Helwan University

Faculty of Computer Science

Programming Language 2

Object oriented design using JAVA

Dr. Ayman Ezzat

Email: ayman@fcih.net

Web: www.fcih.net/ayman

Unit 5: Binary Files, Collections, Packages and abstraction

Outline

1. **Binary Files**
2. **Introduction**
3. **Java libraries**
   - The `java.util` package
4. **Abstraction in class hierarchies**

Binary Files 1/3

Remember Interfaces !
Serializable has no functions in JAVA

```
public class Students implements Serializable{
     public int id;
    public String FullName;
    public int age;
    public double []SubjectsFall;
    public Students()
    {
        SubjectsFall=new double[5];


    }
```

Binary Files 2/3

```
Students Ahmed=new Students();
Ahmed.FullName="Ahmed Mohamed Ibrahim";
Ahmed.id=12;
Ahmed.SubjectsFall[0]=45;
Ahmed.SubjectsFall[1]=56;
Ahmed.SubjectsFall[2]=95;

ObjectOutputStream Bin=new ObjectOutputStream(new FileOutputStream("d:\\JavaFiles\\StorageStudentsDB.bin"));
Bin.writeObject(Ahmed);
Bin.close();
ObjectInputStream Inp=new ObjectInputStream(new FileInputStream("d:\\JavaFiles\\StorageStudentsDB.bin"));
Students x=(Students)Inp.readObject();
System.out.println(x.FullName);
```

Writing and reading object in One LINE !

Casting again

Binary Files 3/3 Extending Binary to arrays

```
Students []Array=new Students[5];
double []Scores={32,45,88,98.5,30.5};
Array[0]=new Students(1,"First Students",19,Scores);
Array[1]=new Students(2,"Second Students",29,Scores);
Array[2]=new Students(3,"Third Students",39,Scores);
Array[3]=new Students(4,"Forth Students",49,Scores);
Array[4]=new Students(5,"Fifth Students",59,Scores);
Bin=new ObjectOutputStream(new FileOutputStream("d:\\JavaFiles\\StorageStudentsDBArray.bin"));
Bin.writeObject(Array);
Bin.close();
Array=null;
Inp=new ObjectInputStream(new FileInputStream("d:\\JavaFiles\\StorageStudentsDBArray.bin"));
Array=(Students[])Inp.readObject();
System.out.println(Array[4].FullName);
```

Read Once

Information hiding

Access modifiers (Cont'd)

(2) The protected and default access levels

- **default access** of a member means it has **no access modifier**.
- **Accessing protected/default members**: classes in the same package can access **protected and default members** using a **qualified name**.

```
package AppletUser;
public class UserV3
{
    // protected instance variable
    protected int numOfAccesses;

    // default access level method
    void setNumOfAccesses(int n)
    {
        numOfAccesses = n;
    }
}
```

```
import AppletUser.*; // this class does not compile!
public class TestUserV3
{
    public static void main (String[] args)
    {
        // this method has a UserV3 reference
        UserV3 myUser;

        // initialize the reference
        myUser = new UserV3();

        // qualified access to instance variable numOfAccesses
        // is not allowed!
        myUser.numOfAccesses = 1;

        // qualified access to setNumOfAccesses method
        // is not allowed!
        myUser.setNumOfAccesses(1);
    }
}
```

• The class TestUserV3 does not compile, **because** it is not in the same package as UserV3, and therefore it cannot access either the protected or the default access level members of UserV3.

• Instead of importing the classes in the AppletUser package, the TestUserV3 class would have to be **preceded by the words "package".**

3. Information hiding

Access modifiers (Cont'd)

(2) The protected and default access levels (Cont'd)

- **Inheritance of protected members**:
  - Any subclass inherits any **protected** and **public** members of its superclass.

- **Inheritance of default members**:
  - **Default members** are inherited **only** by subclasses in **the same package** as the superclass.
    - i.e. any subclass in **the same package** inherits the **protected, public**, and **default** access level members of its superclass.
  - **Default access** level is sometimes known as '**package access**' for this reason.

- **Protected**: A level between private and public members, it can be accessed inside the class, from driven classes but never accessed from objects.

1. Introduction

This unit discusses the following:

(1) Packages:

Java Packages: The Java language provides the concept of a **package** as a way to group a number of related classes.

- **Standard Java Packages:** They are predefined packages which form **libraries of classes** available for programmers to use. They provide facilities including
  - input/output,
  - mathematical functions,
  - graphical user interface components
  - and data structures.
- **Developer defined package**: A Java program has one or more packages written for that application by the developer, together with standard Java packages.

(2) interfaces and abstract classes:

Interfaces are a powerful way of specifying what a class should do, without specifying in detail how it should implement this.

Abstract classes are useful in defining groups of classes related by inheritance.

2. Java libraries

- Standard packages = Class libraries = Java API (application programming interface)
  - Example: java.net package may be referred to as the networking API.

Table 1 Some packages from the Java class libraries

Package	*Provides classes for*
`java.applet`	programs (applets) that can be run from a web page
`java.awt`	Abstract Windowing Toolkit (AWT) – basic graphical user interface (GUI) components such as windows, fonts, colours, events, buttons and scroll bars
`java.io`	low-level input/output – for example, reading data from files or displaying on screen
`java.lang`	basic classes for the language – automatically imported and used by all Java programs
`java.net`	communication across a network, using clients, servers, sockets and URLs
`javax.swing`	creation of more sophisticated, platform-independent GUIs, building on the AWT capabilities
`java.util`	general utility classes, especially collection classes (data structures)

2. Java libraries

Accessing standard library classes

Method1: by writing the **fully qualified name**, for example:

```
java.util.Scanner sc = new java.util.Scanner(System.in);
```

Method2: by **importing** the library near the start of your source file. You can then refer to the class by its **simple name**, for example:

```
import java.util.Scanner;     OR      import java.util.*;
...
Scanner sc = new Scanner(System.in);
```

Notes:

1) **The wild card (*)**: Using the wild card instead of a specific class name allows the programmer to use any of the public items (mainly classes) contained in the java.util package.

2) **Import-on-demand:** Using a wild card import declaration will not make your program any larger or slower because the compiler will include only the classes you actually use.

2. Java libraries

3)Name clash: it means two classes with the **same name** in different packages. For example:

```
import java.util.*;
import java.sql.*;
...
Date d1 = new Date(DATE_IN); //Date is ambiguous
```

→ both libraries have a class with the name `Date` and thus we get an **ambiguous name**.

Solution (1): to use the fully qualified name of the Date class (i.e. Method1 above)

```
java.sql.Date d1 = new java.sql.Date(DATE_IN);
```

Solution (2): to import only those classes you actually use, for example:

```
import java.util.ArrayList;
import java.sql.*
```

Note that: The statement `import java.awt.*` only makes the classes in this package visible, not any of the subpackages. For example: If you need to use a class in `java.awt` and `java.awt.event`, then you must write:

```
import java.awt.*;
import java.awt.event.*;
```

4) The special case of `java.lang`:
The only exception to this requirement for explicit naming of imported classes is the `java.lang` package. Since this package contains classes that every Java program will use, you do not need to include an import declaration for it.

2. Java libraries

Creating your own packages

- To define a collection of classes as a package, precede the source code of each of the classes with the keyword `package`, for example: `package general;`
- Package names sometimes have several components, separated by a dot, to reflect the hierarchical organization of packages.
  - Each component of the package name typically corresponds to a subdirectory within the base directory for this project or the base directory for Java libraries
  - For example:

```
package general.utilities;
```

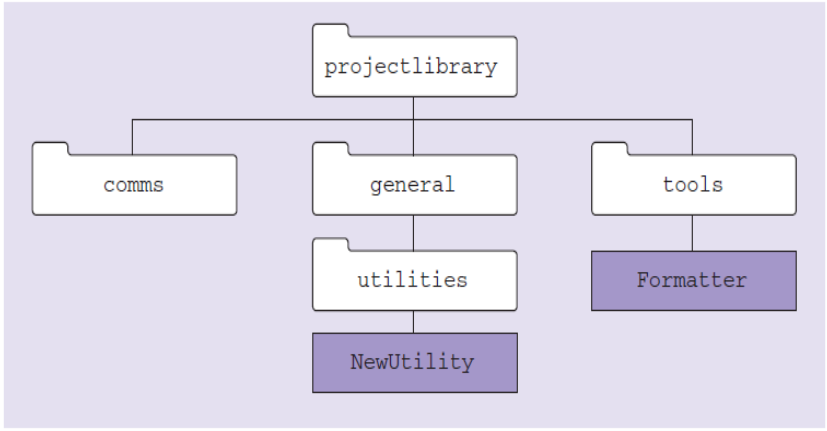


Figure 1 A directory structure for class files in a software project (class files shown shaded)

2. java.util package

The `java.util` package

- This package consists of a number of utility classes, which provide **data structures** and other **facilities**, including the following.
  - **Java Collections Framework** is used for storing and managing objects of any Java type.
    - **A collection** is an object that can be used for storing, retrieving and manipulating groups of items. For example: `LinkedList`, **`ArrayList`**, `HashSet`, `TreeSet`, **`HashMap`** `and TreeMap`. We shall discuss them further in later sections.
  - **'Legacy' collection classes** implement various **simple data structures**.
    - These include the `Vector`, `Hashtable`, `Properties`, `Dictionary`, `Stack` `and BitSet` classes.
    - They are widely used in older Java software, so it is useful to know about them. However, they are not recommended for use in new programs – you should use the Java 2 collection classes instead.
  - **The `Calendar` class and its subclass `GregorianCalendar`** implement dates in a variety of representations.
  - **The `StringTokenizer` class** is useful for processing text input.
  - **The `Random` class** implements methods used for the generation of random numbers.

2. java.util package

- **A collection** is an object that can be used for storing, retrieving and manipulating groups of items.
  - Example: a telephone directory (a collection of names mapped to phone numbers).
- These items may be primitive data or objects.
  - **Homogeneous collections** store data that is all of the same type – for example, an array of *int* can store only *int* values.
  - **Heterogeneous collections** may contain data of a variety of types.
- A **generic class** has at least one member of an unspecified type. A programmer must then specify the desired type when creating an instance of the class. For example:

```
ArrayList<String> myList = new ArrayList<String>(10);
```

  - The angle brackets can be read as 'of', for example **ArrayList of Strings**.
  - The type(s) you provide on instantiation **appear in the API** as single letters in angle brackets after the name of the class, e.g. **ArrayList<E>.**

2. java.util package

(1) The `ArrayList` class

- Objects of the **generic class ArrayList** are similar to **arrays**, but have differences:
  - **Similarities**
    - Both can store a number of references
    - the data can be accessed via an index.
  - **Differences**
    - An ArrayList can automatically **increase in capacity** as necessary
    - The ArrayList object requests **more space** from the Java run-time system, if necessary.
    - The ArrayList is **not ideal in its use of memory**.
      - As soon as an ArrayList requires more space than its current capacity, the system will allocate more memory space for it. The extension is **more than required** for the items about to be added. So there will **often be empty locations** within an ArrayList – that is, **the size will often be less than the capacity**.

- Two important terms in connection with ArrayList objects:
  - **The capacity** of an ArrayList object is the maximum number of items it can currently hold (without being extended).
  - **The size** of an ArrayList object is the current number of items stored in the ArrayList.

2. java.util package

(1) The `ArrayList` class (Cont'd)

How to declare an ArrayList?

Ex1:
```
ArrayList<String> list1 = new ArrayList<String>();
```

- This declaration of an ArrayList specifies that it will hold String references
  - The type "String" can be replaced by any object type.
- This declaration sets up an empty ArrayList The initial capacity is set to a default value. The ArrayList capacity is expanded automatically as needed when items are added.

Ex2:
```
ArrayList<Date> list2 = new ArrayList<Date>(100);
```

- This declaration of an ArrayList specifies that it will hold Date references
  - The type "Date" can be replaced by any object type.
- This declaration can be used to specify the initial capacity –in this case 100 elements. This can be more efficient in avoiding repeated work by the system in extending the list, if you know approximately the required initial capacity.

Interfaces

Interfaces from Java Libraries: the `Iterator` interface

- This interface allows traversing / iterating through any **collection**, such as an **`ArrayList`**.
- The following abstract methods are defined for any class that implements the iterator interface:
  - **`boolean hasNext()`** - returns true if the iteration has more elements
  - **`E next()`** returns the next element in the iteration(E is the type the interface is implemented for)
  - **`void remove()`** - removes from the underlying collection the last element returned by the iterator; note that it is important to precede any call to remove() by a call to next()!

- **Note that it is possible to declare a variable of an interface type**
  - **any class that implements the Iterator interface** may be referenced by such a variable.
    - **ArrayList class implements Iterator interface, & thus may be referenced by a variable of Iterator type.**

```
ArrayList<String> X = new ArrayList<String>();

...

Iterator<String> listX = X.iterator();
while (listX.hasNext())
        System.out.print(listX.next() + " ");
```

Interfaces

Interfaces from Java Libraries: the `Comparable` interface

- This interface allows comparing or sorting objects.
- Any class that implements the **`Comparable interface`** must have a method **`compareTo`**.
  - the type that the comparison will involve is added in <>.
  - The compareTo method allows the class to define an appropriate measure of comparison

```
public interface Comparable<T>
{
    int compareTo(T o);
}
```

Example

In class below, comparison is based on salary.

```
class Employee implements Comparable <Employee>
{
    private String name;
    private String address;
    private float salary;
    ...
    public int compareTo (Employee otherEmployee)
    {
        if (salary > otherEmployee.salary) return 1;
        if (salary < otherEmployee.salary) return -1;
        return 0;
    }
```

How to use the 'comparable' Employee class?

Usage1:

```
Employee e1 = new Employee("ali", "kw", 1000);
Employee e2 = new Employee("hmd", "kw", 2000);
System.out.println(e1.compareTo(e2));
```

Usage2:

- **`Arrays`** is a special class in **java.util**, with a static method called **`sort`**, which can be used to arrange the objects of an array in order, **given that** the objects have **`CompareTo`** method.

```
Employee [ ] staff = new Employee [STAFF_COUNT] ;
...
Arrays.sort(staff);
```

2. java.util package

(1) `ArrayList` class: basic methods (Cont'd)

adding	`void add(int index, E el)`	inserts `el` at the position indicated by `index`
	`boolean add(E el)`	adds `el` to the end of the `ArrayList`, increasing its size by one
getting	`public E get(int index)`	returns the element at the specified position in this `ArrayList`
removing	`boolean remove(Object obj)`	removes the first occurrence, if any, of the specified element in this `ArrayList`
	`E remove(int index)`	removes the element at the specified position in this `ArrayList` and returns its reference
setting	`E set(int index, E el)`	replaces the element at the specified position in this `ArrayList` with the specified element, returning a reference to the removed element
	`Iterator <E> iterator()`	returns an iterator over the elements in this `ArrayList` in proper sequence

An example of the use of the `add` method is shown below:

```
ArrayList<User> list = new ArrayList<User>();
User u = new User("John");
list.add(u);
```

Removing items is straightforward. For example, the code:

```
User u2 = list.remove(0);
```

will remove the user object stored at position 0 in the list.

2. java.util package

(1) The `ArrayList` class – An Example

- **What is the output of the following code?**

```
ArrayList<String> x = new ArrayList<String>(3);
x.add("FirstItem");
x.add("SecondItem");
x.add("ThirdItem");
x.add("FourthItem");
System.out.println(x.toString());
System.out.println(x.get(0));
x.remove("ImaginaryItem");
System.out.println(x.toString());
x.remove("FourthItem");
System.out.println(x.toString());
System.out.println(x.remove(0));
System.out.println(x.toString());
System.out.println(x.remove("SecondItem"));
System.out.println(x.toString());
```

- **Solution:**

```
[FirstItem, SecondItem, ThirdItem, FourthItem]
FirstItem
[FirstItem, SecondItem, ThirdItem, FourthItem]
[FirstItem, SecondItem, ThirdItem]
FirstItem
[SecondItem, ThirdItem]
true
[ThirdItem]
```

2. java.util package

(1) The `ArrayList` class – Another Example

- We construct a data structure that contains: **(1) the details of a computer**, and **(2) all the computers to which it is connected in a local network**.
- The first part of the class, which describes the linkage, is shown below:

```
class LinkingInfo
{
    private String computer;
    private ArrayList<String> linkedComputers;
    ...
```

- **The string `computer`** contains the name of the computer whose details are documented by this class.
- **The ArrayList `linkedComputers`** contains the names of the computers that are linked to that computer.
- We will assume that a number of methods are required:
  - `String getComputerName()` returns the name of the computer whose details are in this object.
  - `int getNumberOfLinks()` returns the number of computers that are linked to the first computer.
  - `boolean isLinked(String comp1)` returns true if the computer comp1 is linked to the computer documented by this object; otherwise it returns false.
  - `void addComputer(String comp1)` adds the name of computer comp1 to the list of linked computers. If the computer is already in the list of linked computers, then a ComputerErrorException exception is raised with the message 'Computer already in links'.
  - `void removeComputer(String comp1)` removes the computer named comp1 from the linked computers. If the computer is not in the list of linked computers, then ComputerErrorException exception is raised with the message 'Computer not in links'.

2. java.util package

(1) The `ArrayList` class – Another Example (Cont'd)

We shall define **two constructors:**

- **The first** is provided with the name of the computer for which the linking information is to be held and an initial value of the number of locations to hold linked computers.
- **The second** has a single argument, the name of the computer, and uses the default initial capacity.

The code for these two constructors is shown below:

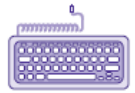
```
public LinkingInfo (String computerName, int capacity)
{
    linkedComputers = new ArrayList<String> (capacity);
    computer = computerName;
}
```

```
public LinkingInfo (String computerName)
{
    linkedComputers = new ArrayList<String> ();
    computer = computerName;
}
```

- We now need to develop some of the methods. The code for the first two methods is shown below. Method getComputerName is trivial as it merely returns the name of the computer.

```
public String getComputerName ()
{
    return computer;
}
```

```
public int getNumberOfLinks ()
{
    return linkedComputers.size();
}
```



Activity 5.3
Completing the coding of the `LinkingInfo` class.

- The getNumberOfLinks method returns the number of linked computers. This uses the size method, which returns the number of items in an ArrayList. This is why we did not need an instance variable for the LinkingInfo class to record the number of computers that are linked.

2. java.util package

Object wrapper classes

Java employed **object wrappers**, which 'wrap around' or encapsulate all the **primitive data types** and allow them to be **treated as objects (See table)**.

Primitive type	*Object wrapper class*
`int`	`Integer`
`long`	`Long`
`short`	`Short`
`byte`	`Byte`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`
`void`	`Void`

2. java.util package

Object wrapper classes (Cont'd)

Examples:

```
ArrayList <Integer> holder =
            new ArrayList <Integer> ();
holder.add(3);
System.out.println(holder.remove(0)+2);
```

Output: 5

```
ArrayList <Character> holder =
                  new ArrayList <Character> ();
holder.add('Z');
System.out.println("The answer is "+ holder.remove(0));
```

Output:
The answer is Z

```
ArrayList<Character> holder = new ArrayList<Character>();
holder.add('a'); holder.add('b'); holder.add('c');
for (Character i : holder) {
    System.out.print(i);
}
```

Output: abc

Notice the for-each statement is written in this format:

```
for (Classname element_identifier: collection_identifier)
```

More Example of array list

```
RacingCar Honda=new RacingCar();
Honda.CurrentSpeed=5;
Honda.MaxSpeed=180;
Honda.ModelName="Honda Civic";
RacingCar BMW=new RacingCar();
BMW.ModelName="BMW z3";
ArrCars.add(Honda);
ArrCars.add(BMW);
System.out.println(ArrCars.size());
System.out.println(ArrCars.get(0).ModelName);
for (RacingCar x:ArrCars)
{
    System.out.println("My model name is "+x.ModelName);
}
```

Print List size

Access element at pos zero

Java Heterogeneous array of objects

```
Object[] arr = new Object[6];

arr[0] = new String("First Pair");
arr[1] = new Integer(1);
arr[2] = new String("Second Pair");
arr[3] = new Integer(2);
arr[4] = new String("Third Pair");
arr[5] = new Integer(3);
System.out.println(arr[3]);
```

2. java.util package

(2) The HashMap class

- In many applications we require two types of object to be associated with each other
  - Example: “user-name” and “password” (see figure)
- A **hash table** can store pairs of linked items in **key–value** pairs.
- In Java, the programmer must specify the **type** of both the key and the **value**; for example, <String, Integer>

Key (user's name)	Value (password)
Einstein	Ahjgeteu
Mandela	Khaiwncoiw
Van Beethoven	nu8we92
Peron	LAmdA3hx
Mao	hwn3jh8kp
Tagore	Slheynvsa
Dickinson	BjKtQhPy7K

Figure 2 A logical view of a **HashMap** object

How to declare a HashMap?

Similar to ArrayList but with the types of both the key and value.

```
HashMap<String,String> table1 = new HashMap<String,String>();
HashMap<String,String> table2 = new HashMap<String,String>(10);
```

2. java.util package

(2) The HashMap class (Cont'd)

Table 5 Basic methods of the HashMap class

Method signature	*Description*
`V get(Object key)`	returns the value to which the key is mapped in this hash map
`V put(K key, V value)`	maps the key to the specified value in this hash map, creating a new key–value pair if necessary
`V remove(Object key)`	removes the key and corresponding value from this hash map

Example:

```
HashMap<String, String> table1 = new HashMap<String, String>();
table1.put("Einstein", "Ahjgeteu");
table1.put("Mandella", "Khaiwncoiw");
System.out.println("Einstein's password is "+table1.get("Einstein"));
for (String name : table1.keySet())
    System.out.println("Name:"+name+", PW:"+table1.get(name));
```

Output:

```
Einstein's password is Ahjgeteu
Name:Einstein, PW:Ahjgeteu
Name:Mandella, PW:Khaiwncoiw
```

Key (user's name)	Value (password)
Einstein	Ahjgeteu
Mandela	Khaiwncoiw

Hashmap Encryption

```
HashMap<String, String> MyTable=new HashMap<String,String>();
MyTable.put("A", "$%"); MyTable.put("B", "5@"); MyTable.put("C", "~t");
MyTable.put("D", "^a"); MyTable.put("E", "as"); MyTable.put("F", "C?");
MyTable.put("G", "W&"); MyTable.put("H", "!o");
String Word="BAD";
String EncText="";
for (int i=0;i<Word.length();i++)
{
    String name=""+Word.charAt(i);
    EncText+=MyTable.get(name);

}
System.out.println(EncText);
```

Hashmap Versus ArrayList

	ArrayList	HashMap
Interface	implements List Interface	implementation of Map interface
Memory consumption	Low	High
Order	Maintain Order	Key value pair no order
Duplicates	Allow duplicates	Overwrite duplicate
NULL	can have any number of null elements	HashMap allows one null key and any number of null values
get method	By Index	fetched by specifying the corresponding key

Hashmap duplicates

```
MyTable.put("A", "DD");MyTable.put("A", "$%");
```

4. Abstraction in class hierarchies

- **An abstract class** defines a common message protocol and a common set of instance variables for its subclasses.
- One **cannot create instances** of abstract classes.
- Abstract classes are specified in the class header using the Java keyword abstract

```
public abstract class Employee
{
    ...
}
```

- Abstract classes **usually** define **abstract methods**, that are then overridden in the concrete classes derived from them.
- It is possible to declare variables of an abstract class type
- it is not possible to create objects of an abstract class (*see example after 2 slides*).

- Note that a fully implemented class, with no abstract methods, is known as a **concrete class**.
  - We also use the term **concrete method** to identify fully implemented methods, as opposed to abstract methods.

4. Abstraction in class hierarchies

When you should not use an abstract class

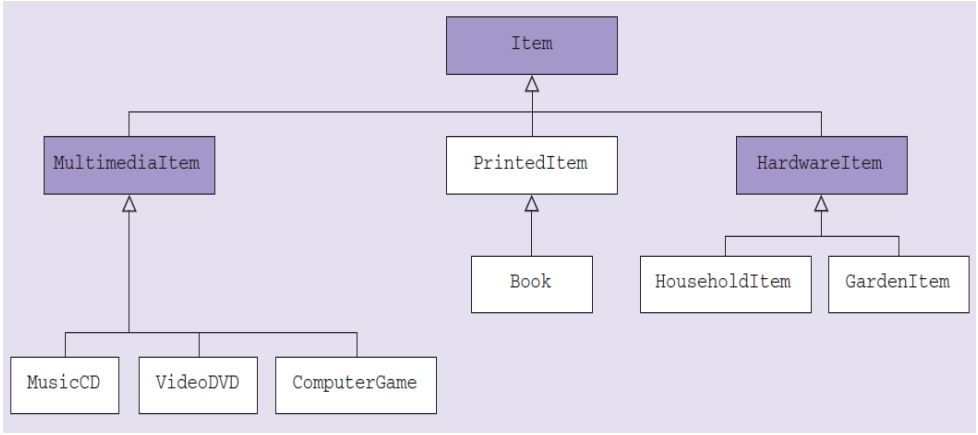
- A class type should be declared abstract only if the intent is that concrete subclasses can be created to complete the implementation.

Designing an abstract class hierarchy

- Construct an abstract class hierarchy **by factoring out common** behavior in a number of subclasses.

Example

A system needs to keep track of a wide variety of items. We can define a number of classes to model the various sorts of items, e.g. Book, MusicCD, VideoDVD, ComputerGame, and so on. We find that there are certain common features between many of the classes. For example, each class will need instance variables, itemName, itemCode and unitPrice, and associated methods for accessing or modifying this data. Instead of repeatedly implementing these separately in each class, we can **factor out** the common behavior by defining a more general class Item, from which all these classes inherit. The Item class will define instance variables itemName, itemCode and unitPrice.



The associated methods for accessing or modifying this data may be either **concrete** or **abstract**, depending on the circumstances; for example, getName is likely to be the same for all subclasses of Item, so it can be implemented as a concrete method. If returning the unit price depends on factors that differ between types of item, then getUnitPrice could be left as an abstract method to be overridden by each specialized subclass. In this case, Item must be declared as an abstract class. We can apply this factoring at a number of levels, if appropriate. So, for example, we may be able to identify commonality between MusicCD, VideoDVD and ComputerGame. In this case we can factor out the common aspects and define a superclass, say, MultimediaItem. This could be either concrete or abstract, as before, depending on how similar the common behavior is.

4. Abstraction in class hierarchies

Abstract classes and polymorphism

- Consider the class hierarchy on the right. Assume we have defined an array of Employee references as follows:

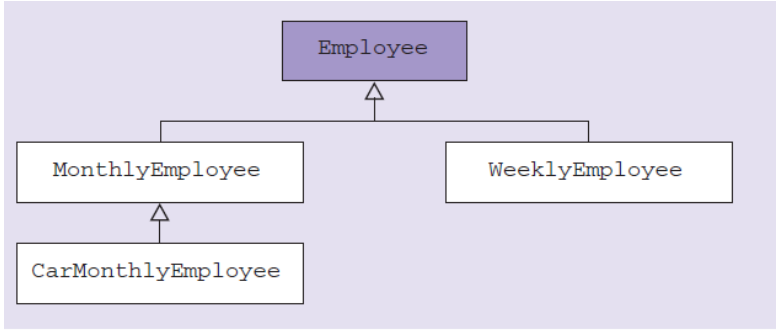
```
Employee[] staff = new Employee [10];
```

NOTE: **We are not creating Employee objects; only Employee references.**

We can then store references to objects of any subclass of Employee in the array, e.g.

```
staff[0] = new WeeklyEmployee(…)

...
```



An example of a class hierarchy with an abstract class (shaded)

- We can take advantage of the common behavior imposed on the hierarchy of classes by the Employee abstract class. If we want to display the wages for all employees in the **staff** array, we write the code on the right. Here we are using the **calculatePay** method, which we know every employee object must have regardless of whether it is a **MonthlyEmployee** object, a **WeeklyEmployee** object or any other subclass of **Employee**. This is an example of **polymorphism**.
  - A message to which objects of more than one class can respond to is polymorphic or to show polymorphism

```
public void displayPayroll ()
{
    for (int e = 0; e < staff.length; e++)
    {
        int pay = staff[e].calculatePay();
        String name = staff[e].getName();
        System.out.println(name + " " + pay);
    }
}
```

Example of Abstract method

```
public abstract class Item {
    public String ItemName;
    protected  int UnitPrice;
    public abstract void setUnitPrice(int x);
    public int getPrice()
    {
        return this.UnitPrice;
    }
}
```

Declaring properties

Function with implementation

4. Abstraction in class hierarchies

Abstract classes compared to interfaces

DIFFERENCES

Interface:

- An interface is **not a class**.
- All its **methods are abstract** and it may contain only constants.
  - The methods are implicitly treated as abstract and the constants are treated as if declared as public static final.
- An interface can be **implemented by any number of unrelated classes**, which declare this using the implements keyword. **A class may implement any number of interfaces**.

Abstract class:

- An abstract class uses the keyword `abstract` in the class header.
- It normally has **at least one abstract method**, either defined within the class or inherited from a superclass, and its abstract methods must be explicitly declared abstract. It **can also have concrete methods** (that is, it may be fully implemented) and instance variables, unlike an interface.
- It can only constrain its own concrete subclasses, by requiring them to implement its abstract methods – it cannot constrain any other classes.

SIMILARITIES

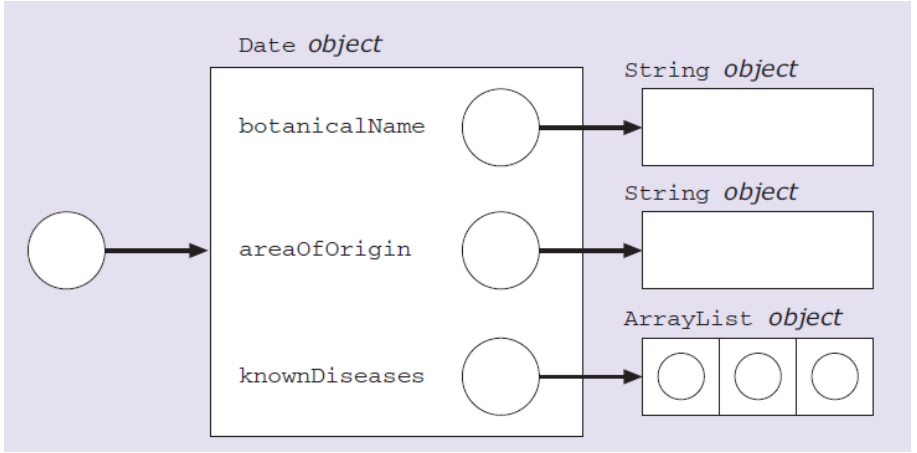
- They can both place requirements on objects of other classes.
- Both can have abstract methods (although neither need have abstract methods).
- **You cannot create objects** of an abstract class type or an interface type. You can, however, create **reference variables of either type. These are normally used to refer to an object** of a subclass of the abstract type or to an object of a class that implements the interface, respectively.
- Both can inherit: the abstract class from another class; the interface only from another interface.

4. Abstraction in class hierarchies

Composition as opposed to Inheritance

- composition, also known as aggregation, occurs when objects of a class contain objects of another class – an object of the class is 'composed' of objects from one or more other classes.
- In the example below, a Date object is composed of two String objects, an ArrayList object.

```
public class Date
{
    private String botanicalName;
    private String areaOfOrigin;
    private ArrayList knownDiseases;
```



Composition of composition

Choosing between composition and inheritance

- Use Inheritance when there is a “IS –A” relationship between classes or simply type of.
- If you just want to reuse code then this means child class inherit many things with no need.
- Don’t use inheritance for polymorphism, as in this case Interfaces are better.
- Use Composition “Has-A” relationship. Big class use facilities offered by another class.
- Composition tends to be more flexible than inheritance.
- Inheritance hierarchies easier to read than network of compositions.
- Composition tends to be more common.