Java Programming, Comprehensive
Lecture 4

Bineet Sharma

Agenda

- Object Oriented Programming
  - Inheritance
  - Interface
  - Polymorphism
- Nested classes
- Enumeration

OOP Motivation: Code Reuse

- Java allows multiple ways to reuse, time tested codes, through:
  - **Composition**: Embed one class into another – defines a '**has-a**' relationships. Car has four wheels. Product class has a String:

```
import java.text.NumberFormat;

public class Product
{
    // the instance variables
    private String code;
    private String description;
    private double price;

    // the constructor
    public Product(){
        code = "";
        description = "";
        price = 0;
    }
…
}
```

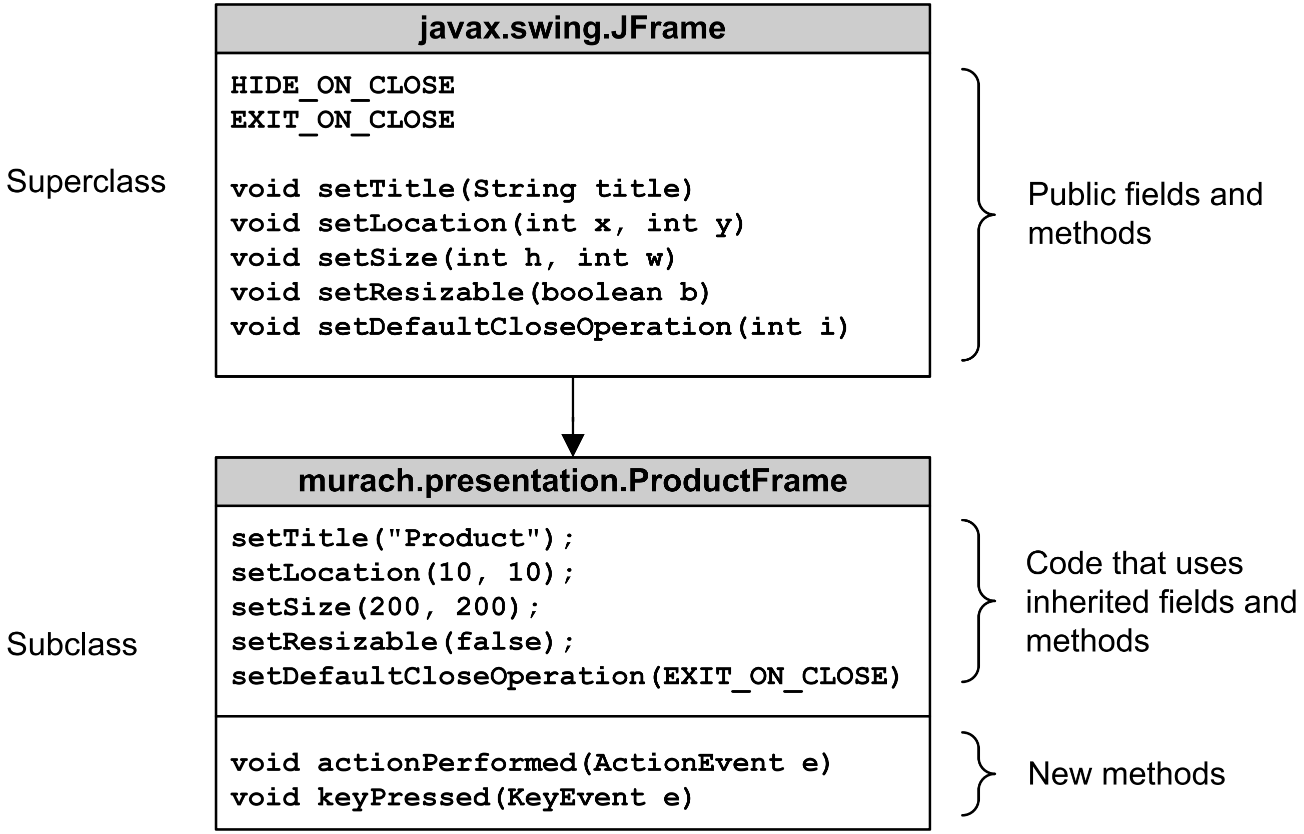
OOP Motivation: Code Reuse

- Java allows multiple ways to reuse, time tested codes, through:
  - **Inheritance**: Copy (inherit) ***all*** of the properties of one class into another class – defines a '**is-a**' relationships. Honda is a Car. ProductFrame is a JFrame:

```
import javax.swing.JFrame;

public class ProductFrame extends JFrame
{
    // the instance variables
    …
    // the constructor
    public ProductFrame()
    {
        //other initialization
        code = "";
        description = "";
        price = 0;
    }
…
}
```

OOP Motivation: Inheritance

- How inheritance works



Superclass

javax.swing.JFrame

```
HIDE_ON_CLOSE
EXIT_ON_CLOSE

void setTitle(String title)
void setLocation(int x, int y)
void setSize(int h, int w)
void setResizable(boolean b)
void setDefaultCloseOperation(int i)
```

Public fields and methods

Subclass

murach.presentation.ProductFrame

```
setTitle("Product");
setLocation(10, 10);
setSize(200, 200);
setResizable(false);
setDefaultCloseOperation(EXIT_ON_CLOSE)
```

Code that uses inherited fields and methods

```
void actionPerformed(ActionEvent e)
void keyPressed(KeyEvent e)
```

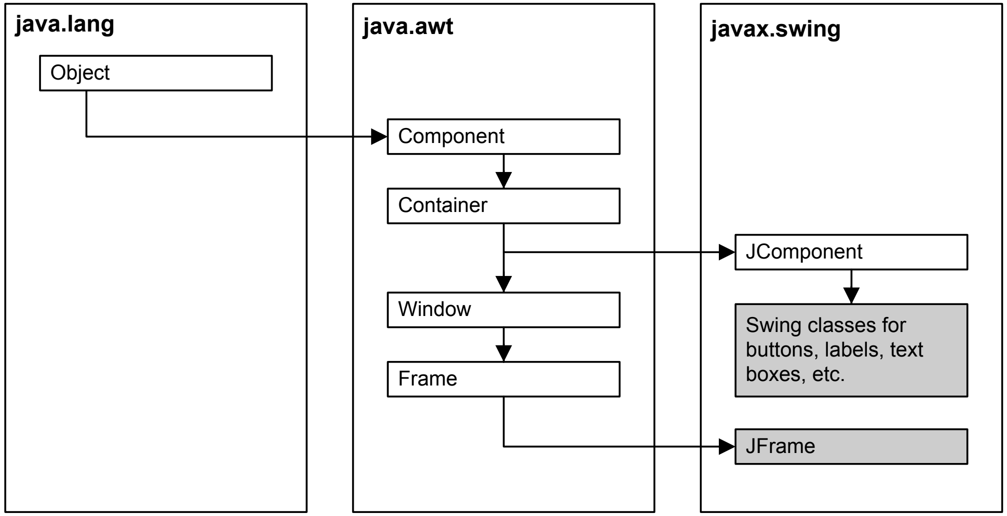
New methods

OOP Motivation: Inheritance

- Inheritance lets you create a new class based on an existing class
- The new class inherits the states (fields) and behaviors (methods) from old class
- The new class is called the *derived* class, *child* class or *subclass*
- The old class is called *base* class, *parent* class, or *superclass*
- A subclass extends superclass by adding new fields, constructors, and methods
- A subclass can *override* the superclass methods as well

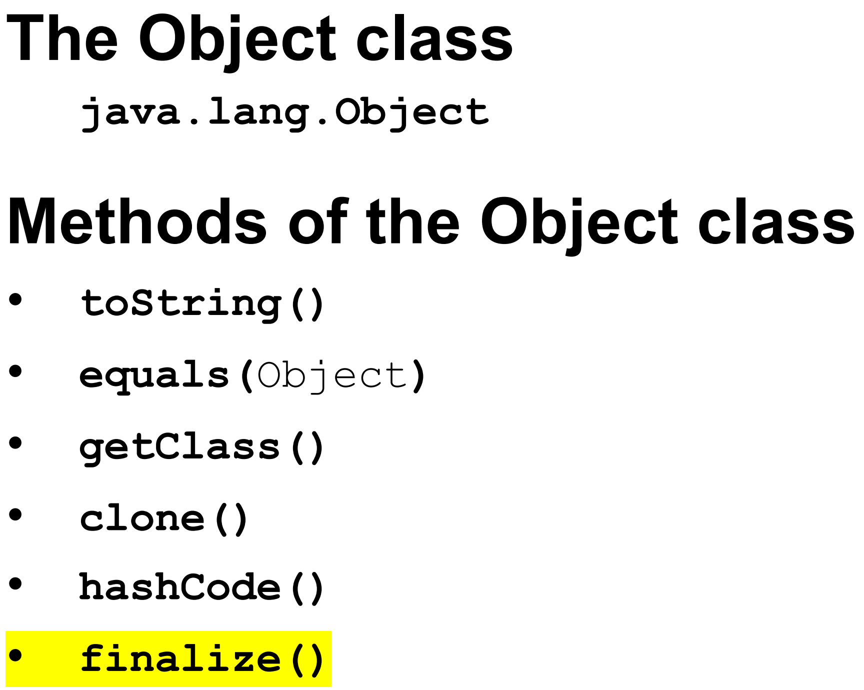
OOP Motivation: Inheritance

- The inheritance hierarchy for Swing forms and controls



- *Object* is mother of all classes (java.lang package). All Java classes are implicitly or explicitly derived from Object class
- *Swing* classes inherit Component and Container classes of java.awt package

OOP Motivation: Inheritance



The Object class

```
java.lang.Object
```

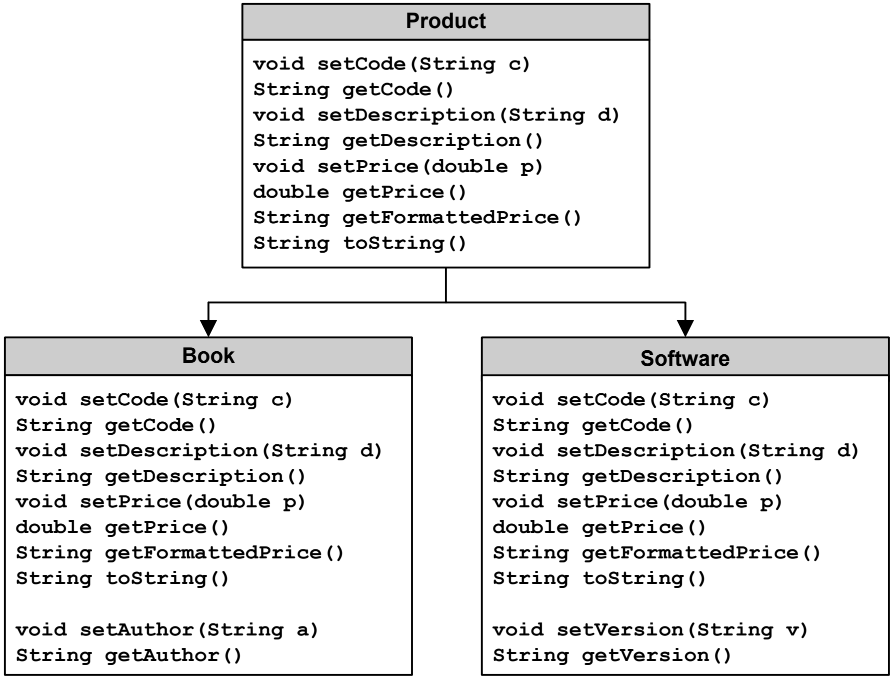
Methods of the Object class

- `toString()`
- `equals(Object)`
- `getClass()`
- `clone()`
- `hashCode()`
- `finalize()`

- *Object* is mother of all classes (java.lang package)
  - hashCode() identifies the location of object in memory
  - finalize() is called before the GC reclaims the memory

OOP Motivation: Inheritance

- Main classes for a Product application



Product

```
void setCode(String c)
String getCode()
void setDescription(String d)
String getDescription()
void setPrice(double p)
double getPrice()
String getFormattedPrice()
String toString()
```

Book

```
void setCode(String c)
String getCode()
void setDescription(String d)
String getDescription()
void setPrice(double p)
double getPrice()
String getFormattedPrice()
String toString()

void setAuthor(String a)
String getAuthor()
```

Software

```
void setCode(String c)
String getCode()
void setDescription(String d)
String getDescription()
void setPrice(double p)
double getPrice()
String getFormattedPrice()
String toString()

void setVersion(String v)
String getVersion()
```

- Book has only setAuthor and getAuthor as new methods
- Software has only setVersion and getVerson as new methods

OOP Motivation: Inheritance

- Use inheritance to create generic superclasses
  - Superclasses implement common elements
  - Allows you to lump commonality in higher level
    - Product, Shape, Bike, Animal, Person, Company, Equation, SateTax
  - You can also inherit from Java API classes.
    - PigLatinDate could inherit from Date (java.util package) class
  - Subclass **is a** superclass and more, hence, you can use subclass to represent superclass
    - Book object can be used whenever a Product object is called for
    - This is classic polymorphism (taking many shapes)

OOP Motivation: Inheritance

- Creating a superclass
  - Superclass is created as any other class
  - Provide all known common functionality in superclass
    - Use access modifiers to limit accessibility to state and behaviors
      - `private`
      - `public`
      - `protected`
      - *no keyword coded*
    - All states should be *private* – with public getters and setters as needed
    - Selectively provide other *public* methods as needed
    - Can provide *protected* members if want direct access by subclass
    - Provide **default constructors** – constructor without parameter
    - Override toString()

OOP Motivation: Inheritance

- The code for the Product superclass

```
import java.text.NumberFormat;

public class Product
{
    private String code;
    private String description;
    private double price;
    protected static int count = 0;   // a protected
                                      // static variable

    public Product()
    {
        count++;
        code = "";
        description = "";
        price = 0;
    }
```

OOP Motivation: Inheritance

- The code for the Product superclass (cont.)

```
public class Product
{
   ...
   //more constructors
   public Product(String c, String d, double p){
        super();
        count++;
        code = c;
        description = d;
        price = p;
    }

    //specialized methods
    public String getFormattedPrice()
    {
        NumberFormat currency =
                NumberFormat.getCurrencyInstance();
        return currency.format(price);
    }
}
```

OOP Motivation: Inheritance

- The code for the Product superclass (cont.)

```
        // get and set accessors for the code, description,
        // and price instance variables
        @Override  // annotation – directive to compiler
        public String toString()
        {
            return "Code:        " + code + "\n" +
                   "Description: " + description + "\n" +
                   "Price:       " +
                   this.getFormattedPrice() + "\n";
        }

        // create public access for the count variable
        public static int getCount()
        {
            return count;
        }
    }
```

OOP Motivation: Inheritance

- The syntax for creating subclasses

To declare a subclass

```
public class SubclassName extends SuperClassName{}
```

To call a superclass constructor

```
super(argumentList)
```

To call a superclass method

```
super.methodName(argumentList)
```

OOP Motivation: Inheritance

- The code for a Book class (Subclassed from Product)

```
public class Book extends Product {
    private String author;

    public Book(){
        super();

    // call constructor of Product (super) class as
    // first statement
    // by default, default constructor of super
    // class is called so super() is optional here

        author = "";
        //count++;  //don’t update here, super does
Product superclass
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public String getAuthor(){
        return author;
    }
```

OOP Motivation: Inheritance

- The code for a Book class (cont.)

```
   // override the toString method
    @Override
    public String toString()
    {
        // call method of Product superclass
        return super.toString() +
            "Author:      " + author + "\n";
    }
}
```

OOP Motivation: Inheritance

- Three versions of the toString method

The toString method in the Product superclass

```
public String toString()
{
    return "Code:        " + code + "\n" +
           "Description: " + description + "\n" +
           "Price:       " + this.getFormattedPrice() + "\n";
}
```

The toString method in the Book class

```
public String toString()
{
    return super.toString() +
        "Author:      " + author + "\n";
}
```

The toString method in the Software class

```
public String toString()
{
    return super.toString() +
        "Version:     " + version + "\n";
}
```

OOP Motivation: Inheritance

- Using overridden methods? Polymorphism (dynamic binding) in action:

```
Book b = new Book();
b.setCode("java");
b.setDescription("Murach's Beginning Java");
b.setPrice(49.50); b.setAuthor("Steelman");

Software s = new Software();
s.setCode("txtp");
s.setDescription("TextPad");
s.setPrice(27.00); s.setVersion("4.7.3");

Product p;    //p is not yet an object, just a reference
p = b;        //implicit casting.  Subclass can be assigned
              //to superclass, now p is pointing to object
System.out.println(p.toString());  // calls toString from
                                   // the Book class
p = s;
System.out.println(p.toString());  // calls toString from
                                   // the Software class
```

OOP Motivation: Inheritance

- There is a ‘Class’ object associated for every objects
  - It holds information about the object
    - Called RTTI: RunTime Type Information
  - Use it to find more information regarding an object at runtime.

The Class class

`java.lang.Class`

Common method

- `getName()`

OOP Motivation: Inheritance

Code that displays an object’s type

```
Product p = new Book();   // create a Book object and
                          // assign it to a Product
                          // variable
Class c = p.getClass();   // get the Class object for
                           //the product

System.out.println("Class name: " + c.getName());
                          // print the object type
```

The console

```
Class name: Book
```

OOP Motivation: Inheritance

Code that tests an object's type

```
Product p = new Book();  // create a Book object
if (p.getClass().getName().equals("Book"))
    System.out.println("This is a Book object");
```

The console

```
This is a Book object
```

An easier way to test an object's type

```
Product p = new Book();  // create a Book object
if (p instanceof Book)
    System.out.println("This is a Book object");
```

The console

```
This is a Book object
```

OOP Motivation: Inheritance

- Casting Objects:
  - Java can implicitly cast a subclass to a superclass. So, you can use subclass whenever a superclass is called for
    - For example a Software object can be specified whenever a Product is expected (upcasting is implicit)
  - You need to explicitly cast a superclass object when a reference to one of its subclasses is required
    - You must explicitly cast a Product object to Software if a Software object is expected. If it is not a valid inheritance, you get a *ClassCastException (downcasting needs to be explicit)*
  - The subclass methods will not be available to a downcasted superclass object
    - For example, you can't call *setVersion* if you store a Software object on a Product reference.

OOP Motivation: Inheritance

- Casting objects (cont.)

```
Book b = new Book();

b.setCode("java");
b.setDescription("Murach's Beginning Java");
b.setPrice(49.50);
b.setAuthor("Steelman");

//change the assignment to Base class
Product p = new Software();  //upcasting is implicit

p.setCode("txtp");
p.setDescription("TextPad");
p.setPrice(27.00); //can I do this? p.setVersion("4.7.3");

Software s;
s = (Software) p;      //downcasting needs to be explicit

s.setVersion("4.7.3"); //is it error?compile/run time?
System.out.println(s.toString()); // calls Software version
```

OOP Motivation: Inheritance

- Casting objects (cont.). Can we do this?

```
Book b = new Book();

b.setCode("java");
b.setDescription("Murach's Beginning Java");
b.setPrice(49.50);
b.setAuthor("Steelman");

//create the object using base class
Product p = new Product();

p.setCode("txtp");
p.setDescription("TextPad");
p.setPrice(27.00);

Software s;
s = (Software) p;        //downcasting needs to be explicit

s.setVersion("4.7.3"); //is it error?compile/run time?
System.out.println(s.toString());
```

OOP Motivation: Inheritance

- Casting examples that use the Product and Book classes

```
Book b = new Book();
b.setCode("java");
b.setDescription("Murach's Beginning Java");
b.setAuthor("Andrea Steelman");
b.setPrice(49.50);

Product p = b;              // cast Book object to a
                            // Product object
p.setDescription("Test");  // OK - method in Product class
//p.setAuthor("Test");      // not OK - method not in
                            // Product class

b = (Book) p;               // cast the Product object back
                            // to a Book object
b.setAuthor("Test");        // OK - method in Book class

Product p2 = new Product();
Book b2 = (Book) p2;      // will throw ClassCastException
                          // because p2 is a Product object
// not a Book object.  It is like child knows about parent,
//but parent don't know how many children they have
```

OOP Motivation: Inheritance

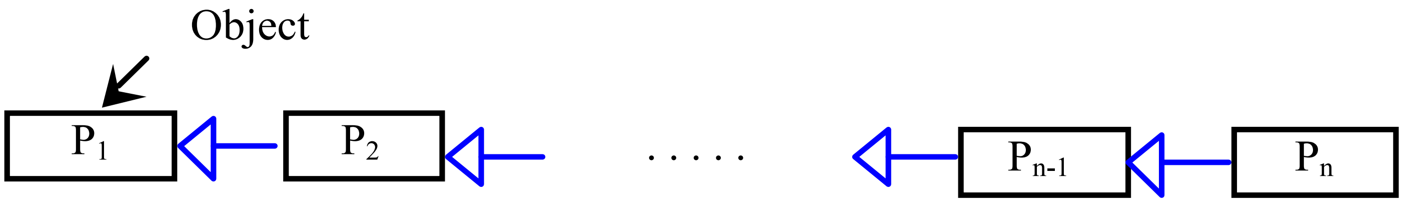
- Why is casting necessary?
- Let see with some examples:
  - Suppose you want to do this:
    - Object o = new Software();
    - Software s = o; //a compiler error would occur
  - You may ask, why so? How come I can do this though?
    - Software s = new Software ();
    - Object o = s;
  - This is simply because a Software object is always an instance of Object, but an Object is not necessarily an instance of Student.
    - You may see that, but, compiler is not smart enough to see it
    - Software s = (Software) o; // that is why we need explicit casting

OOP Motivation: Inheritance

- Why is casting necessary? (cont.)
- Other way to look at it is:
  - All apples are fruits but not all fruits are apples.
- Consider that Fruit is a super class and Apple and Orange classes are derived from it.
  - An apple is a fruit, so you can always safely assign an instance of Apple to a variable (reference) of Fruit.
  - However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of Fruit to a variable of Apple.
    - Fruit f = new Apple(); //this is fine
    - Apple a = (Apple) new Fruit(); //this needs casting (downcasting)

OOP Motivation: Inheritance

- How dynamic binding works?
  - Suppose an object o is an instance of class P_n which is derived from P_{n-1} which is derived from P_{n-2} and so on lastly from P_1. Here P_1 is Object class
  - If o invokes a method p, the JVM, during runtime, searches the implementation for the method p in P_n, P_{n-1}, P_{n-2}, …, P_2, & P_1 in that order
  - It stops stops once found and that implementation is invoked



Since o is an instance of P_n, o is also an instance of P_{n-1}, P_{n-2}, . . . , P_2, & P_1

OOP Motivation: Inheritance

- Providing compare functionality to object:
  - Use *equals* method of Object class to test if both references point to the same memory location
  - Override *equals* to test if both references have same data
  - This is how the equals method of Object class works:

Both variables refer to the same object

```
Product product1 = new Product();
Product product2 = product1;
if (product1.equals(product2))     // expression returns
                                   // true
```

Both variables refer to different objects that store the same data

```
Product product1 = new Product();
Product product2 = new Product();
if (product1.equals(product2))     // expression returns
                                   // false
```

OOP Motivation: Inheritance

- Instead, if you want content of two objects to be compared, instead of references
  - Then, override the equals method of the Object class

```
@Override
public boolean equals(Object object)
{   //make sure it is compatible
    if (object instanceof Product){
        Product product2 = (Product) object;
        if (
            code.equals(product2.getCode()) &&
            description.equals(
                product2.getDescription()) &&
            price == product2.getPrice()
        )
            return true;
    }
    return false;
}
```

OOP Motivation: Inheritance

- Override the equals method of the Object class(cont.)

The equals method of the LineItem class.

LineItem has a Product class and a quantity as instance variables

```
@Override
public boolean equals(Object object){
    if (object instanceof LineItem){
        LineItem li = (LineItem) object;
        if (
            product.equals(li.getProduct()) &&
            quantity == li.getQuantity()
        )
            return true;
    }
    return false;
}
```

OOP Motivation: Inheritance

- **Concrete class:** A class which has all methods fully defined
- **Abstract class**: A class which is not yet fully defined. It still has some methods which are empty and subclass MUST define those methods
- Abstract class can't be instantiated to create an object. But, it can be used in inheritance chain
- Motivation of Abstract class:
  - Functionality of the class is not yet clear
  - Dictates certain order of functionality to derived classes

OOP Motivation: Inheritance

- Abstract class contains:
  - Already defined methods, fields, constants like regular class
  - Additionally, contains abstract methods which are undefined
- Abstract methods <u>can't have private access</u>
- Subclass must define all abstract methods to be concrete
- Any class with an abstract method is also abstract.
  - Meaning, you don't need to write 'abstract' in the class definition if you already have a method which is abstract
- In the mean time, you don't need an abstract method for a class to be abstract,
  - just write *abstract* in class definition
- A class with all defined methods can also be abstract

OOP Motivation: Inheritance

- An abstract Product class

```
public abstract class Product
{
    private String code;
    private String description;
    private double price;

    // regular constructors and methods for instance
    // variables

    @Override
    public String toString()
    {
        return "Code:        " + code + "\n" +
               "Description: " + description + "\n" +
               "Price:       " + this.getFormattedPrice()
                               + "\n";
    }

    // an abstract method
    abstract String getDisplayText();
}
```

OOP Motivation: Inheritance

- A class that inherits the abstract Product class

```
public class Book extends Product
{
    private String author;

    // regular constructor and methods for the Book class

    // implement the abstract method
    @Override
    public String getDisplayText()
    {
        return super.toString() +
            "Author:      " + author + "\n";
    }
}
```

OOP Motivation: Inheritance

- **Final keyword:** Final keyword is used for
  - class, methods, and to create constants
- Use *final* keyword to prevent
  - a class from inheriting, that makes all methods automatically final, ex. String class
  - To prevent only individual methods from overriding, use *final* in method definition
  - To prevent in-advertent assignment to a parameter, you can use *final* for parameters as well
- Use *final* for performance boost as well,
  - however minor, as compiler don't have to worry about inheritance and polymorphism (stops virtualization of methods)

OOP Motivation: Inheritance

Example 1: A final class

```
public final class Book extends Product
{
    // all methods in the class are automatically final
}
```

Example 2: A final method

```
public final String getVersion()
{
    return version;
}
```

Example 3: A final parameter

```
public void setVersion(final String version)
{
    // version = "new value"; // not allowed
    this.version = version;
}
```

OOP Motivation: Inheritance

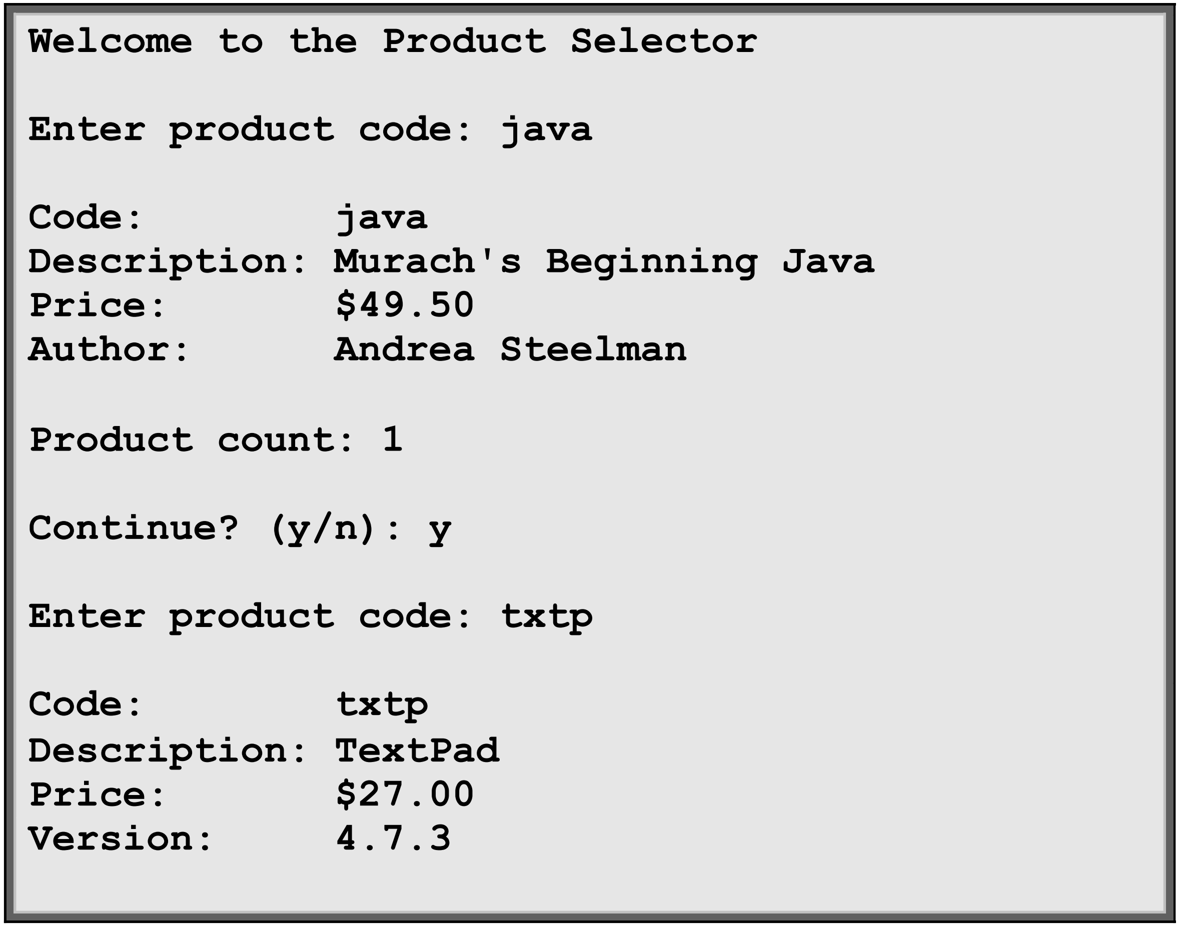
Putting it all Together

Code Walk Through

ProductApp

OOP Motivation: Inheritance

- Code walk through of the ProductApp application



```
Welcome to the Product Selector

Enter product code: java

Code:        java
Description: Murach's Beginning Java
Price:       $49.50
Author:      Andrea Steelman

Product count: 1

Continue? (y/n): y

Enter product code: txtp

Code:        txtp
Description: TextPad
Price:       $27.00
Version:     4.7.3
```

OOP Motivation: Interface

- Interface is other way to define and dictate the properties of objects
- *Interface* defines a set of public methods
- Interface provides only signatures of public method (no code) and it also can define public constants
- All class must define those public methods who want to implement this interface
- Interface is really a contract for a class to follow
- Example: A Printable interface that defines a print method

```
public interface Printable{
    public abstract void print();
}
```

OOP Motivation: Interface

- A Product class that implements the Printable interface

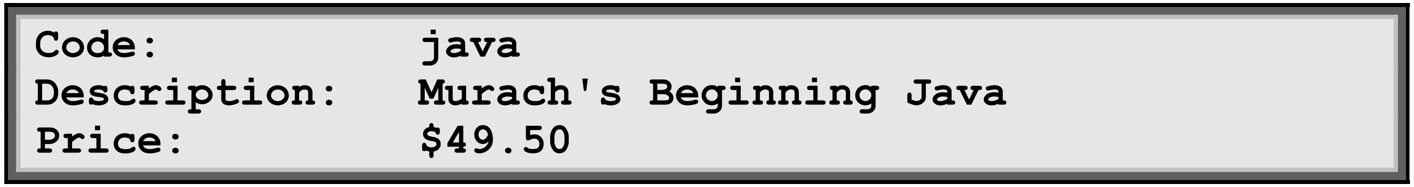
```
import java.text.NumberFormat;

public class Product implements Printable{
    private String code;
    private String description;
    private double price;

    public Product(
    String code, String description, double price){
        this.code = code;
        this.description = description;
        this.price = price;
    }
    // get and set methods for the fields
    // implement the Printable interface
    public void print() {
        System.out.println("Code:          " + code);
        System.out.println(
            "Description:   " + description);
        System.out.println("Price:         " +
            this.getFormattedPrice());
    }
}
```

OOP Motivation: Interface

- Code that uses the print method of the Product class

```
Printable product = new Product(
        "java", "Murach's Beginning Java", 49.50);
product.print();
```

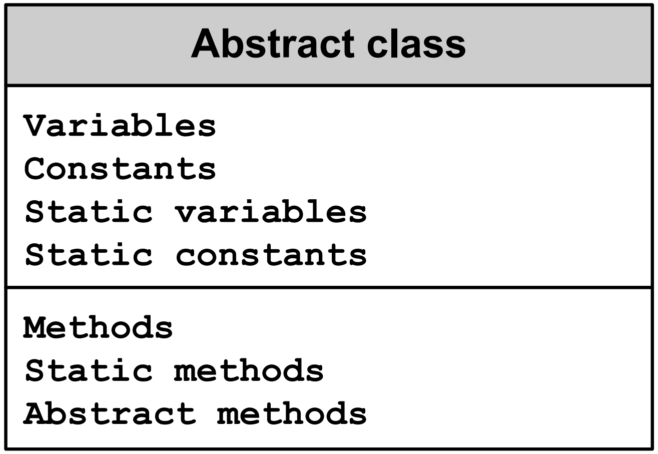
- The output



```
Code:           java
Description:    Murach's Beginning Java
Price:          $49.50
```

OOP Motivation: Interface

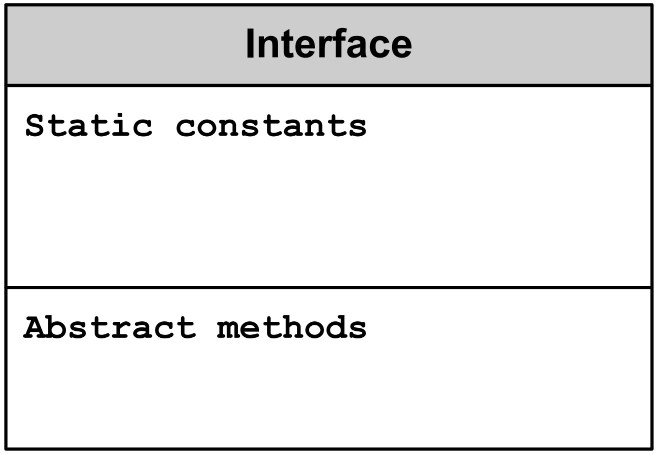
- Abstract class and Interface are close cousins. And, the uses could get confusing
- JDK Pre 1.8



Abstract class

```
Variables
Constants
Static variables
Static constants
```

```
Methods
Static methods
Abstract methods
```



Interface

```
Static constants
```

```
Abstract methods
```

OOP Motivation: Interface

A Printable interface

```
public interface Printable
{
    public abstract void print();
}
```

A Printable abstract class

```
public abstract class Printable
{
    public abstract void print();
}
```

OOP Motivation: Interface

Advantages of an abstract class

- An abstract class can use instance variables and constants as well as static variables and constants. Interfaces can only use static constants.
- An abstract class can define regular methods that contain code as well as abstract methods that don’t contain code. An interface can only define abstract methods.
- An abstract class can define static methods. An interface can’t. (pre JDK 1.8)

Advantages of an interface

- A class can only directly inherit one other class, but it can directly implement multiple interfaces.
- Any object created from a class that implements an interface can be used wherever the interface is accepted.

OOP Motivation: Interface

- Java API defines many interfaces that can be implemented
  - For example: Cloneable, Comparable, EventListener etc.

Interface (java.long)	Methods
`Cloneable`	None *(tagging interface)*
`Comparable`	`int compareTo(Object o)`

Interface (java.awt.event)	Methods
`EventListener`	None

`WindowListener`	`void windowActivated(WindowEvent e)`
	`void windowClosed(WindowEvent e)`
	`void windowClosing(WindowEvent e)`
	`void windowDeactivated(WindowEvent e)`
	`void windowDeiconified(WindowEvent e)`
	`void windowIconified(WindowEvent e)`
	`void windowOpened(WindowEvent e)`

`ActionListener`	`void actionPerformed(ActionEvent e)`

OOP Motivation: Interface

- How to work with interface

The syntax for declaring an interface

```
public interface InterfaceName
{
    type CONSTANT_NAME = value;                 // field
    returnType methodName([parameterList]);     // method
}
```

An interface that defines one method

```
public interface Printable
{
    void print();
}
```

An interface that defines three methods

```
public interface ProductWriter
{
    boolean addProduct(Product p);
    boolean updateProduct(Product p);
    boolean deleteProduct(Product p);
}
```

OOP Motivation: Interface

- How to work with interface

An interface that defines constants

```
public interface DepartmentConstants
{
    int ADMIN = 1;
    int EDITORIAL = 2;
    int MARKETING = 3;
}
```

A tagging interface with no members

```
public interface Cloneable
{
}
```

The syntax for implementing an interface

```
public class ClassName implements Interface1[,
    Interface2]...{}
```

- All methods are automatically public and abstract
- All fields are automatically public, static and final

OOP Motivation: Interface

- An Employee class that implements two interfaces

```
import java.text.NumberFormat;

public class Employee implements Printable,
                                 DepartmentConstants
{
    private int department;
    private String firstName;
    private String lastName;
    private double salary;

    public Employee(int department, String lastName,
                    String firstName, double salary)
    {
        this.department = department;
        this.lastName = lastName;
        this.firstName = firstName;
        this.salary = salary;
    }
```

OOP Motivation: Interface

- An Employee class that implements two interfaces (cont.)

```
public void print()
    {
        NumberFormat currency =
            NumberFormat.getCurrencyInstance();
        System.out.println(
            "Name:\t" + firstName + " " + lastName);
        System.out.println(
            "Salary:\t" + currency.format(salary));

        String dept = "";
        if (department == ADMIN)
            dept = "Administration";
        else if (department == EDITORIAL)
            dept = "Editorial";
        else if (department == MARKETING)
            dept = "Marketing";

        System.out.println("Dept:\t" + dept);
    }
}
```

OOP Motivation: Interface

- Refer this Product superclass for next example

```
public class Product {
    private String code, description;
    private double price;
    protected static int count = 0;

    public Product(){
        count ++;
    }
    // more accessors and mutators code for vars

    @Override  // annotation - directive to compiler
    public String toString(){
        return "Code:        " + code + "\n" +
               "Description: " + description + "\n" +
               "Price:       " +
               this.getFormattedPrice() + "\n";
    }

    // create public access for the count variable
    public static int getCount() {
        return count;
    }
}
```

OOP Motivation: Interface

- The syntax for inheriting a class and implementing an interface

```
public class SubclassName extends SuperclassName
implements Interface1[, Interface2]...{}
```

- A Book class that inherits Product and implements Printable

```
public class Book extends Product implements Printable
{
    private String author;

    public Book(String code, String description,
                double price, String author)
    {
        super(code, description, price);
        this.author = author;
    }
```

OOP Motivation: Interface

- A Book class that inherits Product and implements Printable (cont.)

```
    public void setAuthor(String author){
        this.author = author;
    }

    public String getAuthor(){
        return author;
    }

    @Override
    public String toString() {
        return super.toString() + "Author:      "
                                + author + "\n";
    }

    // implement the Printable interface
    public void print() {
        System.out.println(toString());
    }
}
```

OOP Motivation: Interface

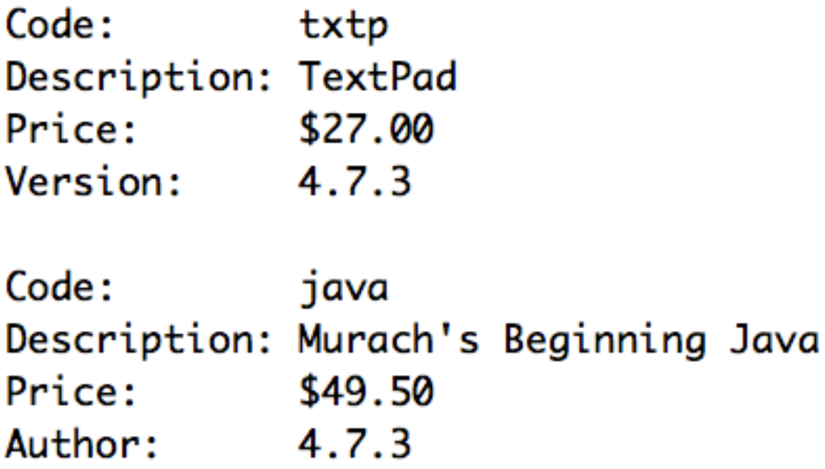
A method that accepts a Printable object

```
private static void printMultiple(Printable p, int count)
{ //Printable is interface
    for (int i = 0; i < count; i++)
        p.print();
}
```

Code that passes a Product object to the method

```
Book book = new Book(
    "java", "Murach's Beginning Java", 49.50, "4.7.3");
printMultiple(book, 2); //Book is a class
```

Resulting output



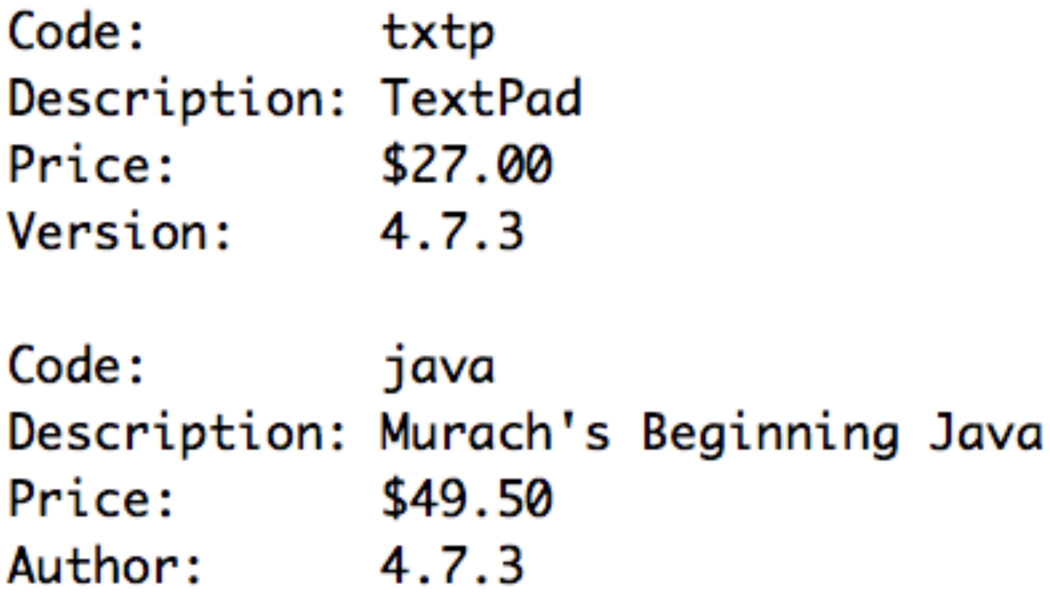
```
Code:        txtp
Description: TextPad
Price:       $27.00
Version:     4.7.3

Code:        java
Description: Murach's Beginning Java
Price:       $49.50
Author:      4.7.3
```

OOP Motivation: Interface

Code that passes a Printable object to the method

```
Printable printable = new Book(
    "java", "Murach's Beginning Java", 49.50, "4.7.3");
printMultiple(printable, 2); //printable is an interface
```

Resulting output



```
Code:        txtp
Description: TextPad
Price:       $27.00
Version:     4.7.3

Code:        java
Description: Murach's Beginning Java
Price:       $49.50
Author:      4.7.3
```

OOP Motivation: Interface

- The syntax for declaring an **interface inheritance**

```
public interface InterfaceName
    extends InterfaceName1[, InterfaceName2]...
{
    // the constants and methods of the interface
}
```

A ProductReader interface

```
public interface ProductReader
{
    Product getProduct(String code);
    String getProductsString();
}
```

A ProductWriter interface

```
public interface ProductWriter
{
    boolean addProduct(Product p);
    boolean updateProduct(Product p);
    boolean deleteProduct(Product p);
}
```

OOP Motivation: Interface

- A ProductConstants Interface

```
public interface ProductConstants
{
    int CODE_SIZE = 4;
    int DESCRIPTION_SIZE = 40;
}
```

A ProductDAO interface that inherits all three interfaces

```
public interface ProductDAO
    extends ProductReader, ProductWriter,
    ProductConstants
{
}
```

OOP Motivation: Interface

- A class that implements the ProductDAO interface

```
public class ProductDB implements ProductDAO {

    @Override
    public Product getProduct(String code) {
        throw new UnsupportedOperationException(
            "Not supported yet.");
    }

    @Override
    public String getProductsString() {
        throw new UnsupportedOperationException(
            "Not supported yet.");
    }

    @Override
    public boolean addProduct(Product p) {
        throw new UnsupportedOperationException(
            "Not supported yet.");
    }
```

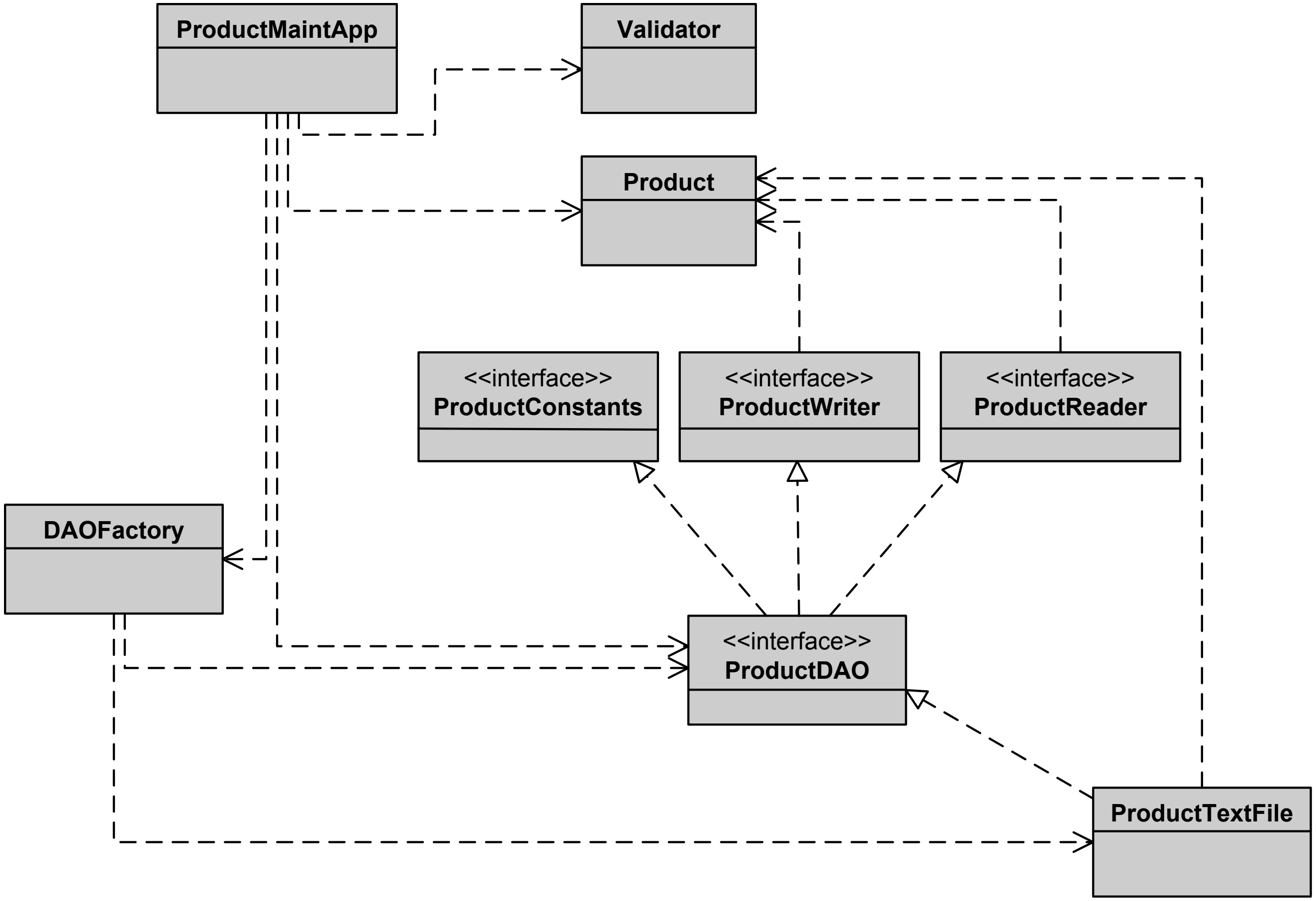
OOP Motivation: Interface

- A class that implements the ProductDAO interface (cont.)

```
    @Override
    public boolean updateProduct(Product p) {
        throw new UnsupportedOperationException(
            "Not supported yet.");
    }

    @Override
    public boolean deleteProduct(Product p) {
        throw new UnsupportedOperationException(
            "Not supported yet.");
    }
}
```

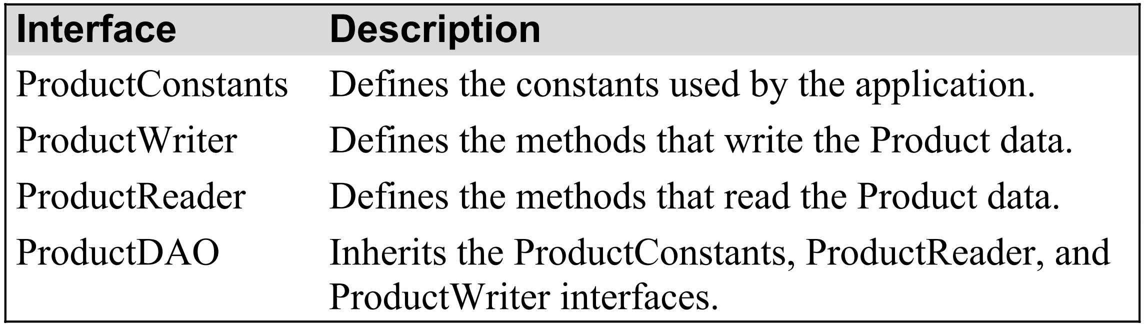
OOP Motivation: Interface

- A class diagram for the Product Maintenance application

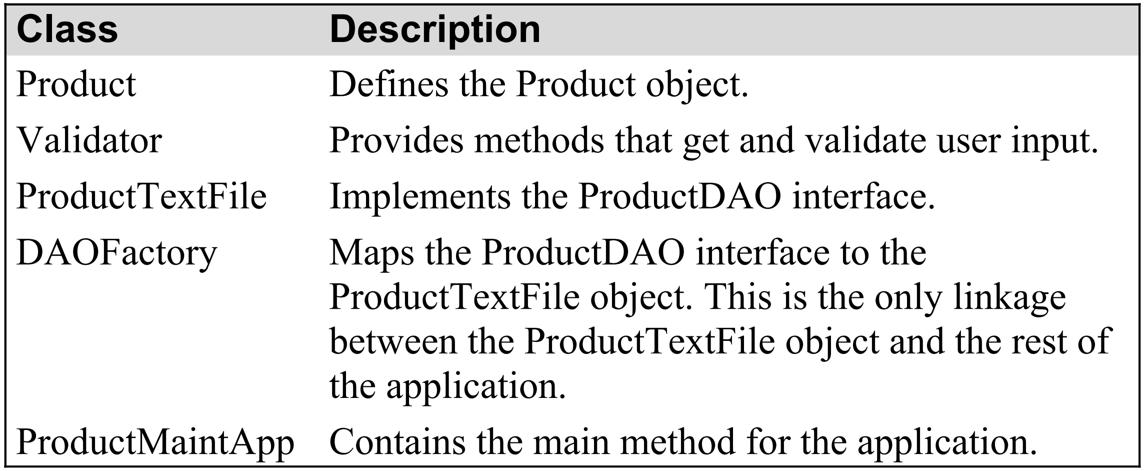


OOP Motivation: Interface

- Interfaces and classes used by the Product Maintenance application



Interface	Description
ProductConstants	Defines the constants used by the application.
ProductWriter	Defines the methods that write the Product data.
ProductReader	Defines the methods that read the Product data.
ProductDAO	Inherits the ProductConstants, ProductReader, and ProductWriter interfaces.



Class	Description
Product	Defines the Product object.
Validator	Provides methods that get and validate user input.
ProductTextFile	Implements the ProductDAO interface.
DAOFactory	Maps the ProductDAO interface to the ProductTextFile object. This is the only linkage between the ProductTextFile object and the rest of the application.
ProductMaintApp	Contains the main method for the application.

OOP Motivation: Inheritance & Interface

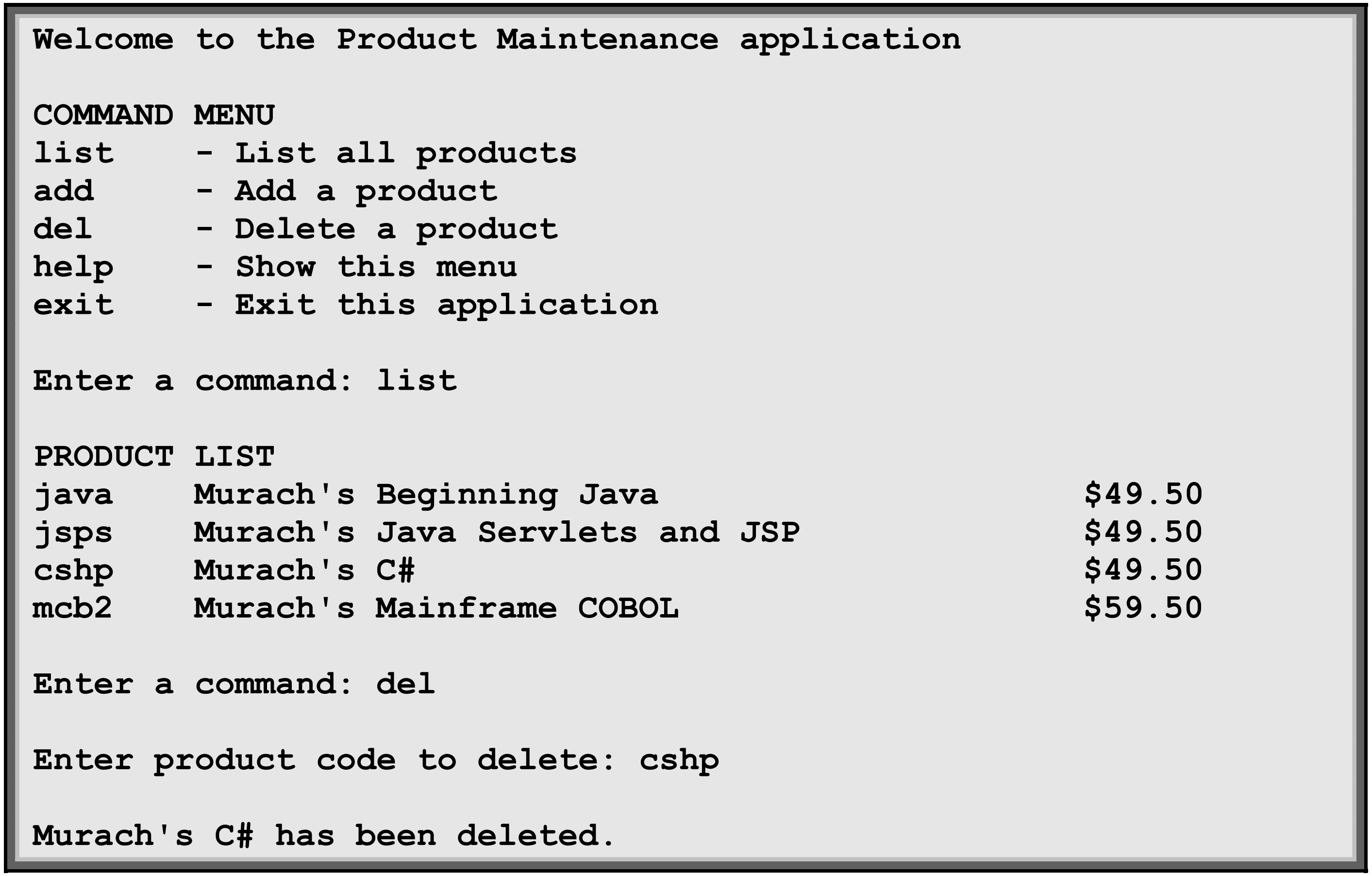
Putting it all Together

Code Walk Through

ProductMaintApp

OOP Motivation: Inheritance & Interface

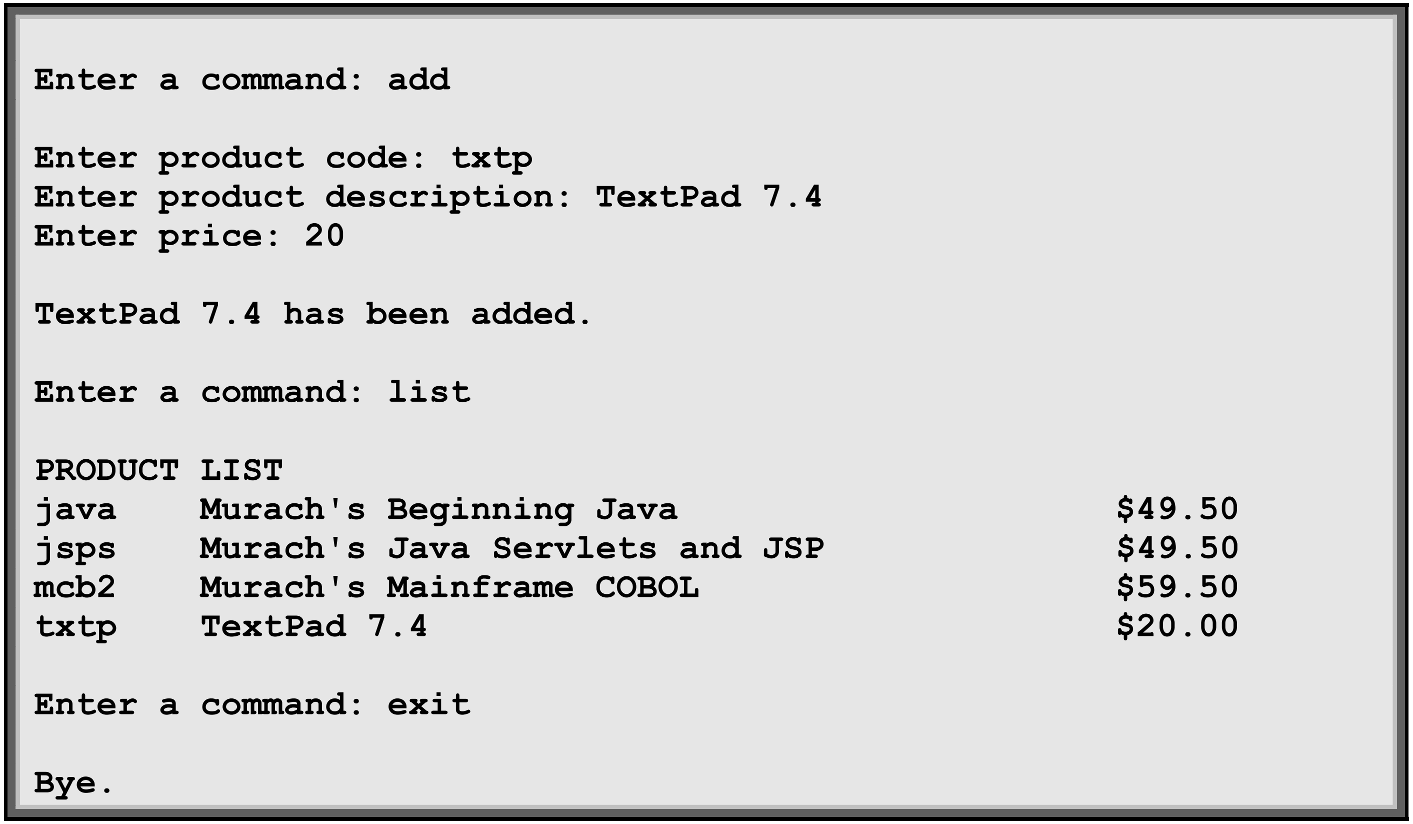
- Code walk through of Product Maintenance Application



```
Welcome to the Product Maintenance application

COMMAND MENU
list    - List all products
add     - Add a product
del     - Delete a product
help    - Show this menu
exit    - Exit this application

Enter a command: list

PRODUCT LIST
java    Murach's Beginning Java                     $49.50
jsps    Murach's Java Servlets and JSP              $49.50
cshp    Murach's C#                                 $49.50
mcb2    Murach's Mainframe COBOL                    $59.50

Enter a command: del

Enter product code to delete: cshp

Murach's C# has been deleted.
```

OOP Motivation: Inheritance & Interface

- Code walk through of Product Maintenance App (cont.)



```
Enter a command: add

Enter product code: txtp
Enter product description: TextPad 7.4
Enter price: 20

TextPad 7.4 has been added.

Enter a command: list

PRODUCT LIST
java    Murach's Beginning Java                     $49.50
jsps    Murach's Java Servlets and JSP              $49.50
mcb2    Murach's Mainframe COBOL                    $59.50
txtp    TextPad 7.4                                 $20.00

Enter a command: exit

Bye.
```

JDK 8: Default Interface Method

- JDK 8 (1.8) allows you to define methods in the interface instead of leaving them as abstract

```
interface MyIF {
    // This is a "normal" interface method declaration.
    // It does NOT define a default implementation.
    int getNumber();

    // This is a default method. Notice that it provides
    // a default implementation.
    default String getString() {
        return "Default String";
    }
}
```

- Provides a mechanism to add new methods to existing interfaces without breaking backwards compatibility

JDK 8: Default Interface Method

- Allows you to include static methods in an interface
- Static methods, by definition, are not abstract

```
interface MyIF2 {
    // This is a "normal" interface method declaration.
    // It does NOT define a default implementation.
    int getNumber();

    // This is a default method. Notice that it provides
    // a default implementation.
    default String getString() {
        return "Default String";
    }

    // This is a static interface method.
    static int getDefaultNumber() {
        return 0;
    }
}
```

JDK 8: Functional Interface

- A *functional interface* is an interface that contains one and only one abstract method.
- Normally, this method specifies the intended purpose of the interface.
- Thus, a functional interface typically represents a single action. For example:
  - The standard interface **Runnable** is a functional interface because it defines only one method: **run()**. Therefore, **run()** defines the action of **Runnable**
  - Furthermore, a functional interface defines a *target type* of a lambda expression.
  - A lambda expression can only be used in a context in which its target type is specified

JDK 8: Functional Interface

- Previously all interface methods were abstract
- Now, an interface method is abstract only if it does not specify a default implementation.
- Because default interface methods are implicitly abstract, there is no need to use the **abstract** modifier.
- Example of a *functional interface:*

```
interface MyNumber {
    double getValue();
}
```

  - Here, method **getValue()** is implicitly abstract, and only method defined by **MyNumber**. Hence **MyNumber** is a functional interface and its function is defined by **getValue()**

OOP Motivation: Nested Classes

- Two classes nested within another class

```
public class OuterClassName {
    // can contain instance variables and methods
    // can contain static variables and methods
    class InnerClassName{
        // can contain instance variables and methods
        // can't contain static variables or methods
        // can access all variables and methods of
        // OuterClass
    }
    static class StaticInnerClassName{
        // can contain instance variables and methods
        // can contain static variables and methods
        // can access static variables and methods of
        // OuterClass
        // can't access instance variables or methods of
        // OuterClass
    }
}
```

- Code walk through TestMiscInnerClass.java in hansoninclass package

OOP Motivation: Nested Classes

- The class files generated for the nested classes

```
OuterClassName.class
OuterClassName$InnerClassName.class
OuterClassName$StaticInnerClassName.class
```

A class nested within a method

```
public class ClassName {
   // code for the outer class
    public void methodName(){
        class InnerClassName{
            // code for the inner class
        }
        // code for the method
    }
}
```

The class files generated for this class

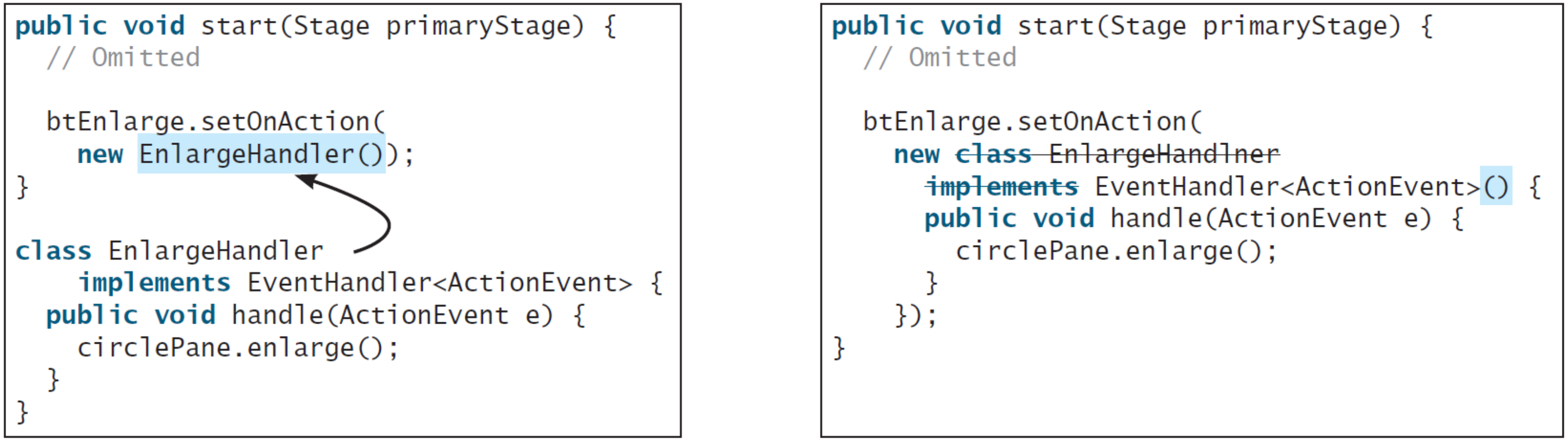
```
ClassName.class
ClassName$InnerClassName.class
```

Anonymous Inner Classes

- Inner class listeners can be shortened using anonymous inner classes.
- An anonymous inner class is an inner class without a name. It combines declaring an inner class and creating an instance of the class in one step.
- An anonymous inner class is declared as follows:

```
new SuperClassName/InterfaceName() {
         // Implement or override methods in superclass or interface
        // Add other methods if necessary
}
```

Anonymous Inner Classes (cont.)

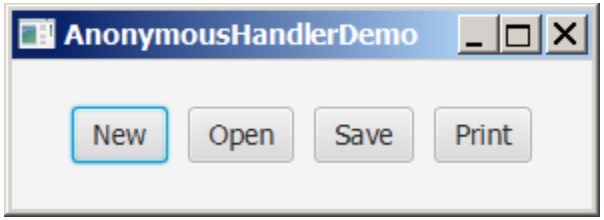


```java
public void start(Stage primaryStage) {
  // Omitted

  btEnlarge.setOnAction(
    new EnlargeHandler());
}

class EnlargeHandler
    implements EventHandler<ActionEvent> {
  public void handle(ActionEvent e) {
    circlePane.enlarge();
  }
}
```

(a) Inner class EnlargeListener

```java
public void start(Stage primaryStage) {
  // Omitted

  btEnlarge.setOnAction(
    new ~~class EnlargeHandlner~~
      ~~implements~~ EventHandler<ActionEvent>() {
      public void handle(ActionEvent e) {
        circlePane.enlarge();
      }
    });
}
```

(b) Anonymous inner class



OOP Motivation: Nested Classes

- More inner class example – graphics programming:

```
public TestInnerClass(String s) {
    super(s);
    setLayout(new FlowLayout());
    Button pushButton = new Button("Go Ahead, Click Me");
    add(pushButton);
    pushButton.addActionListener(this);

    //declare a inner class
    class InnerClass_WA extends WindowAdapter {
        public void windowClosing(WindowEvent e) {
                                    System.exit(0);
        }
    }
    InnerClass_WA icWA = new InnerClass_WA();
    addWindowListener(icWA);
}
```

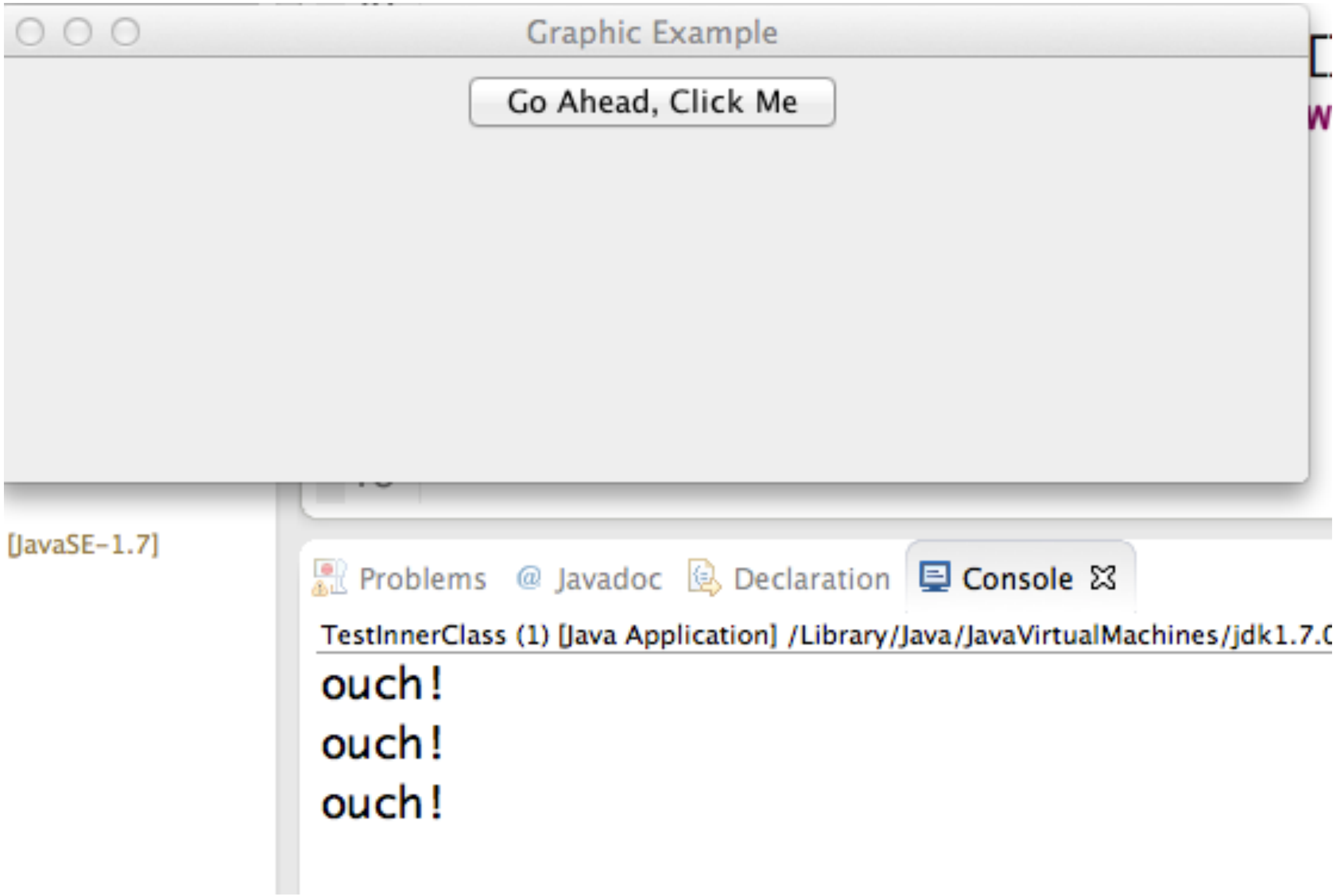
OOP Motivation: Nested Classes

- Inner class example (cont):

```
//This can also be re-written like this

    //instead of fully declaring it
    class InnerClass_WA extends WindowAdapter {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    }
    InnerClass_WA icWA = new InnerClass_WA();
    addWindowListener(icWA);

    //alternately, use anonymous class
    //used extensively in event driven graphics
    //programming (android, desktop, applet)
     addWindowListener(new WindowAdapter()
       {
          //inner anonymous class
          public void windowClosing(WindowEvent e) {
              System.exit(0);
          }
       }
     );
```

OOP Motivation: Inner Class

- Code walk through -- inner class (TestInnerClass.java)



```
TestInnerClass (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.(
ouch!
ouch!
ouch!
```

OOP Motivation: Enumeration

- An *enumeration* defines a type and contains a set of related constants
- The constants are given an int value starting from 0
- Enums are type safe (that means you can't assign to another type)

OOP Motivation: Enumeration

- The syntax of declaring an enumeration

```
public enum EnumerationName
{
    CONSTANT_NAME1[,
    CONSTANT_NAME2]...
}
```

An enumeration that defines three shipping types

```
public enum ShippingType
{
    UPS_NEXT_DAY,
    UPS_SECOND_DAY,
    UPS_GROUND
}
```

A statement that uses the enumeration and one of its constants

```
ShippingType secondDay = ShippingType.UPS_SECOND_DAY;
```

OOP Motivation: Enumeration

- A method that uses the enumeration as a parameter type

```
public static double getShippingAmount(ShippingType st)
{
    double shippingAmount = 2.99;
    if (st == ShippingType.UPS_NEXT_DAY)
        shippingAmount = 10.99;
    else if (st == ShippingType.UPS_SECOND_DAY)
        shippingAmount = 5.99;
    return shippingAmount;
}
```

A statement that calls the method

```
double shippingAmount =
    getShippingAmount(ShippingType.UPS_SECOND_DAY);
// double shippingAmount2 = getShippingAmount(1);
                            // Wrong type, not allowed
```

Two methods of an enumeration constant

- `name()`
- `ordinal()`

OOP Motivation: Enumeration

- An enumeration that overrides the toString method

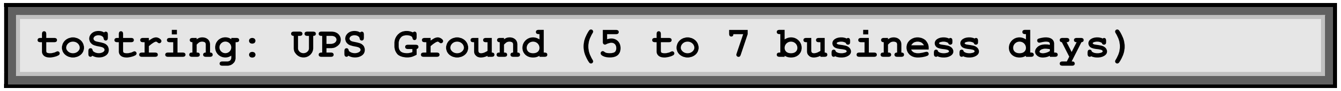
```
public enum ShippingType
{
    UPS_NEXT_DAY,
    UPS_SECOND_DAY,
    UPS_GROUND;

    @Override
    public String toString() //amazing, you can have
                             //method in an enum
    {
        String s = "";
        if (this.ordinal() == 0)
            s = "UPS Next Day (1 business day)";
        else if (this.ordinal() == 1)
            s = "UPS Second Day (2 business days)";
        else if (this.ordinal() == 2)
            s = "UPS Ground (5 to 7 business days)";
        return s;
    }
}
```

OOP Motivation: Enumeration

- Code that uses the overridden toString method

```
ShippingType ground = ShippingType.UPS_GROUND;
System.out.println("toString: " + ground.toString() +
    "\n");
```

Resulting output



```
toString: UPS Ground (5 to 7 business days)
```

How to code a static import statement

```
import static murach.business.ShippingType.*;
```

The code above when a static import is used

```
ShippingType ground = UPS_GROUND;
System.out.println(
    "toString: " + ground.toString() + "\n");
```

Further Reading

- Murach's Java Programming:
  - Chapter 8, 9 (up to page 303 only) & 10 (pages 326 – 331)

Next Lecture

- Arrays
- Collections
- Generics

Summary

- Object Oriented Programming
  - Inheritance
  - Interface
  - Polymorphism
- Nested classes
- Enumeration