This is the second of two chapters on the subject of inheritance.

The previous chapter described how a class can extend another class, to benefit from inheritance of implementation. In this chapter, we show how a class can implement any number of interfaces.

Interfaces play a crucial role in Java applications.

Contents

1. Overview of interfaces
2. Defining and implementing interfaces
3. Using interfaces in the client

Demo project: **DemoInterfaces**

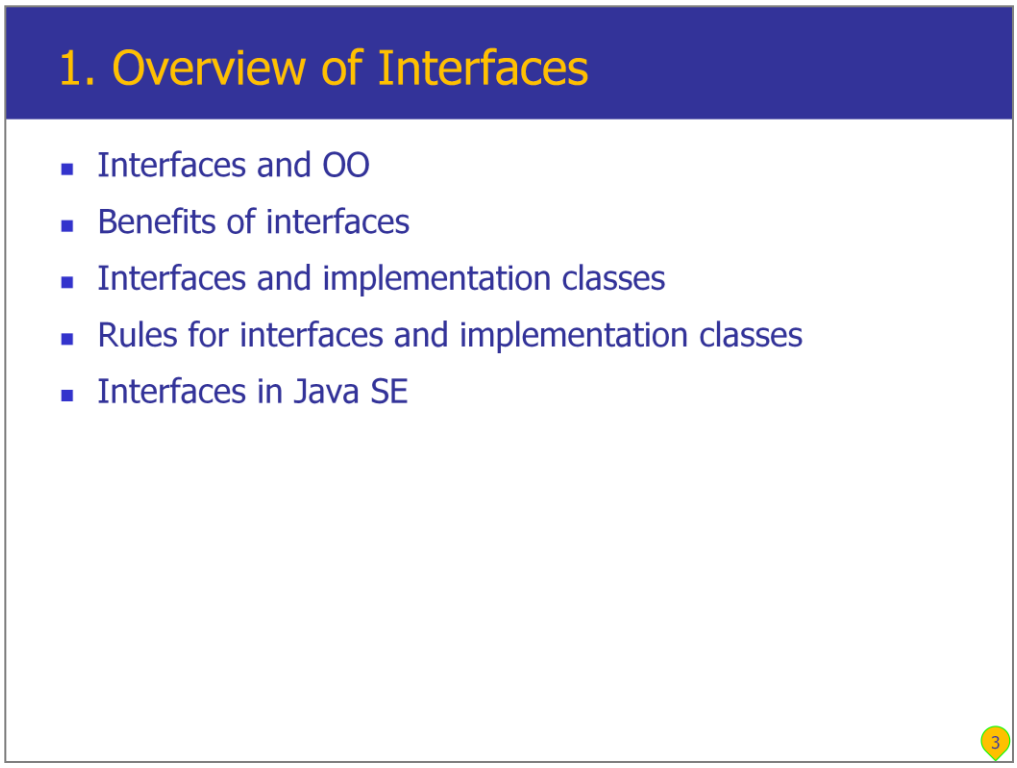
2

Section 1 describes what an interface is, and explains why interfaces are so useful in object oriented development.

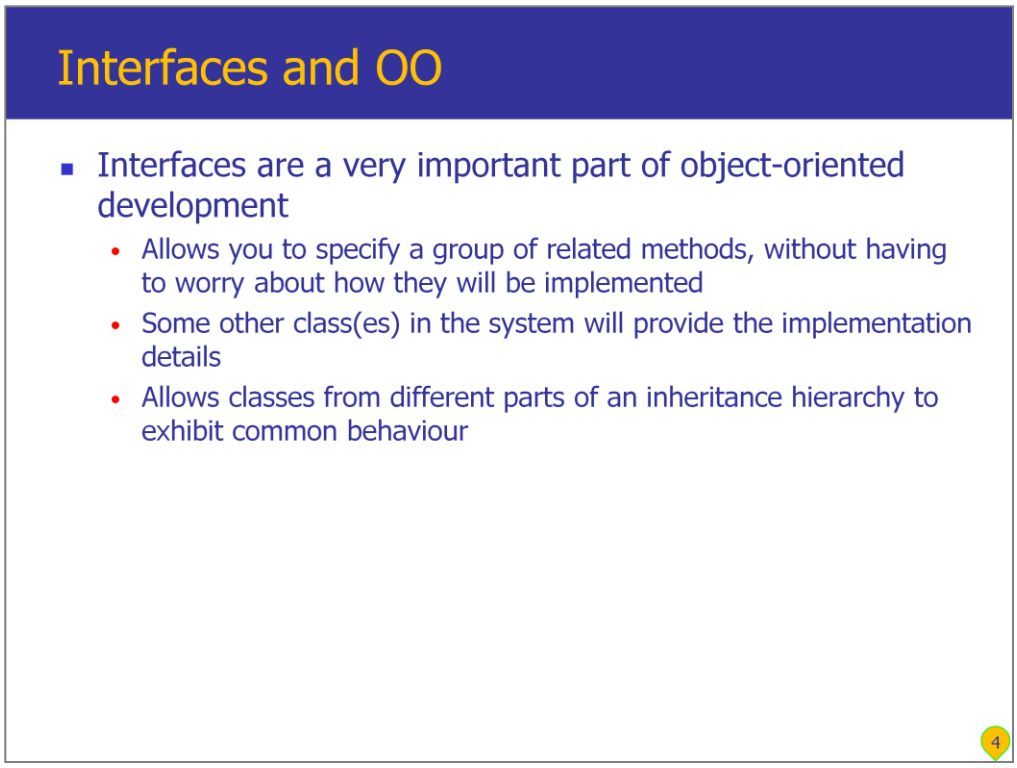
Section 2 shows how to define and implement interfaces in Java. As you'll see, there's an interface keyword that allows you to define interfaces, It's a bit like defining a class, but with empty methods everywhere.

Section 3 outlines the importance of interfaces as far as the client code is concerned. This section will reinforce your understand of the role of interfaces in well-structured object oriented applications.

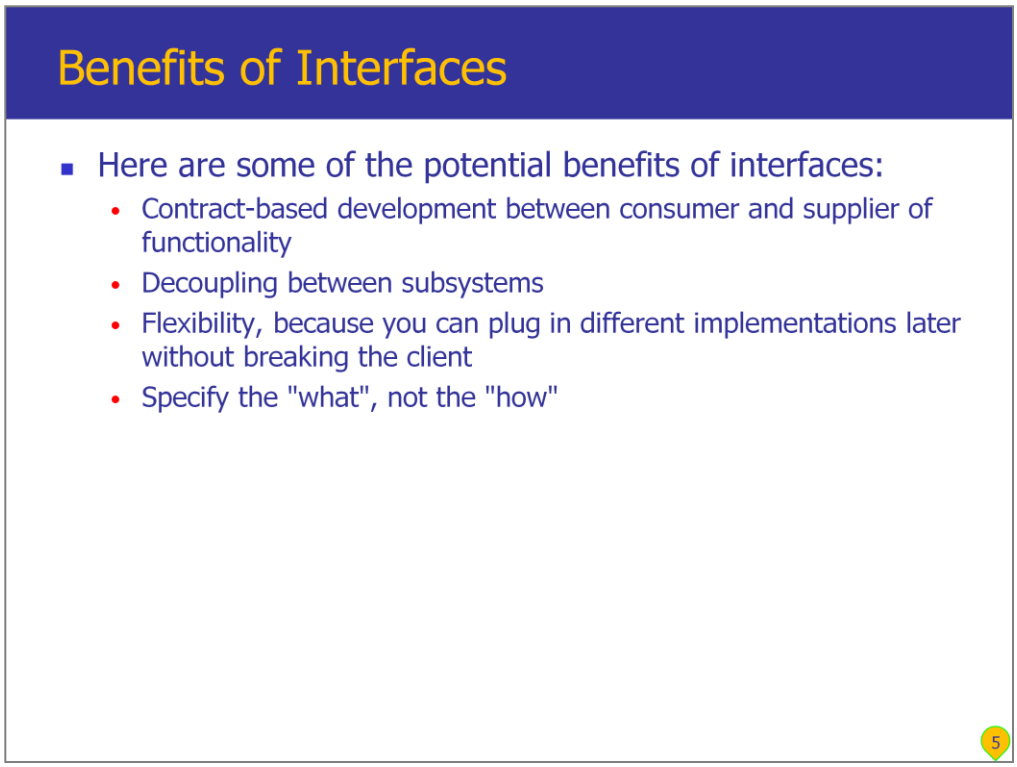
The demos for chapter are located in the `DemoInterfaces` project.



Let's begin our journey by explaining what interfaces are, and why they matter. You might feel interfaces are a bit pointless initially, but hang on in there... Interfaces are very important indeed, and the more experience you get as a Java / OO developer, the more you'll concur with this sentiment.

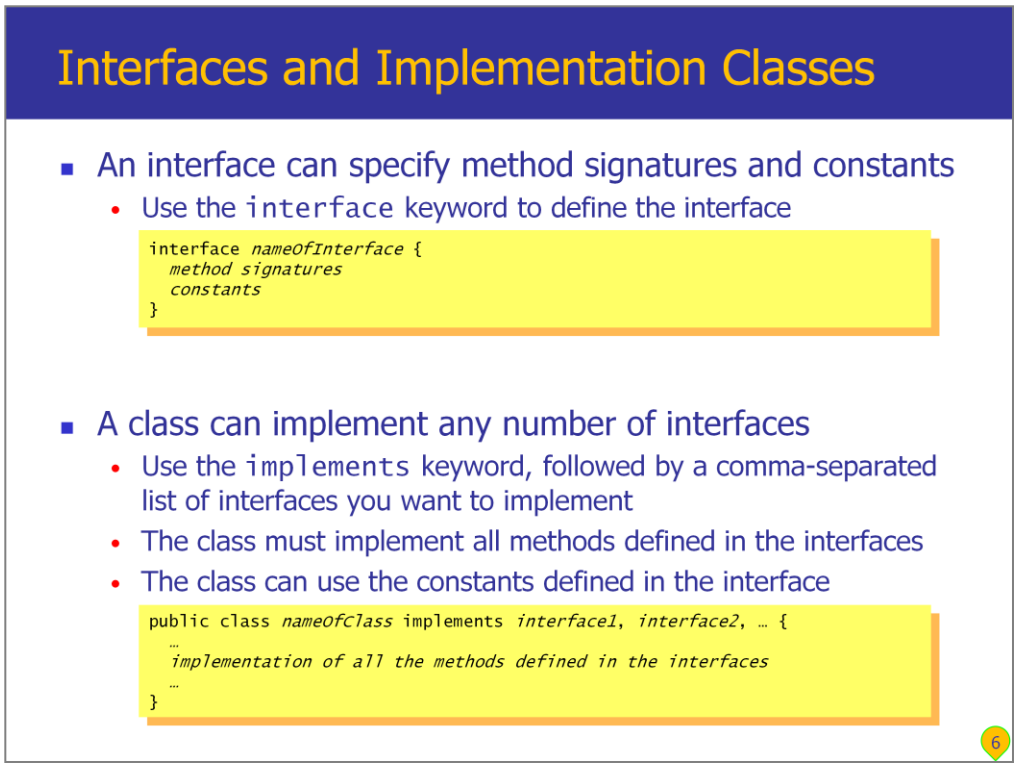


An interface specifies a group of method signatures. An interface doesn't make any assumptions about how these methods will be implemented; that's for other classes to worry about.



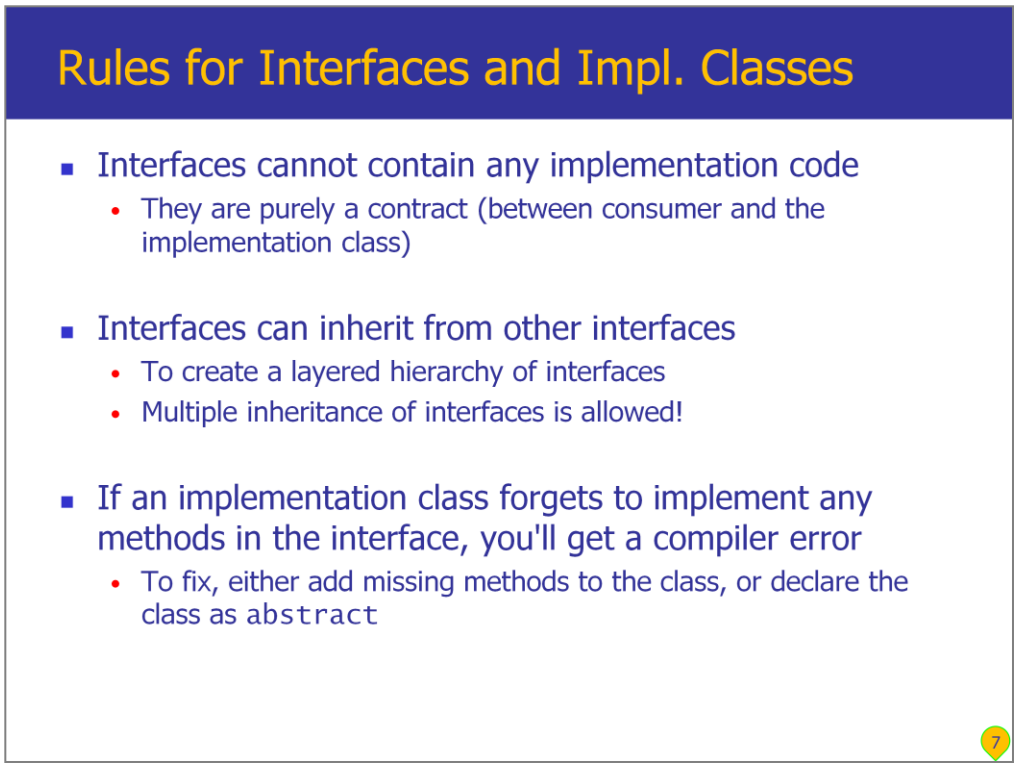
An interface is like a contract between a client developer and a class implementer:

- For client developers, an interface specifies what capabilities they can expect from classes that implement the interface. The client developer is unaware (and doesn't even care) how the methods will be implemented.
- For class implementers, an interface specifies what methods they need to implement. It's up to the class implementer to choose an appropriate implementation strategy, as he or she sees fit. Different classes can implement the interface in different ways, without affecting the client's view of proceedings.



Here's a quick summary of how to define and implement interfaces. We'll revisit this subject later in this chapter, where we'll present some more detailed rules:

- To define an interface in Java, you use the `interface` keyword, followed by the name of the interface.
- A class can implement any number of interfaces. To implement one or more interfaces, append the `implements` keyword after the class name, followed by a comma-separated list of interfaces that you want to implement.

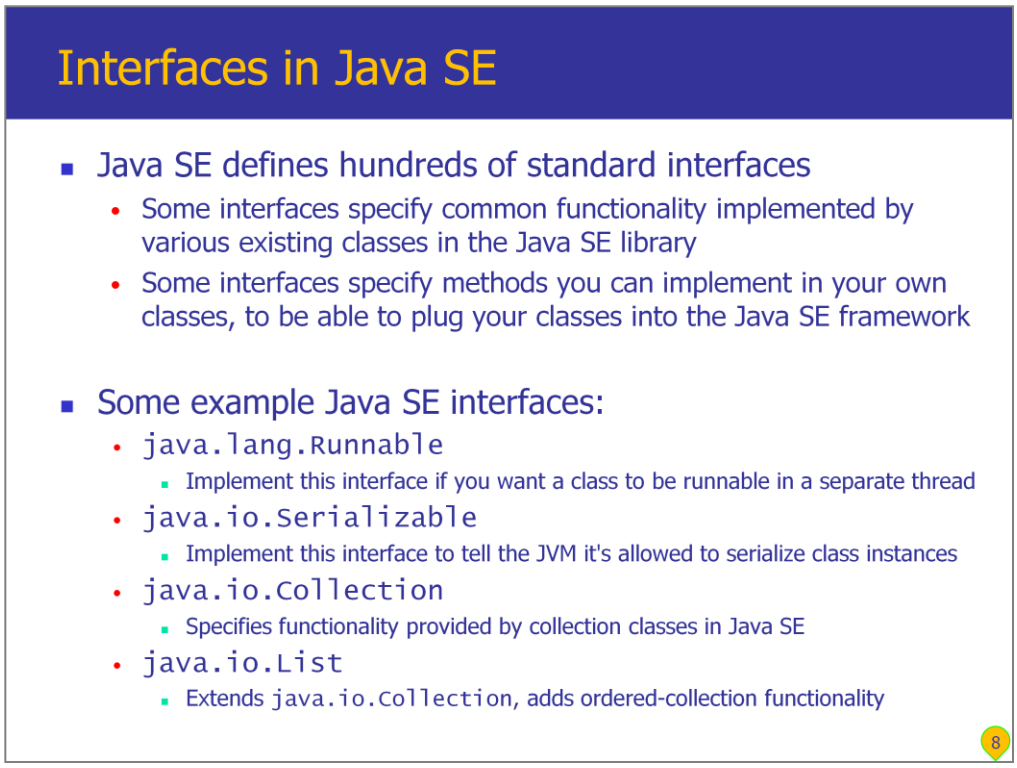


Interfaces are pure specification. They contain just method signatures and constants (i.e. `static final` values). Interfaces cannot contain method code, constructors, or instance variables.

Note that it is allowable for an interface to inherit from other interfaces, using the `extends` keyword. This allows you to create a hierarchy of interface types that layer additional contractual details upon each other.

When a class implements an interface, it must implement all the methods defined in that interface. If the implementation class neglects to implement any methods, then it is effectively an abstract class, and you must decorate it with the `abstract` keyword.

Note: Java 8 makes some drastic changes to the way interfaces work. In Java 8, interfaces can define default implementation bodies for methods. It's a revolution in the concept of interfaces. Java 8 interfaces are out of the scope of this course.



The Java SE library defines a large number of interfaces. You can see these interfaces when you view JavaDoc documentation. For example, take a look at the following web sites and you'll find plenty of interfaces to keep you busy:

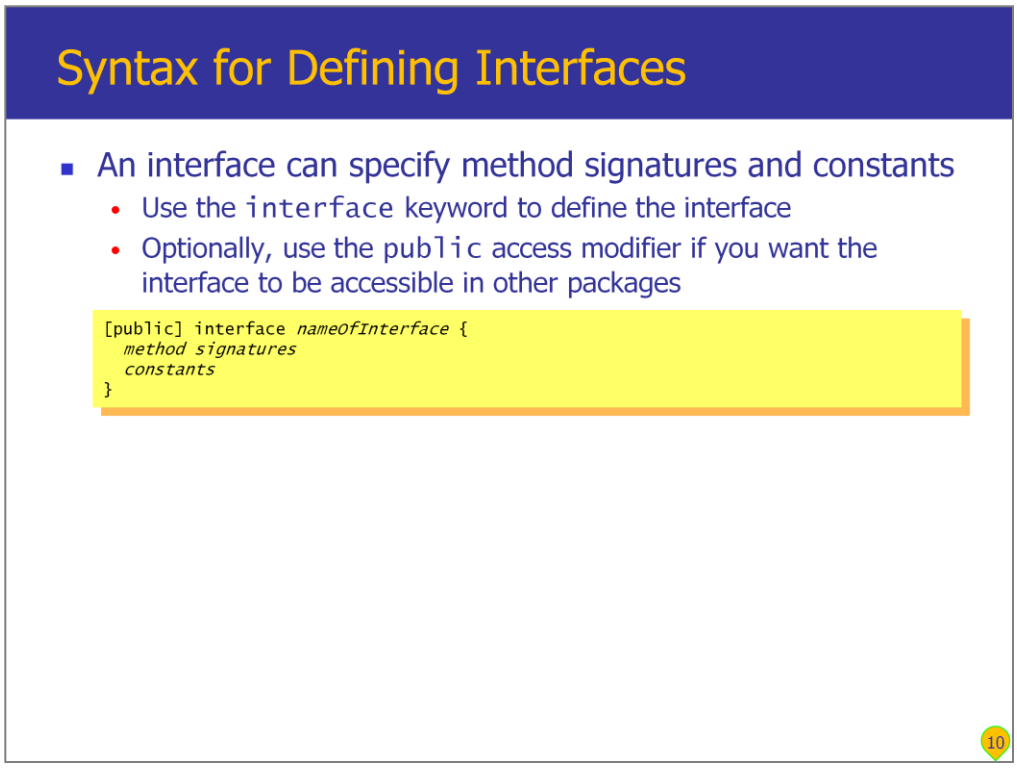
- http://docs.oracle.com/javase/7/docs/api/java/util/package-summary.html
- http://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html
- http://docs.oracle.com/javase/7/docs/api/java/sql/package-summary.html

2. Defining & Implementing Interfaces

- Syntax for defining interfaces
- Defining methods in an interface
- Defining constants in an interface
- Syntax for implementing interfaces
- Example implementation class
- Aside: marker interfaces

9

In this section we dissect the syntax for defining and implementing interfaces in Java 7. Note that the new rules introduced in Java 8 are out of scope for this chapter.



An interface definition is somewhat reminiscent of a class definition, except you use the `interface` keyword rather than the `class` keyword.

Note in particular that the same rules apply about where you define interfaces. A Java file can contain just one `public` interface, plus any number of non-`public` interfaces.

Defining Methods in an Interface

- An interface can specify any number of methods (including zero!)
  - All methods are implicitly `public` and `abstract`
  - The `public` and `abstract` keywords on methods are optional
- Example:

```
public interface Freezable {
  void freeze();
  void unfreeze();
}
```

Freezable.java

- Note: Interface methods must not be:
  - `private` or `protected`
  - `static`, `final`, `native`, or `strictfp`

11

This example shows a typical interface. Note the following points:

- The interface name ends in "able" to indicate it describes a slice of capability that can be implemented by other classes.
- It's a public interface, so it must reside in a Java file with the same name as the interface.
- The interface defines a small number of methods. This is quite typical - interfaces tend to have a relatively few number of methods, to capture the essence of a particular aspect of functionality.
- The interface doesn't contain any implementation code - just method signatures (and potentially constants, as shown in the next slide).

Defining Constants in an Interface

- An interface can specify any number of constants
  - All constants are implicitly `public static final`
  - These keywords on constants are optional
- Example:

```
public interface Loggable {

  PrintStream OUTPUT_STREAM = System.out;

  void logBrief();
  void logVerbose();
}
```

Loggable.java

- Note: Interface constants must not be:
  - `private` or `protected`

12

Interfaces can define constant values, to accompany the method signatures. This is relatively uncommon, but it has its place. Note that constants must be `public`.

The example in the slide shows how a `Loggable` interface can define (and initialize) a constant field to represent the output stream where log messages will be written.

Syntax for Implementing Interfaces

- A class can implement any number of interfaces
  - Use the `implements` keyword, followed by a comma-separated list of interfaces you want to implement
  - The class must implement all methods defined in the interfaces
  - The class can use the constants defined in the interface

```
public class nameOfClass implements interface1, interface2, … {
  …
  implementation of all the methods defined in the interfaces
  …
}
```

- Note:
  - A class can extend one class, as well as implementing interfaces

```
public class nameOfClass extends superclass
                         implements interface1, interface2, … {
  …
}
```

13

This slide formally presents the rules for implementing interfaces. Note in particular that a class can implement any number of interfaces, as well as extending a single class.

Example Implementation Class

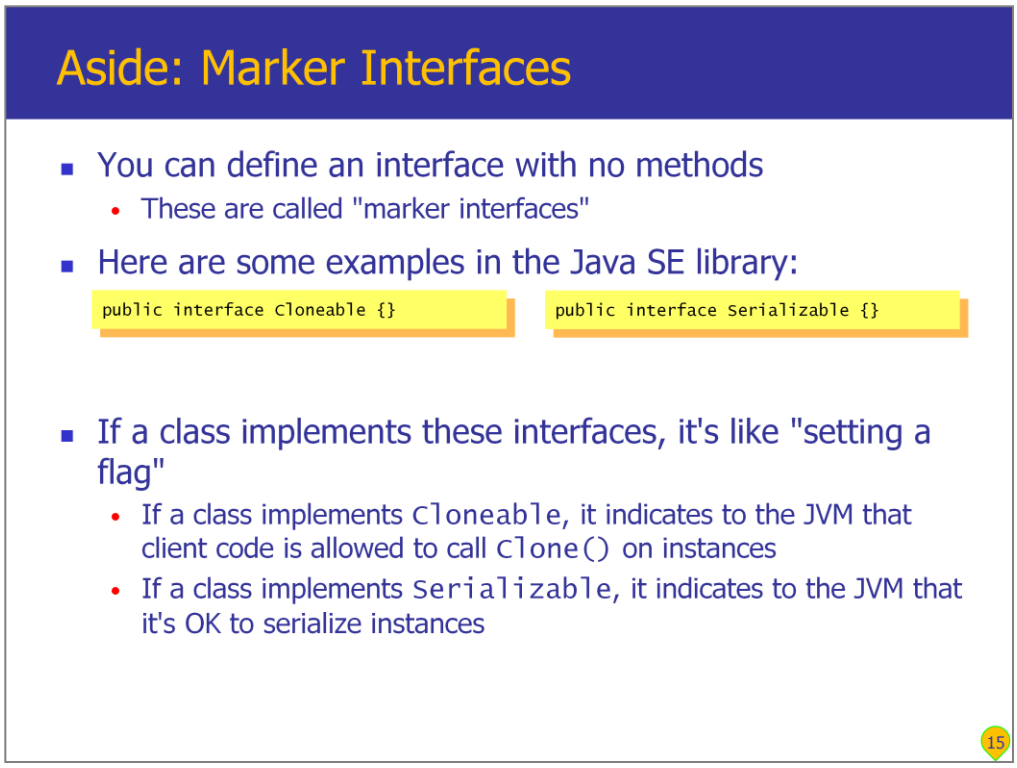
- Here's an example of a class that implements 2 interfaces
  - `Freezable` and `Loggable`

```
public class Calculator implements Freezable, Loggable {

  // Implementation of Freezable.
  public void freeze()   { … }
  public void unfreeze() { … }

  // Implementation of Loggable.
  public void logBrief()   { …}
  public void logVerbose() { … }

  // Plus other members, as for a normal class
  …
}
```

Calculator.java

14

This example shows how a class can implement several interfaces. Note that the class must fulfil its obligations, i.e. it must implement all the methods from all the interfaces

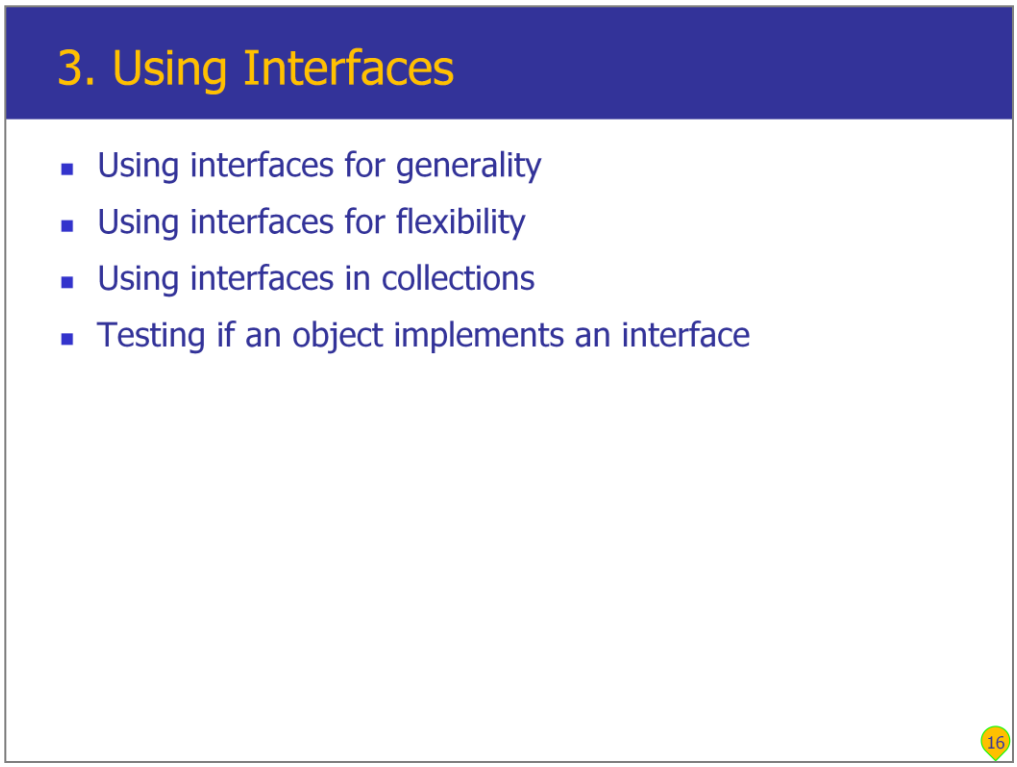


Just before we close our section on syntax, note that there are some interfaces in Java that are empty. These are known as "marker interfaces".

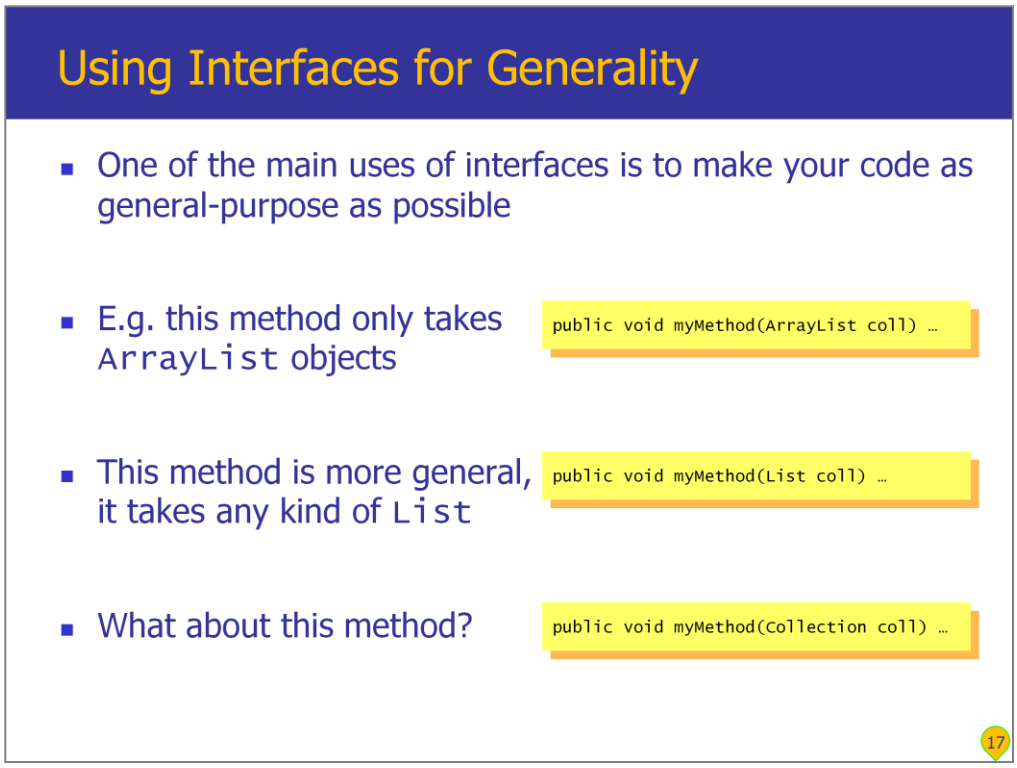
The idea here is that a class can implement such interfaces to indicate that the class offers some aspect of behaviour. For example, if a class implements the `Serializable` interface, it tells the JVM that this class has been designed in such a way that it makes sense to serialize objects to a "flat pack" format. If a class doesn't implement serializable, and you do try to serialize an object, the JVM will throw an exception.

Note that marker interfaces are a little old fashioned, now that Java supports annotations. For example, to indicate that a class is a web service, you use an annotation rather than a marker interface:

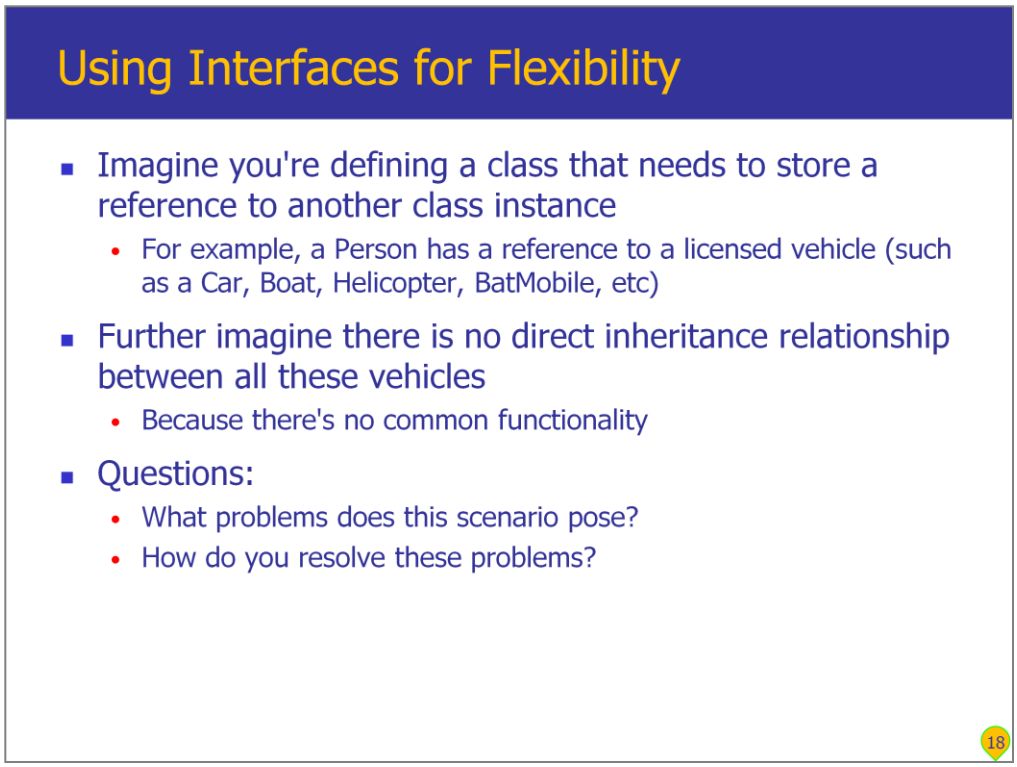
```
@WebService
public class MyWebService {
  …
}
```



This section describes why interfaces matter in OO design, and how they influence the structure of your client code.

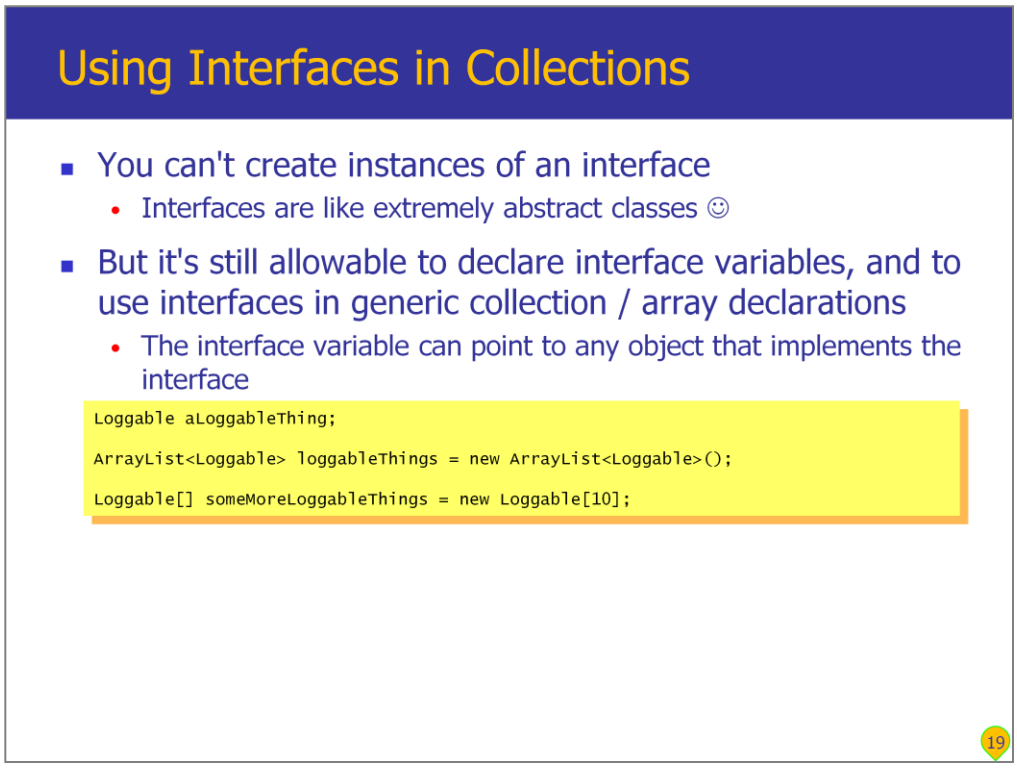


Where possible, method parameters should be as general as purpose. For example, if a method takes an interface parameter, it means the client code can pass in any kind of object that implements the interface. This makes the method more reachable in a wide range of circumstances.

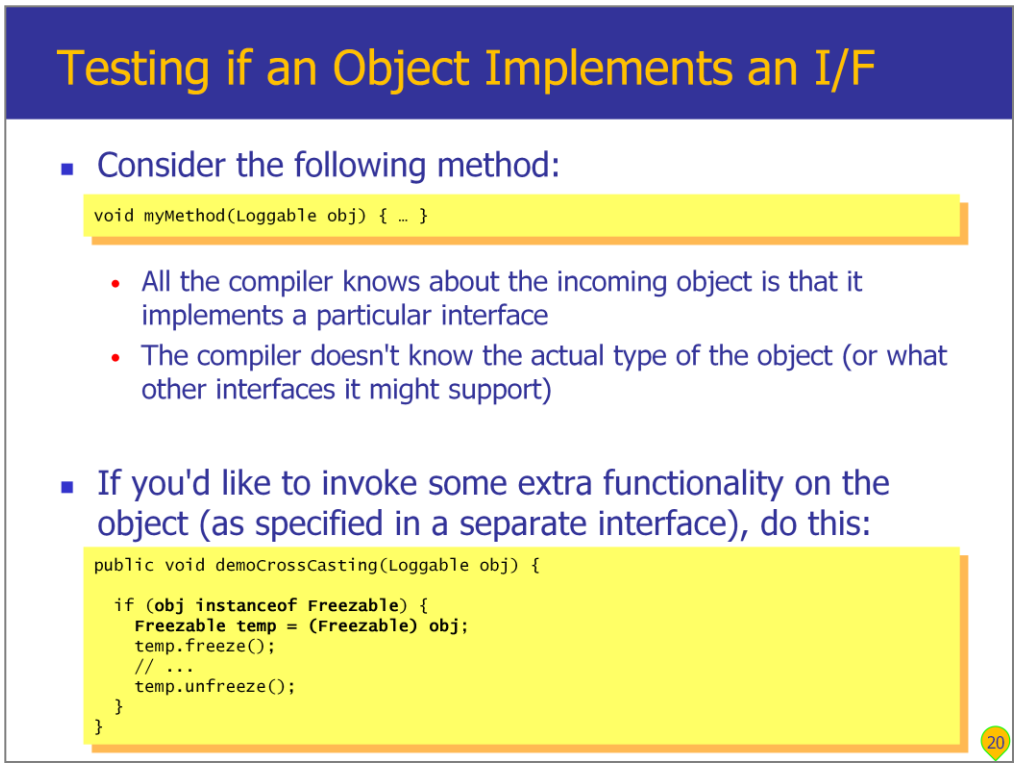


In a real application, objects need to hold references to other objects. This is known as association.

If possible, and if you want to keep your classes as decoupled as possible, then you should consider declaring some instance variables as interface types, rather than as class types. This means a object can hold a reference to any kind of object that implements the interface, which gives you more flexibility to plug in a different kind of object if you want to.



You can create a truly polymorphic collection by using interfaces. For example, the `loggableThings` collection and the `someMoreLoggableThings` array in the slide can hold any kind of object that implements the `Loggable` interface.



You can use the `instanceof` keyword to test whether an object implements a particular interface.

This is useful if a method receives an interface type as a parameter. The client code will pass in an object that implements the interface, but the object might implement other interfaces as well. If you want to utilize some of the capabilities of the other interface, you can test whether the object implements the other interface, and then cross-cast the object reference to that interface type. The example in the slide shows how to do this.

Any Questions?