Inner classes are just what their name suggests: classes that are defined inside something else (e.g. inside another class, or even inside another method). Inner classes are quite similar to nested classes in other languages, but there are more ways to define inner classes in Java than you might expect.

This chapter describes what inner classes are, explains the various syntax options available, and gives you some context to help you decide when to use each mechanism.

Contents

1. Overview of inner classes
2. Regular inner classes
3. Method-local inner classes
4. Anonymous inner classes
5. Static nested class

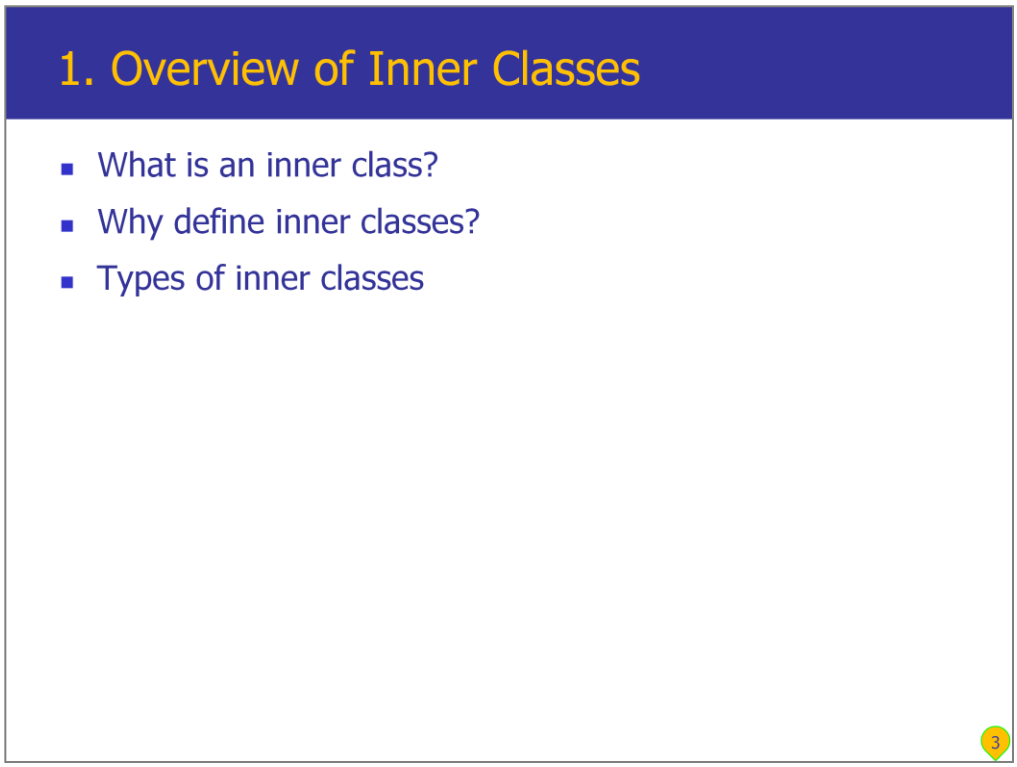
Demo project: **DemoInnerClasses**

2

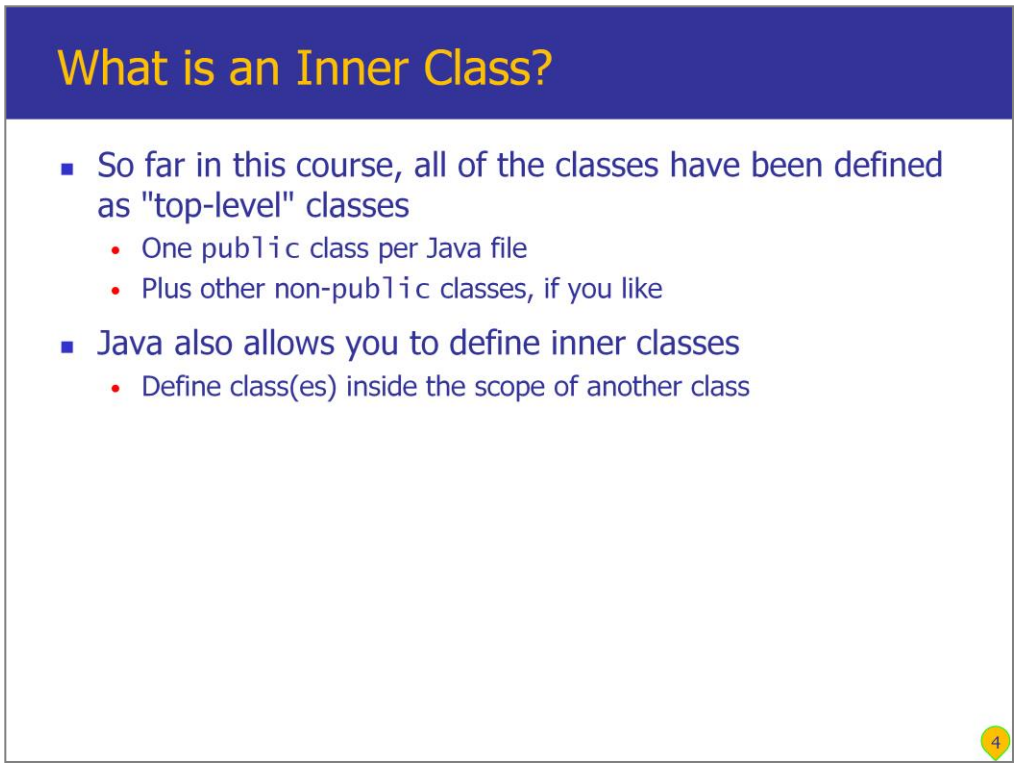
Section 1 gives a quick introduction to the concept of inner classes. We also describe some essential terminology, to set the scene for the rest of the chapter.

The remaining sections look at each of the different types of inner classes, one at a time. Each option has slightly different syntax but dramatically different usage.

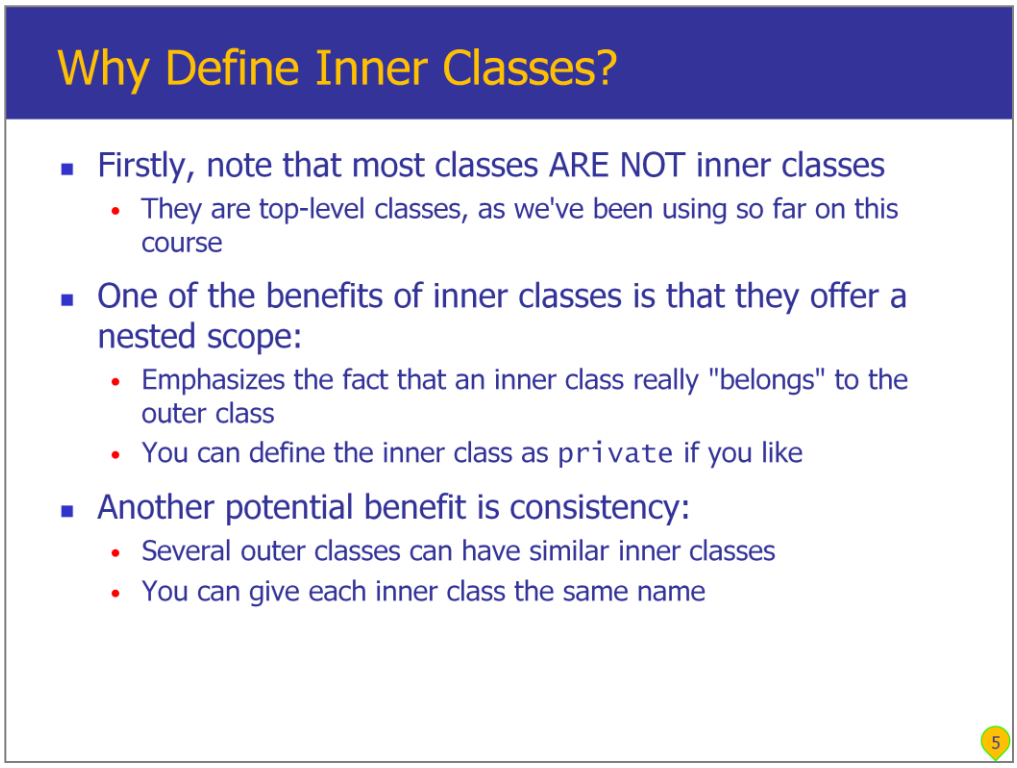
The demos for chapter are located in the `DemoInnerClasses` project.



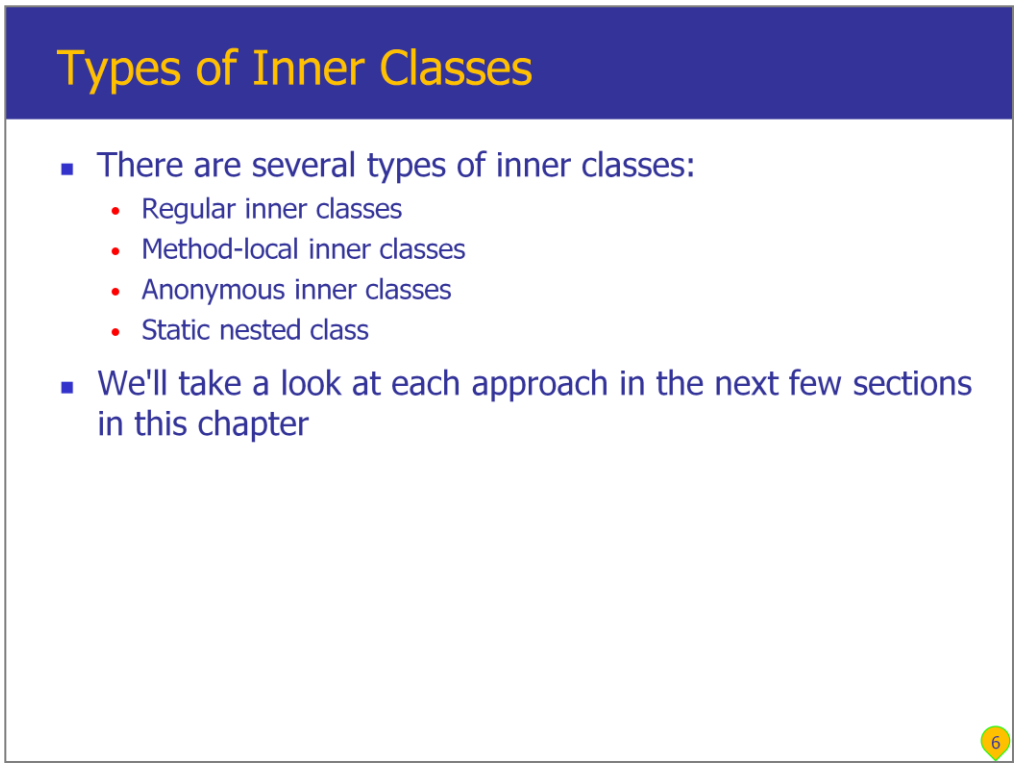
Let's get started. This goal of this section is to outline the different types of inner classes available in Java, so that we can drill into the details in the rest of the chapter.



In a nutshell, an inner class is a class that is defined inside another class or method. There are various reasons for doing this, e.g. to constrain the visibility of a class, to avoid a proliferation of public classes in the system, etc.

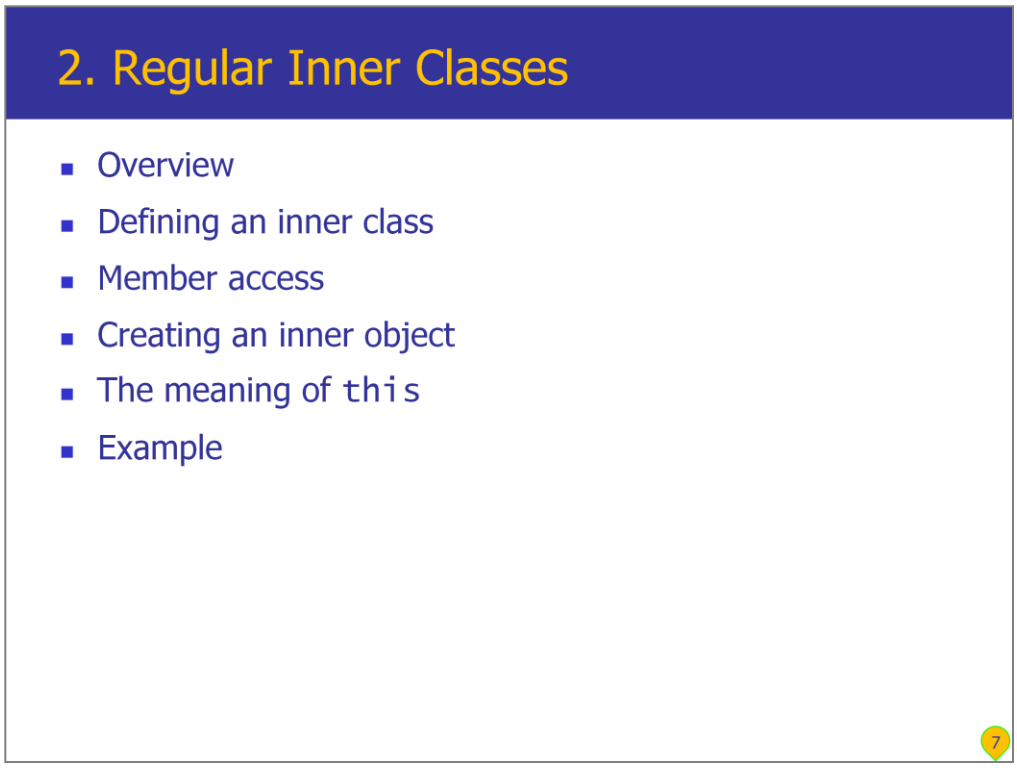


Large systems can contain hundreds or even thousands of classes. Anything you can do to encapsulate some of this detail, i.e. to hide classes from other parts of the system, is a positive outcome. Inner classes give us a convenient way to do this.



Java had four different ways to define inner classes, as outlined above. We'll explore each mechanism separately during this chapter.

Also note that Java 8 supports lambda expressions, which take the notion of anonymous inner classes to a new level. Lambda expressions are out of the scope of this course.



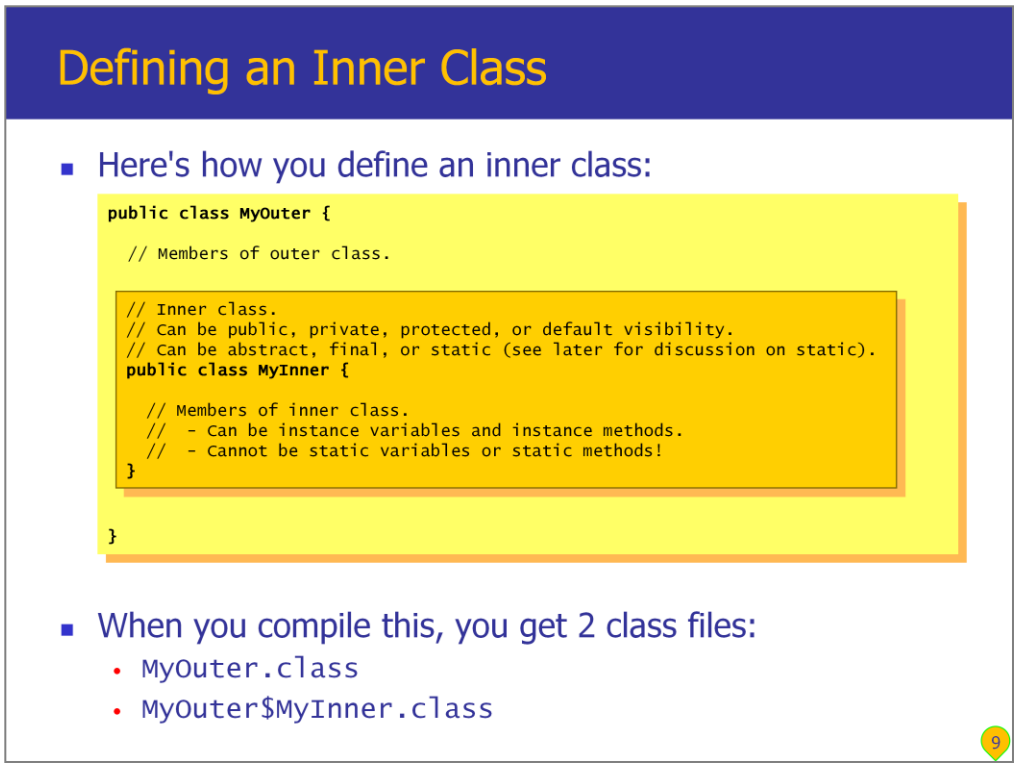
Regular inner classes are probably the simplest way to define inner classes, so this is where we'll start.

Overview

- Sometimes, when you're defining a class, you might find yourself wanted to put some behaviour in a separate dedicated class
  - E.g. to define event-handlers in a GUI application…
  - … you need to define a class that implements the event interface
- The traditional approach:
  - Define 2 completely separate top-level classes
- A more encapsulated approach:
  - Define the "subservient" class inside the "main" class

8

This slide presents a common scenario in programming. Inner classes allow you to define subservient classes inside another class, for reasons of encapsulation.



This slide shows how to define inner classes. Note the rules for defining the inner class, plus the rules for defining members in the inner class.

When you compile the code shown in the slide, you get two separate .class files:

- `MyOuter.class` contains the byte codes for the outer class.
- `MyOuter$MyInner.class` contains the byte codes for the outer class.

Member Access

- In the outer class:
  - Instance methods can access `public` members of inner class
  - Instance methods can't access non`public` members of inner class
  - Static methods can't access anything on inner class
- In the inner class:
  - Can access all the members of the outer class, even if `private`!
  - Allows inner class to interact with its outer class meaningfully

```
public class MyOuter {
  …
  private int anOuterField;

  public class MyInner {
    …
    public void anInnerMethod() {
      anOuterField++;
    }
  }
}
```

10

This slide describes how the outer class can access members in the inner class. These rules seem rather intuitive.

Also note the rules for how the inner class can access members in the outer class. These rules are a lot less intuitive, and might come as quite a surprise. The following slides will help you understand how this all fits together.

Creating an Inner Object (1 of 3)

- When you have an inner class:
  - The inner object MUST be associated with an outer object
- So you can't just create a raw inner object on its own:

```
// Error!
MyInner inner = new MyInner();
```

- Instead, you must create the inner object via the context of an outer object
  - Various ways to do this, see following slides

11

With inner classes, the important thing to understand is what happens with the objects. The essential point is that you can't create an orphan inner object; instead, the inner object must be associated with an instance of the outer class. It's like a spaceship being docked to the mother-ship.

Creating an Inner Object (2 of 3)

- You can define methods in the outer class, with the responsibility of creating inner objects

```
public class MyOuter {
  …
  public MyInner createInner() {
    return new MyInner();
  }
}
```

- Client code:
  - Note the use of `MyOuter.MyInner` to access inner class name

```
// Create an instance of the outer class.
MyOuter myOuter = new MyOuter();

// Use the outer object to create an associated inner object.
MyOuter.MyInner myInner1 = myOuter.createInner();
```

12

The typical arrangement is for the outer class to create an associated instance of the inner class, as shown in the first code block.

The second code block shows client code. We create an instance of the outer class, and once we have the outer object, we invoke its `createInner()` method to create an associated instance of the inner class.

Creating an Inner Object (3 of 3)

- You can create an inner object manually in the client code
  - Using the syntax `outerObject.new`

```
// Create an instance of the outer class.
MyOuter myOuter = new MyOuter();

// Manually create an inner object, associated with an outer object.
MyOuter.MyInner myInner2 = myOuter.new MyInner();
```

- You can create an inner object directly after creating the outer object
  - Using the syntax `new OuterClass().new InnerClass()`

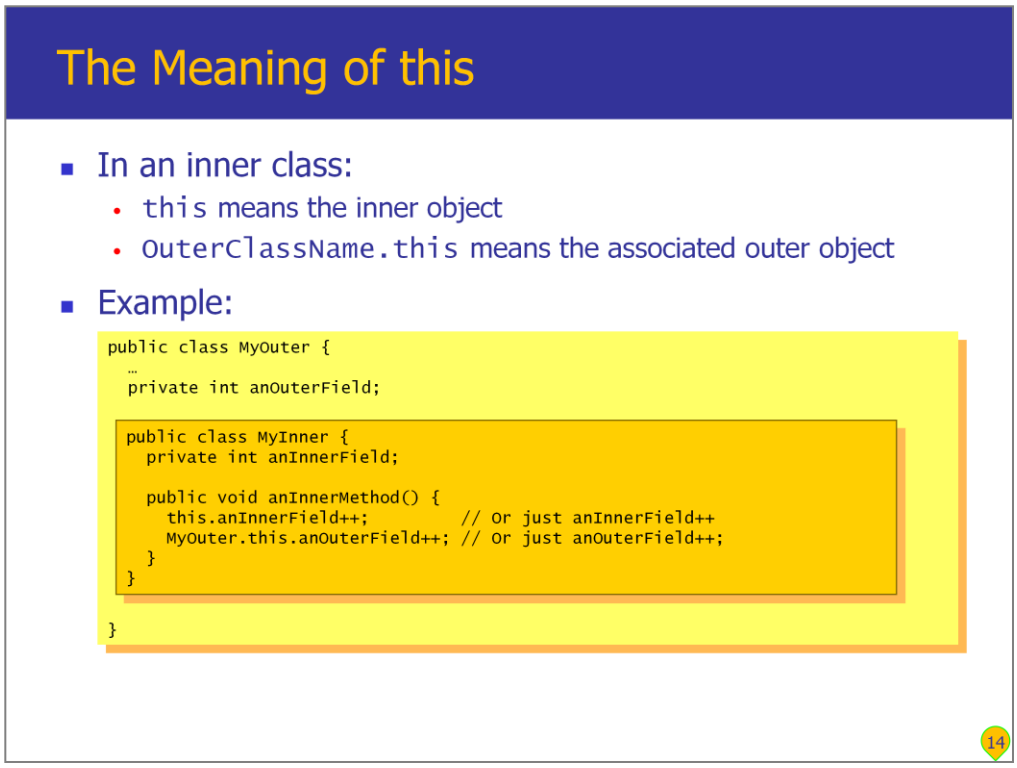
```
// Create an outer object and an inner object "at the same time".
MyOuter.MyInner myInner3 = new MyOuter().new MyInner();
```

13

Another technique is to allow the client code to create an inner object directly. This is a two-stage process, as illustrated in the first code box in the slide:

- First, create an instance of the outer class. Remember, you always need an instance of the outer class before you can create an instance of the inner class.
- Once you have the outer object, you can use the syntax `outerObject.new` to create an instance of an inner object. This syntax looks quite unusual!

The second code block performs both these steps in a single statement. It creates an instance of the outer class and then immediately creates an instance of the inner class. This looks a bit more intuitive perhaps.



Here's another curveball for you to wrestle with… When you're implementing a method in an inner class, you have two "this" references available:

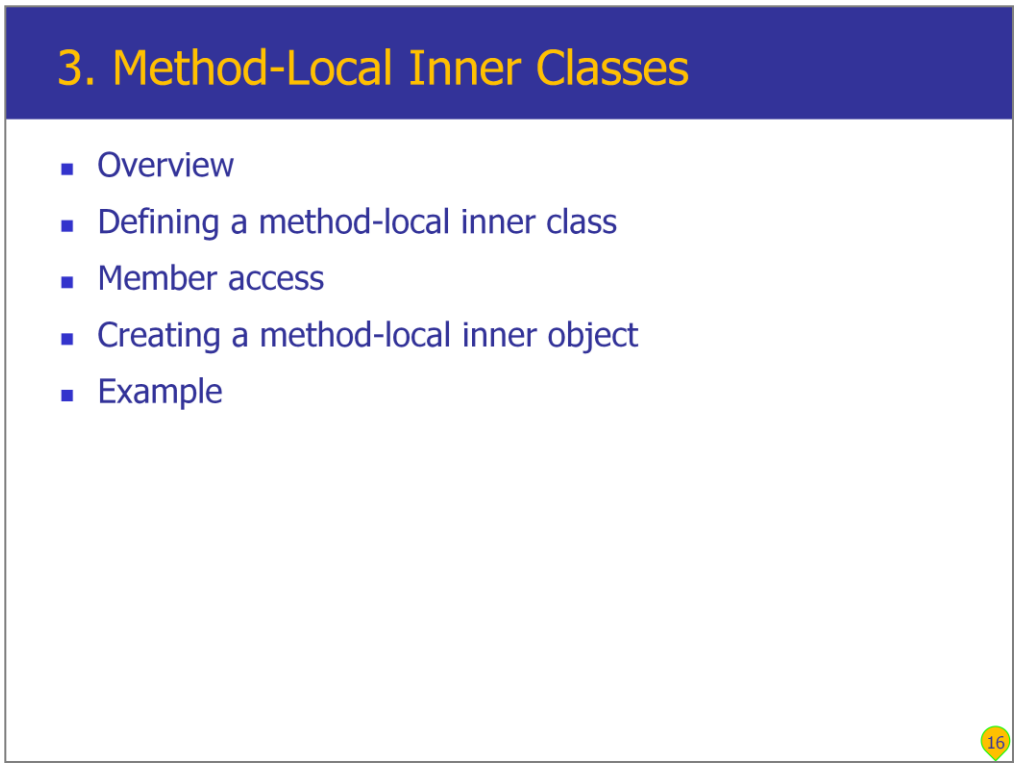
- `this` is a reference to the inner object
- `OuterClassName.this` is a reference to the associated outer object

Example

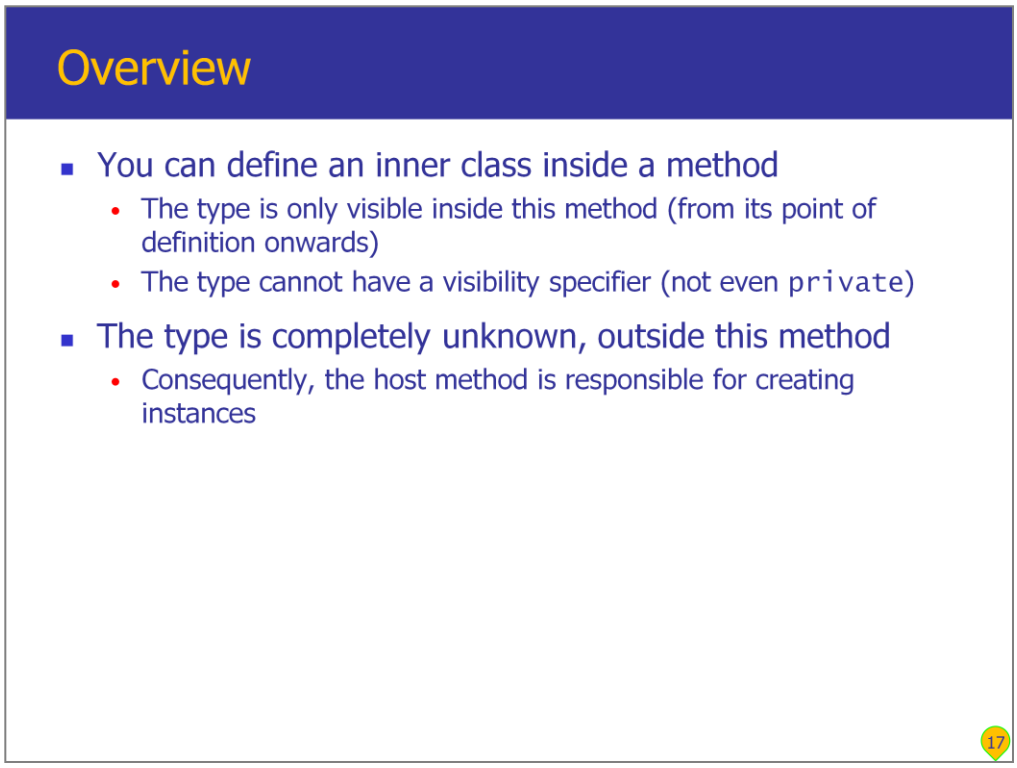
- For a complete example, see:
  - `MyOuterWithInnerClass.java`
- For usage, see:
  - `Main.java`
  - `demoInnerClasses()`

15

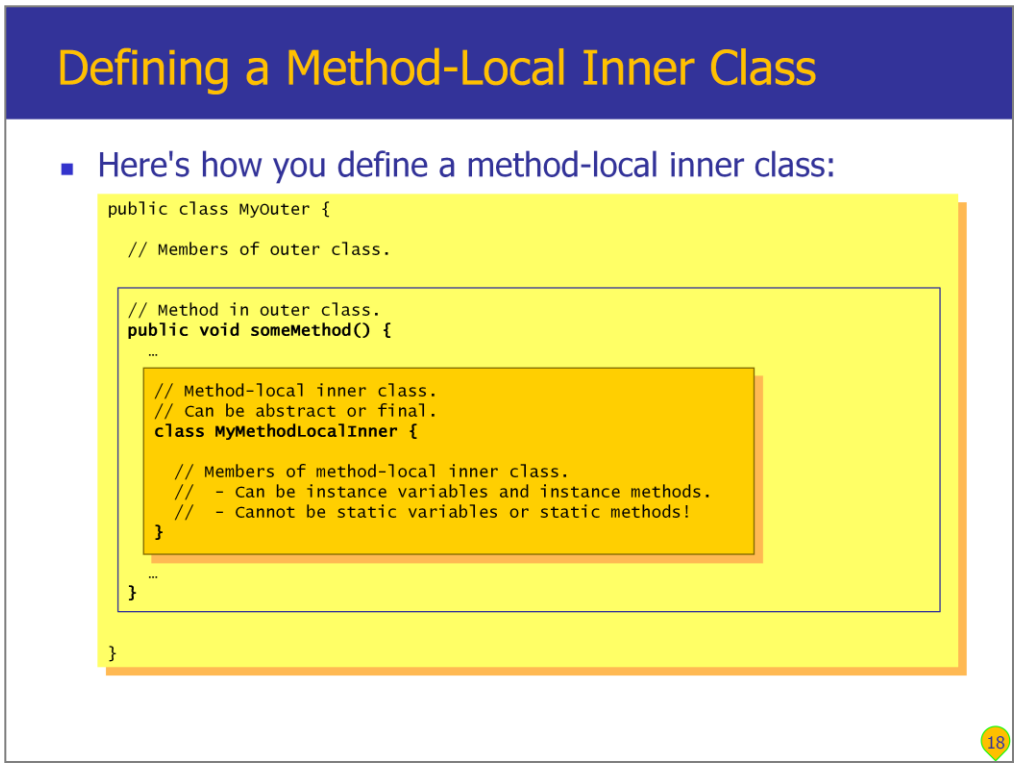
For a full example of how to define and use regular inner classes, see the demo code as described in the slide.



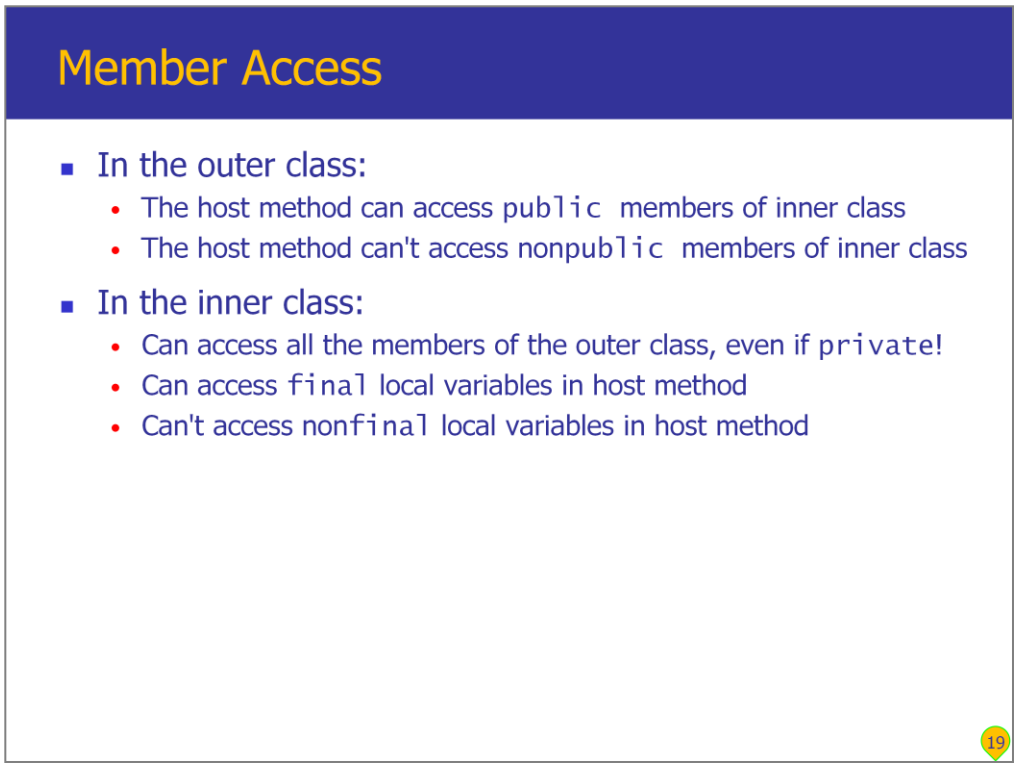
Believe it or not, you can actually define a class inside a method. We'll describe how to do this in this section, then we'll take it a step further in the next chapter when we look at anonymous local classes (which are probably more useful).



Method-local inner classes are defined completely inside a method, and are completely encapsulated by that method. Code outside the method has no notion that the class exists.



This slide shows how to define a class inside a method. Note that the inner class cannot have an access modifier, not even `private`. Also note that the inner class is not allowed to have any `static` members.



When you define a class inside a method, the rest of the code in that method can access the `public` members of the class (but not the non-`public` members of the class).

As regards the inner class, it can access all the class-level members of the outer class (even the `private` ones). However, if it wants to access any of the local variables in the host method, then those variables must be declared as `final` (i.e. the inner class can only access constant variables in the host method).

Creating a Method-Local Inner Object

- The only way to create a method-local inner class:
  - … is inside the host method
  - … after the inner class definition

```
public class MyOuter {

  // Members of outer class.

  // Method in outer class.
  public void someMethod() {
    …
    class MyMethodLocalInner {
      …
    }

    // Create instance of inner class.
    MyMethodLocalInner myInner = new MyIMethodLocalnner();
    …
  }

}
```

20

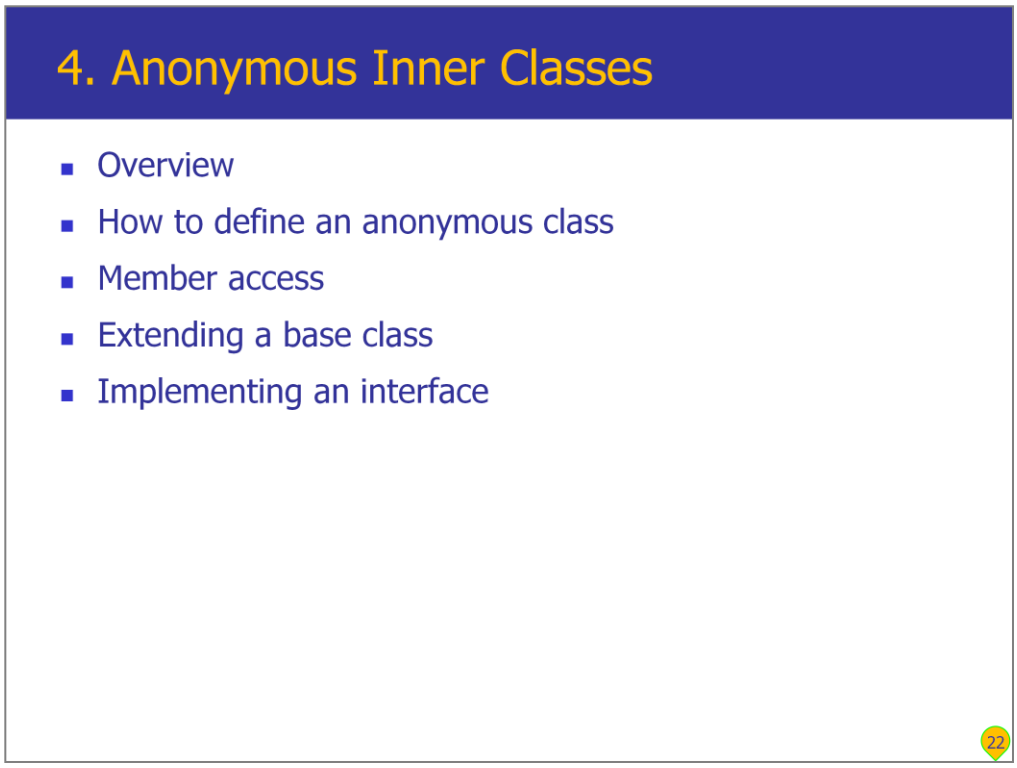
The typical arrangement is for the host method to create an instance of the inner class, as shown in the code above. The host method can then invoke members on the inner instance, as necessary.

Example

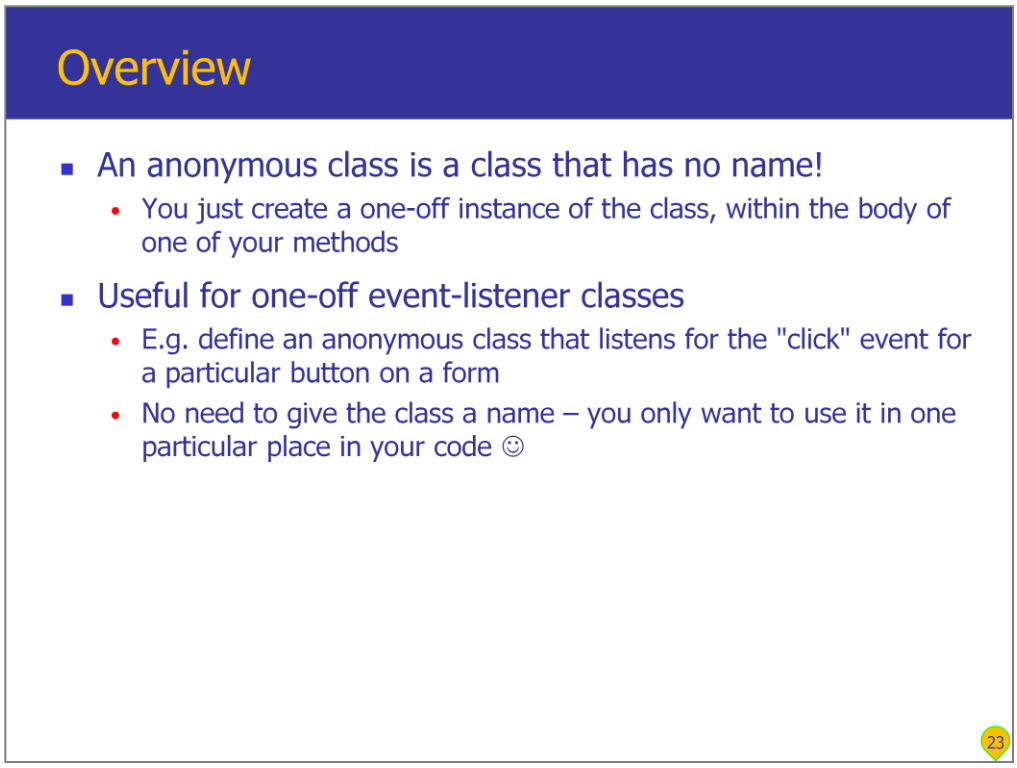
- For a complete example, see:
  - `MyOuterWithMethodLocalInnerClass.java`
  - `hostInner()`
- For usage, see:
  - `Main.java`
  - `demoMethodLocalInnerClasses()`

21

For a full example of how to define and use method-local inner classes, see the demo code as described in the slide.



Anonymous inner classes are similar to method-local inner classes, in the sense that you define a class inside a host method. The difference is that anonymous inner classes don’t have a name. You'll see how this actually works in this section.



The main reason for defining anonymous inner classes is to define one-off event listener classes to handle a specific event.

In Java, the event mechanism is based on interfaces. If a component (e.g. a button) raises events, there's an associated interface that describes the methods the client code must implement to handle these events. These interfaces are often called "listener" interfaces. For example, button clicks are associated with the `ActionListener` interface:

```
public interface ActionListener {
  void actionPerfomed(ActionEvent e);
}
```

Imagine you want to handle a particular button click; to do this, you define a class that implements the `ActionListener` interface. Now imagine you have 5 buttons that require different handling; now you've got to define 5 separate classes, each of which implements `ActionListener` in its own way. The net result is a proliferation of event-listener classes.

Anonymous inner classes avoid the need for global classes to handle events. You can define each event listener as a one-off anonymous class, in-line, without the need for yet another global class cluttering up the global namespace.

How to Define an Anonymous Class

- An anonymous class must either:
  - Inherit from a base class
  - … and override some methods (if it likes)
- Or:
  - Implement an interface
  - … and implement all its methods
- So when you create an instance of an anonymous class, what you are really doing is:
  - Defining a one-off anonymous class (that either inherits from a base class or implements an interface)
  - Creating an instance of that anonymous class

24

The syntax rules for defining an anonymous class requires you to stipulate either a base class that the anonymous class extends, or an interface that the anonymous class implements.

Member Access

- Exactly the same as for method-local inner classes…
- In the outer class:
  - The host method can access `public` members of inner class
  - The host method can't access non`public` members of inner class
- In the inner class:
  - Can access all the members of the outer class, even if `private`!
  - Can access `final` local variables in host method
  - Can't access non`final` local variables in host method

25

Apart from the obvious fact that anonymous classes don't have a name (and that they must specify a superclass or interface), they are the same as for method-local inner classes in all other respects.

Extending a Base Class

- An anonymous inner class that extends a base class…

```
public class MyBase {
  …
}
```

```
public class MyOuter {

  // Members of outer class.

  // Method in outer class.
  public void someMethod() {
    …
    MyBase obj = new MyBase() {
      // Override base-class methods, as appropriate.
    };

    // Invoke methods on obj, as exposed by base class.
    …
  }
}
```

26

This example shows how to define an anonymous class. When we create a new instance of the anonymous class, we indicate that the anonymous class inherits from `MyBase`. This means we can use `MyBase` as a suitable type for the local variable, which will hold a reference to the instance of our anonymous class.

Inside our anonymous class body, we can override methods defined in the base class. In the rest of the client code in the host method, we can invoke these methods (plus inherited members) on the object.

Implementing an Interface

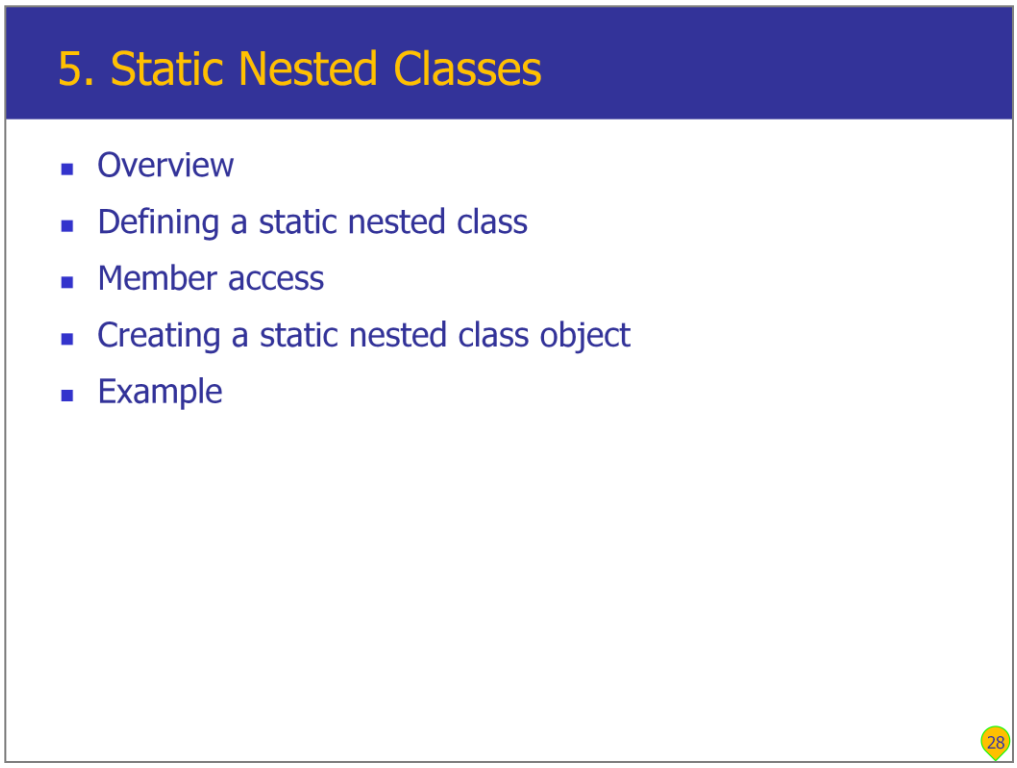
- An anonymous inner class that implements an interface…

```
public interface MyInterface {
  …
}
```

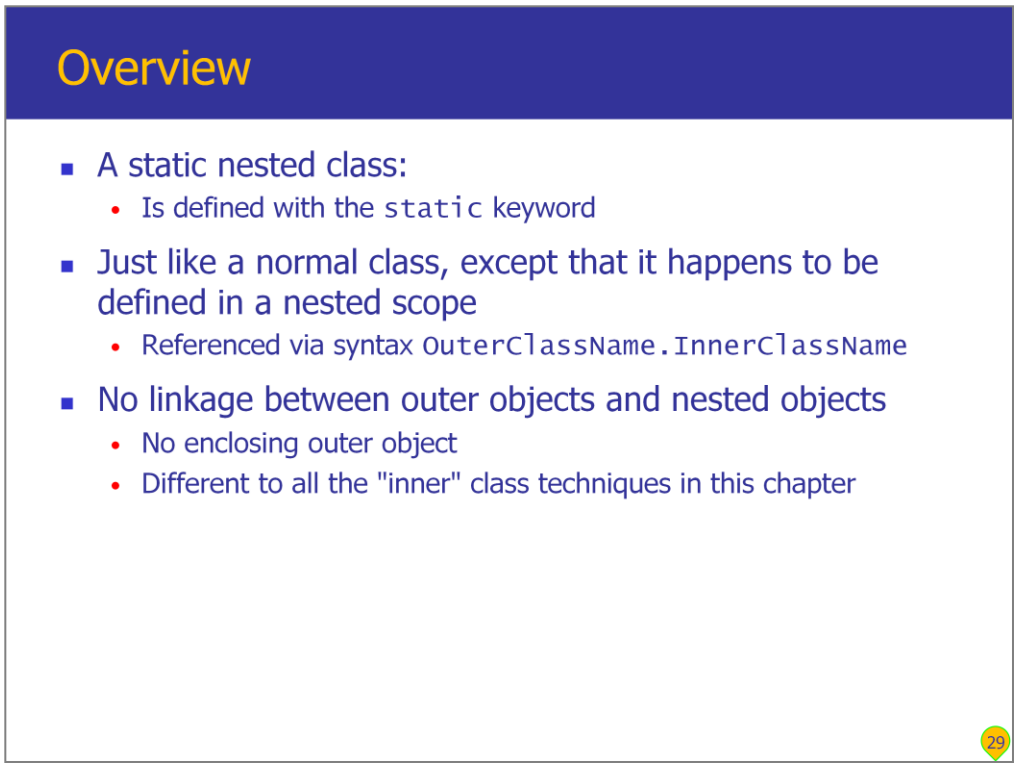
```
public class MyOuter {

  // Members of outer class.

  // Method in outer class.
  public void someMethod() {
    …
    MyInterface obj = new MyInterface() {
      // Implement interface methods.
    };

    // Invoke methods on obj, as exposed by interface.
    …
  }
}
```

27

This example is similar to the previous slide, except that the anonymous class now implements an interface rather than extending a class.

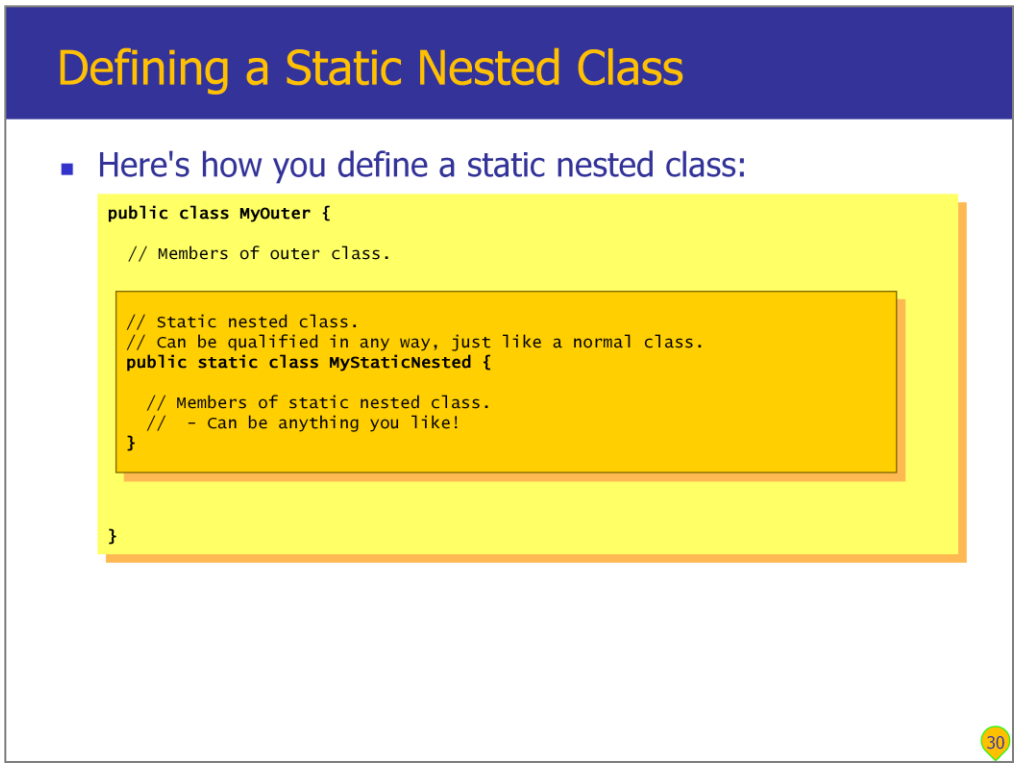


To conclude this chapter, we look at static nested classes. If you're familiar with C++, this is similar to the concept of nested classes in that language.

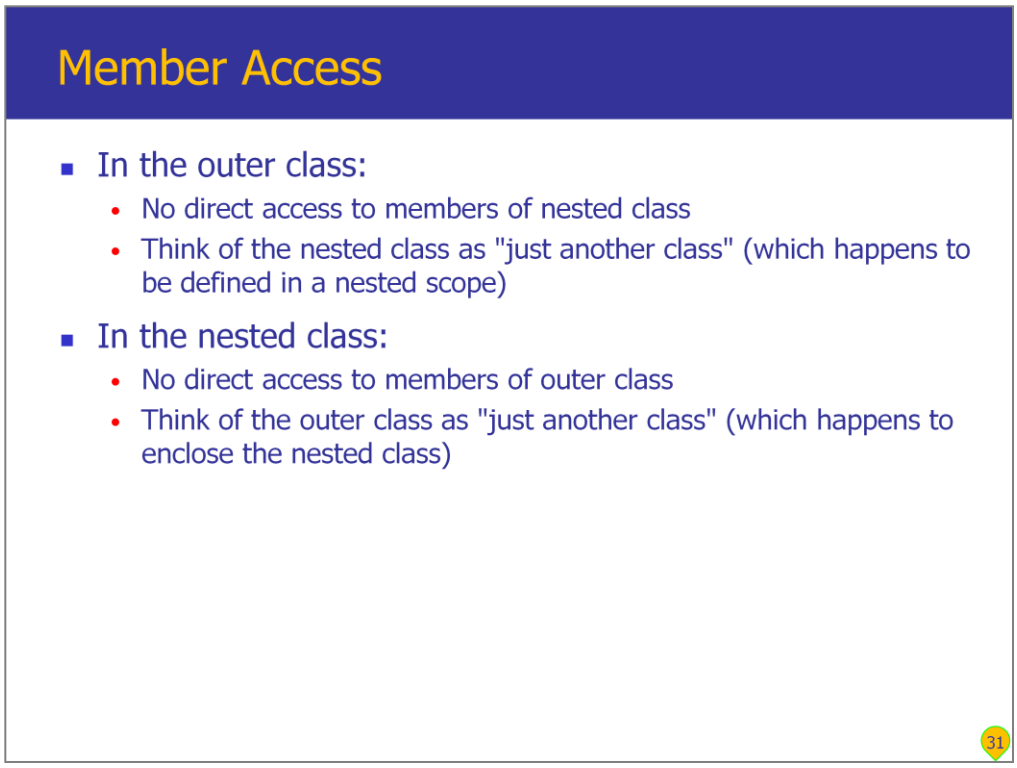


Syntactically, a static nested class is similar to a regular inner class (discussed at the start of this chapter), except that we qualify the class with the `static` keyword.

Behaviourally however, static nested classes are quite different to regular inner classes. Static nested classes do not impose any kind of linkage between an instance of the outer class and an instance of the inner class. Static nested classes purely define a scoping mechanism, whereby the inner class can be hidden from the outside world.



This example shows how to define a static nested class inside an outer class. Note the use of the `static` keyword now - this is the critical point here.



The rules for member access are completely decoupled between the outer class and the inner class. There's no trickery involved here.

Creating a Static Nested Object (1 of 2)

- You can define methods in the outer class, with the responsibility of creating static nested objects

```
public class MyOuter {
  …
  public MyStaticNested createInner() {
    return new MyStaticNested();
  }
}
```

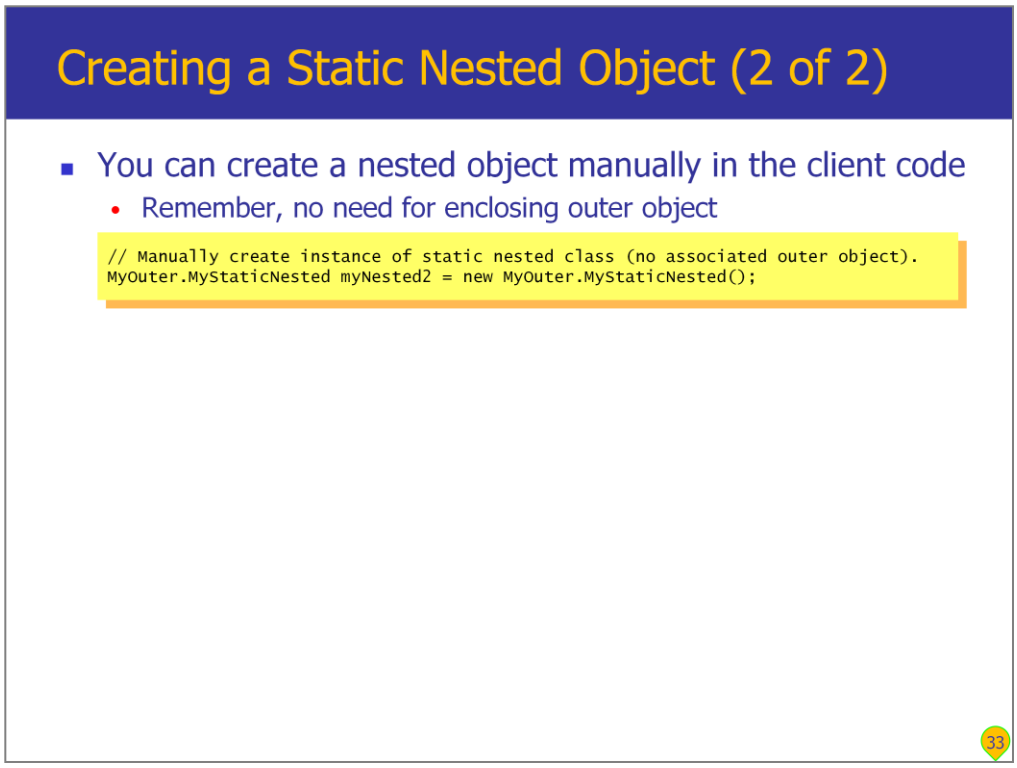
- Client code:
  - Note the use of syntax `MyOuter.MyStaticNested`

```
// Create an instance of the outer class.
MyOuter myOuter = new MyOuter();

// Use the outer object to create an (unrelated) instance of static nested class.
MyOuter.MyStaticNested myNested1 = myOuter.createNested();
```

32

A common arrangement is for the outer class to create a instance of the static nested class, as shown in the first code block.

The second code block shows client code. We create an instance of the outer class, and then invoke its `createNested()` method to create a instance of the static nested class. There is no umbilical cord linking the outer and inner objects - they are two completely separated objects.



Client code is allowed to instantiate the static nested class directly if it wants to, provided the static nested class isn't `private` of course.

The example in the slide shows how to do this. Note that we don't have to create an accompanying instance of the outer class - this is different from the regular inner class approach that we looked at earlier in the chapter.

Example

- For a complete example, see:
  - `MyOuterWithStaticNestedClass.java`
- For usage, see:
  - `Main.java`
  - `demoStaticNestedClasses()`

34

For a full example of how to define and use static nested classes, see the demo code as described in the slide.

Any Questions?