

Week two

Objectives – By Day

- **Java Packages, Interfaces and Advanced Classes**
- Exception Processing and Threads
- Data Structures
- Java Swing
- Swing Interfaces

Day 6 - Java Packages, Interfaces and Advanced Class Development

Modifiers – a set of keywords that are added to the class definition to change its meaning

- `protected`, `public`, `private` – used to control access to a class, method or variable
- `static` – used to create class methods and variables – we've been using this one all along
- `final` – finalizes the implementation of classes, methods and variables
- `abstract` – creates abstract classes and methods
- `synchronized` and `volatile` – are modifiers for threads (more on this later)

Modifiers (2)

To use the modifier – include the keyword in the definition of the class

Modifiers are optional but can enhance/influence the functionality of the program

```
public class RedButton extends javax.swing.JButton {  
    // some code for the class goes here  
}  
private boolean offline;  
static final double WEEKS = 9.5;  
protected static final int MEANING_OF_LIFE = 42;  
public static void main(String[] arguments) {  
    // some code for the method goes here  
}
```

Access Control

Modifiers – public, private, protected and default

Control access to methods and variables

Determines which methods and variables are visible to other classes

Modifiers for AC

Default Access – no access control modifiers
– available for read and change by any other class in the same package – not much access control in play

Private Access – completely hides a method or variable from outside to prevent usage

- Can be used only within its own class
- Not inherited by the subclass
- Variables of public within the class are available as defined
- In the example, the private is available thru getFormat and

```
class Logger {  
    private String format;  
    public String getFormat() {  
        return this.format;  
    }  
}  
  
public void setFormat (String format){  
    if ((format.equals("common") || (format.equals("combined")))) {  
        this.format = format;  
    }  
}
```

Modifiers for AC (2)

Public Access – make a method or variable completely available

- E.g. Color in java.awt have public variables for the color red or blue
- Ability to not limit common variables – there is no benefit in doing so
- Everything we've coded so far has been public
- Inherited by its subclasses

Protected Access – Limits use to only the subclasses of a class or other classes in the same package

- Don't have to be in the same package `protected boolean outOfDate = true;`
- Gives access to subclass but prevents an unrelated class from trying to use it

In this exercise we demonstrate the usage of an Abstract Class while exercising all modifiers

Run as Java Application

Exercise: Modifiers

BaseClass.java

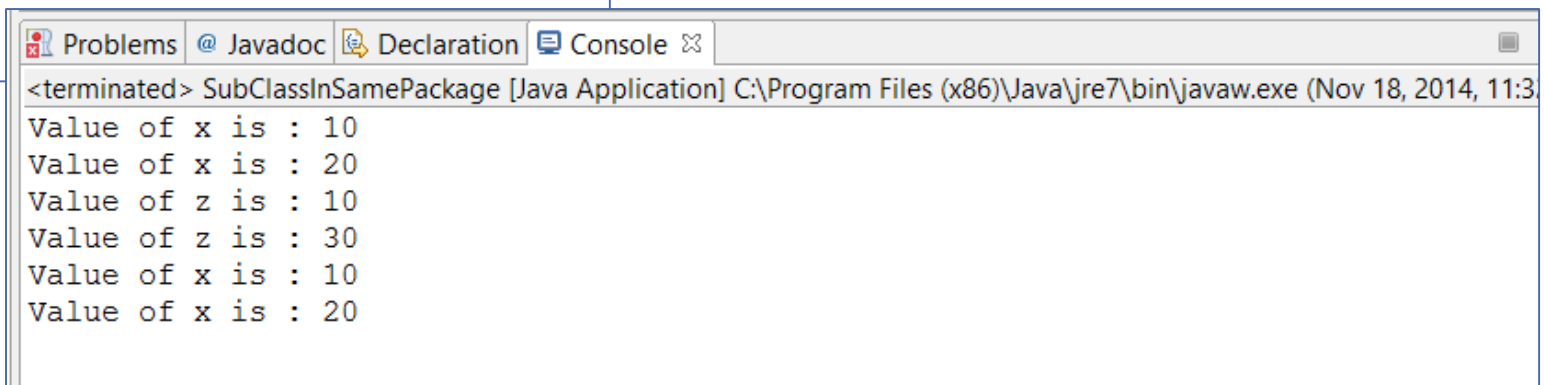
```
1 class BaseClass {
2
3     public int x = 10;
4     private int y = 10;
5     protected int z = 10;
6     int a = 10; //Implicit Default Access Modifier
7     public int getX() {
8         return x;
9     }
10    public void setX(int x) {
11        this.x = x;
12    }
13    private int getY() {
14        return y;
15    }
16    private void setY(int y) {
17        this.y = y;
18    }
19    protected int getZ() {
20        return z;
21    }
22    protected void setZ(int z) {
23        this.z = z;
24    }
```

```
25    int getA() {
26        return a;
27    }
28    void setA(int a) {
29        this.a = a;
30    }
31 }
32
```

Exercise Modifiers – SubClassInSamePackage.java

```
1 public class SubClassInSamePackage extends BaseClass {
2
3     public static void main(String args[]) {
4         BaseClass rr = new BaseClass();
5         rr.z = 0;
6         SubClassInSamePackage subClassObj = new SubClassInSamePackage();
7         //Access Modifiers - Public
8         System.out.println("Value of x is : " + subClassObj.x);
9         subClassObj.setX(20);
10        System.out.println("Value of x is : " + subClassObj.x);
11        //Access Modifiers - Public
12        //    If we remove the comments it would result in a compilation
13        //    error as the fields and methods being accessed are private
14        /*    System.out.println("Value of y is : "+subClassObj.y);
15
16        subClassObj.setY(20);
17    }
```

```
7
8        System.out.println("Value of y is : "+subClassObj.y);*/
9        //Access Modifiers - Protected
10       System.out.println("Value of z is : " + subClassObj.z);
11       subClassObj.setZ(30);
12       System.out.println("Value of z is : " + subClassObj.z);
13       //Access Modifiers - Default
14       System.out.println("Value of x is : " + subClassObj.a);
15       subClassObj.setA(20);
16       System.out.println("Value of x is : " + subClassObj.a);
17   }
```



The screenshot shows an IDE window with tabs for Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of the Java application. The output consists of six lines of text, alternating between the value of 'x' and 'z' as printed by the program.

```
<terminated> SubClassInSamePackage [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Nov 18, 2014, 11:3
Value of x is : 10
Value of x is : 20
Value of z is : 10
Value of z is : 30
Value of x is : 10
Value of x is : 20
```


Exercise: Public Modifier

Create two classes:

MathUtil.java -- accept values from the calling program then compute the cubedRoot(), circumference() or the area().

MyProgram.java – test the methods of the MathUtil class.

Run as a Java Application

```
1 public class MathUtil {
2     public double cubedRoot(int num) {
3         return Math.pow(num, 1.0/3);
4     }
5
6     public double circumference(double radius) {
7         return 2 * Math.PI * radius;
8     }
9
10    public double area(double radius) {
11        return Math.PI * radius * radius;
12    }
13 }
```

```
Puzzle4.java  MathUtil.java  MyProgram.java x
1 public class MyProgram {
2
3     public static void main(String [] args) {
4
5         MathUtil myMathUtil = new MathUtil();
6
7         // area of a circle
8         double circleArea = myMathUtil.area(3);
9
10        // cubed root of 64
11        float root = (float) myMathUtil.cubedRoot(64);
12
13        System.out.println("Area of circle radius 3: " + circleArea);
14        System.out.println("Cubed root of 64: " + root);
15    }
16 }
```

```
Problems  Javadoc  Declaration  Console x
<terminated> MyProgram [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Nov 18, 2014, 10:41:19 PM)
Area of circle radius 3: 28.27433882308138
Cubed root of 64: 4.0
```

Comparing Access Control Methods

Location	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package and also a subclass	No	Yes	Yes	Yes
Same package but not a subclass	No	Yes	Yes	Yes
Different package but a subclass	No	No	Yes	Yes
Different package but not a subclass	No	No	No	Yes

Static Variables and Methods

Static – create class methods and variables

Class variables and methods are accessed using the class by dot notation -- Color.red; or Circle.PI;

Here are a few examples:

```
float circumference = 2 * Circle.PI * getRadius();
```

```
float randomNumber = Math.random();
```

Exercise 1: InstanceCounter.java – Pg 122

Create a class called InstanceCounter.java that uses class and instance variables to keep track of how many objects of that class that can be created

Run as a Java application

InstanceCounter.java

```
1 public class InstanceCounter {
2     private static int numInstances = 0;
3
4     protected static int getCount() {
5         return numInstances;
6     }
7
8     private static void addInstance() {
9         numInstances++;
10    }
11
12    InstanceCounter() {
13        InstanceCounter.addInstance();
14    }
15
16    public static void main(String[] arguments) {
17        System.out.println("Starting with " +
18            InstanceCounter.getCount() + " objects");
19        for (int i = 0; i < 500; ++i)
20            new InstanceCounter();
21        System.out.println("Created " +
22            InstanceCounter.getCount() + " objects");
23    }
24 }
```

Problems @ Javadoc Declaration Console

<terminated> InstanceCounter [Java Application] C:\Program Files\Autopsy-3.0.8\jre\bin\javaw.exe (Oct 23, 2014, 7:51:06 PM)

Starting with 0 objects
Created 500 objects

Final Classes and Methods

final – used with classes, methods and variables to indicate they will never change

The meaning of final is driven by what is being created

- final class cannot be a subclass
- final method cannot be overridden by any subclass
- final variable cannot change its value

final Variables

Constants

Often used with static to make it a class variable (class variables are written in all caps)

```
public static final int TOUCHDOWN = 6;  
public final String TITLE = "Captain";
```

final Methods

Cannot override in a subclass

Declared final in the class declarations

Usually declared as final for performance reasons

Private methods are always final by default

```
public final void getSignature() {  
    // body of method  
}
```


final Classes

Cannot be subclassed by another class – if need a class that behaves like a String (for example) we must build a new class from the beginning

Improves performance

Methods in a final class are automatically final themselves

```
public final class ChatServer {  
    // body of method  
}
```

Abstract Classes and Methods

Abstract Class – A class that never needs to be instantiated directly

Holds common behavior and attributes that are shared by all subclasses

Uses the abstract modifier

Can contain anything a normal class can

Can contain abstract methods (method signatures with no implementation)

Cannot declare an abstract method unless the class is too abstract

```
public abstract class Palette {  
    // ...  
}
```

Exercise: Abstract Class

```
Puzzle4.java  MathUtil.java  MyProgram.java  Animal.java  dog2.java

1 public abstract class Animal {
2     public boolean isAPet = true;
3     public String owner = "Fred";
4
5     public static void sleep(){
6         System.out.println("Sleeping");
7     }
8
9     public static void eat(){
10        System.out.println("eating");
11    }
12
13
14 }
```

Problems Javadoc Declaration Console X

<terminated> dog2 [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Nov 18, 2011)

Sleeping
woof
eating
running

```
public class dog2 extends Animal {
    private int numberOfLegs;
    private boolean hasOwner;

    public dog2() {
        numberOfLegs = 4;
        hasOwner = false;
    }

    private static void bark() {
        System.out.println("woof");
    }

    public static void move() {
        System.out.println("running");
    }

    public static void main(String [] args){
        Animal.sleep();
        bark();
        Animal.eat();
        move();
    }
}
```

Packages

Previous lessons

- We've created classes in the same file as the main or in its own file
- We had to remember which file the class was in
- We had to have the same name as the class – the main class of the file
- Most of what we've done has been marked as public

Packages resolve this issues around using the method

Packages(2)

An organization of a group of classes

Related in purpose, scope or inheritance

Reduces problems with naming collisions

Enables the protection of variables, methods and classes on a larger scale

Packages are imported via the import statement or using the fully qualified name of a class

Packages (3)

Three techniques to accessing the classes of a package

- If the class is in the package `java.lang` (`System` or `Date`), we can use the class name to refer to that class. `java.lang` is automatically available in all programs
- If the class is in some other package, not `java.lang`, we refer to that class by its fully qualified name as in `java.awt.Font`
- If we use classes from a particular package frequently, we can import the individual classes or the entire package then we can refer to the needed class by its class name

Creating Packages

Must have one or more classes or interfaces (more on interfaces in a bit)

- Cannot be empty

Source files of classes or interfaces must reside in the same directory as the package

Name of the package must be unique

Declaring Package Members

Decide which classes and interface should belong in the package

Specify the package declaration in the source file of each class and interface that is part of the package

- Use the syntax: keyword `package` followed by the package name

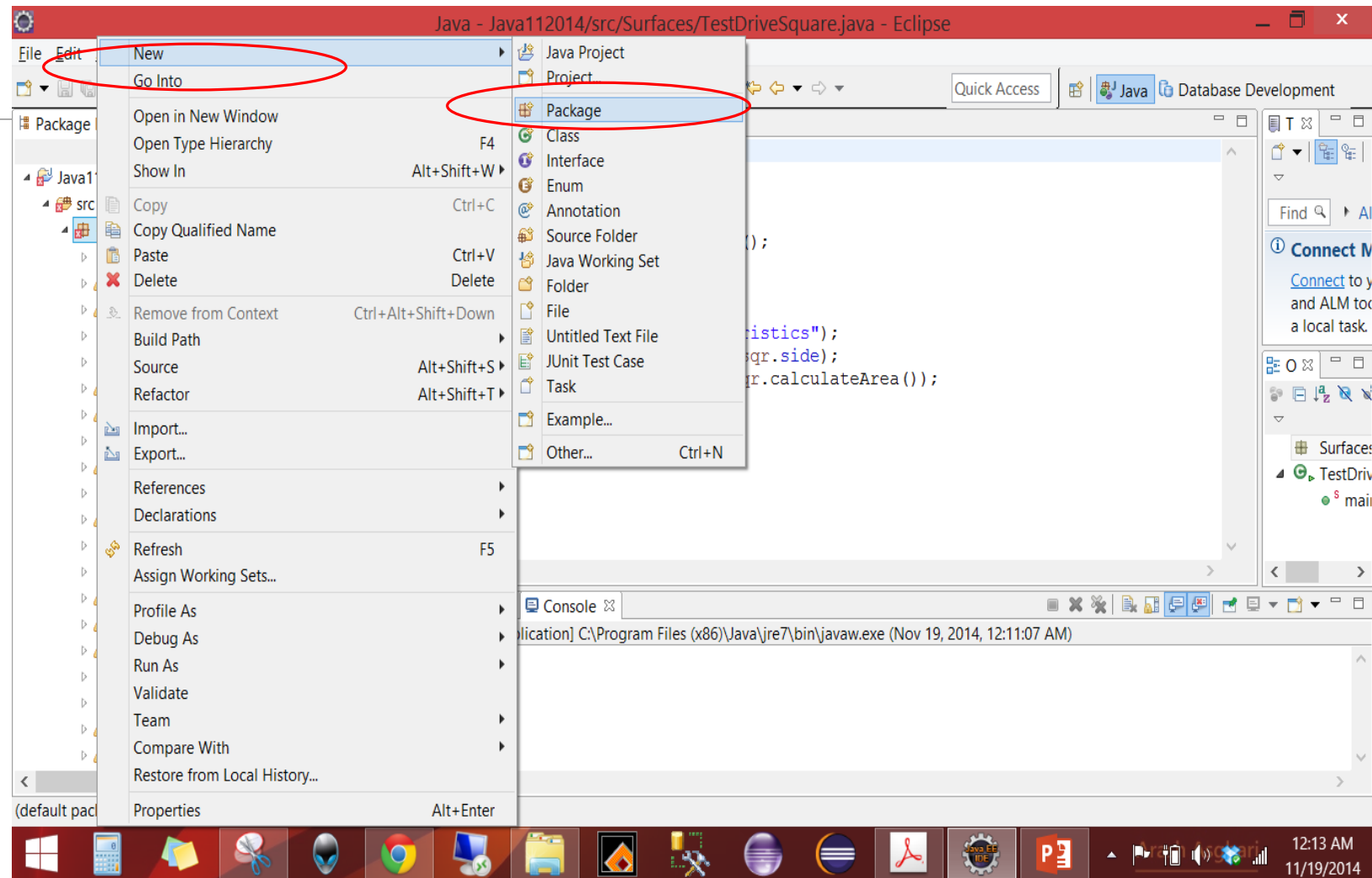
```
package com.appinc-  
4me.myproject;
```

```
/*  
 * File comments...  
 */  
  
package com.mycompany.myproject;  
  
import java.util.*;  
  
class MyClass {  
  
}
```


Exercise: Create and Use a Custom Package

1. Create a Package

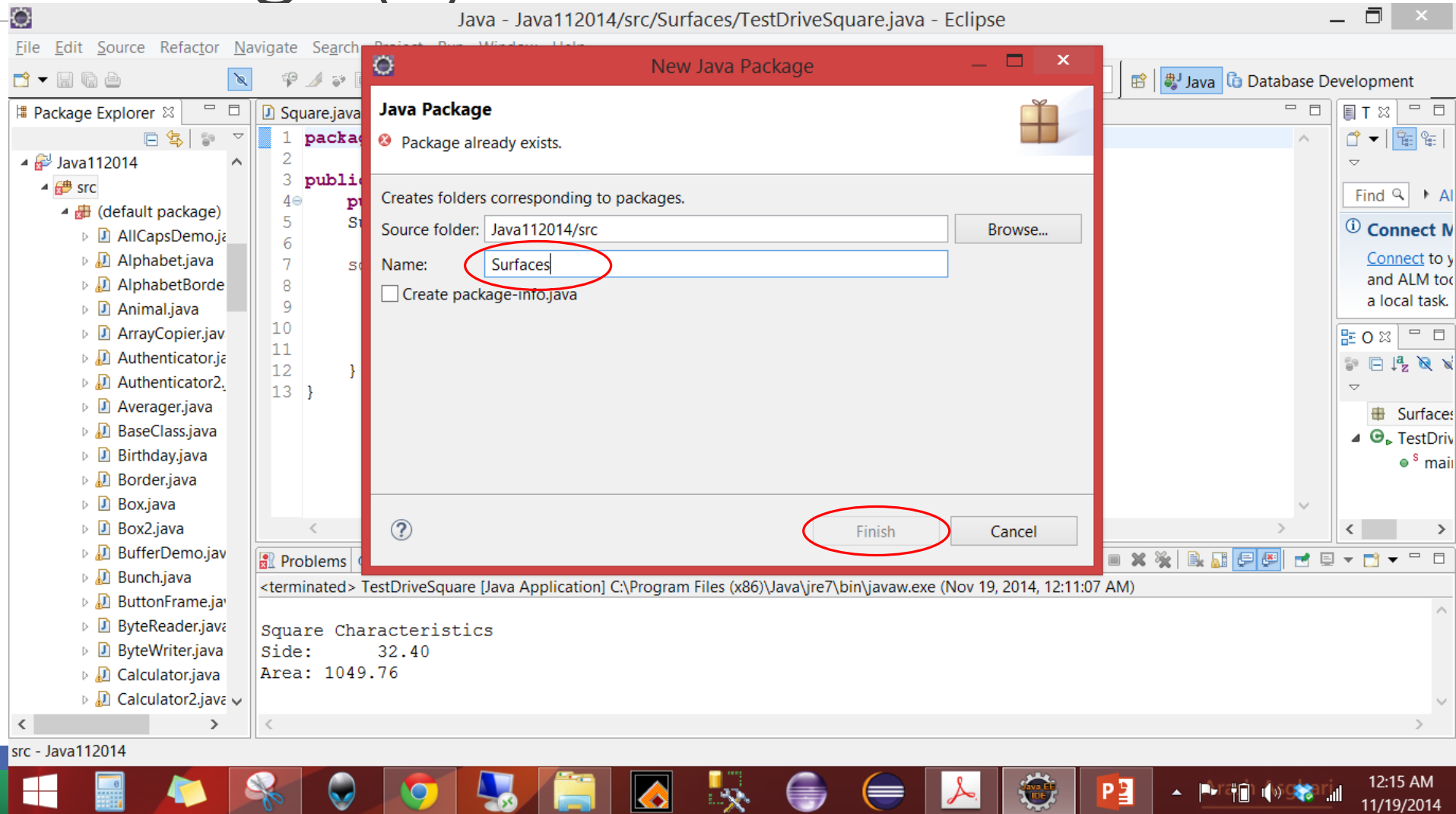
- Right click on src
- Select new
- Select package



Create a Package (2)

2. Give it a name

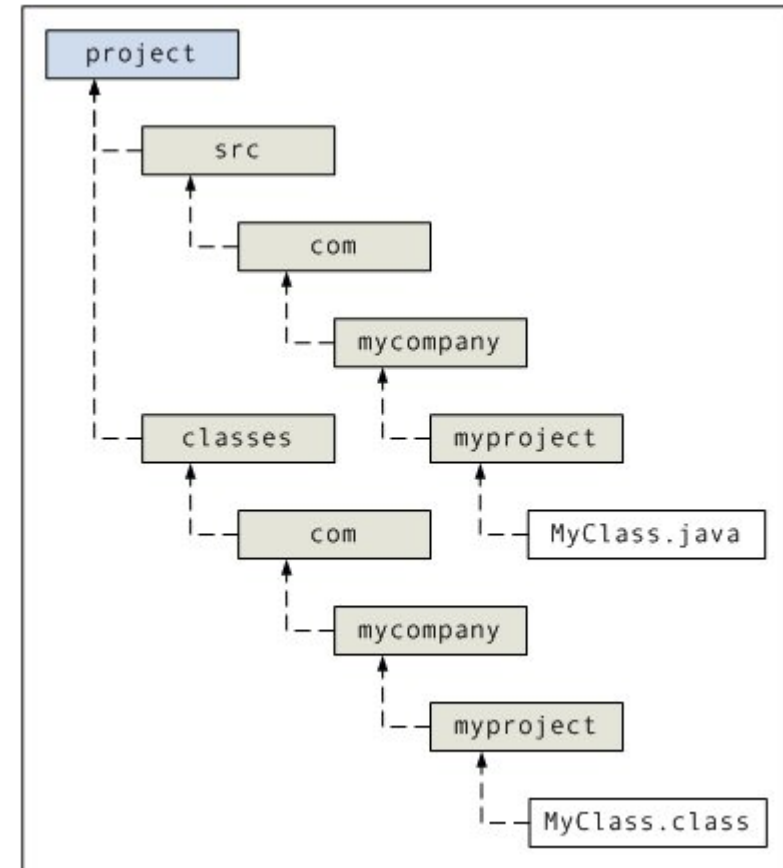
- Name the package Surfaces
- Click Finish



Package and Source File Declaration

Common to split the package name into its components to build the directory hierarchy.

Using Eclipse, we don't have to compile the package



Using Packages

Can be used by many programmers

There are three ways to use a package

- Inline declarations
- Importing only the package members
- Importing the entire package

Packages – Inline declarations

Simply declare the package members we wish to use with its fully qualified name

Using the Vector class of java.util
`java.util.Vector vector;`

Good when we only want to use a single member a few times

We initialize the object by calling its constructor using the fully qualified package name as shown:

```
class Test {  
    java.util.Vector vector;  
  
    Test() {  
        vector = new java.util.Vector();  
    }  
}
```

Packages – Importing a Single Package Member

Used when we need to use the package and its members many times during our code

Once declared, can use its members many times without the fully qualified name

Use the import statement followed by the fully qualified name of the member

```
import java.util.Vector;
class Test {
    Vector vector;

    Test() {
        vector = new Vector();
    }
}
```

Packages – Importing the Entire Package

Used when we need a number of members from a package

Can replace the following with a single statement by using the Wild Card *

```
import java.util.Vector;
import java.util.LinkedList;
import java.util.Hashtable
import java.util.Stack
import java.util.Set

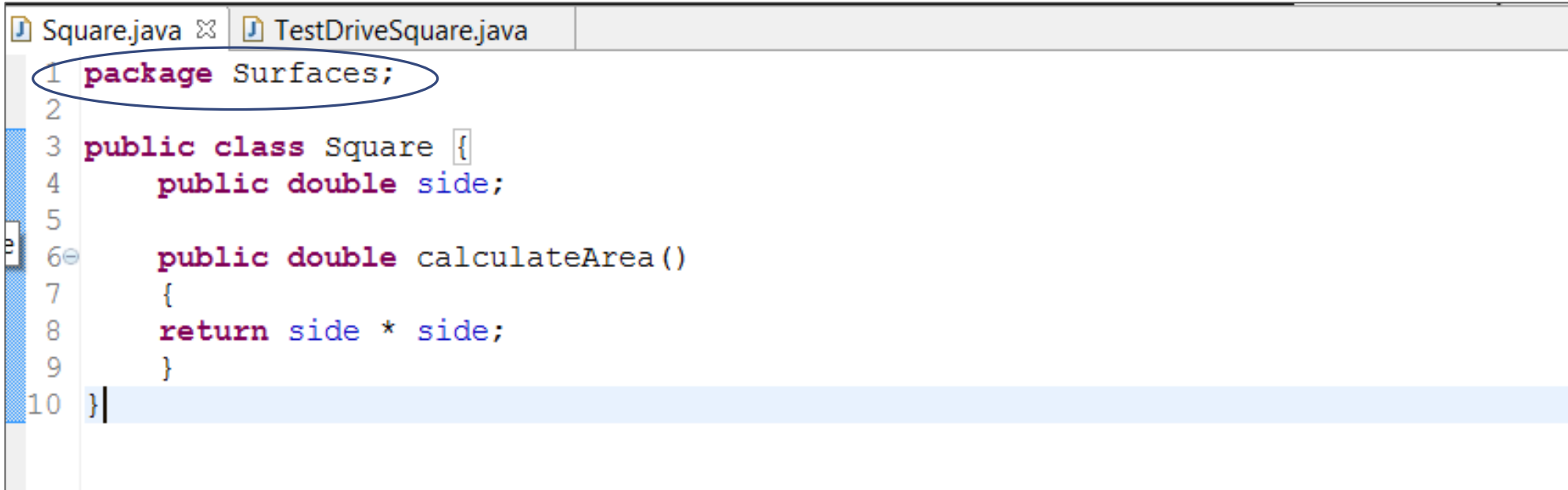
class Test {
    ...
}
```

```
import java.util.*;

class Test {
    ...
}
```

Create a class called Square.java in the Surfaces package

Under the package Surfaces, create the class called Square.java



```
1 package Surfaces;
2
3 public class Square {
4     public double side;
5
6     public double calculateArea()
7     {
8         return side * side;
9     }
10 }
```


Use the Square.java in the TestDriveSquare.java class

Create the tester class for Square.java under the package Surfaces

Run as a Java application

```
1 package Surfaces;
2
3 public class TestDriveSquare {
4     public static void main(String[] args) {
5         Surfaces.Square sqr = new Surfaces.Square();
6
7         sqr.side = 32.40;
8
9         System.out.println("\nSquare Characteristics");
10        System.out.printf("Side:      %.2f", sqr.side);
11        System.out.printf("\nArea: %.2f\n", sqr.calculateArea());
12    }
13 }
```

```
<terminated> TestDriveSquare [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw
Square Characteristics
Side:      32.40
Area: 1049.76
```

Packages – Best Practices

Package by Feature

- Organized according to the feature set – place all items related to a single feature (and only that feature) into a single package
- Placing items that work together in the same package
- Names correspond to important, high-level aspects of the problem domain: `com.app.doctor`, `com.app.patient`, etc.
- Each package contains only the items related to that package feature: `Doctor.java`, `DoctorAction.java`

Package – Best Practices (2)

Highest level packages reflect various abstraction layers
com.app.action, com.app.model, etc.

Implementation over multiple directories

- Can be considered categories

Most use Package by Feature

- Higher modularity
- Easier code navigation
- Higher level of abstraction
- Separates both features and layers
- Minimizes scope
- Better growth style

Design Checklist

Favor placing API (Application Program Interfaces) and implementation into separate packages

- API in higher level packages
- Implementation into lower level packages

Break large APIs into several packages

Abstract Classes and Interfaces

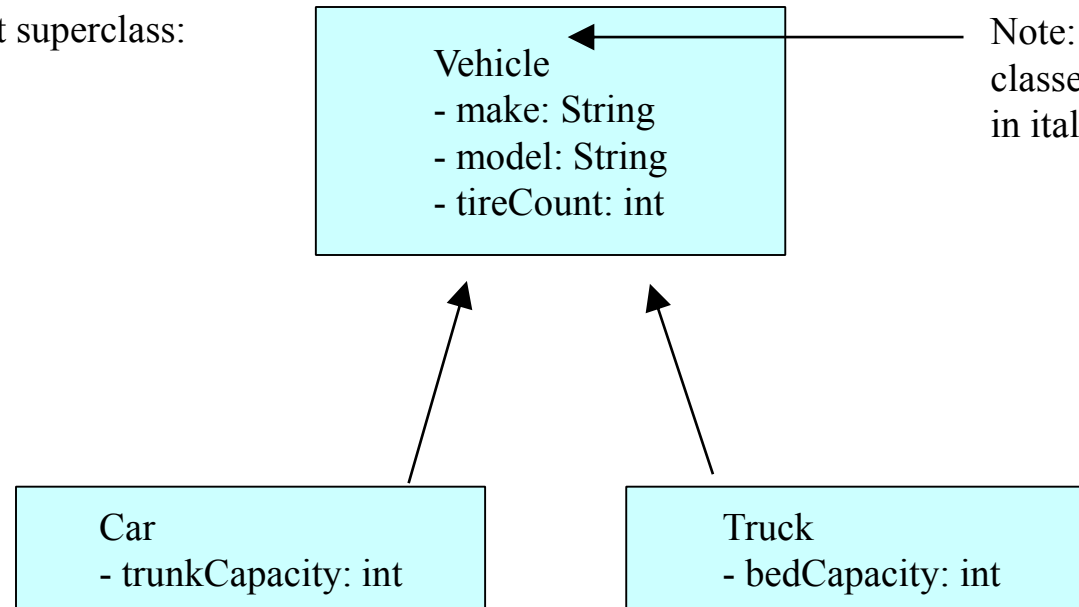
What is an Abstract class?

- Superclasses are created through the process called "generalization"
 - Common features (methods or variables) are factored out of object classifications (ie. classes).
 - Those features are formalized in a class. This becomes the superclass
 - The classes from which the common features were taken become subclasses to the newly created super class
- Often, the superclass does not have a "meaning" or does not directly relate to a "thing" in the real world
 - It is an artifact of the generalization process
- Because of this, abstract classes cannot be instantiated
 - They act as place holders for abstraction

Abstract Class Example

- In the following example, the subclasses represent objects taken from the problem domain.
- The superclass represents an abstract concept that does not exist "as is" in the real world.

Abstract superclass:



Note: UML represents abstract classes by displaying their name in italics.

What Are Abstract Classes Used For?

- Abstract classes are used heavily in Design Patterns
 - Creational Patterns: Abstract class provides interface for creating objects. The subclasses do the actual object creation
 - Structural Patterns: How objects are structured is handled by an abstract class. What the objects do is handled by the subclasses
 - Behavioural Patterns: Behavioural interface is declared in an abstract superclass. Implementation of the interface is provided by subclasses.
- Be careful not to over use abstract classes
 - Every abstract class increases the complexity of your design
 - Every subclass increases the complexity of your design
 - Ensure that you receive acceptable return in terms of functionality given the added complexity.

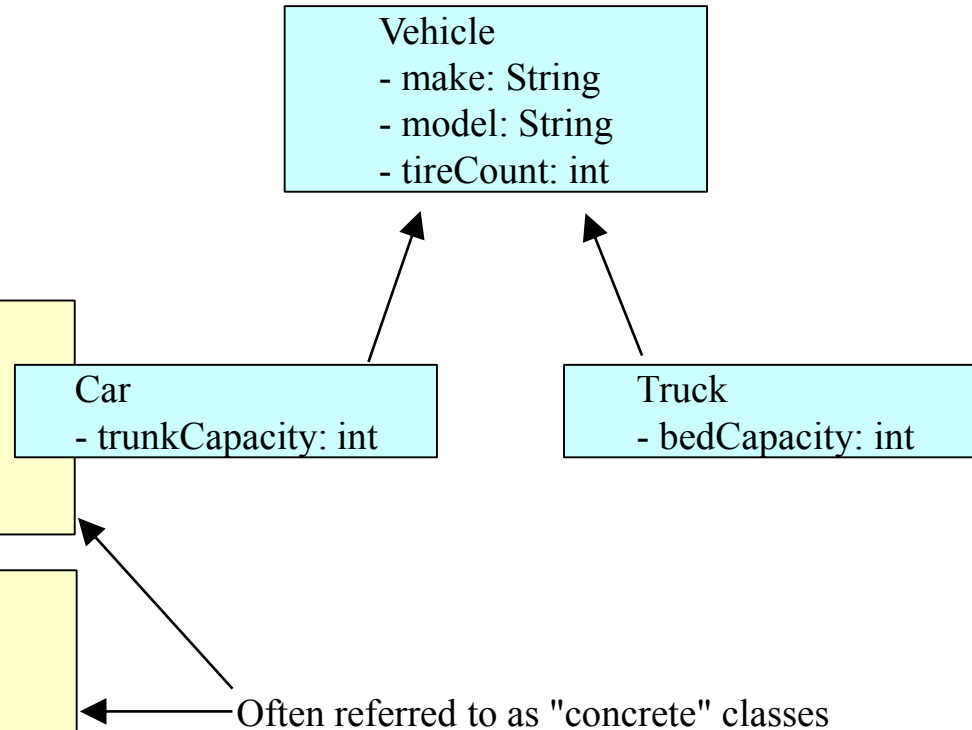
Defining Abstract Classes

- Inheritance is declared using the "extends" keyword
 - If inheritance is not defined, the class extends a class called Object

```
public abstract class Vehicle
{
    private String make;
    private String model;
    private int tireCount;
    [...]
```

```
public class Car extends Vehicle
{
    private int trunkCapacity;
    [...]
```

```
public class Truck extends Vehicle
{
    private int bedCapacity;
    [...]
```

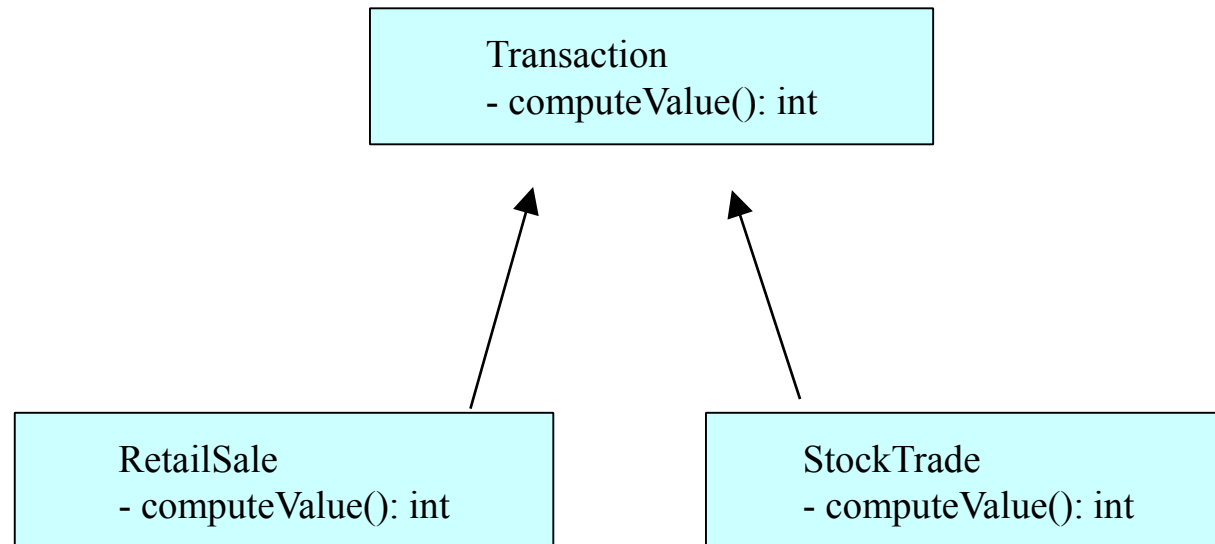


Abstract Methods

- Methods can also be abstract
 - An abstract method is one to which a signature has been provided, but no implementation for that method is given.
 - An Abstract method is a placeholder. It means that we declare that a method must exist, but there is no meaningful implementation for that methods within this class
- Any class which contains an abstract method **MUST** also be abstract
 - Any class which has an incomplete method definition cannot be instantiated (ie. it is abstract)
- Abstract classes can contain both concrete and abstract methods.
 - If a method can be implemented within an abstract class, and implementation should be provided.

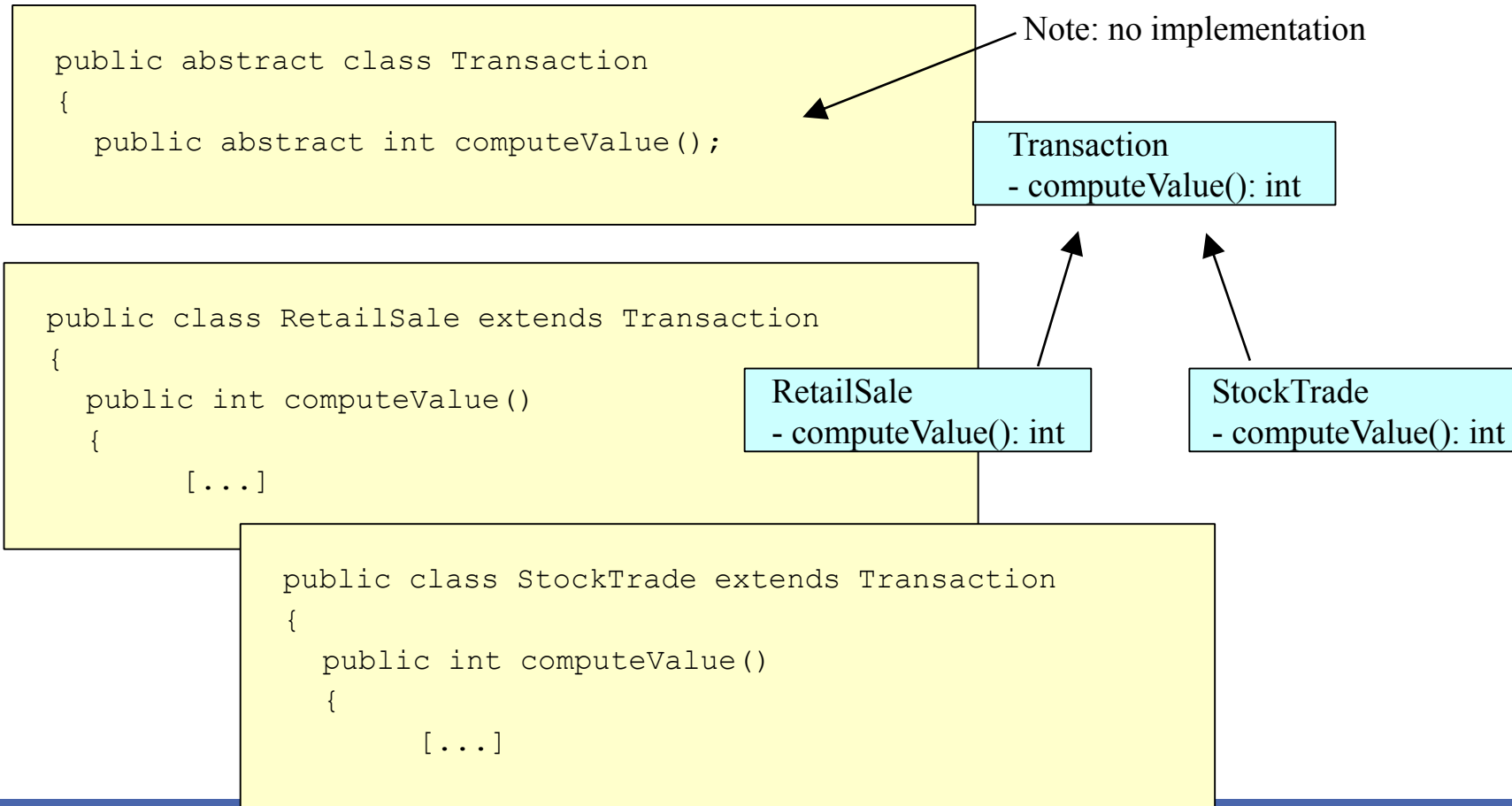
Abstract Method Example

- In the following example, a Transaction's value can be computed, but there is no meaningful implementation that can be defined within the Transaction class.
- How a transaction is computed is dependent on the transaction's type
- Note: This is polymorphism.



Defining Abstract Methods

- Inheritance is declared using the "extends" keyword
 - If inheritance is not defined, the class extends a class called Object



What is an Interface?

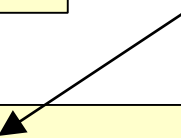
- A named collection of method definitions without implementations
- An interface is similar to an abstract class with the following exceptions:
 - All methods defined in an interface are abstract. Interfaces can contain no implementation
 - Interfaces cannot contain instance variables. However, they can contain public static final variables (ie. constant class variables)
- Interfaces are declared using the "interface" keyword
 - If an interface is public, it must be contained in a file which has the same name.
- Interfaces are more abstract than abstract classes
- Interfaces are implemented by classes using the "implements" keyword.

Declaring an Interface

In Steerable.java:

```
public interface Steerable
{
    public void turnLeft(int degrees);
    public void turnRight(int degrees);
}
```

When a class "implements" an interface, the compiler ensures that it provides an implementation for all methods defined within the interface.



In Car.java:

```
public class Car extends Vehicle implements Steerable
{
    public int turnLeft(int degrees)
    {
        [...]
    }

    public int turnRight(int degrees)
    {
        [...]
    }
}
```

Implementing Interfaces

- A Class can only inherit from one superclass. However, a class may implement several Interfaces
 - The interfaces that a class implements are separated by commas
- Any class which implements an interface must provide an implementation for all methods defined within the interface.
 - NOTE: if an abstract class implements an interface, it NEED NOT implement all methods defined in the interface. HOWEVER, each concrete subclass MUST implement the methods defined in the interface.
- Interfaces can inherit method signatures from other interfaces.

Declaring an Interface

In Car.java:

```
public class Car extends Vehicle implements Steerable, Driveable
{
    public int turnLeft(int degrees)
    {
        [...]
    }

    public int turnRight(int degrees)
    {
        [...]
    }

    // implement methods defined within the Driveable interface
```


Inheriting Interfaces

- If a superclass implements an interface, its subclasses also implement the interface

```
public abstract class Vehicle implements Steerable
{
    private String make;
    [...]
```

```
public class Car extends Vehicle
{
    private int trunkCapacity;
    [...]
```

```
public class Truck extends Vehicle
{
    private int bedCapacity;
    [...]
```

Vehicle
- make: String
- model: String
- tireCount: int

Car
- trunkCapacity: int

Truck
- bedCapacity: int

Multiple Inheritance?

- Some people (and textbooks) have said that allowing classes to implement multiple interfaces is the same thing as multiple inheritance
- This is NOT true. When you implement an interface:
 - The implementing class does not inherit instance variables
 - The implementing class does not inherit methods (none are defined)
 - The Implementing class does not inherit associations
- Implementation of interfaces is not inheritance. An interface defines a list of methods which must be implemented.

Interfaces as Types

- When a class is defined, the compiler views the class as a new type.
- The same thing is true of interfaces. The compiler regards an interface as a type.
 - It can be used to declare variables or method parameters

```
int i;  
Car myFleet[];  
Steerable anotherFleet[];  
  
[...]  
  
myFleet[i].start();  
  
anotherFleet[i].turnLeft(100);  
anotherFleet[i+1].turnRight(45);
```

Abstract Classes Versus Interfaces

- When should one use an Abstract class instead of an interface?
 - If the subclass-superclass relationship is genuinely an "is a" relationship.
 - If the abstract class can provide an implementation at the appropriate level of abstraction
- When should one use an interface in place of an Abstract Class?
 - When the methods defined represent a small portion of a class
 - When the subclass needs to inherit from another class
 - When you cannot reasonably implement any of the methods

Exercise: Interface

Code the following 4 interface modules

Code the driver program to populate an array that is created from the Nose class

- Call each of the interfaces to populate the array
- Use a for loop to display the values of each element

Interface Modules

1

```
Nose.java Picasso.java Clowns.java Acts.java Of76.java
1 interface Nose {
2     public int iMethod();
3 }
```

3

```
Nose.java Picasso.java Clowns.java Acts.java Of76.java
1 class Clowns extends Picasso {
2
3 }
```

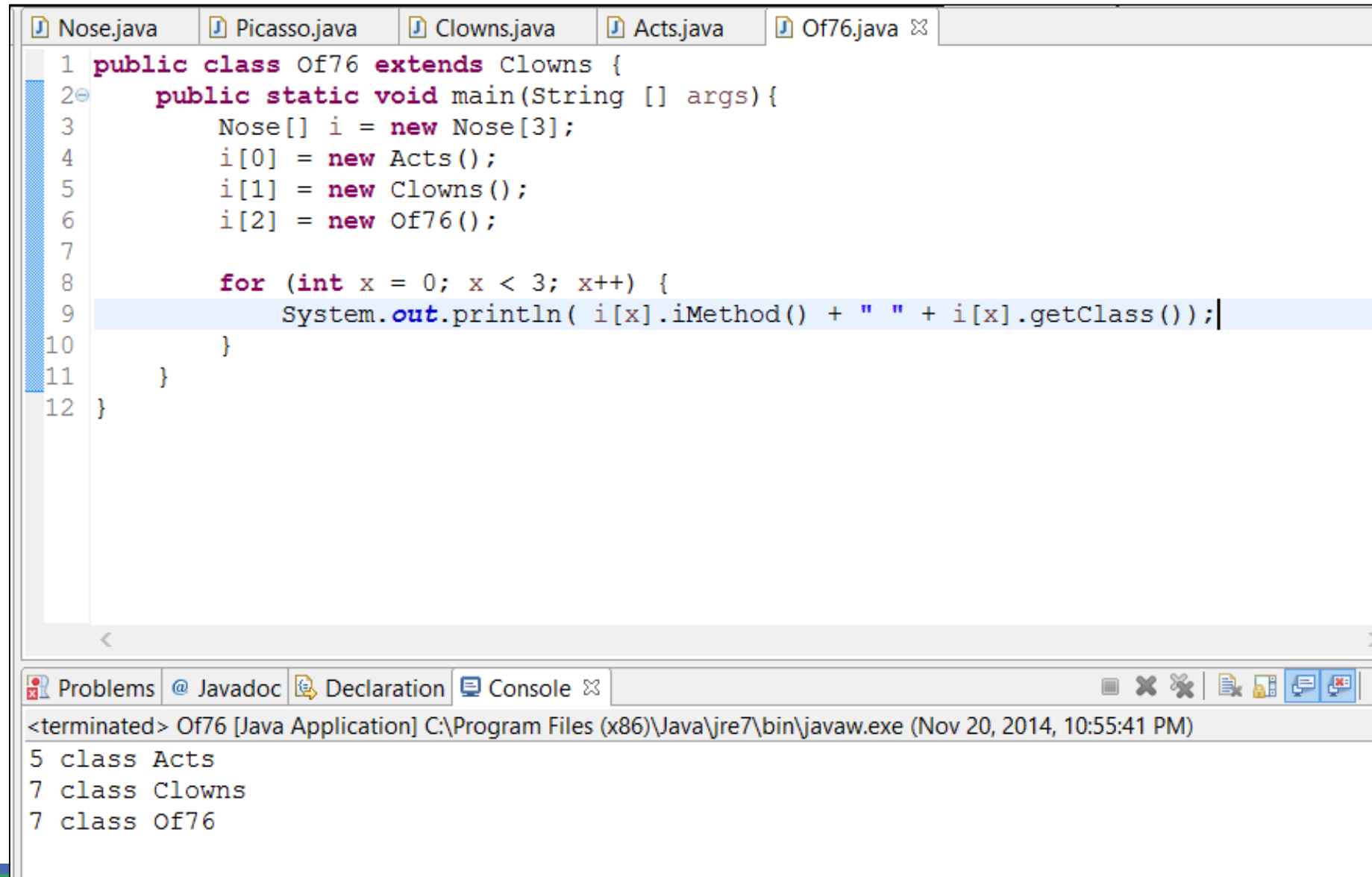
2

```
Nose.java Picasso.java Clowns.java Acts.java Of76.java
1 abstract class Picasso implements Nose {
2     public int iMethod() {
3         return 7;
4     }
5 }
```

4

```
Nose.java Picasso.java Clowns.java Acts.java Of76.java
1 class Acts extends Picasso {
2     public int iMethod () {
3         return 5;
4     }
5 }
```

Driver Module



```
1 public class Of76 extends Clowns {
2     public static void main(String [] args){
3         Nose[] i = new Nose[3];
4         i[0] = new Acts();
5         i[1] = new Clowns();
6         i[2] = new Of76();
7
8         for (int x = 0; x < 3; x++) {
9             System.out.println( i[x].iMethod() + " " + i[x].getClass());
10        }
11    }
12 }
```

<terminated> Of76 [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Nov 20, 2014, 10:55:41 PM)

```
5 class Acts
7 class Clowns
7 class Of76
```

Exercise 1: Online Store Front – page 131

To get our heads around the interface concept, let's complete the following:

Create a Storefront application that uses packages, access controls, interfaces and encapsulation – this store manages the items in an online storefront to handle two main tasks

- Calculate the sales price of each item depending on how much is presently in stock
- Sort items according to sale price

Create two (2) classes, Storefront.java and Item.java in a new package called com.appinc.ecommerce

Exercise 1: continue

1. create a new package
 - Follow the instructions on page 131 – 133
2. Create the class Item.java on page 134 under the new package
3. Create the class Storefront on page 136 under the new package
4. Create the GiftShop application (has the main() method) on page 138 under the new package to run the storefront
5. Run GiftShop.java as a Java Application

Output:

```
Item ID: D01
Name: T SHIRT
Retail Price: $16.99
Price: $11.89
Quantity: 90
```

```
Item ID: C02
Name: LG MUG
Retail Price: $12.99
Price: $9.09
Quantity: 82
```

```
Item ID: C01
Name: MUG
Retail Price: $9.99
Price: $6.99
Quantity: 150
```

```
Item ID: C03
Name: MOUSEPAD
Retail Price: $10.49
Price: $5.25
Quantity: 800
```

Item.java

Line 1: establish the class is part of the package `com.appinc.ecommerce`

Line 3: implement the `Comparable` interface to make it easy to sort the class objects. It has one method `compareTo(Object)` to return an integer – it compares two objects of a class

Lines 10-23: `Item()` constructor takes four `String` objects as arguments and uses them to set up the `ID`, `name`, `retail price` and `quantity` instance variables.

Lines 16-21: the value of the `price` instance variable is set depending on how much of that item is in stock

Line 22: rounds off price

Lines 25-32: Simple accessor method

StoreFront.java

Line 6: Each product is an item object (item.java) and stored in LinkedList instance variable (more on link lists later) called catalog

Line 8-13: creates a new object using the addItem() method

Lines 15-17: getItem() is called catalog.get(int) with an index as an argument returning the object stored at the location in the linked list

Lines 19-21: getItem() and getSize() are the interfaces to the information stored in a private catalog variable.

Lines 23-25: sort() use the implementation of the Comparable interface coded in the item.class – the class method Collections.sort() sorts the linked list

Independent Exercises – Page 140

A. In the package `com.appinc.ecommerce`, create a modified version of the StoreFront project including the Java files `Item.java` and `GiftShop.java` that have **an added `noDiscount` variable** for each item. When `noDiscount` is true, sell the item at the retail price.

All files should be named as follows:

`Storedfront2.java`, `item2.java`, `GiftShop2.java`

B. Create a `ZipCode` class that uses access control to ensure that its `zipCode` instance variable always has 5 digits.

This module should be named `Zipcode.java`

Solutions can be found in the appendix for day 6