Unit-3

Packages

- The main feature of OOP is its ability to support the reuse of code:
  - Using the classes (directly)
  - Extending the classes (via inheritance)
  - Extending interfaces
- The features in basic form limited to reusing the classes within a program

- What if we want to reuse your classes in other programs without physically copying them ?

- In Java, this is achieved by using “packages”, a concept similar to “class libraries” in other languages

- Package is a group of classes, interfaces and other packages

Creating and importing a user defined package

1. Pick a name for your package

Ex : 1. mypackage

2. mypackage.util

– java recommends lower case letters to the package names

2. Choose a directory on your hard drive as the root of your class library

- You need a place on your hard drive to store your classes
- I suggest you create a directory such as ***c:\javaclasses***
- This folder becomes the ***root directory*** for your Java packages

3. Create subdirectories within the package root directory for your package name

-- For example, for the package named **mypackage.util**, create a directory named **mypackage** in the **c:\javaclasses**. Then, in the **mypackage** directory, create a directory named **util**. Thus, the complete path to the directory that contains the classes for the **mypackage.util** package is **c:\javaclasses\mypackage\util**

4. Add the root directory for your package to the **classpath** environment variable

- Do not disturb any directories already listed in the classpath

  – For example, suppose your classpsath is already set to this:

  C:\Program Files\Java\jdk1.5.0_05\lib;

  – Then, you modify it to look like this:

  C:\Program Files\Java\jdk1.5.0_05\lib;c:\javaclasses;

5. Add a package statement at the beginning of each source file

- The package statement creates a package with specified name
- All classes declared within that file belong to the specified package
- For example: **package mypackage.util**;
- The package statement must be the first non-comment statement in the file

6. Save the files for any classes you want to be in a particular package in the directory for that package

-- For example, save the files for a class that belongs to the **mypackage.util** package in **c:\javaclasses\mypackage\util**

Ex:

```
package mypackage.util;
public class Sum
{
    public int sumInt(int a[])
    {

        int s=0;

        for(int i=0;i<a.length;i++)
        {
                s = s+a[i];
        }

        return s;
    }
}
```

```
import mypackage.util.Sum;
class PackageDemo
{
    public static void main( String args[])
    {
        int x[] = {1,2,3,4,5};
        Sum s = new Sum();
        System.out.println(s.sumInt(x));
    }
}
```

Note: This file can be compiled and executed from any place

- In general, a Java source file can contain any (or all) of the following four internal parts:

  - A single package statement (optional).
  - Any number of import statements (optional).
  - A single public class declaration (required).
  - Any number of classes private to the package (optional).

Accessing Classes from Packages

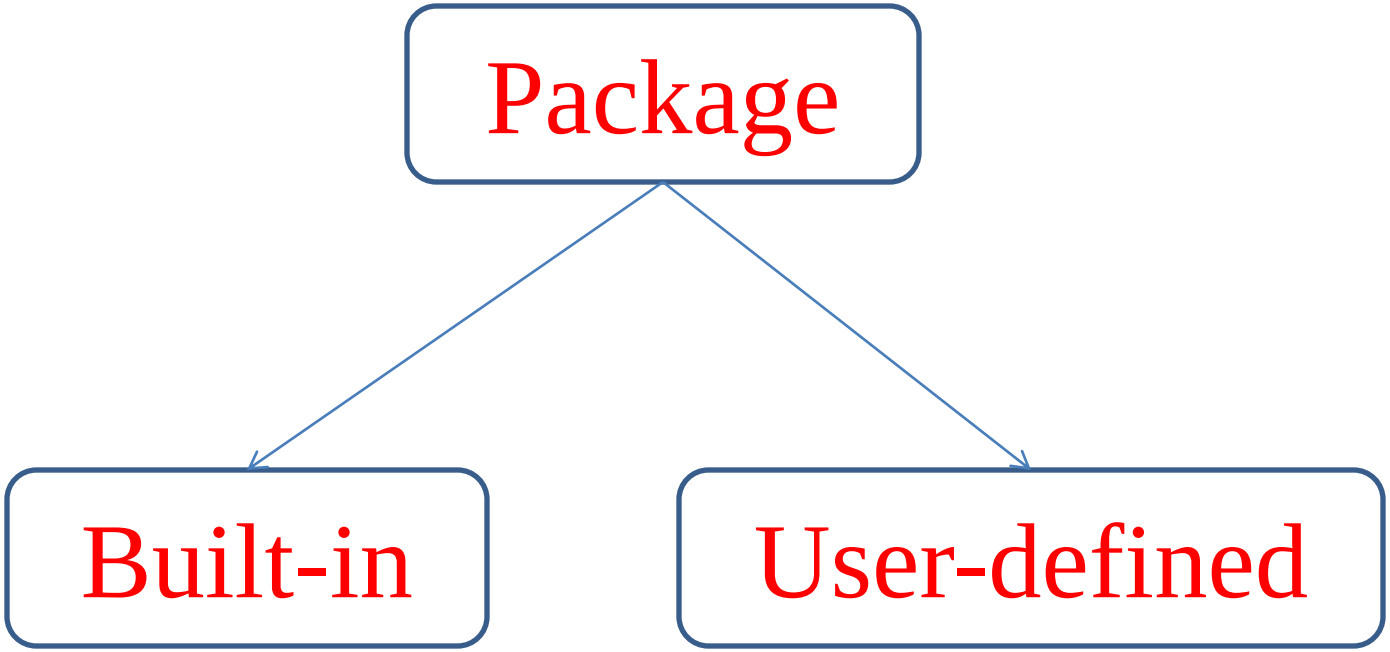
- There are two ways of accessing the classes stored in packages:

1. Using fully qualified class name
   - java.lang.Math.sqrt(x);

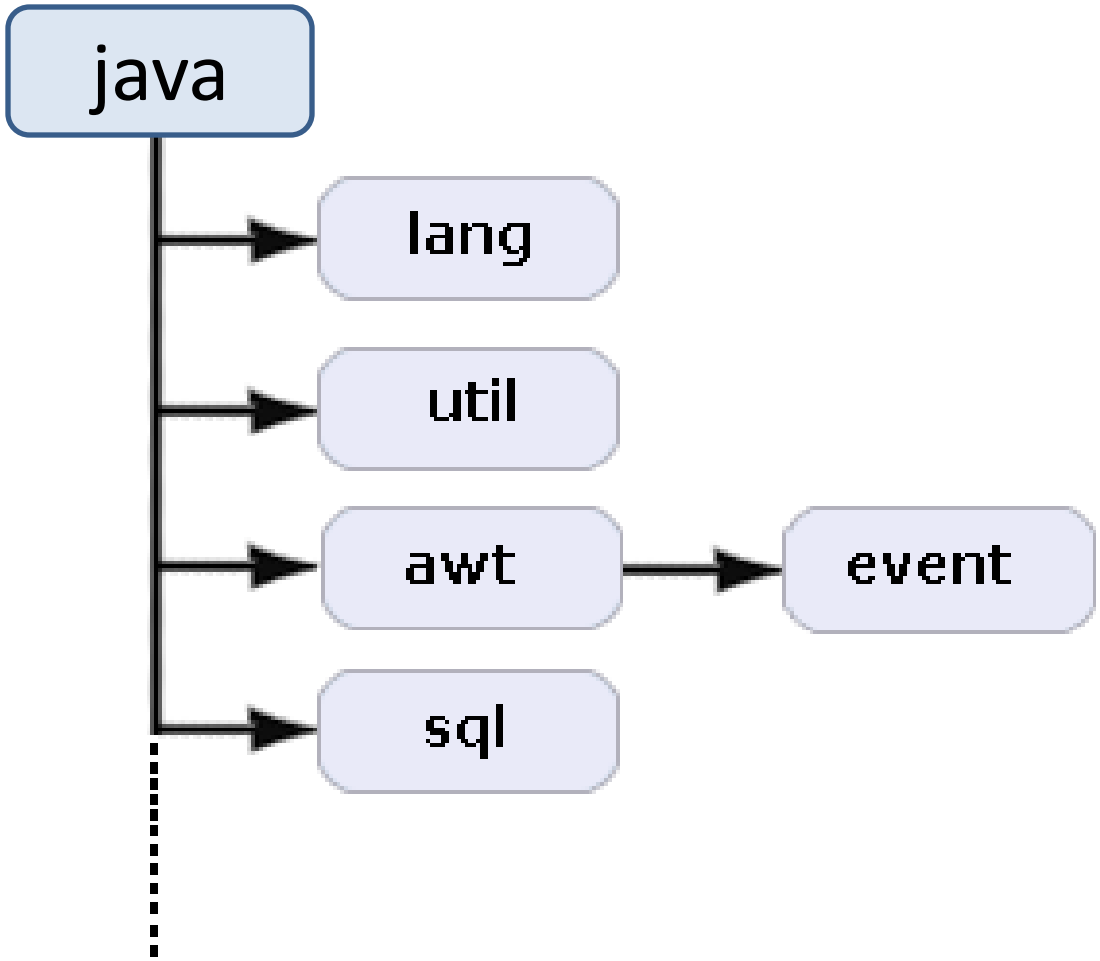
2. Import package and use class name directly
   - import java.lang.Math;
   - Math.sqrt(x);

- Selected or all classes in packages can be imported:
  - import package.ClassName;
  - import package.*;

Types of Packages



Built-In Packages



Built-in examples

- import java.util.Scanner;

  --java.util package contains Scanner class (has methods nextInt(), next(),…)

- import java.lang.Math;

  --java.lang package contains Math class (has methods sqrt(), floor(), …)

User-defined examples

- import mypackage.util.Sum;

  --user defined package mypackage.util contains Sum class

Access specifiers

	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Accessing Classes in the same package (under same directory)

-Assume all files are stored in util package

```
class Add
{
   int addition(int a,int b)
   {
      return a+b;
   }
}
```

-Save file with Add.java

-Accessing Add.class in the following program:

```
class ExDemo
{
  public static void main(String args[])
  {
    Add ad = new Add();
    System.out.println("Sum is "+ad.addition(100,200));
  }
}
```

-Save file with ExDemo.java

-The output of ExDemo.java is 300

Note: access specifier for class and method in Add.java is default

Accessing Classes from other Packages

default access modifier

```
package abcpackage;

public class Addition {
   /* Since we didn't mention any access modifier here, it would
    * be considered as default.
    */
   int addTwoNumbers(int a, int b){
         return a+b;
   }
}
```

-Assume the following file is stored in D:\src folder

```
/* We are importing the abcpackage
 * but still we will get error because the
 * class we are trying to use has default access
 * modifier.
 */
import abcpackage.*;
public class Test {
   public static void main(String args[]){
        Addition obj = new Addition();
        /* It will throw error because we are trying to access
         * the default method in another package
         */
        obj.addTwoNumbers(10, 21);
   }
}
```

protected access modifier

```
package abcpackage;
public class Addition {

   protected int addTwoNumbers(int a, int b){
        return a+b;
   }
}
```

-Assume the following file is stored in D:\src folder

```
import abcpackage.*;
class Test extends Addition{
   public static void main(String args[]){
        Test obj = new Test();
        System.out.println(obj.addTwoNumbers(11, 22));
   }
}
```

public access modifier

```
package abcpackage;

public class Addition {

   public int addTwoNumbers(int a, int b){
         return a+b;
   }
}
```

-Assume the following file is stored in D:\src folder

```
import abcpackage.*;
class Test{
   public static void main(String args[]){
      Addition obj = new Addition();
      System.out.println(obj.addTwoNumbers(100, 1));
   }
}
```

Note:

We can declare class (interface) as either public or default only.

I/O Programming

Introduction

- So far we have used variables and arrays for storing data inside a program. This approach poses the following limitations:
  - The data is lost when the program terminates.
  - It is difficult to handle large volumes of data.
- We can overcome this problem by storing data on secondary storage devices such as hard disks.

C Input/Output Revision

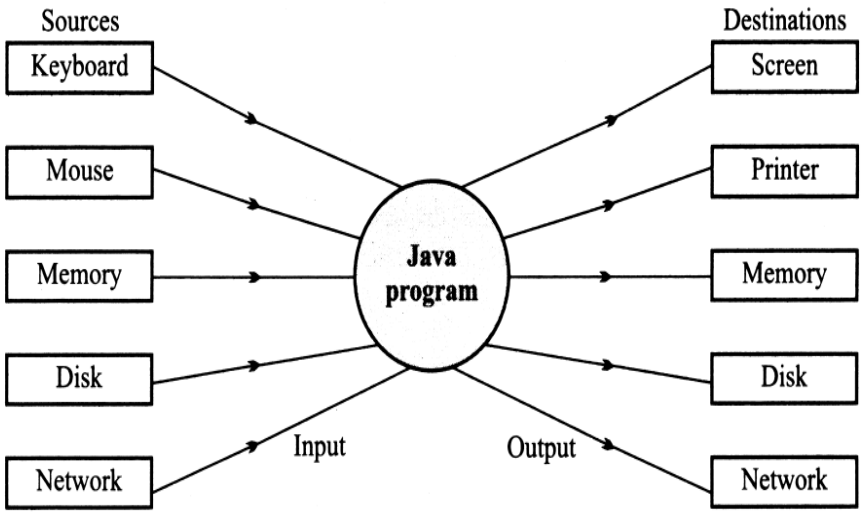
```
FILE *fp;

fp = fopen(“file.txt”, “w”);
fscanf(fp, ……);
fpintf(fp, …..);
fread(……., fp);
fwrite(……….., fp);
```

I/O and Data Movement

- The flow of data into a program (input) may come from different devices such as keyboard, mouse, memory, disk, network, or another program.

- The flow of data out of a program (output) may go to the screen, printer, memory, disk, network, another program.

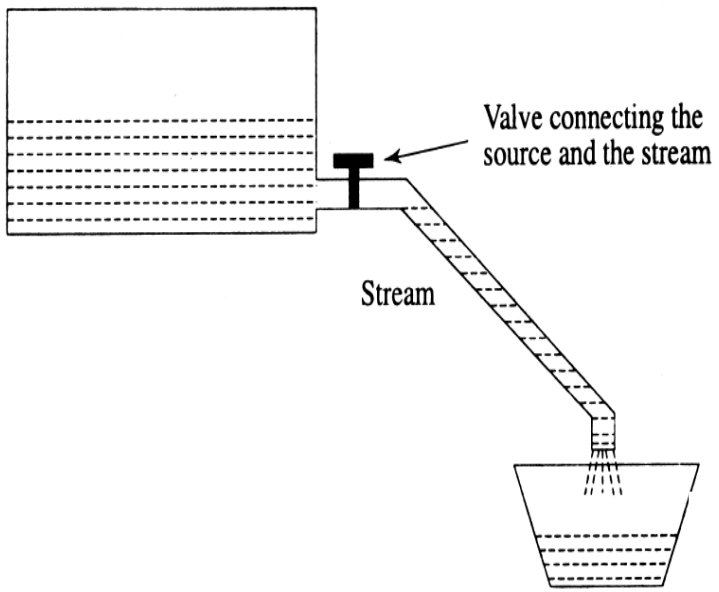


Relationship of Java program with I/O devices

Streams

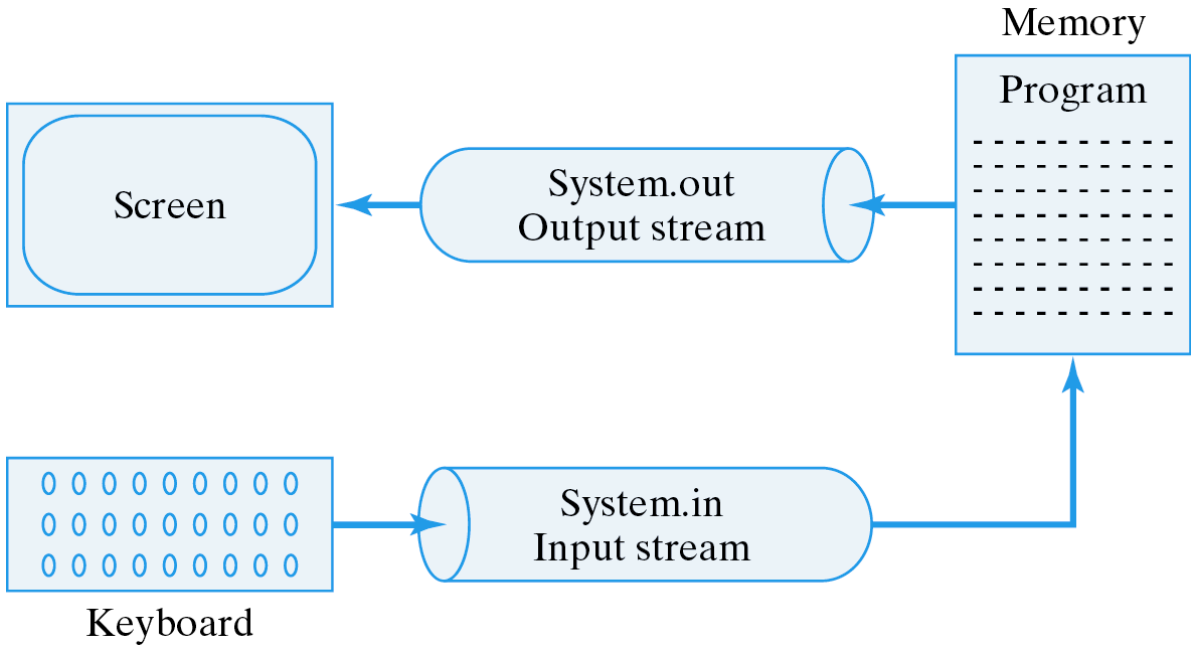
- Java uses the concept of Streams to represent the ordered sequence of data.

- A Stream is a path along which data flows (like a river or pipe along which water flows).



Conceptual view of a stream

- A *stream, in java,* is an object that reads data from a source (keyboard, mouse, memory, disk, network, or another program.) or writes data to a source (screen, printer, memory, disk, network, another program).



Java Stream Classes

- Input/Output related stream classes are defined in java.io package.

- Java i/o classes are categorised into two types:
  - *Byte streams*
  - *Character Streams*

Streams

Byte Streams	Character streams
Operated on 8 bit (1 byte) data.	Operates on 16-bit (2 byte) unicode characters.
Input streams/Output streams	Readers/ Writers

Class	Description
FileInputStream	Provides methods for reading bytes from a file
FileOutputStream	Provides methods for writing bytes to a file
DataInputStream	Provides methods for reading Java's primitive data types
DataOutputStream	Provides methods for writing Java's primitive data types
FileReader	Provides methods for reading characters from a file
BufferedReader	Provides buffering for character input streams

FileInputStream

- The **FileInputStream** class creates a **Stream** that can be used to read bytes from a file.

- Its two most common constructors are shown below:

  FileInputStream(String *filepath)*

  FileInputStream(File *fileObj)*

- Here***, filepath*** *is the path of a file***,** and *fileObj is a* ***File*** *object that describes the file.*
- *Ex:* FileInputStream("d:/in.txt"*)*

  FileOutputStream("out.txt");

Methods in **FileInputStream**

Method	Description
int read( )	Returns an integer representation of the next available byte of input. **–1** is returned when the end of the file is encountered.
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Further read attempts will generate an **IOException.**
long skip(long *numBytes)*	*Ignores (that is, skips) numBytes bytes of input,* returning the number of bytes actually ignored.

FileOutputStream

- **FileOutputStream** creates a **Stream** that can be used to write bytes to a file.
- Its commonly used constructors are shown below:

FileOutputStream(String *filePath)*

FileOutputStream(File *fileObj)*

FileOutputStream(String *filePath*, *boolean append)*

- Here, *filePath* is the path of a file, and *fileObj* is a File object that describes the file. If append is true, the file is opened in append mode**.**

Methods in **FileOutputStream**

Method	Description
void write(int *b)*	*Writes a single byte to an output stream.*
void close( )	Closes the output stream. Further write attempts will generate an **IOException.**

Ex:

```
import java.io.*;
public class FileInputOutputExample
{
    public static void main(String[] args) throws Exception
    {
            FileInputStream is = new FileInputStream("in.txt");
           FileOutputStream os = new FileOutputStream("out.txt");
            int c;
                    while ((c = is.read()) != -1)
                    {
                            System.out.print((char) c);
                           os.write(c);
                    }
         is.close();
        os.close();
    }
}
```

in.txt
abcdef

DataOutputStream

- DataOutputStream takes an OutputStream (Ex. FileOutputStream) and uses this to do higher-level i/o, such as writeInt, writeDouble, etc
- Constructor
  - DataOutputStream(OutputStream out)
- Methods
  - public void writeBoolean(boolean v) throws IOException
  - public void writeByte(int b) throws IOException
  - public void writeShort(int s) throws IOException
  - public void writeChar(int c) throws IOException
  - public void writeInt(int i) throws IOException
  - public void writeLong(long l) throws IOException
  - public void writeFloat(float f) throws IOException
  - public void writeDouble(double d) throws IOException

DataInputStream

- DataInputStream takes an InputStream (Ex. FileInputStream) and uses this to do higher-level i/o, such as readInt, readDouble, etc
- Constructor
  - DataInputStream(InputStream in)
- Methods
  - public boolean readBoolean(boolean v) throws IOException
  - public byte readByte(int b) throws IOException
  - public short readShort(int s) throws IOException
  - public char readChar(int c) throws IOException
  - public int readInt(int i) throws IOException
  - public long readLong(long l) throws IOException
  - public float readFloat(float f) throws IOException
  - public double readDouble(double d) throws IOException

Ex:

```
import java.io.*;
public class DataOutStream_And_DataInputStream_Example
{       public static void main(String[] args) throws IOException
        {
                DataOutputStream  dos= new  DataOutputStream( new FileOutputStream("data.txt")  );

                dos.writeInt(123);
                dos.writeFloat(123.45F);
                dos.writeLong(789);
                dos.close();

                DataInputStream   dis= new  DataInputStream( new  FileInputStream("data.txt") ) ;

                int  a= dis.readInt();
                float  b = dis.readFloat();
                long c = dis.readLong();
                dis.close();

                System.out.println("a = " + a);
                System.out.println("b = " + b);
                System.out.println("c = " + c);
    }
}
```

Ex:

```
import java.io.*;
public class KeyboardReading
{
  public static void main(String args[]) throws IOException
  {
    DataInputStream dis = new DataInputStream(System.in);
    DataOutputStream dos = new DataOutputStream(System.out);
    System.out.println("Enter name: ");
    String str1 = dis.readLine();
    System.out.println("Enter an integer number: ");
    String str2 = dis.readLine();
    int x = Integer.parseInt(str2);
    System.out.println("Enter a double value: ");
    String str3 = dis.readLine();
    double y = Double.parseDouble(str3);
    //System.out.println("Name:"+str1+","+"Integer:"+x+","+"Double:"+y);
    dos.writeBytes("Name:"+str1+","+"Integer:"+x+","+"Double:"+y);
    dis.close();
    dos.close();
  }
}
```

FileReader

- The FileReader class creates a Reader that can be used to read characters from a file.

- Its two most commonly used constructors are shown below:
  - FileReader(String *filePath)*
  - FileReader(File *fileObj)*

Ex:

```
import java.io.*;
public class FileReaderExample
{
    public static void main(String[] args) throws Exception
    {
        FileReader fis = new FileReader("in.txt");

        int c;
            while ((c = fis.read()) != -1)
            {
                System.out.print( (char)c );

            }
        fis.close();

    }
}
```

BufferedReader

- It is a character based input stream class.
- It reads more characters than initially needed and store them in a buffer.
- So when the buffered reader's read() method is called, the data is read from the buffer rather than from the file.
- When the buffer is empty, the buffered stream refills the buffer.
- It improves the performance of I/O
- One long disk access takes less time than many smaller ones.
- It reads large amount of data at a time into the buffer.

- BufferedReader Constructors
- It has two constructors
  - BufferedReader(Reader *inputstream*)
  - BufferedReader(Reader *inputstream*, int *bufSize*)

Ex: BufferedReader with a FileReader

```
import java.io.*;
public class BufferedReaderExample
{
    public static void main(String[] args) throws Exception
    {
        FileReader fis = new FileReader("in.txt");
        BufferedReader br = new BufferedReader(fis);
        int c;
                while ((c = br.read()) != -1)
                {
                        System.out.print( (char)c );
                }
        fis.close();
    }
}
```

Example (BufferedReader)

```
import java.io.*;
public class KeyboardReading2
{
  public static void main(String args[]) throws IOException
  {
    InputStreamReader ir = new InputStreamReader(System.in);
    BufferedReader dis = new BufferedReader(ir);
    System.out.println("Enter name: ");
    String str1 = dis.readLine();
    System.out.println("Enter an integer number: ");
    String str2 = dis.readLine();
    int x = Integer.parseInt(str2);
    System.out.println("Enter a double value: ");
    String str3 = dis.readLine();
    double y = Double.parseDouble(str3);
    System.out.println("Name:"+str1+","+"Integer:"+x+","+"Double:"+y);
    dis.close();
  }
}
```

Collections

- A **collection** is a group of objects

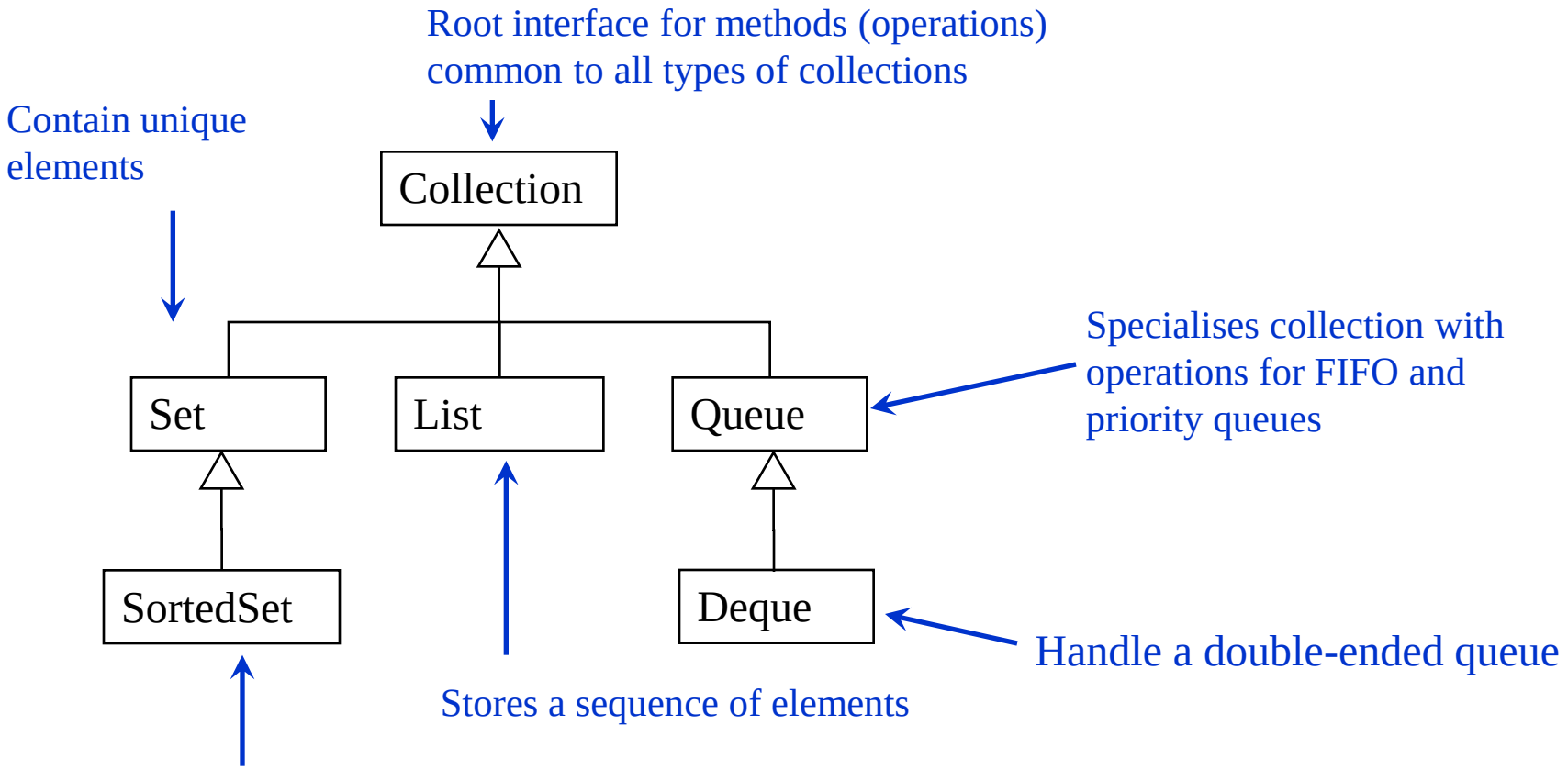
- The classes and interfaces of the **collections framework** are in package **java.util**

Collections frame work contains the following

- **Interfaces**
- **Implementations -**These are the classes
- **Algorithms -**These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces

Interfaces

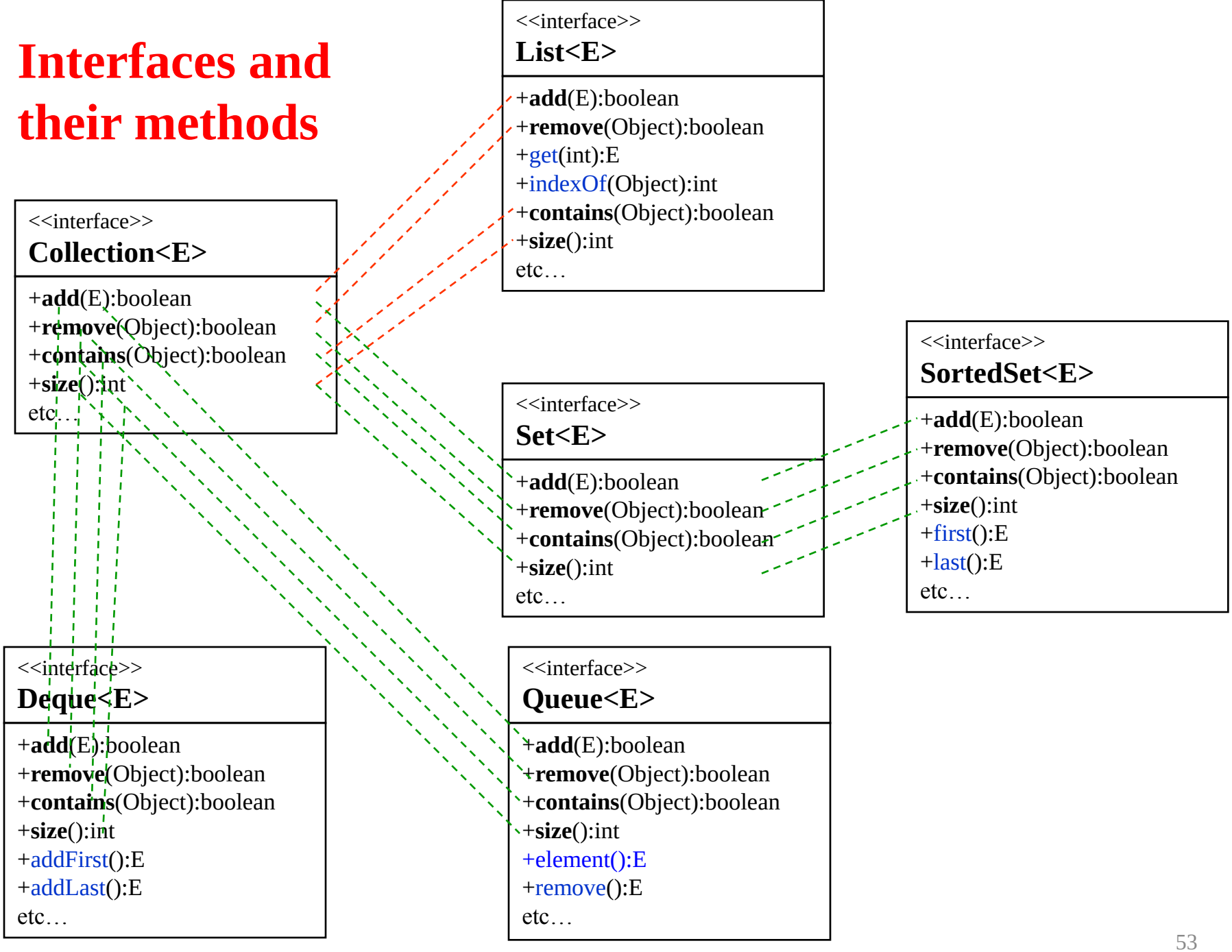
Generalisation



Specialisation

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy
List	Extends **Collection** to handle sequences (lists of objects)
Set	Extends **Collection** to handle sets, which must contain unique elements
Queue	Extends **collection** to handle special types of lists in which elements are removed only from the head
Deque	Extends **Queue** to handle a double-ended queue
SortedSet	Extends **Set** to handle sorted sets

Interfaces and their methods



The Collection Interface

- The **Collection** interface is the foundation upon which the collections framework is built

- It must be implemented by any class that defines a collection

- **Collection** is a generic interface that has this declaration:

  interface Collection<E>

  here E specifies the type of objects that the collection will hold

The methods defined by Collection

Method	Description
boolean add(E obj)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection, and the collection does not allow duplicates
Boolean remove(Object obj)	Removes one instance of *obj* from collection . Returns true if the element was removed. Otherwise returns false
void clear( )	Removes all elements from the invoking collection
Boolean isEmpty()	Returns true if the collection is empty. Otherwise, returns false
int size()	Returns the number of elements held in the collection

The List Interface

- The **List** interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements
- Elements can be inserted or accessed by their position in the list, using a zero-based index
- A list may contain duplicate elements
- **List** is a generic interface that has this declaration:

  interface List<E>

  here E specifies the type of objects that the list will hold

The methods defined by List

Method	Description
void add(int index, E obj)	Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten
E get(int *index*)	Returns the object stored at the specified index within the invoking collection
int indexOf(Object *obj*)	Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned
int lastIndexOf(Object *obj*)	Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned
E remove(int *index*)	Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one

The Set Interface

- The **Set** extends **Collection** and declares the behavior of a collection that does not allow duplicate elements
- Therefore, the **add( )** method returns **false** if an attempt is made to add duplicate elements to a set.
- It does not define any additional methods of its own
- **Set** is a generic interface that has this declaration:

  interface Set<E>

  here E specifies the type of objects that the Set will hold

The SortedSet Interface

- The **SortedSet** interface extends **Set** and declares the behavior of a set sorted in ascending order
- **SortedSet** is a generic interface that has this declaration:

interface SortedSet<E>

here E specifies the type of objects that the set will hold

The methods defined by SortedSet

Method	Description
E first()	Returns the first element in the invoking sorted set
E last()	Returns the last element in the invoking sorted set
SortedSet<E> subSet(E start, E end)	Returns SortedSet that includes those elements between start and end. Elements in the returned collection are also referenced by the invoking object

The queue interface

- The queue interface extends Collection and declares the behavior of a queue, which is often first-in, first-out list
- Queue is a generic interface that has this declaration:

  interface Queue<E>

  here E specifies the type of objects that the queue will hold

Methods defined by a queue

Method	Description
E element()	Returns the element at the head of the queue. The element is not removed. It throws **NoSuchElementException** if the queue is empty
E remove()	Removes the element at the head of the queue and returns that element. It throws **NoSuchElementException** if the queue is empty
E peek()	Returns the element at the head of the queue. It returns null if the queue is empty. The element is not removed
E poll()	Returns the element at the head of the queue. The element is removed. It returns null if the queue is empty

The deque interface

- It was added by Java SE 6
- It extends **queue** and declares the behavior of a double ended queue
- Double ended queue can function as first-in, first-out queues or as last-in, first-out stacks

Methods defined by deque

Method	Description
void addFisrt(E obj)	Adds obj to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space
void addLast(E obj)	Adds obj to the tail of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space
E getFirst()	Returns the first element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty
E getLast()	Returns the last element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty
E removeFirst()	Returns and removes the first element. It throws **NoSuchElementException** if the deque is empty
E removeLast()	Returns and removes the last element. It throws **NoSuchElementException** if the deque is empty

The Collection Classes

- Collection classes are classes that implement collection interfaces
- Some of the classes provide full implementations that can be used as-it-is
- Others are abstract classes

Class	description
AbstractCollection	Implements most of the **Collection** interface
AbstractList	Extends **AbstractCollection** and implements most of the **List** interface
AbstractSet	Extends **AbstractCollection** and implements most of the **Set** interface
AbstractQueue	Extends **AbstractCollection** and implements most of the **Queue** interface
AbstractSequentialList	Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements
LinkedList	Implements a linked list by extending **AbstractSequentialList**
ArrayList	Implements a dynamic array by extending **AbstractList**

The ArrayList Class

- The **ArrayList** class extends **AbstractList** and implements the **List** interface
- Arraylist is a generic class that has this declaration:

  class ArrayList<E>

  here E specifies type of objects that the list will hold
- **ArrayList** has the constructors shown here:

  ArrayList( )

  ArrayList(Collection *c*)

  ArrayList(int *capacity*)

Ex:-

```
// Demonstrate ArrayList.
import java.util.*;
class ArrayListDemo {
    public static void main(String args[]) {
ArrayList<String> al = new ArrayList<String>();
    System.out.println("Initial size of al: " + al.size());
    al.add("C");
    al.add("A");
    al.add("E");
    al.add("B");
    al.add("D");
    al.add("F");
    al.add(1, "A2");
    System.out.println("Size of al after additions: " + al.size());
    System.out.println("Contents of al: " + al);
    al.remove("F");
    al.remove(2);
    System.out.println("Size of al after deletions: " + al.size());
    System.out.println("Contents of al: " + al);
  }
}
```

Output:

```
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

The LinkedList Class

- The **LinkedList** class extends **AbstractSequentialList** and implements the **List , Deque,** and **Queue** interfaces
- LinkedList Class is a generic class that has this declaration:

  class LinkedList<E>

  here E specifies type of objects that the list will hold
- It provides a linked-list data structure
- It has the two constructors, shown here:

  LinkedList( )

  LinkedList(Collection *c*)
- The first constructor builds an empty linked list. The second constructor builds a linked list that is initialized with the elements of the collection *c*

Ex:-

```
// Demonstrate LinkedList.
import java.util.*;
class LinkedListDemo {
public static void main(String args[]) {
// create a linked list
LinkedList<String> llist = new LinkedList<String>();
llist.add("F");
llist.add("B");
llist.add("D");
llist.add("E");
llist.add("C");
llist.addLast("Z");
llist.addFirst("A");
llist.add(1, "A2");
System.out.println("Original contents of llist: " + llist);
// remove elements from the linked list
llist.remove("F");
llist.remove(2);
System.out.println("Contents of llist after deletion: " + llist);
```

```
// remove first and last elements
llist.removeFirst();
llist.removeLast();
System.out.println("llist after deleting first and last: "+ llist);
}
}
```

Output:

```
Original contents of llist: [A, A2, F, B, D, E, C, Z]
Contents of llist after deletion: [A, A2, D, E, C, Z]
llist after deleting first and last: [A2, D, E, C]
```

The Collection Algorithms

- The Collections Framework defines several algorithms
- These algorithms are defined as static methods within the **Collections** class

Method	Description
static int binarySearch(List *list*, Object *value*)	Searches for *value* in *list*. The list must be sorted. Returns the position of *value* in *list*,or−1 if *Value* is not found
static void sort(List *list*)	Sorts the elements of *list* as determined by their natural Ordering
static Object max(Collection *c*)	Returns the maximum element in *c* as determined by natural ordering. The collection need not be sorted

Method	Description
static Object min(Collection *c*)	Returns the minimum element in *c* as determined by natural ordering
static void reverse(List *list*)	Reverses the sequence in *list*

Example: (binary search)

```
import java.util.*;

public class BinarySearchDemo {
   public static void main(String args[]) {

   ArrayList<String> arlst=new ArrayList<String>();

    arlst.add("PROVIDES");
    arlst.add("QUALITY");
    arlst.add("TP");
    arlst.add("TUTORIALS");

  int index=Collections.binarySearch(arlst, "QUALITY");

    System.out.println("'QUALITY' is available at index: "+index);
    }
}
```

Example: (sort)

```
import java.util.*;

public class SortDemo {
   public static void main(String args[]) {

   ArrayList<String> arlst=new ArrayList<String>();

   arlst.add("QUALITY");
   arlst.add("PROVIDES");
   arlst.add("TUTORIALS");
   arlst.add("TP");

   System.out.println("List value before: "+arlst);

   Collections.sort(arlst);

   System.out.println("List value after sort: "+arlst);
   }
}
```

Example: (max, min and reverse)

```
import java.util.*;
public class MaxMinRev {
   public static void main(String args[]) {
   ArrayList<Integer> arlst=new ArrayList<Integer>();

   arlst.add(10);
   arlst.add(20);
   arlst.add(30);
   arlst.add(40);
   arlst.add(50);

   System.out.println("List values: "+arlst);
   System.out.println("Minimum is :"+Collections.min(arlst));
   System.out.println(“Maximum is :"+Collections.max(arlst));

   Collections.reverse(arlst);

   System.out.println("List values after reverse: "+arlst);
   }
}
```