# 2

# Java OOP Review


ORACLE

# Objectives

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

- Create Java classes
- Use encapsulation in Java class design
- Construct abstract Java classes and subclasses
- Override methods
- Use virtual method invocation
- Use `varargs` to specify variable arguments
- Apply the `final` keyword in Java
- Distinguish between top-level and nested classes
- Use enumerations

# Java Language Review

This lesson is a review of fundamental Java and programming concepts. It is assumed that students are familiar with the following concepts:

- The basic structure of a Java class
- Program block and comments
- Variables
- Branching constructs
- Iteration with loops
- Overloading of methods
- Encapsulation
- Inheritance
- Polymorphism
- Abstract Classes

## A Simple Java Class: `Employee`

A Java class is often used to represent a concept.

```
1 package com.example.domain;
2 public class Employee {
3     public int empId;
4     public String name;
5     public String ssn;
6     public double salary;
7
8     public Employee () {
9     }
10
11     public int getEmpId () {
12       return empId;
13    }
14 }
```

Package declaration

Class declaration

Fields

Constructor

Method



A Java class is often used to store or represent data for the construct that the class represents. For example, you could create a model (a programmatic representation) of an employee. An `Employee` object defined by using this model contains values for `empId`, `name`, Social Security Number (`ssn`), and `salary`.

A constructor is used to create an instance of a class. Unlike methods, constructors do not declare a return type and are declared with the same name as their class. Constructors can take arguments, and you can declare more than one constructor.

# Encapsulation: Private Data, Public Methods

One way to hide implementation details is to declare all the fields `private`.

- The `Employee` class currently uses `public` access for all of its fields.
- To encapsulate the data, make the fields `private`.

```
public class Employee {

    private int empId;
    private String name;
    private String ssn;
    private double salary;

 //... constructor and methods
}
```

Declaring fields private prevents direct access to this data from a class instance.
// illegal!
emp.salary = 1_000_000_000.00;

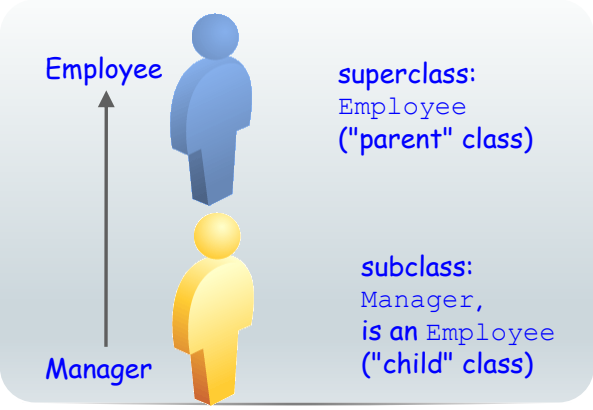


In Java, we accomplish encapsulation through the use of visibility modifiers (and also through the module system introduced in the lesson “Modules Overview”). Declaring Java fields private makes it invisible outside of the methods in the class itself.

In this example, the fields `custID`, `name`, and `amount` are now marked private, making them invisible outside of the methods in the class itself.

## Subclassing

In an object-oriented language like Java, subclassing is used to define a new class in terms of an existing one.




```
public class Manager extends Employee
{ }
```

The keyword **extends** creates the inheritance relationship



In the case of a subclass access to its superclass, it has ***access*** to all superclass fields but only inherits the nonprivate attributes and methods.

The code snippet in the slide demonstrates the Java syntax for subclassing.

The diagram in the slide demonstrates an inheritance relationship between the `Manager` class and, its parent, the `Employee` class.

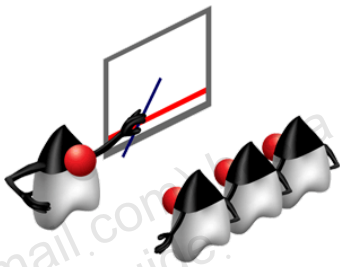
- The `Manager` class, by extending the `Employee` class, inherits all of the nonprivate data fields and methods from `Employee`.
- Since a manager is also an employee, then it follows that Manager has all of the same attributes and operations of Employee.

**Note:** The Manager class declares its own constructor. Constructors are *not* inherited from the parent class. There are additional details about this in the next slide.

# Constructors in Subclasses

Although a subclass inherits all of the methods and fields from a parent class, it doesn't inherit constructors. There are two ways to gain a constructor:

- Write your own constructor.
- Use the default constructor.
  - If you do not declare a constructor, a default no-arg constructor is provided for you.
  - If you declare your own constructor, the default constructor is no longer provided.





Every subclass inherits the nonprivate fields and methods from its parent (superclass). However, the subclass does not inherit the constructor from its parent. It must provide a constructor.

The *Java Language Specification* includes the following description:

"Constructor declarations are not members. They are never inherited and therefore are not subject to hiding or overriding."

# Using `super`

To construct an instance of a subclass, it is often easiest to call the constructor of the parent class.

- In its constructor, `Manager` calls the constructor of `Employee`.
- The `super` keyword is used to call a parent's constructor.
- It must be the first statement of the constructor.
- If it is not provided, a default call to `super()` is inserted for you.
- The `super` keyword may also be used to invoke a parent's method or to access a parent's (nonprivate) field.

```
super (empId, name, ssn, salary);
```

## 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 |



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



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.

# Inheritance: 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 () + 23 " 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.**

## Final Methods

A method can be declared `final`. Final methods may not be overridden.

```
public class MethodParentClass {
    public final void printMessage() {
        System.out.println("This is a final method");
    }
}
```

```
public class MethodChildClass extends MethodParentClass {
    // compile-time error
    public void printMessage() {
        System.out.println("Cannot override method");
    }
}
```



### Performance Myths

There is little to no performance benefit when you declare a method as final. Methods should be declared as final only to disable method overriding.

# Final Classes

A class can be declared `final`. Final classes may not be extended.

```
public final class FinalParentClass { }
```

```
// compile-time error
public class ChildClass extends FinalParentClass { }
```

## 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.

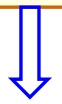
In this case, `Employee` is a good base class to start from. But how do you know what object type is passed in? You learn the answer in the next slide.

# Overriding methods of `Object` Class

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

- All classes subclass `Object` by default.
  - You don't have to declare that your class extends `Object`. The compiler does that for you.

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



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

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

# Overriding methods of `Object` Class: `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.

# Overriding methods of `Object` Class: `equals` Method

The `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. For 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!
```

- In case you want to test the contents of the `Employee` object, you 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 methods of `Object` Class: `equals` Method

For example, 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 the compiler that the 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 methods of `Object` Class: `hashCode` Method

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.

# 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.

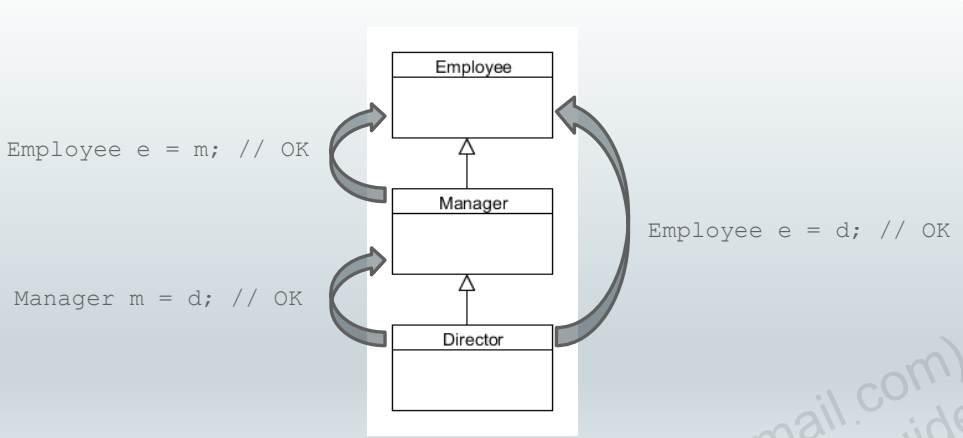
In the slide, for example, you need the `setDeptName` method of the `Manager` class. To satisfy the compiler, you can cast a reference from the generic superclass to the specific class.

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.

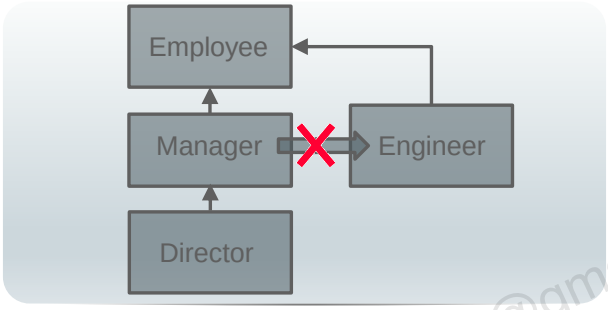
```
Director d = new Director();
Manager m = new Manager();
```



## 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`.

## 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);
```



### 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);
    }
}
```

# 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.

## Nested Classes

A nested class is a class declared within the body of another class:

- Have multiple categories
  - **Inner classes**
    - Member classes
    - Local classes
    - Anonymous classes
  - **Static nested classes**
- Are commonly used in applications with GUI elements
- Can limit utilization of a "helper class" to the enclosing top-level class



An **inner** nested class is considered part of the outer class and inherits access to all the private members of the outer class.

A **static** nested class is not an inner class, but its declaration appears similar to an additional `static` modifier on the nested class. Static nested classes can be instantiated before the enclosing outer class and, therefore, are denied access to all nonstatic members of the enclosing class.

**Note:** Anonymous classes are covered in detail in the lesson titled “Interfaces and Lambda Expressions.”

### Reasons to Use Nested Classes

The following information is obtained from http://download.oracle.com/javase/tutorial/java/javaOO/nested.html.

- **Logical Grouping of Classes**
  - If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- **Increased Encapsulation**
  - Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- **More Readable, Maintainable Code**
  - Nesting small classes within top-level classes places the code closer to where it is used.

## Example: Member Class

```
public class BankEMICalculator {
   private  String CustomerName;
   private  String AccountNo;
   private  double loanAmount;
   private  double monthlypayment;
   private EMICalculatorHelper helper = new EMICalculatorHelper();

    /*Setters ad Getters*/

      private class EMICalculatorHelper {
        int loanTerm = 60;
        double interestRate = 0.9;
        double interestpermonth=interestRate/loanTerm;

    protected double calcMonthlyPayment(double loanAmount)
     {
      double EMI= (loanAmount * interestpermonth) / ((1.0) - ((1.0) /    Math.pow(1.0 +
    interestpermonth, loanTerm)));
     return(Math.round(EMI));
     }
   }
}
```

Inner class, `EMICalculatorHelper`



The example in the slide demonstrates an inner class, `EMICalculatorHelper`, which is defined in the `BankEMICalculator` class.

# What Are Enums?

- An enum is a special data type that represents a fixed set of constants.
- For example:
  1. Directions
  2. Colors of the rainbow
  3. Planets in the solar system
- Defining an enum:

```
public enum Directions {

NORTH,SOUTH, EAST, WEST//; semi-colon is optional here

}
```

enum values are declared in capitals as they are constants

- Accessing an enum values:

  Enum constants can be accessed as `Directions.EAST` and `Directions.SOUTH`.

## Complex Enums

Enums can have fields, methods and constructors.

```
public enum Department {

  HR("DEPT-01"), OPERATIONS("DEPT-02"), LEGAL("DEPT-03"), MARKETING("DEPT-
     O4");//semi-colon is not optional here

  Department(String deptCode){
    this.deptCode=deptCode;
  }

  private String deptCode;

  public String getDeptCode(){
    return deptCode;

 }
}
```



### Enum Constructors

You may not instantiate an enum instance with `new`.

## Methods in Enums

Enum has a few special methods and are very useful.

- `values()`: Returns an array of all enum constants of that enum type

```
public class EnumTest {
  public static void main(String args[]) {

    for(Department dept:Department.values()){
      System.out.println(dept+" Department Code:
     "+dept.getDeptCode());
    }
  }
}
```

Output:

```
HR Department Code: DEPT-01
OPERATIONS Department Code: DEPT-02
LEGAL Department Code: DEPT-03
MARKETING Department Code: DEPT-04
```




All enums implicitly extend java.lang.Enum<E>. Methods name(), ordinal() and valueOf(), inherited from Enum<E>, are available on all enums.

## Methods in Enums

- `ordinal()`: Returns an `int` value equal to the enum constant's ordinal position in enum declaration, starting from the value 0

```
public class EnumTest {
public static void main(String args[]) {
  for(Department dept:Department.values()){
    System.out.println("dept+"ordinal value-> "+dept.ordinal());
  }
 }
}
```

Output:

```
HR ordinal value-> 0
OPERATIONS ordinal value-> 1
LEGAL ordinal value-> 2
MARKETING ordinal value-> 3
```



### Enum Constructors

You cannot instantiate an enum instance with `new` operator.

# Summary

In this lesson, you should have learned how to:

- Create Java classes
- Use encapsulation in Java class design
- Construct abstract Java classes and subclasses
- Override methods
- Use virtual method invocation
- Use varargs to specify variable arguments
- Apply the final keyword in Java
- Distinguish between top-level and nested classes
- Use enumerations

# Practice 2: Overview

This practice covers the following topics:

- Practice 2-1: Overriding and overloading methods
- Practice 2-2: Using Java enumerations

# 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.

# Quiz

Which two of the following should an abstract method not have to compile successfully?

a. A return value
b. A method implementation
c. Method parameters
d. `private` access

# Quiz

A final field (instance variable) can be assigned a value either when declared or in all constructors.

a. True
b. False