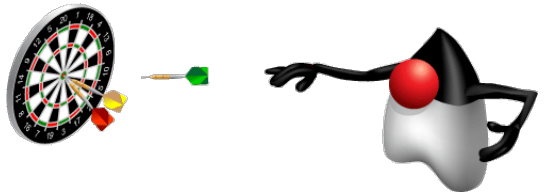
Overriding Methods, Polymorphism, and Static Classes

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

Protected Access Control: Example

```
1 package demo;
2 public class Foo {
3     protected int result = 20;
4     int num= 25;
5 }
```

subclass-friendly declaration

```
1 package test;
2 import demo.Foo;
3 public class Bar extends Foo {
4      private int sum = 10;
5      public void reportSum () {
6          sum += result;
7          sum +=num;
8      }
9 }
```

compiler error

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.

```
1 package demo;
2 public class Foo3 {
3    private int result = 20;
4    protected int getResult() {
5     return this.result;
6    }
7 }
```

```
1 package test;
2 import demo.Foo3;
3 public class Bar3 extends Foo3 {
4     private int sum = 10;
5     public void reportSum() {
6         sum += getResult();
7     }
8 }
```

Overriding Methods

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

```
 3 public class Employee {
 4     private int empId;
 5     private String name;
14     // Lines omitted
15
16     public String getDetails() {
17       return "ID: " + empId + " Name: " + name;
18     }
```

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:

```
 3 public class Manager extends Employee {
 4     private String deptName;
17     // Lines omitted
18
19     @Override
20     public String getDetails() {
21       return super.getDetails () +
22         " Dept: " + deptName;
23     }
```

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

Invoking an Overridden Method

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

```
 5   public static void main(String[] args) {
 6     Employee e = new Employee(101, "Jim Smith",
 7         "011-12-2345", 100_000.00);
 8     Manager m = new Manager(102, "Joan Kern",
 9         "012-23-4567", 110_450.54, "Marketing");
10
11     System.out.println(e.getDetails());
12     System.out.println(m.getDetails());
13   }
```

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

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

Virtual Method Invocation

- What happens if you have the following?

```
 5   public static void main(String[] args) {
 6     Employee e = new Manager(102, "Joan Kern",
 7         "012-23-4567", 110_450.54, "Marketing");
 8
 9     System.out.println(e.getDetails());
10   }
```

- 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*.

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() { ... }
}
```

```
 3 public class BadManager extends Employee {
 4     private String deptName;
 5     // lines omitted
20     @Override
21     private String getDetails() { // Compile error
22       return super.getDetails () +
23         " Dept: " + deptName;
24     }
```

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):

```
 3 public class BadBonus {
 4   public double getBonusPercent(Employee e){
 5     return 0.01;
 6   }
 7
 8   public double getBonusPercent(Manager m){
 9     return 0.03;
10   }
11
12   public double getBonusPercent(Engineer e){
13     return 0.01;
14   }
// Lines omitted
```

not very object-oriented!

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.

Using the `instanceof` Keyword

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

```
 3 public class GoodBonus {
 4   public static double getBonusPercent(Employee e){
 5     if (e instanceof Manager){
 6       return 0.03;
 7     }else if (e instanceof Director){
 8       return 0.05;
 9     }else {
10       return 0.01;
11     }
12   }
13 }
```

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.

`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) { ... }
```

Overriding `equals` in `Employee`

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

```
1 @Override
2 public boolean equals (Object o) {
3     boolean result = false;
4     if ((o != null) && (o instanceof Employee)) {
5         Employee e = (Employee)o;
6         if ((e.empId == this.empId) &&
7             (e.name.equals(this.name)) &&
8             (e.ssn.equals(this.ssn)) &&
9             (e.salary == this.salary)) {
10                result = true;
11        }
12     }     return result;
13 }
```

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.

```
1 @Override //generated by NetBeans
2 public int hashCode() {
3     int hash = 7;
4     hash = 83 * hash + this.empId;
5     hash = 83 * hash + Objects.hashCode(this.name);
6     hash = 83 * hash + Objects.hashCode(this.ssn);
7     hash = 83 * hash + (int)
(Double.doubleToLongBits(this.salary) ^
(Double.doubleToLongBits(this.salary) >>> 32));
8     return hash;
9 }
```

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 Using Variable Arguments

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

```
1 public class Statistics {
2     public float average(int... nums) {
3         int sum = 0;
4         for (int x : nums) {  // iterate int array nums
5             sum += x;
6         }
7         return ((float) sum / nums.length);
8     }
9 }
```

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.

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:

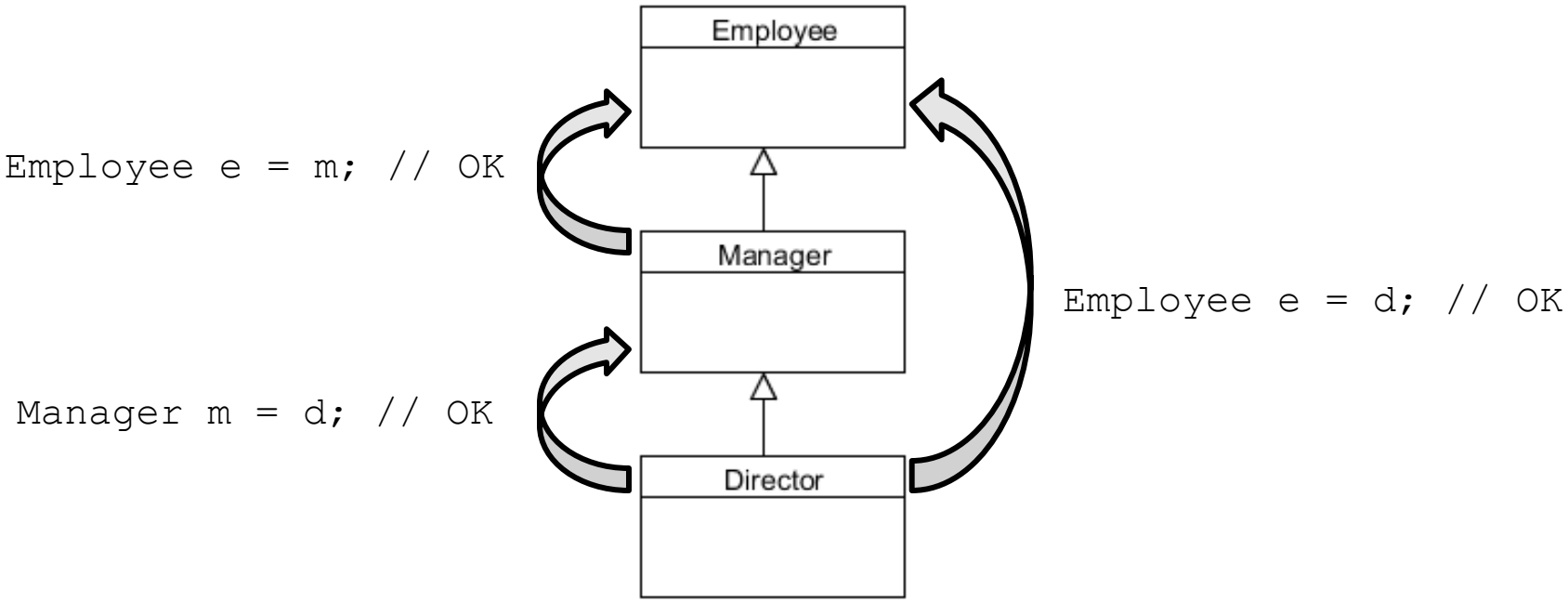
```
 4   public static void main(String[] args) {
 5     Employee e = new Manager(102, "Joan Kern",
 6         "012-23-4567", 110_450.54, "Marketing");
 7
 8     if (e instanceof Manager){
 9       Manager m = (Manager) e;
10       m.setDeptName("HR");
11       System.out.println(m.getDetails());
12     }
13   }
```

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

Upward Casting Rules

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

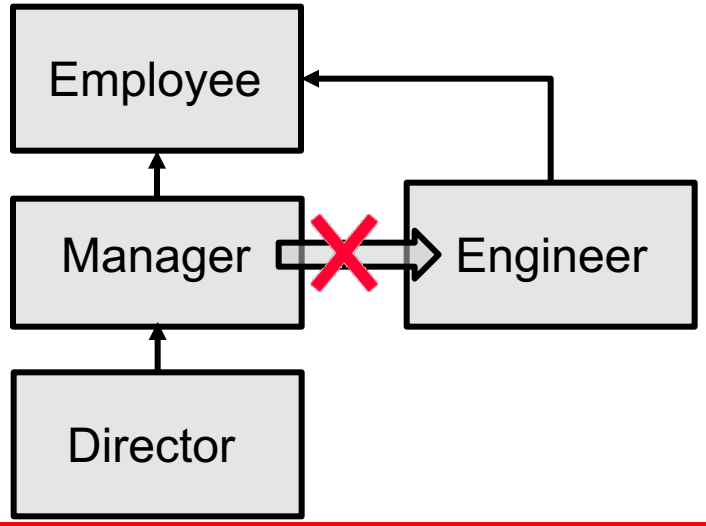
```
Director d = new Director();
Manager m = new Manager();
```



Downward Casting Rules

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

```
 5      Employee e = new Manager(102, "Joan Kern",
 6          "012-23-4567", 110_450.54, "Marketing");
 7
 8      Manager m = (Manager)e; // ok
 9      Engineer eng = (Manager)e; // Compile error
10      System.out.println(m.getDetails());
```



`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

Using Static Variables and Methods: Example

```
 3 public class A01MathTest {
 4   public static void main(String[] args) {
 5     System.out.println("Random: " + Math.random() * 10);
 6     System.out.println("Square root: " + Math.sqrt(9.0));
 7     System.out.println("Rounded random: " +
 8         Math.round(Math.random()*100));
 9     System.out.println("Abs: " + Math.abs(-9));
10   }
11 }
```

Implementing Static Methods

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

```
 3 import java.time.LocalDate;
 4
 5 public class StaticHelper {
 6
 7     public static void printMessage(String message) {
 8         System.out.println("Messsage for " +
 9          LocalDate.now() + ": " + message);
10     }
11
12 }
```

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

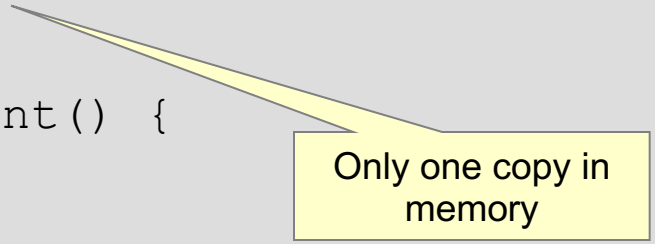
Static variables are variables that can be accessed even if the class they are declared in has not been instantiated.

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

Defining Static Variables

```
 4 public class StaticCounter {
 5     private static int counter = 0;
 6
 7     public static int getCount() {
 8         return counter;
 9     }
10
11     public static void increment(){
12       counter++;
13     }
14 }
```



Using Static Variables

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

```
 5   public static void main(String[] args) {
 6     System.out.println("Start: " + StaticCounter.getCount());
 7     StaticCounter.increment();
 8     StaticCounter.increment();
 9     System.out.println("End: " + StaticCounter.getCount());
10}
```

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

Static Initializers

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

```
 3 public class A04StaticInitializerTest {
 4   private static final boolean[] switches = new boolean[5];
 5
 6   static{
 7     System.out.println("Initializing...");
 8     for (int i=0; i<5; i++){
 9       switches[i] = true;
10     }
11   }
12
13   public static void main(String[] args) {
14     switches[1] = false; switches[2] = false;
15     System.out.print("Switch settings: ");
16     for (boolean curSwitch:switches){
17       if (curSwitch){System.out.print("1");}
18       else {System.out.print("0");}
19     }
```

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();
    }
}
```

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

Singleton Pattern

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

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

    private SingletonClass() {}

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

1

2

3

Singleton: Example

```
 3 public final class DbConfigSingleton {
 4   private final String hostName;
 5   private final String dbName;
 6   //Lines omitted
10   private static final DbConfigSingleton instance =
11             new DbConfigSingleton();
12
13   private DbConfigSingleton(){
14     // Values loaded from file in practice
15     hostName = "dbhost.example.com";
16     // Lines omitted
20   }
21
22   public static DbConfigSingleton getInstance() {
23     return instance;
24   }
```

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.

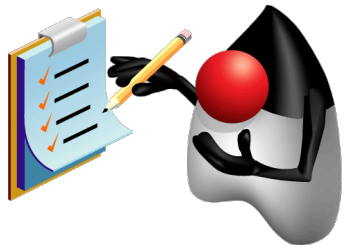
Example: Creating Immutable class in Java

```
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;

}
}
```

Summary

In this lesson, you should have learned how to:

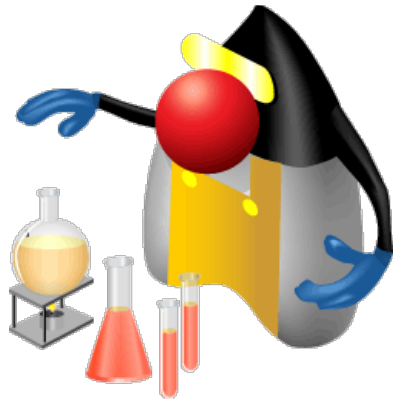
- Use access levels: `private`, `protected`, default, and `public`.
- Override methods
- Use virtual method invocation
- Use `varargs` to specify variable arguments
- Use the `instanceof` operator to compare object types
- Use upward and downward casts
- Model business problems by using the `static`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



Practice 4-1 Overview: Overriding Methods and Applying Polymorphism

This practice covers the following topics:

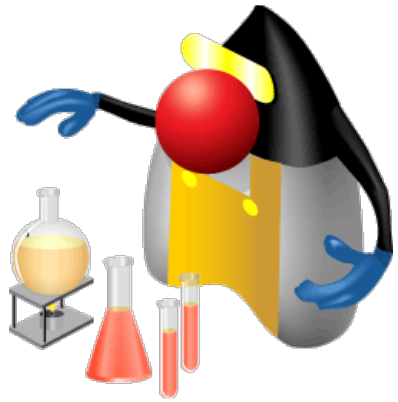
- Modifying the `Employee`, `Manager`, and `Director` classes; overriding the `toString()` method
- Creating an `EmployeeStockPlan` class with a grant stock method that uses the `instanceof` keyword



Practice 4-2 Overview: Overriding Methods and Applying Polymorphism

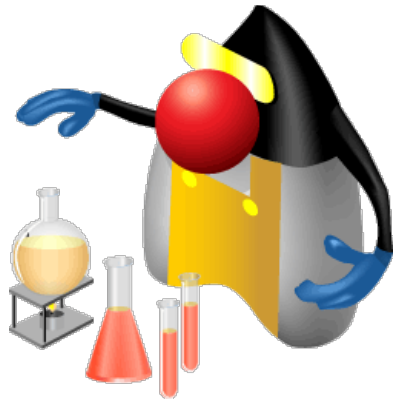
This practice covers the following topics:

- 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.