4

Overriding Methods, Polymorphism, and Static Classes

Objectives

After completing this lesson, you should be able to do the following:

- Use access levels: `private`, `protected`, default, and `public`
- Override methods
- Use virtual method invocation
- Use `varargs` to specify variable arguments
- Use the `instanceof` operator to compare object types
- Use upward and downward casts
- Model business problems by using the `static` keyword
- Implement the singleton design pattern

Using Access Control

- You have seen the keywords `public` and `private`.
- There are four access levels that can be applied to data fields and methods.
- Classes can be default (no modifier) or `public`.

Modifier (keyword)	Same Class	Same Package	Subclass in Another Package	Universe
`private`	Yes			
`default`	Yes	Yes		
`protected`	Yes	Yes	Yes	
`public`	Yes	Yes	Yes	Yes

ORACLE

The table in the slide illustrates access to a field or method marked with the access modifier in the left column.

The access modifier keywords shown in this table are `private`, `protected`, and `public`.

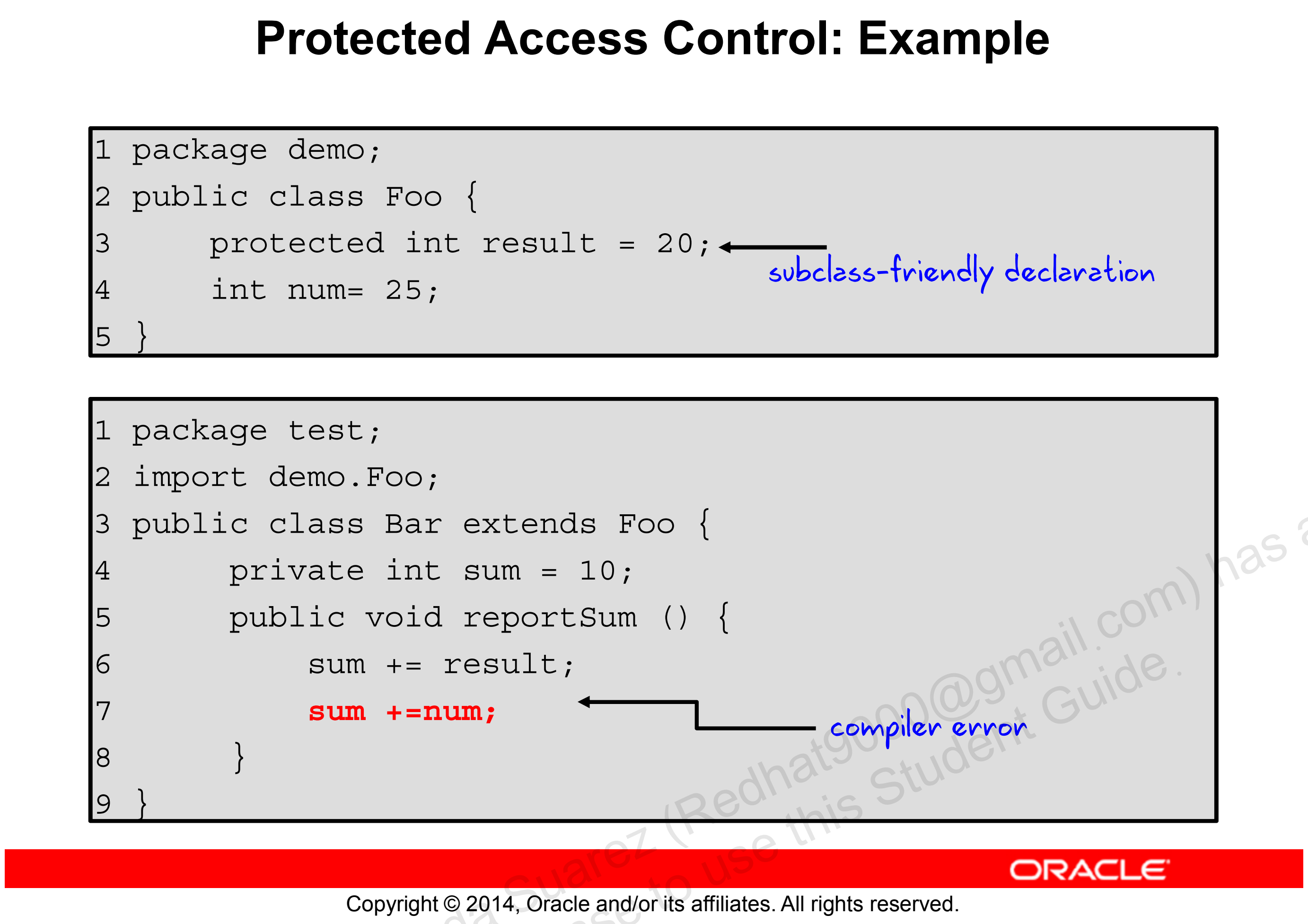
When a keyword is absent, the ***default*** access modifier is applied.

`private`: Provides the greatest control over access to fields and methods. With `private`, a data field or method can be accessed only within the same Java class.

`default`: Also called package level access. With default, a data field or method can be accessed within the same class or package. A default class cannot be subclassed outside its package.

`protected`: Provides access within the package and subclass. Fields and methods that use `protected` are said to be "subclass-friendly." Protected access is extended to subclasses that reside in a package different from the class that owns the protected feature. As a result, `protected` fields or methods are actually more accessible than those marked with default access control.

`public`: Provides the greatest access to fields and methods, making them accessible anywhere: in the class, package, subclasses, and any other class.



Protected Access Control: Example

```
package demo;
public class Foo {
    protected int result = 20;
    int num= 25;
}
```

subclass-friendly declaration

```
package test;
import demo.Foo;
public class Bar extends Foo {
     private int sum = 10;
     public void reportSum () {
         sum += result;
         sum +=num;
     }
}
```

compiler error

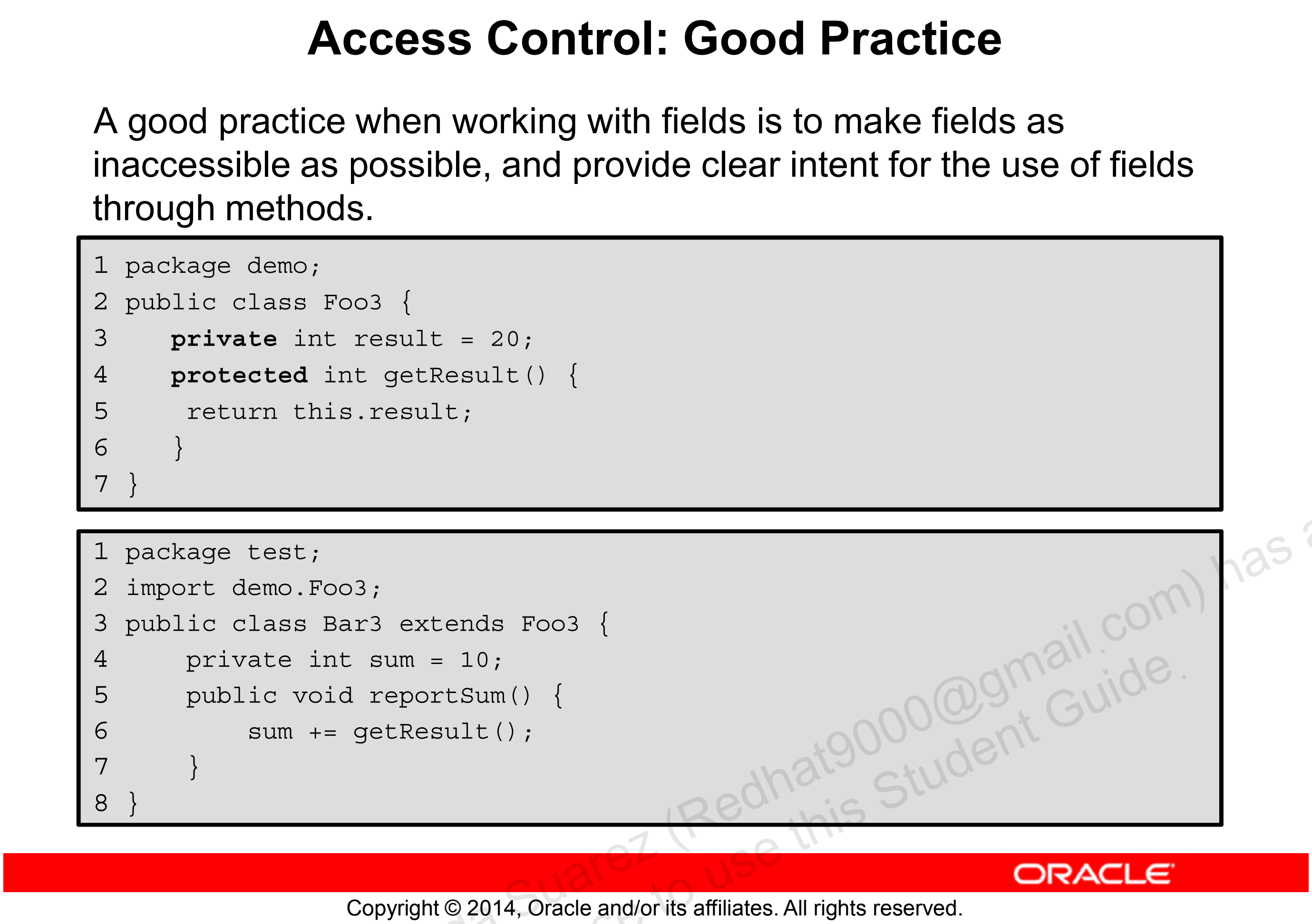
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this example, there are two classes in two packages. Class `Foo` is in the package `demo`, and declares a data field called `result` with a `protected` access modifier.

In the class `Bar`, which extends `Foo`, there is a method, `reportSum`, that adds the value of `result` to `sum`. The method then attempts to add the value of `num` to sum. The field `num` is declared using the default modifier, and this generates a compiler error. Why?

Answer: The field `result`, declared as a `protected` field, is available to all subclasses—even those in a different package. The field `num` is declared as using default access and is only available to classes and subclasses declared in the same package.

This example is from the `JavaAccessExample` project.



Access Control: Good Practice

A good practice when working with fields is to make fields as inaccessible as possible, and provide clear intent for the use of fields through methods.

```
package demo;
public class Foo3 {
   private int result = 20;
   protected int getResult() {
    return this.result;
   }
}
```

```
package test;
import demo.Foo3;
public class Bar3 extends Foo3 {
    private int sum = 10;
    public void reportSum() {
        sum += getResult();
    }
}
```

A slightly modified version of the example using the `protected` keyword is shown in the slide. If the idea is to limit access of the field result to classes within the package and the subclasses (package-protected), you should make the access explicit by defining a method purposefully written for package and subclass-level access.

Overriding Methods

Consider a requirement to provide a String that represents some details about the `Employee` class fields.

```
public class Employee {
    private int empId;
    private String name;
    // Lines omitted

    public String getDetails() {
      return "ID: " + empId + " Name: " + name;
    }
```

Although the `Employee` class has getters to return values for a print statement, it might be nice to have a utility method to get specific details about the employee. Consider a method added to the `Employee` class to print details about the `Employee` object.

In addition to adding fields or methods to a subclass, you can also modify or change the existing behavior of a method of the parent (superclass).

You may want to specialize this method to describe a `Manager` object.

Overriding Methods

In the `Manager` class, by creating a method with the same signature as the method in the `Employee` class, you are *overriding* the `getDetails` method:

```
public class Manager extends Employee {
    private String deptName;
    // Lines omitted

    @Override
    public String getDetails() {
      return super.getDetails () +
        " Dept: " + deptName;
    }
```

A subclass can invoke a parent method by using the `super` keyword.

When a method is overridden, it replaces the method in the superclass (parent) class.

This method is called for any `Manager` instance.

A call of the form `super.getDetails()` invokes the `getDetails` method of the parent class.

Note: If, for example, a class declares two public methods with the same name, and a subclass overrides one of them, the subclass still inherits the other method.

Invoking an Overridden Method

- Using the previous examples of `Employee` and `Manager`:

```
public static void main(String[] args) {
  Employee e = new Employee(101, "Jim Smith",
      "011-12-2345", 100_000.00);
  Manager m = new Manager(102, "Joan Kern",
      "012-23-4567", 110_450.54, "Marketing");

  System.out.println(e.getDetails());
  System.out.println(m.getDetails());
}
```

- The correct `getDetails` method of each class is called:

```
ID: 101 Name: Jim Smith
ID: 102 Name: Joan Kern Dept: Marketing
```

During run time, the Java Virtual Machine invokes the `getDetails` method of the appropriate class. If you comment out the `getDetails` method in the `Manager` class shown in the previous slide, what happens when `m.getDetails()` is invoked?

Answer: Recall that methods are inherited from the parent class. So, at run time, the `getDetails` method of the parent class (`Employee`) is executed.

Virtual Method Invocation

- What happens if you have the following?

```
public static void main(String[] args) {
  Employee e = new Manager(102, "Joan Kern",
      "012-23-4567", 110_450.54, "Marketing");

  System.out.println(e.getDetails());
}
```

- During execution, the object’s runtime type is determined to be a `Manager` object:

```
ID: 102 Name: Joan Kern Dept: Marketing
```

- At run time, the method that is executed is referenced from a `Manager` object.
- This is an aspect of polymorphism called *virtual method invocation*.

Compiler Versus Runtime Behavior

The important thing to remember is the difference between the compiler (which checks that each method and field is accessible based on the strict definition of the class) and the behavior associated with an object determined at run time.

This distinction is an important and powerful aspect of polymorphism: The behavior of an object is determined by its runtime reference.

Because the object you created was a `Manager` object, at runtime, when the `getDetails` method was invoked, the runtime reference is to the `getDetails` method of a `Manager` class, even though the variable `e` is of the type `Employee`.

This behavior is referred to as *virtual method invocation*.

Note: If you are a C++ programmer, you get this behavior in C++ only if you mark the method by using the C++ keyword `virtual`.

Accessibility of Overriding Methods

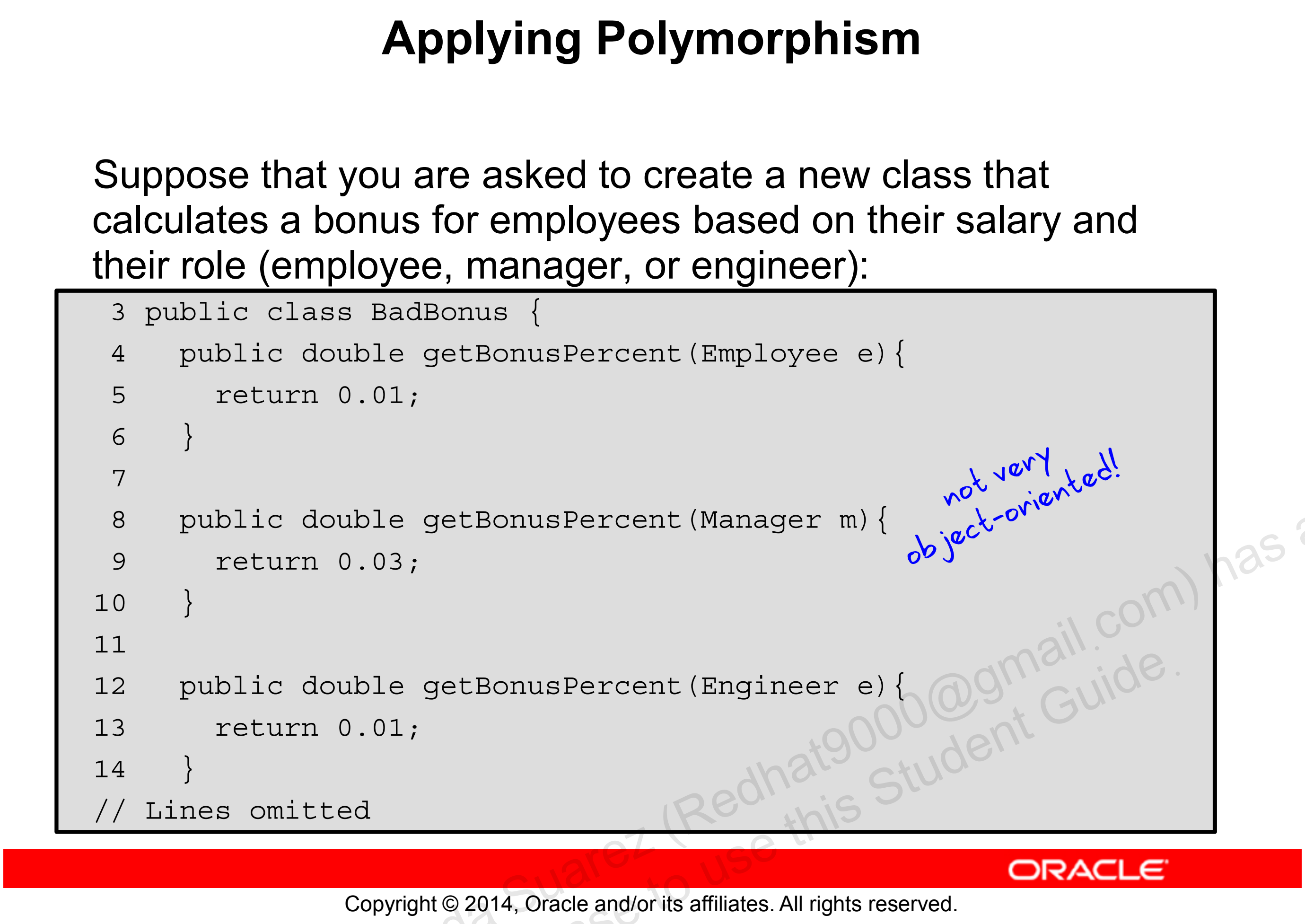
The overriding method cannot be less accessible than the method in the parent class.

```
public class Employee {
    //... other fields and methods
    public String getDetails() { ... }
}
```

```
public class BadManager extends Employee {
    private String deptName;
    // lines omitted
    @Override
    private String getDetails() { // Compile error
      return super.getDetails () +
        " Dept: " + deptName;
    }
```

To override a method, the name and the order of arguments must be identical.

By changing the access of the Manager `getDetails` method to `private`, the **`BadManager` class will not compile.**



Applying Polymorphism

Suppose that you are asked to create a new class that calculates a bonus for employees based on their salary and their role (employee, manager, or engineer):

```
public class BadBonus {
  public double getBonusPercent(Employee e){
    return 0.01;
  }

  public double getBonusPercent(Manager m){
    return 0.03;
  }

  public double getBonusPercent(Engineer e){
    return 0.01;
  }
// Lines omitted
```

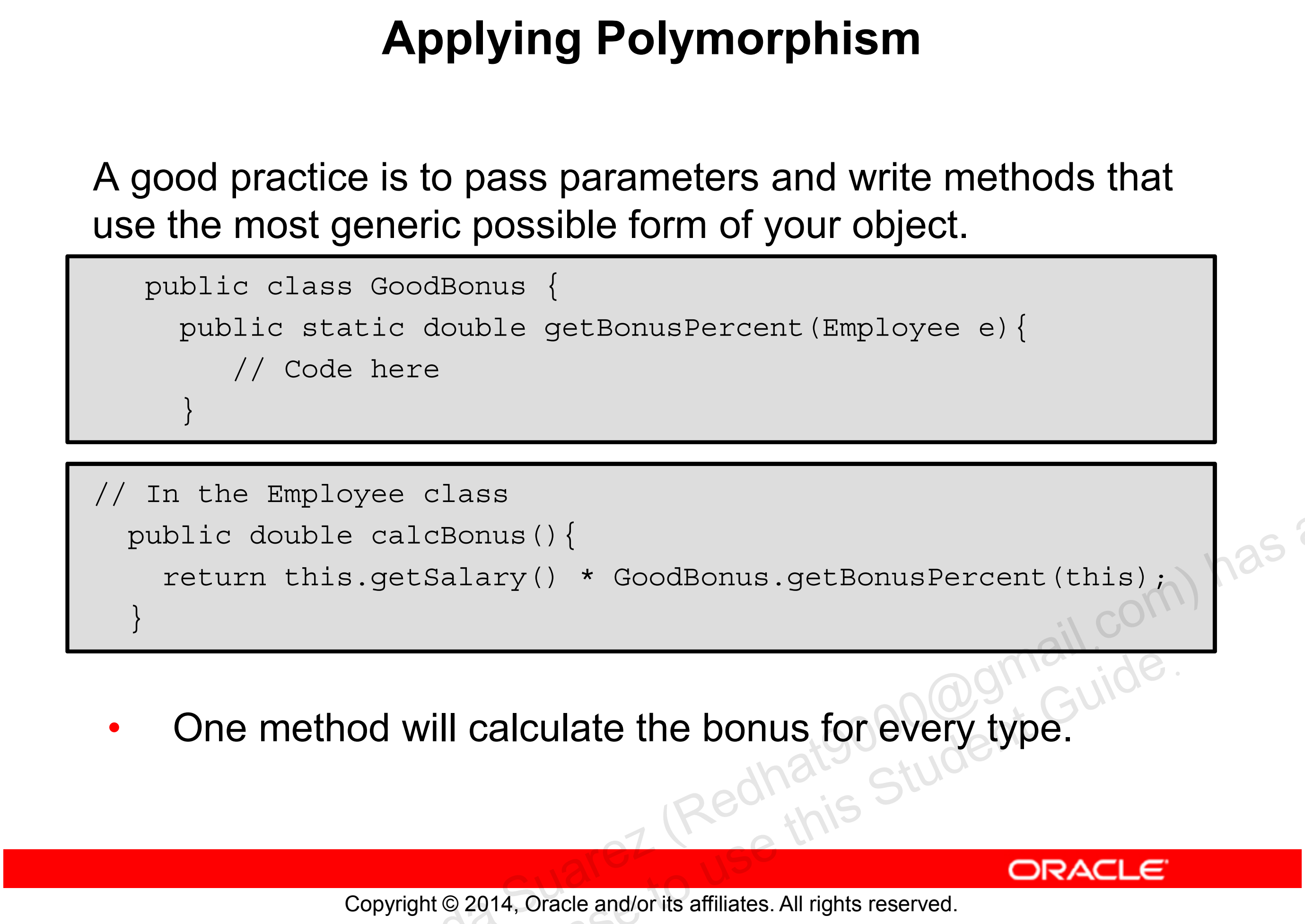
not very object-oriented!

Design Problem

What is the problem in the example in the slide? Each method performs the calculation based on the type of employee passed in, and returns the bonus amount.

Consider what happens if you add two or three more employee types. You would need to add three additional methods, and possibly replicate the code depending upon the business logic required to compute shares.

Clearly, this is not a good way to treat this problem. Although the code will work, this is not easy to read and is likely to create much duplicate code.



Applying Polymorphism

A good practice is to pass parameters and write methods that use the most generic possible form of your object.

```
   public class GoodBonus {
     public static double getBonusPercent(Employee e){
        // Code here
     }
```

```
// In the Employee class
  public double calcBonus(){
    return this.getSalary() * GoodBonus.getBonusPercent(this);
  }
```

- One method will calculate the bonus for every type.

Use the Most Generic Form

A good practice is to design and write methods that take the most generic form of your object possible.

In this case, `Employee` is a good base class to start from. But how do you know what object type is passed in? You learn the answer in the next slide.

Using the `instanceof` Keyword

The Java language provides the `instanceof` keyword to determine an object's class type at run time.

```
public class GoodBonus {
  public static double getBonusPercent(Employee e){
    if (e instanceof Manager){
      return 0.03;
    }else if (e instanceof Director){
      return 0.05;
    }else {
      return 0.01;
    }
  }
}
```

In the `GoodBonus` class, the `getBonusPercent` method uses the `instanceof` operator to determine what type of `Employee` was passed to the method.

Overriding Object methods

The root class of every Java class is `java.lang.Object`.

- All classes will subclass `Object` by default.
- You do not have to declare that your class extends `Object`. The compiler does that for you.

```
public class Employee { //... }
```

is equivalent to

```
public class Employee extends Object { //... }
```

- The root class contains several nonfinal methods, but there are three that are important to consider overriding:
  - `toString`, `equals`, and `hashCode`

`Object toString` Method

The `toString` method returns a String representation of the object.

```
Employee e = new Employee (101, "Jim Kern", ...)
System.out.println (e);
```

- You can use `toString` to provide instance information:

```
public String toString () {
    return "Employee id:  " + empId + "\n"+
           "Employee name:" + name;
}
```

- This is a better approach to getting details about your class than creating your own `getDetails` method.

The `println` method is overloaded with a number of parameter types. When you invoke `System.out.println(e);` the method that takes an `Object` parameter is matched and invoked. This method in turn invokes the `toString()` method on the object instance.

Note: Sometimes you may want to be able to print out the name of the class that is executing a method. The `getClass()` method is an `Object` method used to return the `Class` object instance, and the `getName()` method provides the fully qualified name of the runtime class. `getClass().getName(); //` returns the name of this class instance. These methods are in the `Object` class.

`Object equals` Method

The `Object equals` method compares only object references.

- If there are two objects x and y in any class, x is equal to y if and only if x and y refer to the same object.
- Example:

```
Employee x = new Employee (1,"Sue","111-11-1111",10.0);
Employee y = x;
x.equals (y); // true
Employee z = new Employee (1,"Sue","111-11-1111",10.0);
x.equals (z); // false!
```

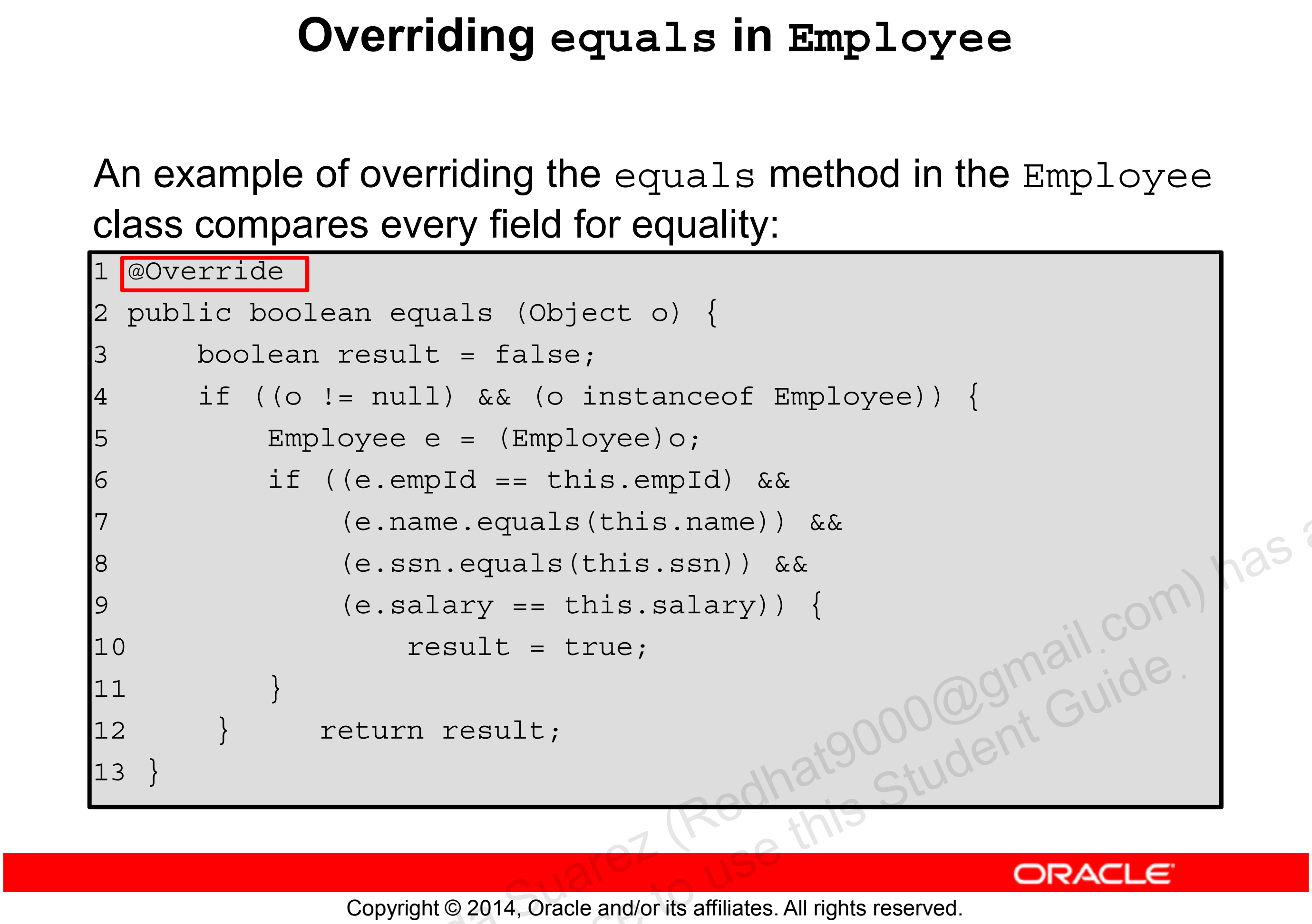
- Because what we really want is to test the contents of the Employee object, we need to override the `equals` method:

```
public boolean equals (Object o) { ... }
```

The `equals` method of `Object` determines (by default) only if the values of two object references point to the same object. Basically, the test in the `Object` class is simply as follows:

If x == y, return true.

For an object (like the `Employee` object) that contains values, this comparison is not sufficient, particularly if we want to make sure there is one and only one employee with a particular ID.



Overriding `equals` in `Employee`

An example of overriding the `equals` method in the `Employee` class compares every field for equality:

```
@Override
public boolean equals (Object o) {
    boolean result = false;
    if ((o != null) && (o instanceof Employee)) {
        Employee e = (Employee)o;
        if ((e.empId == this.empId) &&
             (e.name.equals(this.name)) &&
             (e.ssn.equals(this.ssn)) &&
             (e.salary == this.salary)) {
                result = true;
        }
    }     return result;
}
```

This simple `equals` test first tests to make sure that the object passed in is not null, and then tests to make sure that it is an instance of an `Employee` class (all subclasses are also employees, so this works). Then the `Object` is cast to `Employee`, and each field in `Employee` is checked for equality.

Note: For `String` types, you should use the `equals` method to test the strings character by character for equality.

`@Override` annotation

This annotation is used to instruct compiler that method annotated with `@Override` is an overridden method from super class or interface. When this annotation is used the compiler check is to make sure you actually are overriding a method when you think you are. This way, if you make a common mistake of misspelling a method name or not correctly matching the parameters, you will be warned that you method does not actually override as you think it does. Secondly, it makes your code easier to understand when you are overriding methods.

Overriding `Object` `hashCode`

The general contract for `Object` states that if two objects are considered equal (using the `equals` method), then integer hashcode returned for the two objects should also be equal.

```
@Override //generated by NetBeans
public int hashCode() {
    int hash = 7;
    hash = 83 * hash + this.empId;
    hash = 83 * hash + Objects.hashCode(this.name);
    hash = 83 * hash + Objects.hashCode(this.ssn);
    hash = 83 * hash + (int)
(Double.doubleToLongBits(this.salary) ^
(Double.doubleToLongBits(this.salary) >>> 32));
    return hash;
}
```

Overriding `hashCode`

The Java documentation for the `Object` class states:

"... It is generally necessary to override the `hashCode` method whenever this method [`equals`] is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes."

The `hashCode` method is used in conjunction with the `equals` method in hash-based collections, such as `HashMap`, `HashSet`, and `Hashtable`.

This method is easy to get wrong, so you need to be careful. The good news is that IDEs such as NetBeans can generate `hashCode` for you.

To create your own hash function, the following will help approximate a reasonable hash value for equal and unequal instances:

1) Start with a nonzero integer constant. Prime numbers result in fewer hashcode collisions.
2) For each field used in the `equals` method, compute an `int` hash code for the field. Notice that for the `Strings`, you can use the `hashCode` of the `String`.
3) Add the computed hash codes together.
4) Return the result.

Methods Using Variable Arguments

A variation of method overloading is when you need a method that takes any number of arguments of the same type:

```
public class Statistics {
    public float average (int x1, int x2) {}
    public float average (int x1, int x2, int x3) {}
    public float average (int x1, int x2, int x3, int x4) {}
}
```

- These three overloaded methods share the same functionality. It would be nice to collapse these methods into one method.

```
Statistics stats = new Statistics ();
float avg1 = stats.average(100, 200);
float avg2 = stats.average(100, 200, 300);
float avg3 = stats.average(100, 200, 300, 400);
```

ORACLE

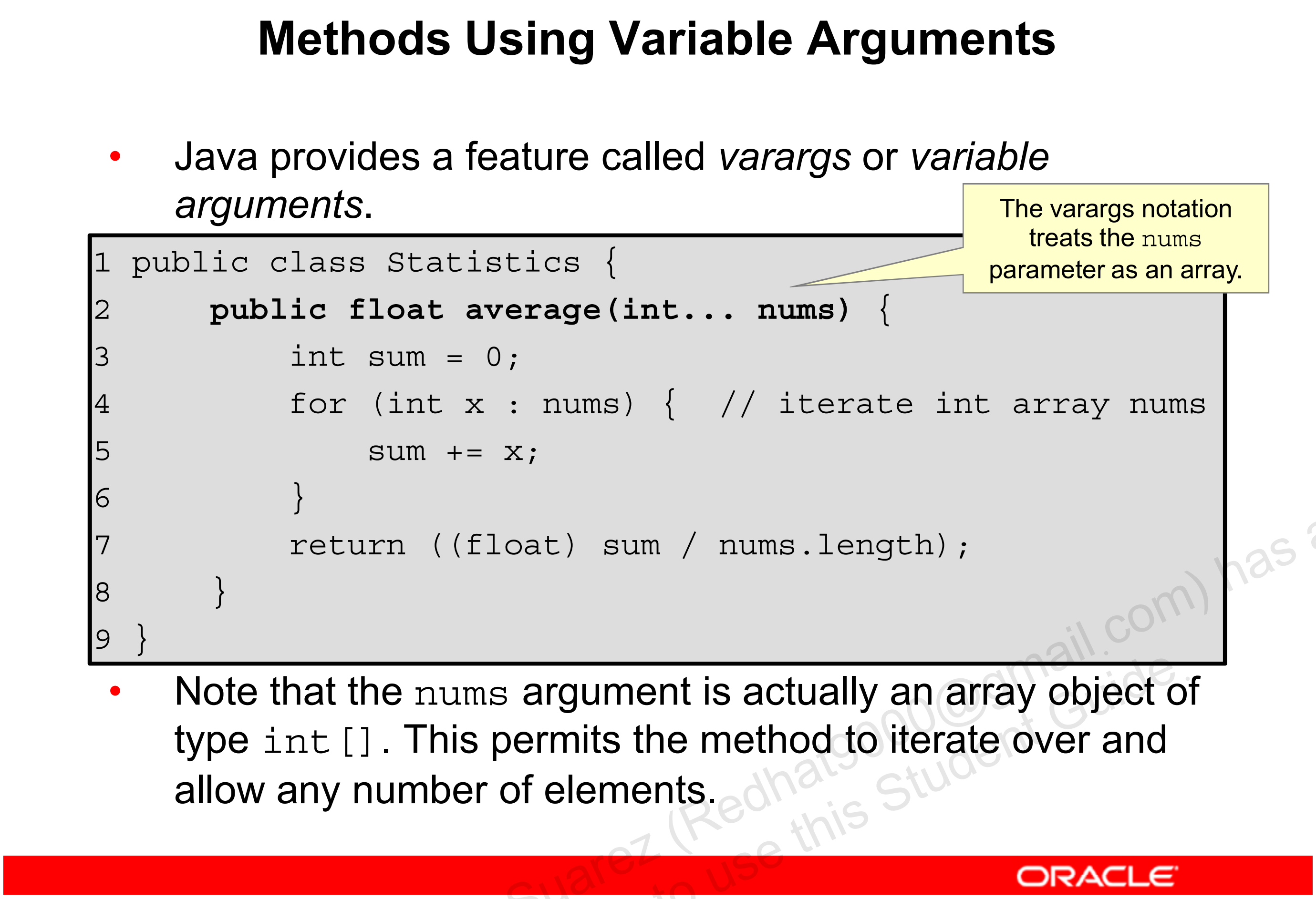
Methods with a Variable Number of the Same Type

One case of overloading is when you need to provide a set of overloaded methods that differ in the number of the same type of arguments. For example, suppose you want to have methods to calculate an average. You may want to calculate averages for 2, 3, or 4 (or more) integers.

Each of these methods performs a similar type of computation—the average of the arguments passed in, as in this example:

```
public class Statistics {
    public float average(int x1, int x2) { return (x1 + x2) / 2; }
    public float average(int x1, int x2, int x3) {
        return (x1 + x2 + x3) / 3;
    }
    public float average(int x1, int x2, int x3, int x4) {
        return (x1 + x2 + x3 + x4) / 4;
    }
}
```

Java provides a convenient syntax for collapsing these three methods into just one and providing for any number of arguments.



Methods Using Variable Arguments

- Java provides a feature called *varargs* or *variable arguments*.

```
public class Statistics {
    public float average(int... nums) {
        int sum = 0;
        for (int x : nums) {  // iterate int array nums
            sum += x;
        }
        return ((float) sum / nums.length);
    }
}
```

The varargs notation treats the `nums` parameter as an array.

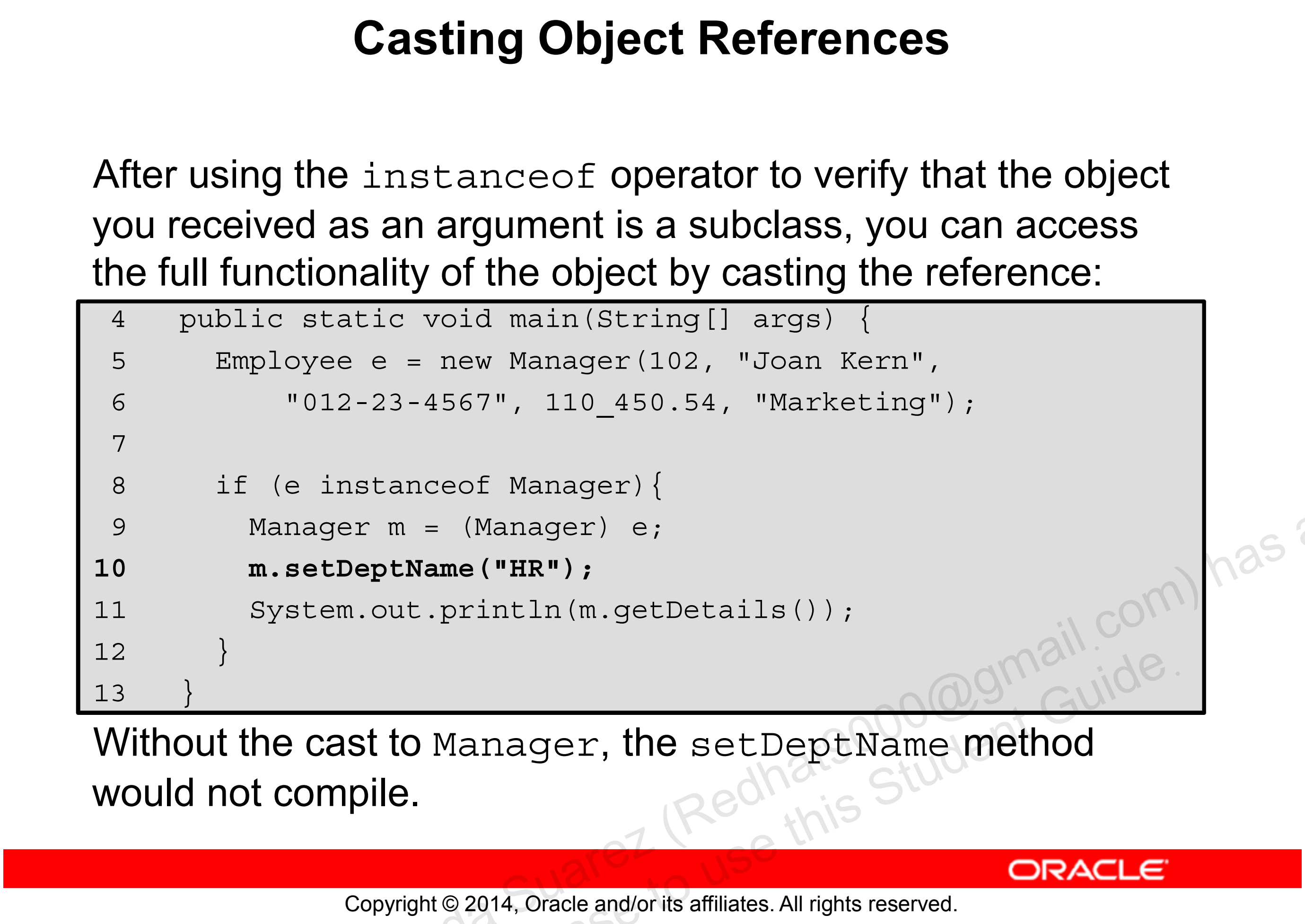
- Note that the `nums` argument is actually an array object of type `int[]`. This permits the method to iterate over and allow any number of elements.

Using Variable Arguments

The `average` method shown in the slide takes any number of integer arguments. The notation `(int... nums)` converts the list of arguments passed to the `average` method into an array object of type `int`.

Note: Methods that use varargs can also take no parameters—an invocation of `average()` is legal. You will see varargs as optional parameters in use in the NIO.2 API in the lesson titled “Java File I/O.” To account for this, you could rewrite the `average` method in the slide as follows:

```
public float average(int... nums) {
    int sum = 0; float result = 0;
        if (nums.length > 0) {
            for (int x : nums)   // iterate int array nums
                sum += x;
            result = (float) sum / nums.length;
        }
        return (result);
    }
}
```



Casting Object References

After using the `instanceof` operator to verify that the object you received as an argument is a subclass, you can access the full functionality of the object by casting the reference:

```
public static void main(String[] args) {
  Employee e = new Manager(102, "Joan Kern",
      "012-23-4567", 110_450.54, "Marketing");

  if (e instanceof Manager){
    Manager m = (Manager) e;
    m.setDeptName("HR");
    System.out.println(m.getDetails());
  }
}
```

Without the cast to `Manager`, the `setDeptName` method would not compile.

Although a generic superclass reference is useful for passing objects around, you may need to use a method from the subclass.

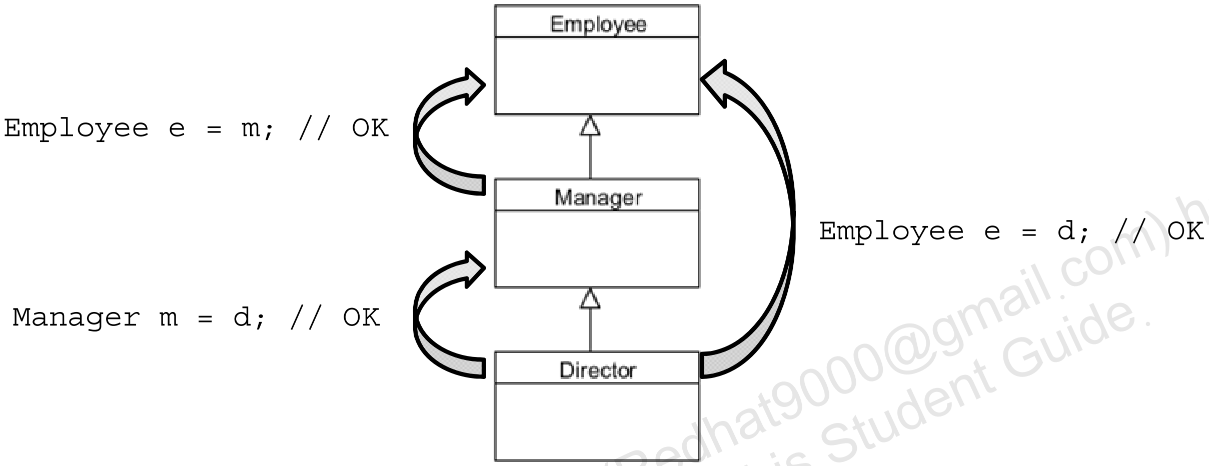
In the slide, for example, you need the `setDeptName` method of the `Manager` class. To satisfy the compiler, you can cast a reference from the generic superclass to the specific class.

However, there are rules for casting references. You see these in the next slide.

Upward Casting Rules

Upward casts are always permitted and do not require a cast operator.

```
Director d = new Director();
Manager m = new Manager();
```



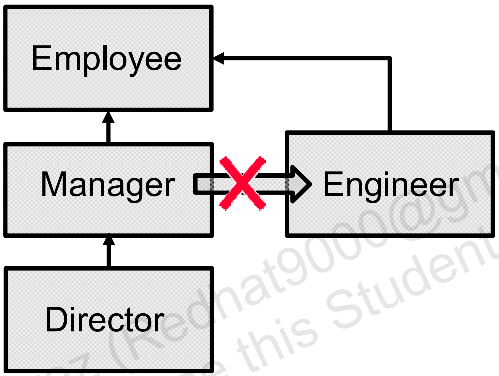
```
Employee e = m; // OK
Manager m = d; // OK
Employee e = d; // OK
```

Downward Casting Rules

For downward casts, the compiler must be satisfied that the cast is possible.

```
     Employee e = new Manager(102, "Joan Kern",
         "012-23-4567", 110_450.54, "Marketing");

     Manager m = (Manager)e; // ok
     Engineer eng = (Manager)e; // Compile error
     System.out.println(m.getDetails());
```



Downward Casts

With a downward cast, the compiler simply determines if the cast is possible; if the cast down is to a subclass, then it is possible that the cast will succeed.

Note that at run time, the cast results in a `java.lang.ClassCastException` if the object reference is of a superclass and not of the class type or a subclass.

Finally, any cast that is outside the class hierarchy will fail, such as the cast from a `Manager` instance to an `Engineer`. A `Manager` and an `Engineer` are both employees, but a `Manager` is not an `Engineer`.

`static` Keyword

The `static` modifier is used to declare fields and methods as class-level resources.

Static class members:

- Can be used without object instances
- Are used when a problem is best solved without objects
- Are used when objects of the same type need to share fields
- Should *not* be used to bypass the object-oriented features of Java unless there is a good reason

Static Methods

Static methods are methods that can be called even if the class they are declared in has not been instantiated.

Static methods:

- Are called class methods
- Are useful for APIs that are not object oriented
  - `java.lang.Math` contains many static methods
- Are commonly used in place of constructors to perform tasks related to object initialization
- Cannot access nonstatic members within the same class

ORACLE

Using Static Variables and Methods: Example

```
public class A01MathTest {
  public static void main(String[] args) {
    System.out.println("Random: " + Math.random() * 10);
    System.out.println("Square root: " + Math.sqrt(9.0));
    System.out.println("Rounded random: " +
        Math.round(Math.random()*100));
    System.out.println("Abs: " + Math.abs(-9));
  }
}
```

ORACLE

The `java.lang.Math` class contains methods and constants for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric.

The methods and constants in the `Math` class are all static, so you call them directly from the class as the examples in the slide demonstrates.

Here is some sample output from this example:

```
Random: 7.064916553599695
Square root: 3.0
Rounded random: 35
Abs: 9
```

Implementing Static Methods

- Use the static keyword before the method
- The method has parameters and return types like normal

```
import java.time.LocalDate;

public class StaticHelper {

    public static void printMessage(String message) {
        System.out.println("Messsage for " +
         LocalDate.now() + ": " + message);
    }

}
```

Static methods or class methods may be without an instantiating an object.

Static Method Limitations

Static methods can be used before any instances of their enclosing class have been created. If a class contains both static and instance components, the instance components cannot be accessed for a static context.

Calling Static Methods

```
double d = Math.random();
StaticHelper.printMessage("Hello");
```

When calling static methods, you should:

- Qualify the location of the method with a class name if the method is located in a different class than the caller
  - Not required for methods within the same class
- Avoid using an object reference to call a static method

Static Variables

Static variables are variables that can be accessed even if the class they are declared in has not been instantiated.

Static variables are:

- Called class variables
- Limited to a single copy per JVM
- Useful for containing shared data
  - Static methods store data in static variables.
  - All object instances share a single copy of any static variables.
- Initialized when the containing class is first loaded

Class Loading

Application developer-supplied classes are typically loaded on demand (first use). Static variables are initialized when their enclosing class is loaded. An attempt to access a static class member can trigger the loading of a class.

Defining Static Variables

```
public class StaticCounter {
    private static int counter = 0;

    public static int getCount() {
        return counter;
    }

    public static void increment(){
      counter++;
    }
}
```

Only one copy in memory

A static variable is defined when the `static` keyword precedes the type definition for a variable. Static variables are useful for containing shared data, all object instances share a single copy of any static variables.

Using Static Variables

```
double p = Math.PI;
```

```
public static void main(String[] args) {
  System.out.println("Start: " + StaticCounter.getCount());
  StaticCounter.increment();
  StaticCounter.increment();
  System.out.println("End: " + StaticCounter.getCount());
}
```

When accessing static variables, you should:

- Qualify the location of the variable with a class name if the variable is located in a different class than the caller
  - Not required for variables within the same class
- Avoid using an object reference to access a static variable

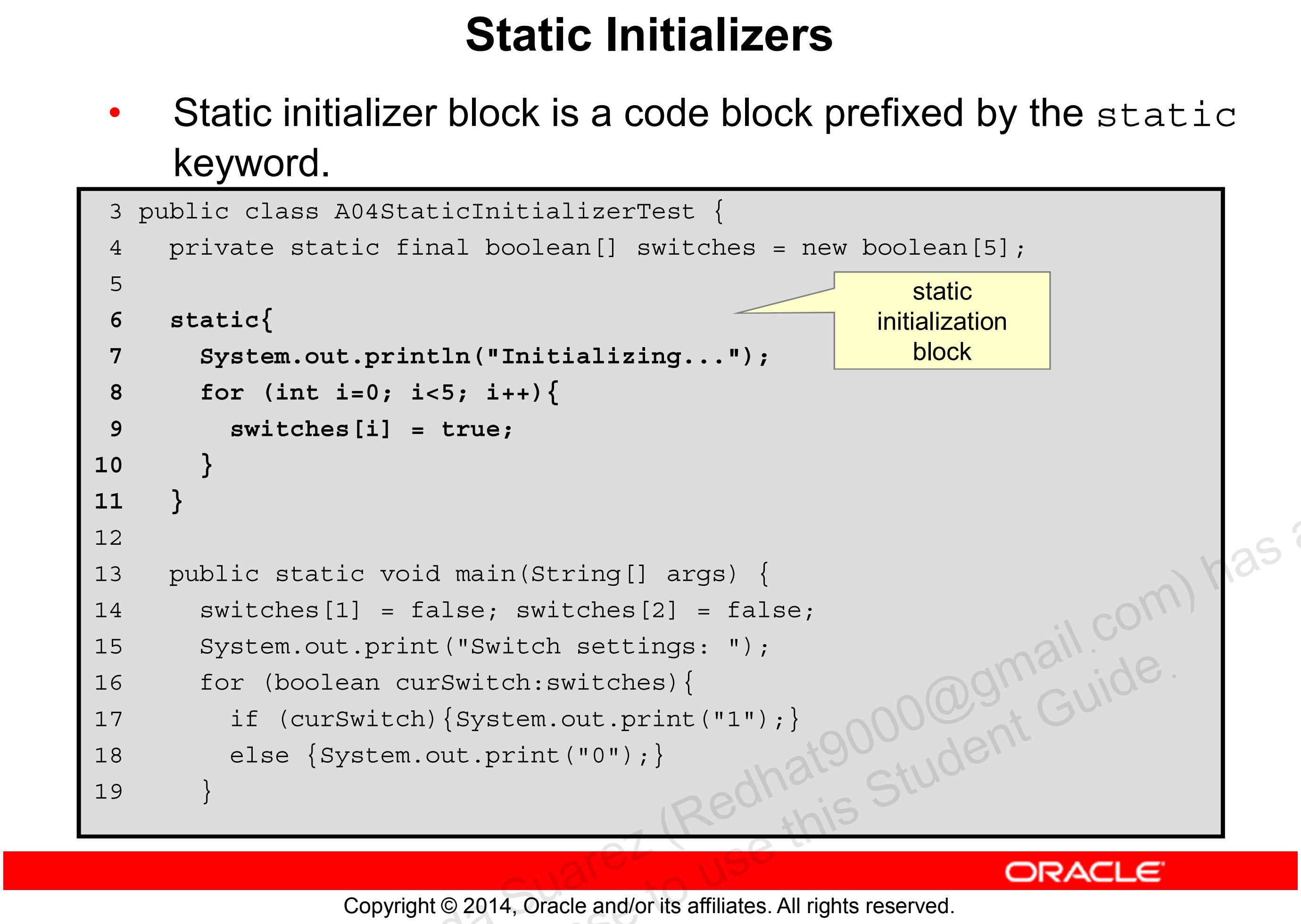
ORACLE

Object References to Static Members

Just as using object references to static methods should be avoided, you should also avoid using object references to access static variables. Using a `private` access level prevents direct access to `static` variables.

Sample output:

```
Start: 0
End: 2
```



Static Initializers

- Static initializer block is a code block prefixed by the `static` keyword.

```
public class A04StaticInitializerTest {
  private static final boolean[] switches = new boolean[5];

  static{
    System.out.println("Initializing...");
    for (int i=0; i<5; i++){
      switches[i] = true;
    }
  }

  public static void main(String[] args) {
    switches[1] = false; switches[2] = false;
    System.out.print("Switch settings: ");
    for (boolean curSwitch:switches){
      if (curSwitch){System.out.print("1");}
      else {System.out.print("0");}
    }
```

static initialization block

Here are some key facts about static initialization blocks:

- They are executed only once when the class is loaded.
- They are used to initialize static variables.
- A class can contain one or more static initializer blocks.
- They can appear anywhere in the class body.
- The blocks are called in the order that they appear in the source code.

Consider using static initializers when nontrivial code is required to initialize static variables.

Static Imports

A static import statement makes the static members of a class available under their simple name.

- Given either of the following lines:

```
import static java.lang.Math.random;
import static java.lang.Math.*;
```

- Calling the `Math.random()` method can be written as:

```
public class StaticImport {
    public static void main(String[] args) {
        double d = random();
    }
}
```

Overusing static import can negatively affect the readability of your code. Avoid adding multiple static imports to a class.

Design Patterns

Design patterns are:

- Reusable solutions to common software development problems
- Documented in pattern catalogs
  - *Design Patterns: Elements of Reusable Object-Oriented Software*, written by Erich Gamma et al. (the “Gang of Four”)
- A vocabulary used to discuss design

Design Pattern Catalogs

Pattern catalogs are available for many programming languages. Most of the traditional design patterns apply to any object-oriented programming language. One of the most popular books, *Design Patterns: Elements of Reusable Object-Oriented Software*, uses a combination of C++, Smalltalk, and diagrams to show possible pattern implementations. Many Java developers still reference this book because the concepts translate to any object-oriented language.

You learn more about design patterns and other Java best practices in the *Java Design Patterns* course.

Singleton Pattern

The singleton design pattern details a class implementation that can be instantiated only once.

```
public class SingletonClass {
1   private static final SingletonClass instance =
            new SingletonClass();

2   private SingletonClass() {}

    public static SingletonClass getInstance() {
3       return instance;
    }
}
```

Implementing the Singleton Pattern

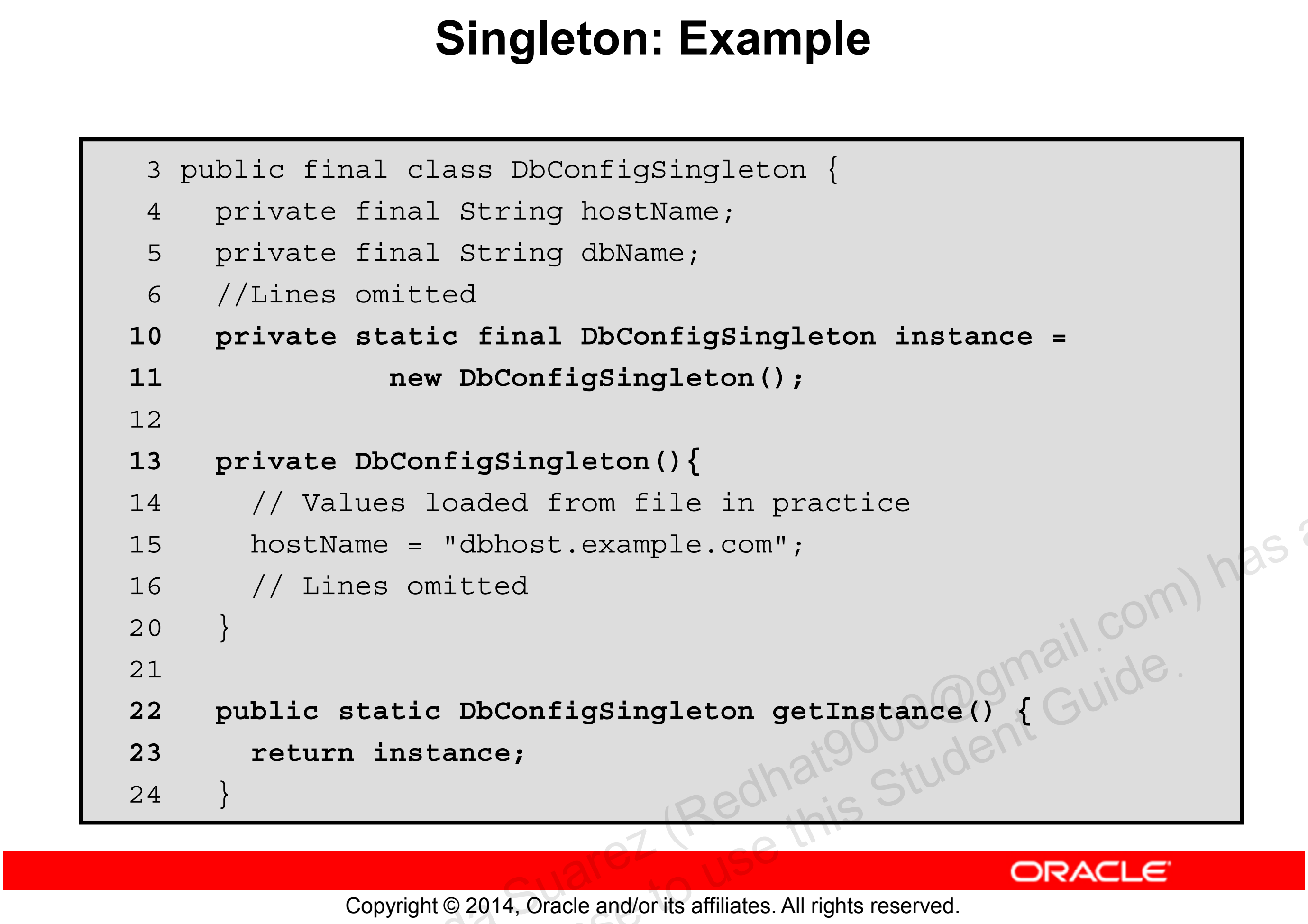
A singleton is a class for which only one instance can be created and it provides a global point of access to this instance.

Singletons are useful to provide a unique source of data or functionality to other Java Objects. For example, you may use a singleton to access your data model from within your application or to define logger which the rest of the application can use.

To implement the singleton design pattern:

1. Use a static reference to point to the single instance. Making the reference final ensures that it will never reference a different instance.
2. Add a single private constructor to the singleton class. The private modifier allows only "same class" access, which prohibits any attempts to instantiate the singleton class except for the attempt in step 1.
3. A public factory method returns a copy of the singleton reference. This method is declared static to access the static field declared in step 1. Step 1 could use a public variable, eliminating the need for the factory method. Factory methods provide greater flexibility (for example, implementing a per-thread singleton solution) and are typically used in most singleton implementations.
4. In singleton pattern, it restricts the creation of instance until requested first time(Lazy initialization). To obtain a singleton reference, call the `getInstance` method:

```
SingletonClass ref = SingletonClass.getInstance();
```



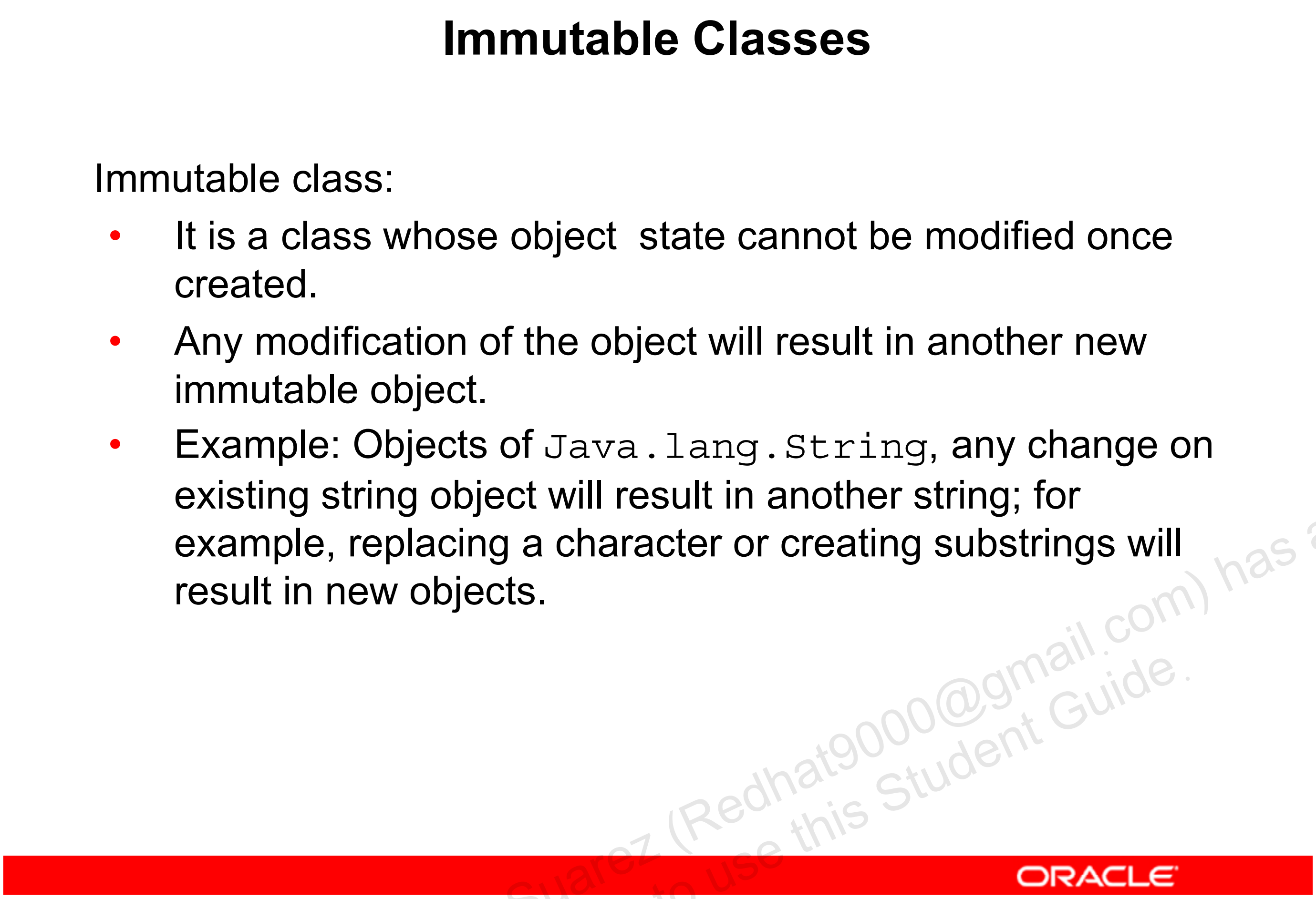
Singleton: Example

```
public final class DbConfigSingleton {
  private final String hostName;
  private final String dbName;
  //Lines omitted
  private static final DbConfigSingleton instance =
            new DbConfigSingleton();

  private DbConfigSingleton(){
    // Values loaded from file in practice
    hostName = "dbhost.example.com";
    // Lines omitted
  }

  public static DbConfigSingleton getInstance() {
    return instance;
  }
```

Singletons are great for storing data shared by an entire application. In this example, some database configuration data is stored in a Singleton.



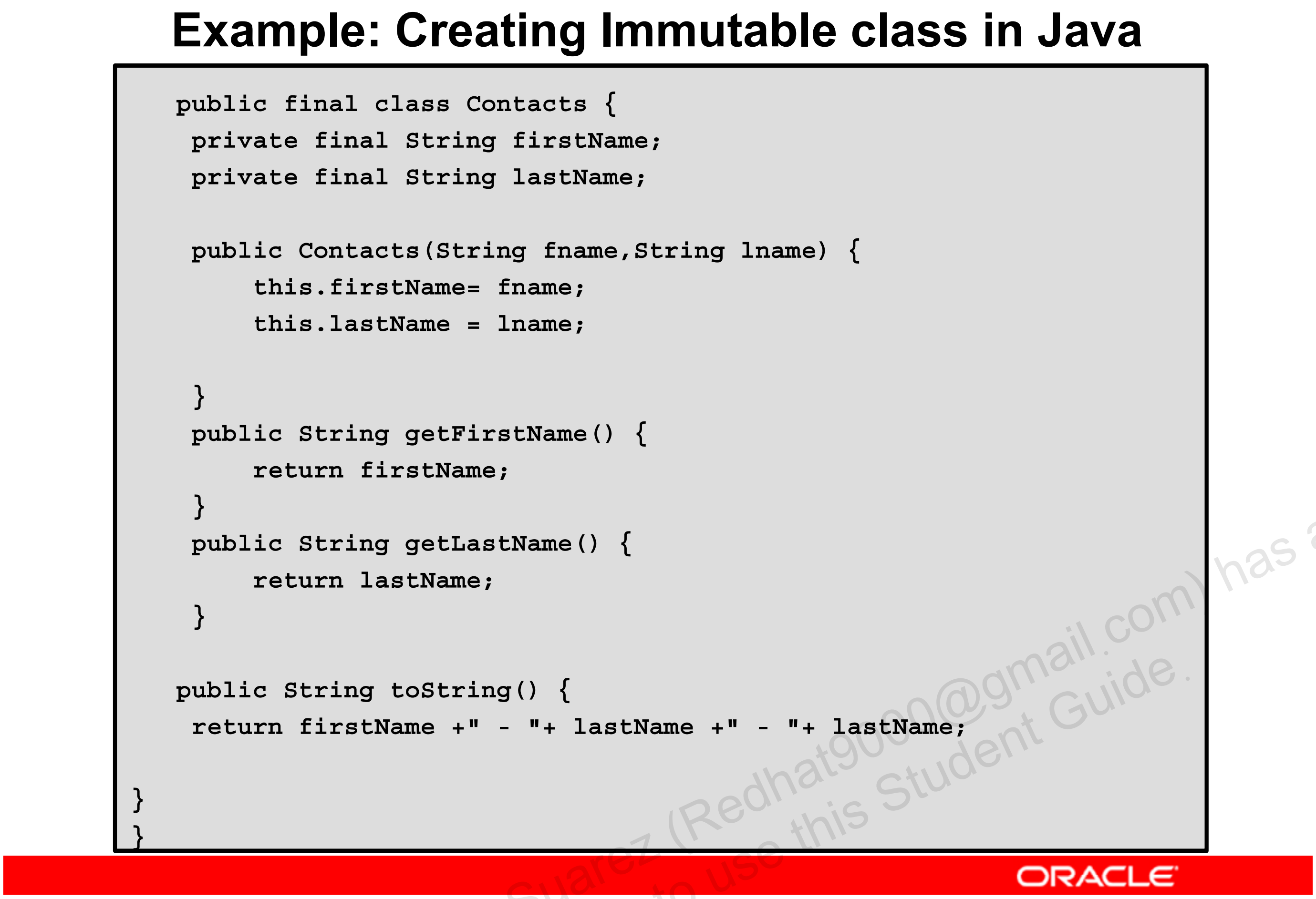
Immutable Classes

Immutable class:

- It is a class whose object state cannot be modified once created.
- Any modification of the object will result in another new immutable object.
- Example: Objects of `Java.lang.String`, any change on existing string object will result in another string; for example, replacing a character or creating substrings will result in new objects.

Rules to create a immutable class in Java:

- State of immutable object can not be modified after construction. Any modification would result in a new immutable object.
- Declare the class as final so it cannot be extended.
- All fields are declared private so that direct access is not allowed.
- Setter methods are not provided for variables.
- All fields of immutable class should be final.
- All the fields are initialized via a constructor.
- Object should be final in order to restrict subclass from altering immutability of parent class.



Example: Creating Immutable class in Java

```
    public final class Contacts {
     private final String firstName;
     private final String lastName;

     public Contacts(String fname,String lname) {
         this.firstName= fname;
         this.lastName = lname;

     }
     public String getFirstName() {
         return firstName;
     }
     public String getLastName() {
         return lastName;
     }

    public String toString() {
     return firstName +" - "+ lastName +" - "+ lastName;

 }
 }
```

ORACLE

The example in the slide demonstrates an immutable class in Java where all fields of class remain immutable and the class is also made final to avoid risk of immutability through inheritance.

Benefits of immutable classes in Java

1. Immutable objects are by default thread-safe, and can be shared without synchronization in concurrent environment.
2. Immutable object boost performance of Java application by reducing synchronization in code.
3. Reusability, you can cache immutable object and reuse them, much like String literals and Integers.

Note: If Immutable class has many optional and mandatory fields, you can also use Builder Design Pattern to make a class Immutable in Java.

Summary

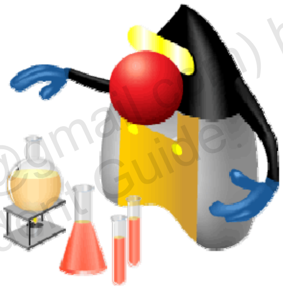
In this lesson, you should have learned how to:

- Use access levels: `private`, `protected`, default, and `public`.
- Override methods
- Use virtual method invocation
- Use `varargs` to specify variable arguments
- Use the `instanceof` operator to compare object types
- Use upward and downward casts
- Model business problems by using the `static` keyword
- Implement the singleton design pattern

Practice 4-1 Overview: Overriding Methods and Applying Polymorphism

This practice covers the following topics:

- Modifying the `Employee`, `Manager`, and `Director` classes; overriding the `toString()` method
- Creating an `EmployeeStockPlan` class with a grant stock method that uses the `instanceof` keyword



Practice 4-2 Overview: Overriding Methods and Applying Polymorphism

This practice covers the following topics:

- Fixing compilation errors caused due to casting
- Identifying runtime exception caused due to improper casting

Practice 4-3 Overview: Applying the Singleton Design Pattern

This practice covers using the `static` and `final` keywords and refactoring an existing application to implement the singleton design pattern.

Quiz

Suppose that you have an `Account` class with a `withdraw()` method, and a `Checking` class that extends `Account` that declares its own `withdraw()` method. What is the result of the following code fragment?

```
1   Account acct = new Checking();
2   acct.withdraw(100);
```

a. The compiler complains about line 1.

b. The compiler complains about line 2.

c. Runtime error: incompatible assignment (line 1)

d. Executes `withdraw` method from the `Account` class

e. Executes `withdraw` method from the `Checking` class

Quiz

Suppose that you have an `Account` class and a `Checking` class that extends `Account`. The body of the `if` statement in line 2 will execute.

```
1  Account acct = new Checking();
2  if (acct instanceof Checking) { // will this block run? }
```

a. True

b. False

Quiz

Suppose that you have an Account class and a Checking class that extends Account. You also have a Savings class that extends Account. What is the result of the following code?

```
1  Account acct1 = new Checking();
2  Account acct2 = new Savings();
3  Savings acct3 = (Savings)acct1;
```

a. acct3 contains the reference to acct1.

b. A runtime ClassCastException occurs.

c. The compiler complains about line 2.

d. The compiler complains about the cast in line 3.