# Interfaces

In the Java programming language, an *interface* is a reference type, similar to a class, that can contain *only* constants, method signatures, and nested types. There are no method bodies. Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.

* *Interfaces* are contracts
* An *interface* is a reference type, similar to a class except that it can contain *only* constants, method signatures, and nested types.
* There are no method bodies.
* Interfaces cannot be instantiated—they can only be  *implemented* by classes or *extended* by other interfaces.
* Method signatures have no braces and are terminated with a semicolon.
* To use an interface, you write a class that *implements* the interface.

An interface declaration consists of modifiers, the keyword interface, the interface name, a comma-separated list of parent interfaces (if any), and the interface body.

Unlike interfaces, abstract classes can contain fields that are not static and final, and they can contain implemented methods. Such abstract classes are similar to interfaces, except that they provide a partial implementation, leaving it to subclasses to complete the implementation. If an abstract class contains *only* abstract method declarations, it should be declared as an interface instead.

Multiple interfaces can be implemented by classes anywhere in the class hierarchy, whether or not they are related to one another in any way. Think of Comparable or Cloneable, for example.

# Clonable

Clone method belongs to the “class” object (the highest class object)

Has no methods

**Implementing the Cloneable interface :** interface Cloneable {}

The clone() method of object class can be redefined.

Another way to make a deep copy is to use serialization and write the object to a stream later you can read the object to a different object reference.

Remember deep local copies are needed sometimes.

public Employee clone() throws CloneNotSupportedException

{

// call Object.clone()

Employee cloned = (Employee) super.clone();

// clone mutable fields

cloned.hireDay = (Date) hireDay.clone();

return cloned;

}

**The clone() Method**

If a class, or one of its superclasses, implements the Cloneable interface, you can use the clone() method to create a copy from an existing object. To create a clone, you write:

*aCloneableObject*.clone();

Object's implementation of this method checks to see whether the object on which clone() was invoked implements the Cloneable interface. If the object does not, the method throws a CloneNotSupportedException exception.

The clone() method must be declared as protected Object clone() throws CloneNotSupportedException

or:

public Object clone() throws CloneNotSupportedException if you are going to write a clone() method to override the one in Object.

If the object on which clone() was invoked does implement the Cloneable interface, Object's implementation of the clone() method creates an object of the same class as the original object and initializes the new object's member variables to have the same values as the original object's corresponding member variables.

# Comparable

public class Student implements Comparable

This means it must implement the method  
 public int compareTo(Object o)

Notice that the parameter is an Object

In order to implement this interface, our parameter must also be an Object, even if that’s not what we want

public int compareTo(Object o) throws ClassCastException {  
 if (o instanceof Student)  
 return score - ((Student)o).score;  
 else  
 throw new ClassCastException("Not a Student!");  
}

A ClassCastException should be thrown if we are given a non-Student parameter

class Author implements Comparable<Author>{

String firstName;

String lastName;

@Override

public int compareTo(Author other){

// compareTo should return < 0 if this is supposed to be

// less than other, > 0 if this is supposed to be greater than

// other and 0 if they are supposed to be equal

int last = this.lastName.compareTo(other.lastName);

return last == 0 ? this.firstName.compareTo(other.firstName) : last;

}

}

Packages.

Packages are used to group related classes together. Create a directory same name as the package and store the .class file there. Other programs can use the classes in the package by providing an import statement.The package can be a subdirectory of the current package or even add it to the java libraries (not recommended).

Import package-name.\*;

If different programs are to use the package then set the environment variable classpath to the location of the package. Package access can be summarized with the visibility modifiers. Access same class and same package. Not subclasses or other classes in different packages.

# Swing dialogues/Settings

showMessageDialog(***<parent>***, ***<message>***)

showInputDialog(***<parent>***, ***<message>***)

**JOptionPane.showMessageDialog**(null,"How's the weather?");

JMenuBar menubar = new JMenuBar();

frame.setJMenuBar(menubar);

// create the File menu

JMenu fileMenu = new JMenu("File");

menubar.add(fileMenu);

JMenuItem openItem = new JMenuItem("Open");

fileMenu.add(openItem);

openItem.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

openFile();

}

});

# Layouts

GridLayout: public GridLayout(int rows, int columns)

FlowLayout: Container panel = new JPanel(new FlowLayout());

BorderLayout: frame.add(new JButton("east"), BorderLayout.EAST);

frame.add(new JButton("center"), BorderLayout.CENTER);

BoxLayout: Box.createHorizontalBox() Box.createVerticalBox()

Strut: fixed size.

**Component createHorizontalStrut(int width)**

**Component createVerticalStrut(int height)**

Box.*createRigidArea (****new Dimension (0, 10)***

Glue: fills available space.

**Component createHorizontalGlue()**

**Component createVerticalGlue()**

# JcheckBox

pizzaLabel = new JLabel("choose your topping: ");

topping= new JCheckBox[NUM\_TOPPINGS];

topping[0] = new JCheckBox("Extra cheese",false);

topping[1] = new JCheckBox("Peperoni",false);

topping[2] = new JCheckBox("Anchovies",false);

topping[3] = new JCheckBox("Mushrooms",false);

topping[4] = new JCheckBox("Olives",false);

orderButton = new JButton("Place order");

orderButton.addActionListener(new ActionListener(){public void actionPerformed(ActionEvent event){

processOrder();

}

});

### What is a class?

### What is an interface?

### What is the difference between a class and an interface in Java?

### A teacher has a list of student names (Strings) from her class. How does she sort this list?

### Using code explain your answer.

Implement comparable inside an array sorting class to return a list in the correct order.

### What if the teacher is using objects of type Student. Student contains instance variables as

### follows: String name, String address, int age, String course.

### How does she sort this list into alphabetical order? What if she wanted to sort this list using another instance variable say course or age. How is this achieved?

### What is a Design Pattern? Give one example?

MVC pattern

### Show the code required to allow the user to enter a string and store that string in a text file.

### File patientData = new File("patient\_data.txt");

Scanner procScanner = **new** Scanner(procedureData);

FileWriter saveFile = new FileWriter("patient\_data.txt");

PrintWriter fileWrite = new PrintWriter(saveFile);

### Describe how you would write an object to a file. What part of the object would you not want to save and how is this achieved.

Serializable, Jaxb