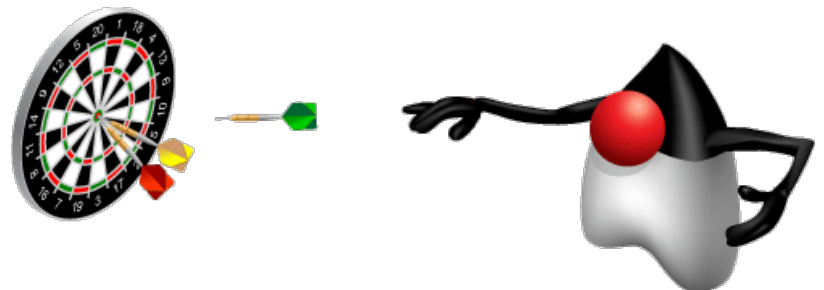


Abstract and Nested Classes

Objectives

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

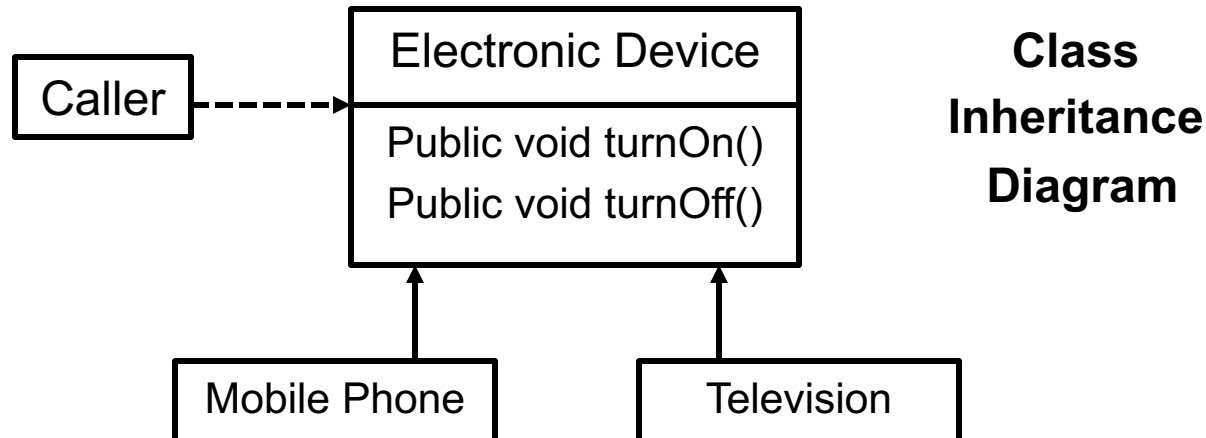
- Design general-purpose base classes by using abstract classes
- Construct abstract Java classes and subclasses
- Apply the `final` keyword in Java
- Distinguish between top-level and nested classes



Modeling Business Problems with Classes

Inheritance (or subclassing) is an essential feature of the Java programming language. Inheritance provides code reuse through:

- **Method inheritance:** Subclasses avoid code duplication by inheriting method implementations.
- **Generalization:** Code that is designed to rely on the most generic type possible is easier to maintain.



Enabling Generalization

Coding to a common base type allows for the introduction of new subclasses with little or no modification of any code that depends on the more generic base type.

```
ElectronicDevice dev = new Television();  
dev.turnOn(); // all ElectronicDevices can be turned on
```

Always use the most generic reference type possible.

Identifying the Need for Abstract Classes

Subclasses may not need to inherit a method implementation if the method is specialized.

```
public class Television extends ElectronicDevice {  
  
    public void turnOn() {  
        changeChannel(1);  
        initializeScreen();  
    }  
    public void turnOff() {}  
  
    public void changeChannel(int channel) {}  
    public void initializeScreen() {}  
  
}
```

Defining Abstract Classes

A class can be declared as abstract by using the `abstract` class-level modifier.

```
public abstract class ElectronicDevice { }
```

- An abstract class can be subclassed.

```
public class Television extends ElectronicDevice { }
```

- An abstract class **cannot** be instantiated.

```
ElectronicDevice dev = new ElectronicDevice(); // error
```

Defining Abstract Methods

A method can be declared as abstract by using the `abstract` method-level modifier.

```
public abstract class ElectronicDevice {  
  
    public abstract void turnOn();  
    public abstract void turnOff();  
}
```

No braces

An abstract method:

- Cannot have a method body
- Must be declared in an abstract class
- Is overridden in subclasses

Validating Abstract Classes

The following additional rules apply when you use abstract classes and methods:

- An abstract class may have any number of abstract and nonabstract methods.
- When inheriting from an abstract class, you must do either of the following:
 - Declare the child class as abstract.
 - Override all abstract methods inherited from the parent class. Failure to do so will result in a compile-time error.

```
error: Television is not abstract and does not override  
abstract method turnOn() in ElectronicDevice
```


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");  
    }  
}
```

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 { }
```

Final Variables

The `final` modifier can be applied to variables.

Final variables may not change their values after they are initialized.

Final variables can be:

- Class fields
 - Final fields with compile-time constant expressions are constant variables.
 - Static can be combined with final to create an always-available, never-changing variable.
- Method parameters
- Local variables

Note: Final references must always reference the same object, but the contents of that object may be modified.

Declaring Final Variables

```
public class VariableExampleClass {  
    private final int field;  
    public static final int JAVA_CONSTANT = 10;  
  
    public VariableExampleClass() {  
        field = 100;  
    }  
  
    public void changeValues(final int param) {  
        param = 1; // compile-time error  
        final int localVar;  
        localVar = 42;  
        localVar = 43; // compile-time error  
    }  
}
```

Nested Classes

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

- 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

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

Enumerations

Java includes a typesafe enum to the language.

Enumerations (enums):

- Are created by using a variation of a Java class
- Provide a compile-time range check

```
public enum PowerState {  
    OFF,  
    ON,  
    SUSPEND;  
}
```

These are references to the only three `PowerState` objects that can exist.

An enum can be used in the following way:

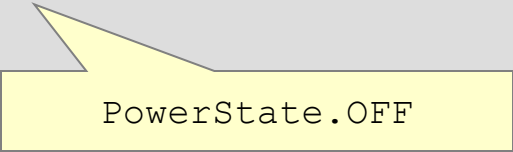
```
Computer comp = new Computer();  
comp.setState(PowerState.SUSPEND);
```

This method takes a `PowerState` reference .

Enum Usage

Enums can be used as the expression in a switch statement.

```
public void setState(PowerState state) {  
    switch(state) {  
        case OFF:  
            //...  
    }  
}
```



PowerState.OFF

Complex Enums

Enums can have fields, methods, and private constructors.

```
public enum PowerState {  
    OFF("The power is off"),  
    ON("The usage power is high"),  
    SUSPEND("The power usage is low");  
  
    private String description;  
    private PowerState(String d) {  
        description = d;  
    }  
    public String getDescription() {  
        return description;  
    }  
}
```

Call a `PowerState` constructor to initialize the public static final `OFF` reference.

The constructor may not be public or protected.

Complex Enums

- Here is the complex enum in action.

```
public class ComplexEnumsMain {  
  
    public static void main(String[] args) {  
        Computer comp = new Computer();  
        comp.setState(PowerState.SUSPEND);  
        System.out.println("Current state: " +  
comp.getState());  
        System.out.println("Description: " +  
comp.getState().getDescription());  
    }  
}
```

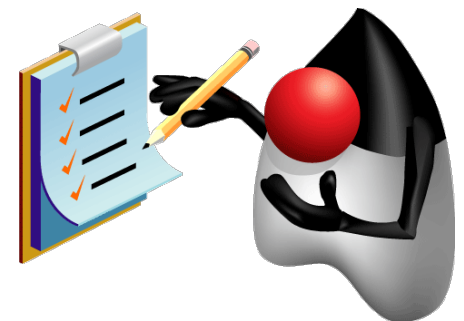
- Output

```
Current state: SUSPEND  
Description: The power usage is low
```

Summary

In this lesson, you should have learned how to:

- Design general-purpose base classes by using abstract classes
- Construct abstract Java classes and subclasses
- Apply the `final` keyword in Java
- Distinguish between top-level and nested classes

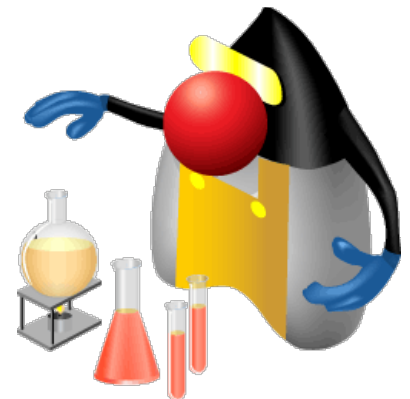


Practice 5-1 Overview:

Applying the Abstract Keyword

This practice covers the following topics:

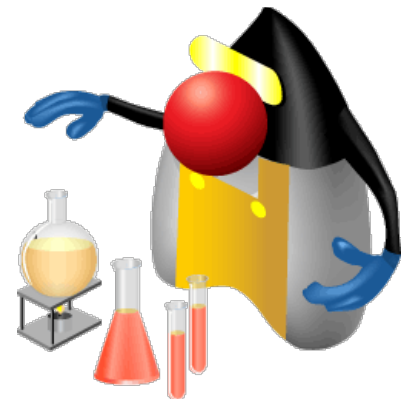
- Identifying potential problems that can be solved using abstract classes
- Refactoring an existing Java application to use abstract classes and methods



Practice 5-2 Overview:

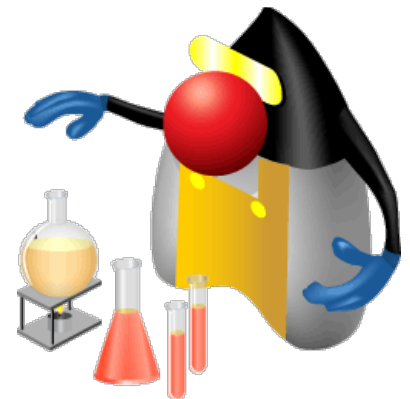
Using Inner Class As a Helper Class

This practice covers using an inner class as a helper class to perform some calculations in an Employee class.



Practice 5-3 Overview: Using Java Enumerations

This practice covers taking an existing application and refactoring the code to use an enum.



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

Which of the following nested class types are inner classes?

- a. Anonymous
- b. Local
- c. Static
- d. Member

Quiz

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

- a. True
- b. False