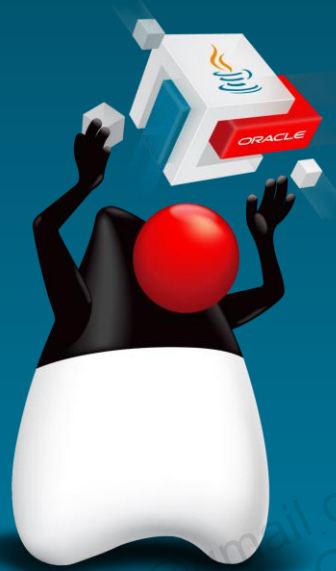


Java Interfaces



ORACLE



4

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Adolfo De+la+Rosa (adolfodelaros2012@gmail.com) has a non-transferable license to use this Student Guide.

Objectives

After completing this lesson, you should be able to:

- Use `default` methods in interfaces
- Identify when it's desirable to implement a `default` method in an interface
- Identify how inheritance rules apply to methods implemented in interfaces
- Use `private` methods in interfaces
- Identify when it's desirable to implement a `private` method in an interface
- Define anonymous classes

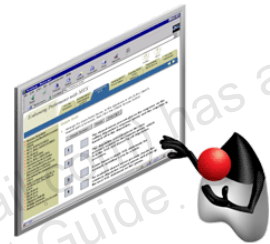


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Java Interfaces

Java interfaces are used to define abstract types. Interfaces:

- Are similar to abstract classes containing only public abstract methods
- Outline methods that must be implemented by a class
 - Methods must not have an implementation {braces}.
- Can contain constant fields
- Can be used as a reference type
- Are an essential component of many design patterns



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

In Java, an interface outlines a contract for a class. The contract outlined by an interface mandates the methods that must be implemented in a class. Classes implementing the contract must fulfill the entire contract or be declared abstract.

Java SE 7 Interfaces

- Interfaces are Java's solution to safely facilitate multiple inheritance.
- Interfaces originally contained only:
 - `static` variables
 - `abstract` methods

An example is the `Accessible` interface. This interface is meant to be implemented in classes for financial products where people access money through deposits and withdrawals.

```
public interface Accessible{  
    public static final double OVERDRAFT_FEE = 25;  
  
    public abstract double verifyDeposit(double amount, int pin);  
    public abstract double verifyWithdraw(double amount, int pin);  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

An example of a nonaccessible financial product would be a savings bond. You're not able to access the money after the bond is purchased.

Implementing Java SE 7 Interface Methods

`abstract` methods must be implemented later.

- If one class implements an interface, you'll write your implementation logic once.
- If many classes implement the same interface, you'll write your implementation logic many times.

What if most classes implement the exact same logic?

- Must you duplicate the same code in many places?
- Isn't code duplication bad?



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Example: Implementing `abstract` Methods

Notice the logic found in these methods:

```
public class BasicChecking implements Accessible{
    ...
    public double verifyDeposit(double amount, int pin){
        //Verify the PIN
        //Verify amount is greater than 0
    }
    public double verifyWithdraw(double amount, int pin){
        //Verify the PIN
        //Verify amount is greater than 0
        //Verify account balance won't go negative
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Example: Duplicating Logic

The same logic is largely duplicated by other classes that implement `Accessible`.

```
public class RestrictedChecking implements Accessible{  
    ...  
    public double verifyDeposit(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
    }  
    public double verifyWithdraw(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
        //Verify account balance won't go negative  
        //Verify the withdrawal is under the transaction limit.  
    }  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

You'll have a chance to work with both the `BasicChecking` and `RestrictedChecking` classes in this lesson's practices. `RestrictedChecking` is a financial product that offers enhanced security by imposing a limit on the amount of money that can be withdrawn in a single transaction.

Implementing Methods in Interfaces

- Java SE 8 allows you to implement special types of methods within interfaces:
 - `static` methods
 - `default` methods
- `default` methods help minimize code duplication.
 - They provide a single location to write and edit code.
 - They can be overridden later if necessary.
 - They're overridden with per-class precision.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Example: Implementing `default` Methods

Previously duplicated logic can be written once in the `Accessible` interface.

```
public interface Accessible{  
    ...  
    public default double verifyDeposit(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
    }  
    public default double verifyWithdraw(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
        //Verify account balance won't go negative  
    }  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Adding the keyword "public" is optional for default methods. A default method is always public.

Example: Inheriting default Methods

Simply let the class implement the `Accessible` interface.

```
public class BasicChecking implements Accessible{  
    ...  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Example: Overriding a default Method

You can override a default method and call the interface's implementation.

- For example, you might override the default method to enhance the verification.

```
public class RestrictedChecking implements Accessible{
    ...
    public double verifyWithdraw(double amount, int pin){
        //Call the interface's implementation
        Accessible.super.verifyWithdraw(amount, pin);




        //Verify the withdrawal is under the transaction limit.
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

After calling the interface's version of the verifyWithdraw method, additional logic can be written into the method to support the features of the RestrictedChecking class.

What About the Problems of Multiple Inheritance?

Multiple inheritance of...	Is possible...	Using this mechanic...
Type		A class implements multiple interfaces.
Behavior		A class implements interfaces containing multiple default methods. Special rules exist to prevent complications.
State		A call to a variable cannot have multiple potential values.*

**This is why multiple inheritance is problematic in C++.*



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

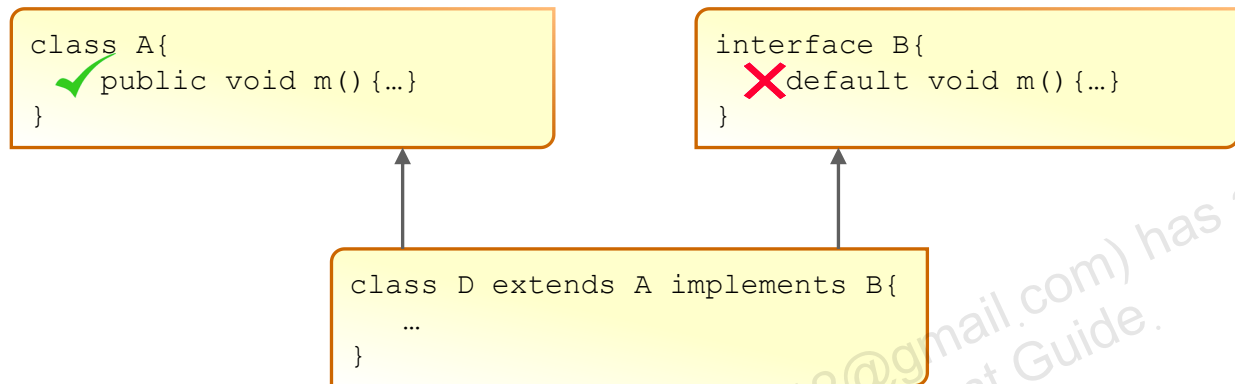
The idea of being able to implement methods within interfaces is still unfamiliar to many developers.

Inheritance Rules of `default` Methods

Rule 1:

A superclass method takes priority over an interface default method.

- The superclass method may be concrete or abstract.
- Only consider the interface default if no method exists from the superclass.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The code for this is found in the project `Ex_04_01_Interfaces`.

If you instantiate an instance of class `D` and call method `m`, the version of the method that is used comes from the superclass `A`.

Inheritance Rules of default Methods

Rule 2:

A subtype interface's default method takes priority over a super-type interface's default method.

```
interface B{  
  ✗ default void m() {...}  
}
```

```
interface C extends B{  
  ✓ default void m() {...}  
}
```

```
class D implements C{  
  ...  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The code for this is found in the project Ex_04_02_Interfaces.

If you instantiate an instance of class D and call method m, the version of the method that is used comes from interface C.

Inheritance Rules of `default` Methods

Rule 3:

If there is a conflict, treat the default method as abstract.

- The concrete class must provide its own implementation.
- This may include a call to a specific interface's implementation of the method.

```
interface B{  
  default void m() {...}  
}
```

```
interface C{  
  default void m() {...}  
}
```

```
class D implements B,C{  
  void m() {  
    B.super.m();  
  }  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The code for this is found in the project `Ex_04_03_Interfaces`.

If you instantiate an instance of class D and call method m, you'll need to specify within class D how the method should be implemented.

Interfaces Don't Replace Abstract Classes

- An interface doesn't let you store the state of an instance.
- An `abstract` class may contain instance fields.
- To avoid complications caused by multiple inheritance of state, a class cannot extend multiple `abstract` classes.

```
abstract class B{  
    int x = 200;  
}
```

```
abstract class C{  
    int x = 300;  
}
```

✗

```
class D extends B,C{  
    int getX(){  
        return x;  
    }  
}
```

Which value of x is returned?



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The code for this is found in the project Ex_04_04_Interfaces.

If the design of your program requires fields to be shared among different classes, you may want to write this field once in a common location. Abstract classes can accommodate this. Interfaces cannot. However, there is a trade-off. Multiple inheritance is not allowed with abstract classes.

What If `default` Methods Duplicate Logic?

Can anything be done to reduce this duplication?

```
public interface Accessible{  
    ...  
    public default double verifyDeposit(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
    }  
    public default double verifyWithdraw(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
        //Verify account balance won't go negative  
    }  
}
```

Duplicated

Duplicated



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Returning to our scenario...

Duplication Between default Methods

One strategy is to put duplicated logic within its own default method.

```
public interface Accessible{  
    ...  
    public default double verifyDeposit(double amount, int pin){  
        verifyTransaction(amount, pin);  
    }  
    public default double verifyWithdraw(double amount, int pin){  
        verifyTransaction(amount, pin);  
        //Verify account balance won't go negative  
    }  
    public default boolean verifyTransaction(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
    }  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

You'll see in the practice solution why the verifyTransaction method returns a boolean.

The Problem with This Approach

- `default` methods must be public.
- They can be called from almost anywhere (including the main method).
 - Returned values may not mean anything outside the context of other methods.
 - It's dangerous if the method returns information you don't want exposed.
- They can be overridden at any time.
 - The result of calling the method may not be predictable.

```
public class TestClass{  
    public static void main(String[] args){  
  
        RestrictedChecking acct1 = new RestrictedChecking(1000, 1111);  
        acct1.verfiyTransaction(200,1111);  
  
    }  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

What if `verifyTransaction` was overridden somewhere in the program to always deposit one million dollars into an account? As long as the method remains public, this undesirable behavior (from the bank's perspective) remains a possibility.

Introducing `private` Methods in Interfaces

- A better strategy is to make the method `private`.
- `private` interface methods are more secure.
 - They can't be called from elsewhere.
 - They limit the risk of exposing sensitive information.
- `private` interface methods lead to more predictable programs.
 - They can't be overridden.
 - They can't be called from a class that implements the interface.
- `private` interface methods lead to more maintainable and readable code.
 - Common logic can be stored and edited in one location.
- `private` interface methods don't lead to inheritance complications.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Example: Using `private` Methods to Reduce Duplication Between default Methods

```
public interface Accessible{  
    ...  
    public default double verifyDeposit(double amount, int pin){  
        verifyTransaction(amount, pin);  
    }  
    public default double verifyWithdraw(double amount, int pin){  
        verifyTransaction(amount, pin);  
        //Verify account balance won't go negative  
    }  
    private boolean verifyTransaction(double amount, int pin){  
        //Verify the PIN  
        //Verify amount is greater than 0  
    }  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

You'll see in the practice solution why the `verifyTransaction` method returns a boolean.

Types of Methods in Java SE 9 Interfaces

Access Modifier and Method Type	Supported?
<code>public abstract</code>	Yes
<code>private abstract</code>	Compiler Error
<code>public default</code>	Yes
<code>private default</code>	Compiler Error
<code>public static</code>	Yes
<code>private static</code>	Yes
<code>private</code>	Yes



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Adding the keyword "public" is optional for abstract methods. An abstract method is always public.
Adding the keyword "public" is optional for default methods. A default method is always public.

Anonymous Inner Classes

- Define a class in place instead of in a separate file.
- An Anonymous class doesn't have a name. Since it doesn't have a name, it cannot have a constructor.
 - You define an anonymous class and create its object at the same time.
 - It's always created using the new operator as part of an expression.
- Why would you do this?
 - Logically group code in one place
 - Increase encapsulation
- Syntax:

```
new <interface-name or class-name> (<argument-list>) {  
    // Anonymous class body goes here  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Creating an anonymous class:

The new operator is used to create an instance of the anonymous class. It's followed by either an existing interface name or an existing class name. Note that the interface name or class name is not the name for the newly created anonymous class. Rather, it is an existing interface/class name. If an interface name is used, the anonymous class implements the interface. If a class name is used, the anonymous class inherits from the class. The <argument-list> is used only if the new operator is followed by a class name.

Anonymous Inner Class: Example

- Example method call with concrete class

```
20 // Call concrete class that implements StringAnalyzer
21 ContainsAnalyzer contains = new ContainsAnalyzer();
22
23 System.out.println("===Contains===");
24 Analyzer.searchArr(strList, searchStr, contains);
```

- Anonymous inner class example

```
22 Analyzer.searchArr(strList, searchStr,
23     new StringAnalyzer() {
24         @Override
25         public boolean analyze(String target, String searchStr) {
26             return target.contains(searchStr);
27         }
28     });
```

- The anonymous inner class, `StringAnalyzer`, and its object is created in place.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The example shows how an anonymous inner class can be substituted for an object. Assume the class `Analyzer` has a static method `searchArr` that performs analysis based on a set of sample strings and a search string. The method takes the following arguments:

- `String[] strList` (the sample strings)
- `String searchStr` (the search String)
- `StringAnalyzer contains` (determines how the search functions)

By having the third argument, you can pass functionality to the `searchArr` method. For example, the search could be for anywhere in the string or just at the start. It may have to match case or not.

Here is the source code for `ContainsAnalyzer` that checks if a string starts with the search string:

```
public class ContainsAnalyzer implements StringAnalyzer {
    public boolean analyze(String target, String searchStr) {
        return target.startsWith(searchStr);
    }
}
```

The anonymous inner class specifies no name but implements almost exactly the same code (but has slightly different search functionality). The syntax is a little complicated as the class is defined where a parameter variable would normally be.

The advantages of passing functionality to a method are explained further in the lesson “Functional Interfaces and Lambda Expressions.”

Summary

In this lesson, you should have learned how to:

- Use `default` methods in interfaces
- Identify when it's desirable to implement a `default` method in an interface
- Identify how inheritance rules apply to methods implemented in interfaces
- Use `private` methods in interfaces
- Identify when it's desirable to implement a `private` method in an interface
- Define anonymous classes



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Practice 4: Overview

This practice covers the following topics:

- Practice 4-1: Java SE 8 Default Methods
- Practice 4-2: Java SE 9 Private Methods



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Quiz



What is true about code duplication?

- A. Duplication makes your code longer. This is good because it makes your colleagues believe you're really smart and capable of handling complex code.
- B. Duplication is good because it builds redundancy into the system.
- C. If you need to make an edit, you'll have to search for all cases where the code is duplicated. This is tedious and inefficient.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Quiz



The `Accessible` interface only verifies if a transaction is appropriate. Why can't this interface also store and modify an account's balance?

- A. `private` interface methods can't return `void`.
- B. Changing balance is a behavior. This could potentially lead to multiple inheritance of behavior.
- C. If the interface had a `balance` instance field, this could potentially lead to multiple inheritance of state.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Interfaces must safely allow for multiple inheritance. For this reason, interfaces cannot store instance variables. Design your interfaces accordingly, as this rule restricts what data an interface's method can access.



Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a
non-transferable license to use this Student Guide.