Université Concordia University
CENTRE FOR CONTINUING EDUCATION

INTRODUCTION TO PROGRAMMING WITH JAVA - CEJV416

Lecture #12

Interfaces

How to work with the abstract keyword

- An *abstract class* is a class that can be inherited by other classes but that you can't use to create an object.
- To declare an abstract class, code the abstract keyword in the class declaration.
- An abstract class can contain fields, constructors, and methods just like other superclasses. It can also contain abstract methods.
- To create an *abstract method*, you code the abstract keyword in the method declaration and you omit the method body.
- Abstract methods cannot have private access. However, they may have protected or default access (no access modifier).
- When a subclass inherits an abstract class, all abstract methods in the abstract class must be overridden in the subclass.
- Any class that contains an abstract method must be declared as abstract.

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();
}
```

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";
    }
}
```

A class that inherits the abstract Product class

```
public class Software extends Product
{
    private String version;

    // regular constructor and methods for the Software
class

    public String getDisplayText()
                          // implement the abstract method
    {
        return super.toString() +
            "Version is:      " + version + "\n";
    }
}
```

How to work with the final keyword

- To prevent a class from being inherited, you can create a *final class*.
- To prevent subclasses from overriding a method of a superclass, you can create a *final method*.
- To prevent a method from assigning a new value to a parameter, you can create a *final parameter*.
- All methods in a final class are automatically final methods.
- Coding the final keyword for classes and methods can result in a minor performance improvement because the compiler doesn't have to allow for inheritance and polymorphism.

A final class

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

A final method

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

A final parameter

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

Exercise 23

- Use the abstract and final keyword

Interfaces

What is an interface?

Why is an interface useful?

How do you define an interface?

How do you use an interface?

What is an interface? Why is an interface useful?

- An interface is a classlike construct that contains only constants and abstract methods.
- In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify common behavior for objects.
  - For example, you can specify that the objects are comparable, edible, cloneable using appropriate interfaces.

Define an Interface

To distinguish an interface from a class, Java uses the following syntax to define an interface:

```
public interface InterfaceName {
  constant declarations;
  method signatures;
}
```

Example:

```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

A Printable interface that defines a print method

```
public interface Printable {

    public abstract void print();
}
```

Interface concepts

- An *interface* defines a set of public methods that can be implemented by a class.
- The interface itself doesn't provide any code to implement the methods. Instead, it merely provides the method signatures.
- A class that *implements* an interface must provide an implementation for each method defined by the interface.
- An interface can also define public constants. Then, those constants are available to any class that implements the 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;
    }
```

A Product class that implements the Printable interface (cont.)

```
    // get and set methods for the fields

    // implement the Printable interface
    @Override
    public void print() {
        System.out.println("Code:           " + code);
        System.out.println(
            "Description:   " + description);
        System.out.println(
            "Price:         " +
            this.getFormattedPrice());
    }
}
```

Code that uses the print method of the Product class

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

Resulting output

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

An abstract class compared to an interface

Abstract class
Variables Constants Static variables Static constants
Methods Static methods Abstract methods

Interface
Static constants
Abstract methods

A Printable interface

```
public interface Printable {

    public abstract void print();
}
```

A Printable abstract class

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

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.

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.

Interfaces of the Java API

- The Java API defines many interfaces that you can implement in your classes.
- An interface that doesn't contain any constants or methods and that is primarily used to identify some aspect of the object is known as a *tagging interface*.

Some interfaces in the java.lang package

Interface	Methods
`Cloneable`	None
`Comparable`	`int compareTo(Object o)`

Some interfaces in the java.util and java.awt.event packages

Interface	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)`

How to code an interface

- Declaring an interface is similar to declaring a class except that you use the interface keyword instead of the class keyword.
- In an interface, all methods are automatically declared public and abstract, and all constants are automatically declared public, static, and final, so the access modifiers are optional.
- Interface methods can’t be static.

Omitting Modifiers in Interfaces

All data fields are *public final static* and all methods are *public abstract* in an interface. For this reason, these modifiers can be omitted, as shown below:

```
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T1 {
  int K = 1;

  void p();
}
```

A constant defined in an interface can be accessed using syntax InterfaceName.CONSTANT_NAME (e.g., T1.K).

22

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);
}
```

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]...{}
```

How to implement an interface

- To declare a class that implements an interface:
  - you use the implements keyword
  - you provide an implementation for each method defined by the interface
- If you forget to implement an interface method, the compiler will issue an error message.
- A class that implements an interface can use any constant defined by that interface.

A 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;
    }
```

A class that implements two interfaces (cont.)

```
    @Override
    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);
    }
}
```

The syntax for inheriting a class and implementing an interface

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

How to inherit a class and implement an interface

- A class can inherit another class and also implement one or more interfaces.
- If a class inherits another class that implements an interface:
  - the subclass automatically implements the interface (but you can code the implements keyword in the subclass for clarity)
  - the subclass has access to any methods of the interface that are implemented by the superclass and can override those methods

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;
    }

    public void setAuthor(String author) {

        this.author = author;
    }

    public String getAuthor() {

        return author;
    }
```

A Book class that inherits Product and implements Printable

```
        // implement the Printable interface
        @Override
        public void print() {
            System.out.println("Code:\t" + super.getCode());
            System.out.println(
                "Title:\t" + super.getDescription());
            System.out.println("Author:\t" + this.author);
            System.out.println(
                "Price:\t" + super.getFormattedPrice());
        }
    }
```

A method that accepts a Printable object

```
private void printMultiple(Printable p, int count) {

    for (int i = 0; i < count; i++)
        p.print();
}
```

Code that passes a Product object to the method

```
Product product = new Product(
    "java", "Murach's Beginning Java", 49.50);
printMultiple(product, 2);
```

Resulting output

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

Code that passes a Printable object to the method

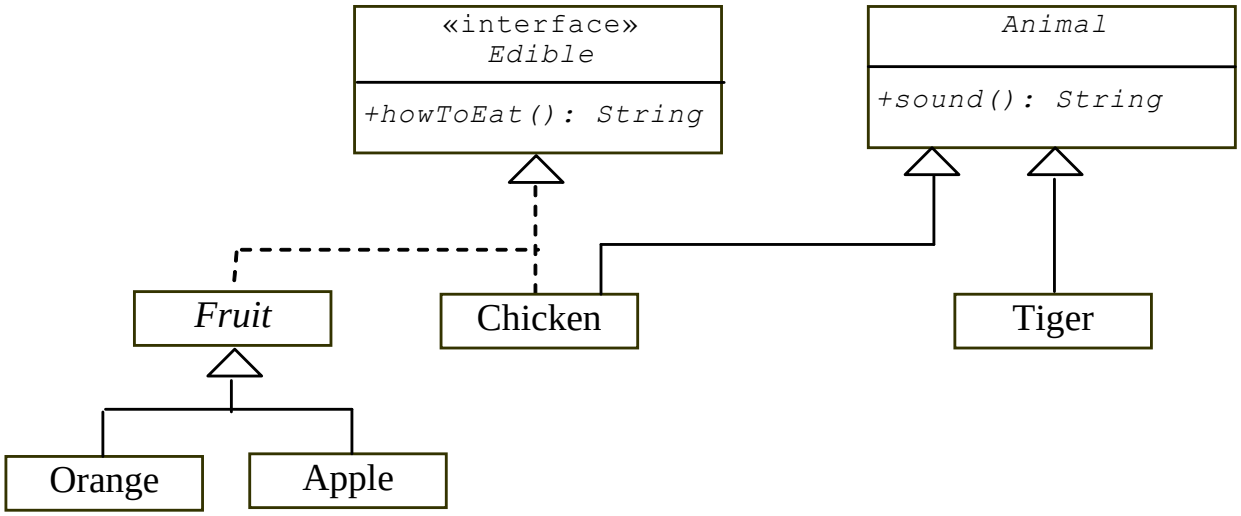
```
Printable product = new Product(
    "java", "Murach's Beginning Java", 49.50);
printMultiple(product, 2);
```

Resulting output

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

Example

You can now use the Edible interface to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the implements keyword. For example, the classes Chicken and Fruit implement the Edible interface (See TestEdible).



Exercise 24

- Create and work with interfaces