the method).
- The lower code box calls the `Collections.sort()` method. The second parameter is a reference to the `myInstanceCompare` method. When the `sort()` method executes, it invokes `myInstanceCompare` using this kind of syntax internally:

```
person1.myInstanceCompare(person2);
```



The previous section showed how to use method references to refer to existing `static` methods or instance methods in a class. You can extend this concept to apply to constructors in a class, as we'll see in this section.

Overview of Constructor References

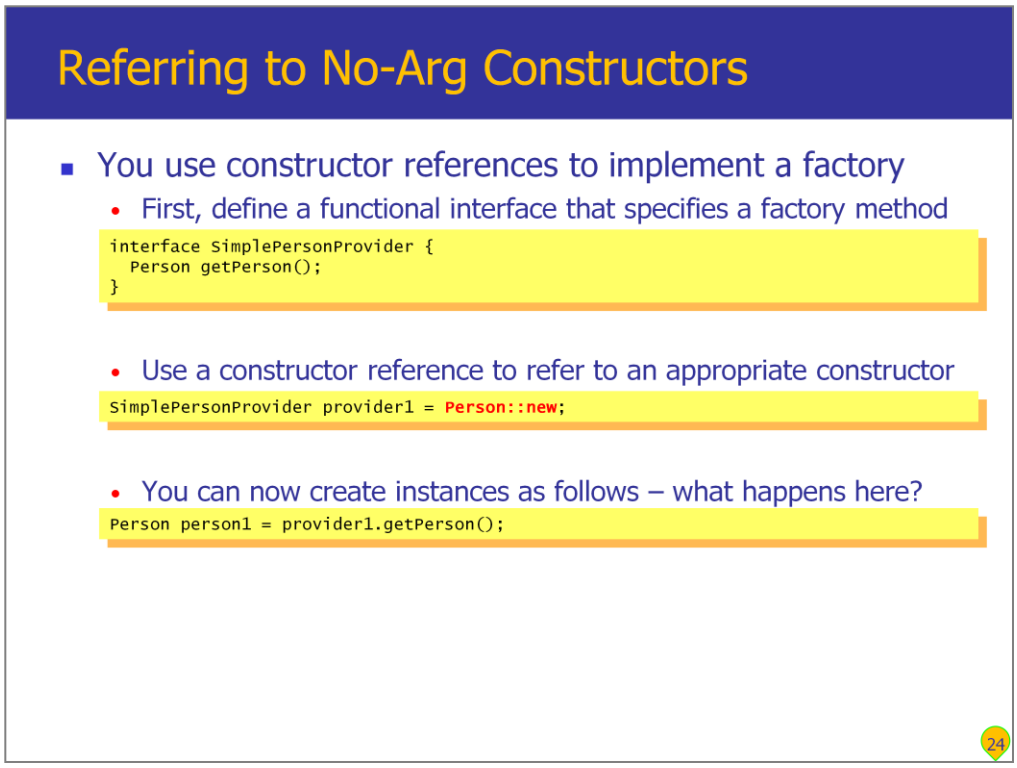
- A constructor reference is similar to a method reference
  - Allows you to refer to constructors for a class
  - Similar to method references, but use new for method name

```
className :: new
```

- If you have overloaded constructors, that's fine!
  - The compiler will figure out which one to call, based on context
- We'll explore an example of constructor references in the next few slides
  - See CtorReferenceDemo.java
  - Also see Person.java

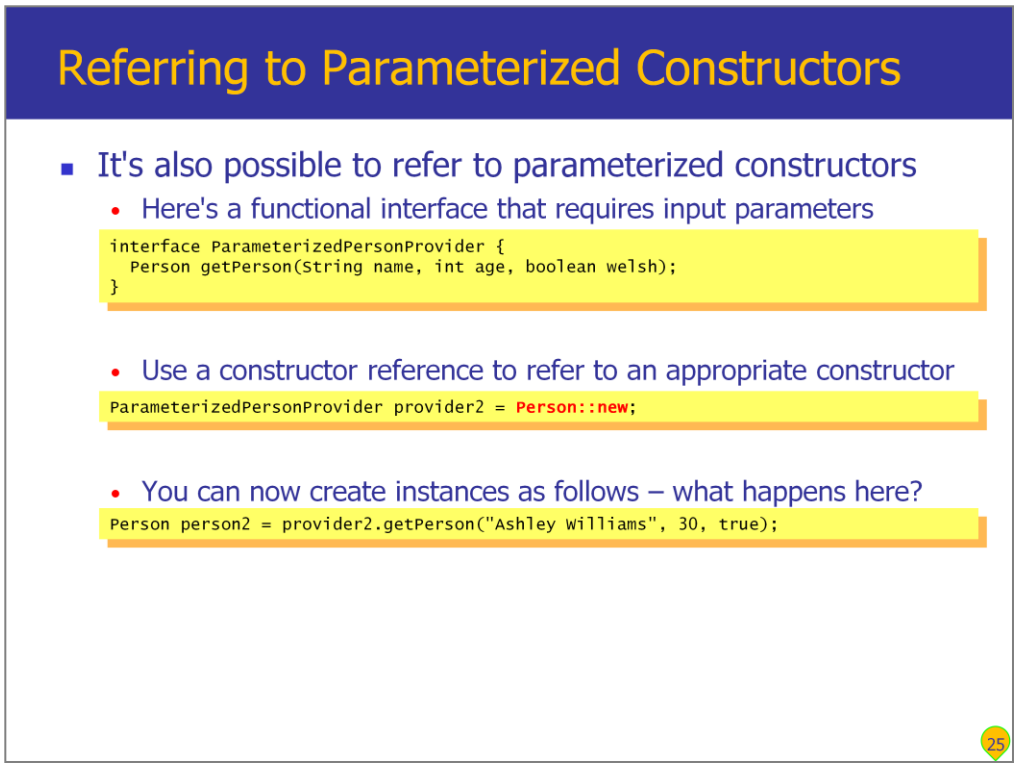
23

This slide shows the syntax for defining constructor references. Basically the syntax is the same as for method references, except that you use new for the method name.



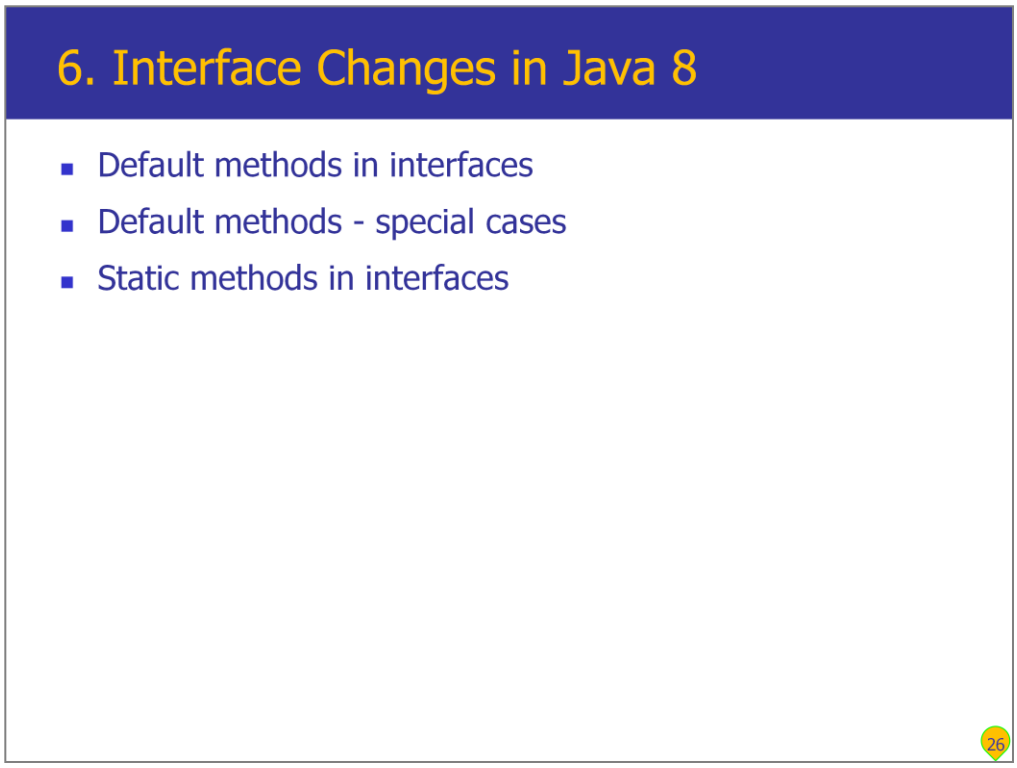
You use constructor references to implement a factory as follows:

- Define a functional interface that specifies a factory method. In other words, the interface must have a single `abstract` method whose responsibility is to return a new object when called. E.g. in the upper code box in the slide, the `SimplePersonProvider` interface defines a `getPerson()` which is expected to return a new `Person` object.
- In traditional Java (i.e. pre-Java 8), the next step would be to define a class that implements the interface and provides a suitable implementation of the factory method. In Java 8, you can instead refer to an existing constructor in a class to act as the factory method. E.g. in the middle code box in the slide, we refer to the no-arg constructor in the `Person` class, and assign it to a `SimplePersonProvider` variable. Effectively, the `Person` constructor fulfils the role of `getPerson()` in the `SimplePersonProvider` interface.
- The final step is for client code to invoke the `getPerson()` method on the `SimplePersonProvider` object, to create a `Person` object. The client code doesn't know or care how the `Person` object was instantiated; all the client code needs to know is that it can call `getPerson()` to get a new `Person` object.



The previous slide showed how to use a constructor reference to refer to a no-arg constructor in a class. You can also refer to parameterized constructors in a class, as shown in this slide. Consider the following points:

- The upper code box shows a functional interface that has a parameterized factory method, i.e. you must supply values that can be used to initialize the new object when it is created.
- The middle code box uses a constructor reference to refer to an appropriate constructor in the `Person` class. The compiler looks for a suitable constructor in the `Person` class, based on the signature of the `getPerson()` factory method (i.e. the `Person` class must have a constructor that takes a `String`, an `int`, and a `boolean`).
- The lower code box shows how to create an instance via the `getPerson()` factory method. At this stage it's just like regular Java programming, i.e. the client code has no idea that the `getPerson()` factory method is actually implemented via a reference to a constructor in the `Person` class.



To conclude this chapter, we're going to look at some important new features available when you define interfaces.

Default Methods in Interfaces

- Java 8 supports default methods in an interface
  - Allows you to provide a concrete implementation for method(s)

```
interface ParentInterface {

    void method1(int i);
    void method2(double d);

    default void method3(String s) {
        System.out.println("ParentInterface.method3 received " + s);
    }
}
```

- Examples:
  - See `DefaultMethodDemo.java`
- Usefulness:
  - Allows you to augment existing interfaces in a non-breaking way
  - Java 8 makes extensive use of default methods!

27

Java 8 allows you to define concrete behaviour for methods in an interface. Simply prefix the method signature with the keyword `default`, and provide a suitable default implementation for the method body.

There are several reasons for this new language feature:

- It allows the Java API authors to "fill in the gaps" in existing interfaces, i.e. to provide a default implementation for all-but-one of the methods in the interface. The idea is to leave the interface with only a single outstanding abstract method. This allows the interface to be used with lambda expressions and method references.
- It also allows the Java API authors (and you) to augment existing interfaces with useful additional concrete behaviour, without breaking any existing code that uses these interfaces.

Default Methods - Special Cases (1 of 3)

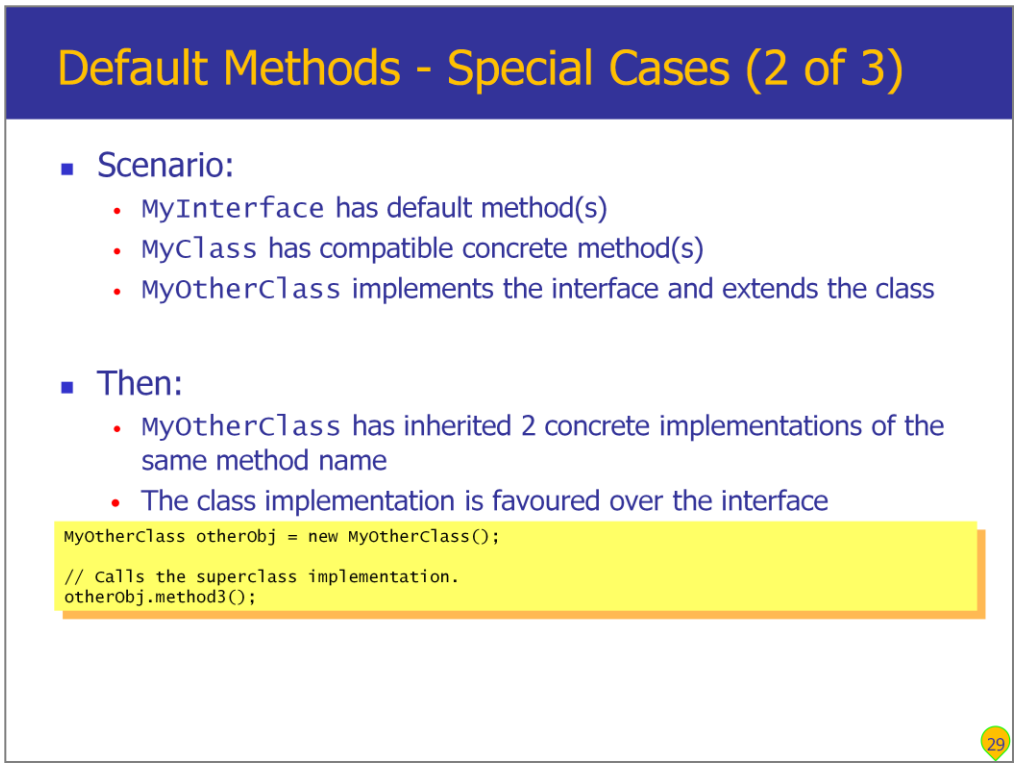
- Scenario:
  - An interface has default method(s)
  - A class implements the interface
- Then:
  - The class can override default method(s) if it wants to
  - The overriding method can access default method implementation

```
class ChildClass implements ParentInterface {

  // Optionally, can override default method(s) from interface.
  @Override
  public void method3(String s) {
      …

      // Can also access default implementation of method.
      ParentInterface.super.method3(s);
  }
}
```

28

On this and the following two slides, we examine a few special cases that occur when you define default methods in an interface.

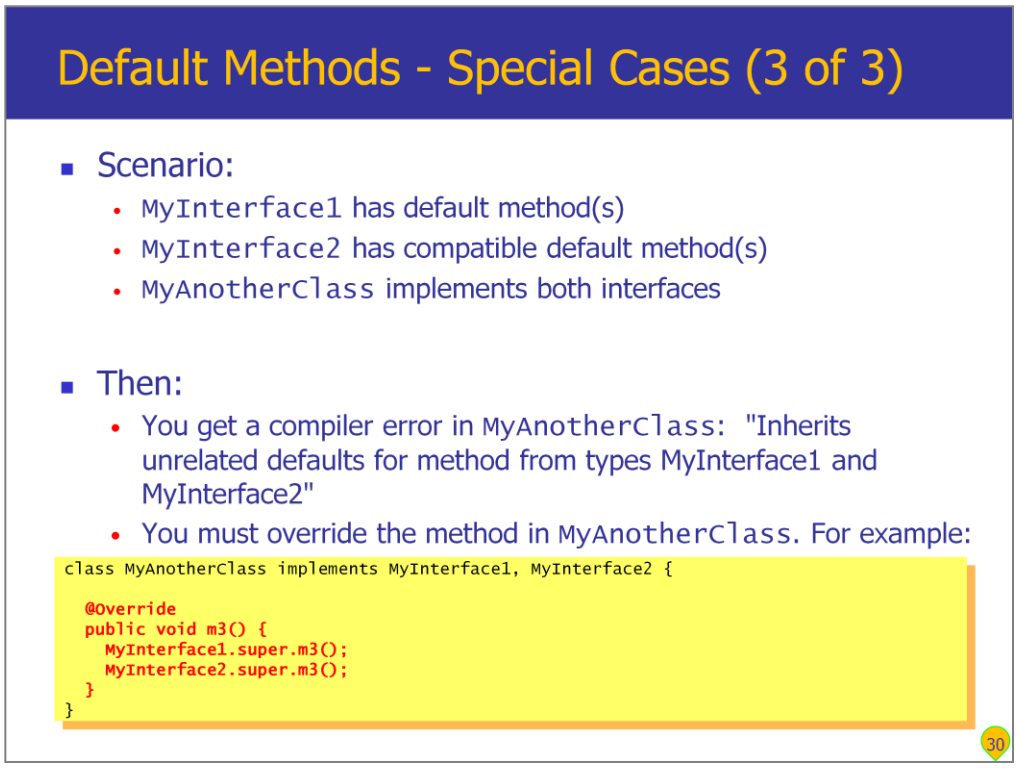
To kick things off, if you define a class that implements an interface that has default method(s), you can override default methods if want to, but you aren't obliged to do so.



Consider the scenario outlined in the slide. You can find sample code for this scenario in `DefaultMethodDemo.java` (see the "Special case 2" comment in the code).

Now consider the following client code, which calls `m3()` on the `MyOtherClass` object. There are two implementations of this method available - one from the superclass (`MyClass`) and one from the interface (`MyInterface`). In this scenario, the implementation from the superclass is always preferred.

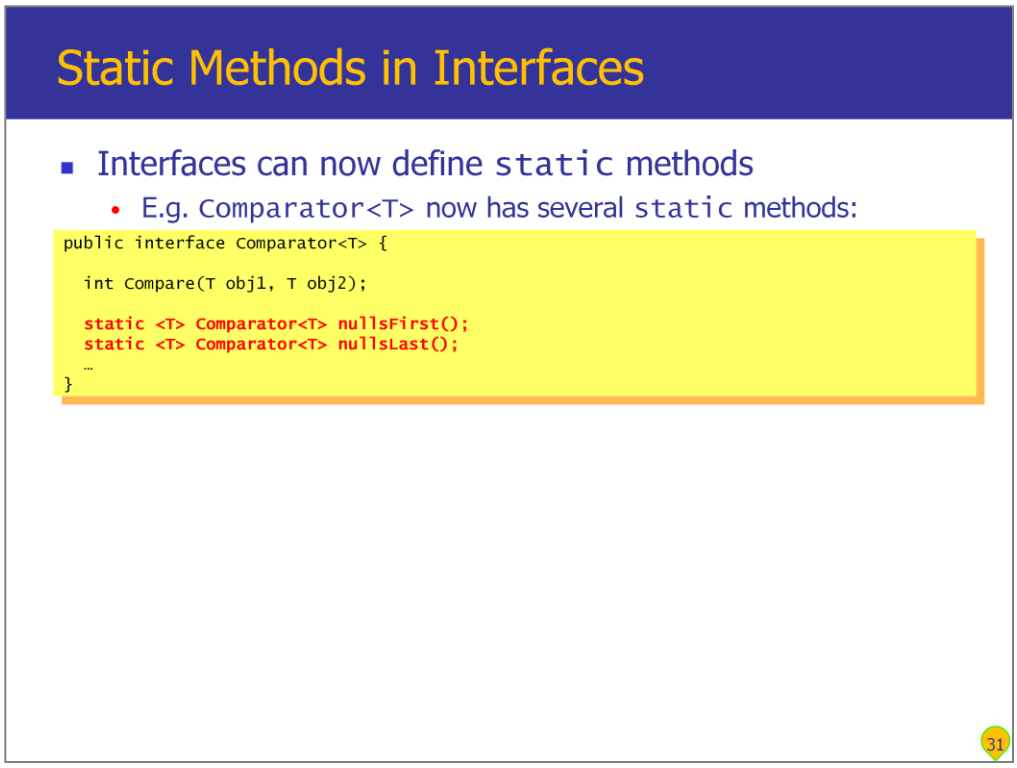
```
MyOtherClass otherObj = new MyOtherClass();
otherObj.m3();
```



Consider the scenario outlined in the slide. You can find sample code in `DefaultMethodDemo.java` (see the "Special case 3" comment in the code).

If a class implements several interfaces, it will inherit concrete implementation of all the default methods from all the interfaces. If the interfaces have compatible default methods (e.g. named `m3()` in the sample code), you'll get a compiler error. For example: "Duplicate default methods named m3 with the parameters () and () are inherited from the types MyInterface2 and MyInterface1".

To resolve the compiler error, the class must implement the offending method itself. It's still possible to invoke the inherited methods, as shown in the code snippet in the slide.



Interfaces in Java 8 can have `static` methods, as shown in the slide (you really couldn't do this in previous versions of Java!).

One of the reasons for allowing `static` methods in Java 8 is to eliminate the need for superfluous companion utility classes to house `static` methods that should really belong in an interface. For example:

- Many traditional Java interfaces (e.g. `Comparator<T>`) have a companion utility class (e.g. `Comparators<T>`) with `static` methods to generate or work with interface instances.
- Now that `static` methods can exist on interfaces, the utility class might be superfluous (put its `public` methods into the interface instead, as `static` methods).

Any Questions?