Java Technologies Week 3

Prepared for: DDU CE Semester 4

Prepared by: Prof. Niyati J. Buch

Topics

- **Package**
  - Organizing Classes and Interfaces in Packages
  - Package as Access Protection
  - Defining Package
  - CLASSPATH Setting for Packages
  - Making JAR Files for Library Packages
  - Import and Static Import
  - Naming Convention for Packages

- **Input/Output Operation in Java**
  - Streams and the new I/O Capabilities
  - Understanding Streams
  - The Classes for Input and Output
  - The Standard Streams
  - Working with File Object, File I/O Basics
  - Reading and Writing to Files
  - Buffer and Buffer Management,
  - Read/Write Operations with File
  - Channel
  - Serializing Objects

Package

- Class name must be unique to avoid name collisions.
- Java provides a mechanism called **package** for partitioning the class name space.
- The package is both a naming and a visibility control mechanism.
- You can define classes inside a package that are not accessible by code outside that package.
- You can also define class members that are accessible only to other members of the same package.
- So related classes can have knowledge of each other and for rest of the world this knowledge is hidden.

Defining a Package

- To create a package, write package statement as first statement of the Java source file.
- Any classes declared within that file will belong to the specified package.
- The package statement defines a namespace in which classes are stored.
- If the package statement is omitted, the class names are put into the default package, which has no name.
- **package mypackage;**
  - this statement will create a package of the name mypackage.

- Java uses file system directories to store packages.
- For example, the .class files for any classes you declare to be part of **mypackage** must be stored in a directory called **mypackage**.
- It is **case sensitive** and directory name must exactly match the package name.

- You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a dot.
- A package hierarchy must be reflected in the file system of your Java development system.
- For example, a package declared as package java.awt.image; needs to be stored in java\awt\image in a Windows environment.

- You cannot rename a package without renaming the directory in which the classes are stored.
- Package names are written in all lower case to avoid conflict with the names of classes or Interfaces.

How does Java runtime system know where to look for packages??

1. By default, Java runtime system uses the current working directory as its starting point. i.e. if your package is in a sub directory of the current directory, it will be found.
2. Specify a directory path/s by setting the CLASSPATH environment variable.
3. Use the –classpath option with java and javac to specify the path to your class.

import for reuse

- A class can use all classes from its own package and all public classes from other packages.
- You can access the public classes in another package in two ways.
  1. The first is simply to add the full package name in front of every class name. For example: java.util.Scanner sc = new java.util.Scanner();
  2. The simpler, and more common, approach is to use the import statement.
- The point of the **import statement** is to give you a shorthand to refer to the classes in the package. Once you use import, you no longer have to give the classes their full names.
- You can import a specific class or the whole package.

  import java.util.*; or import java.util.Date;
- You place import statements at the top of your source files (but below any package statements).

Which HelloWorld will be considered??

```
import mypack.first.HelloWorld;
import mypackage.pack1.HelloWorld;

class Test{
    public static void main(String []args){
        HelloWorld obj = new HelloWorld();
    }
}
```

If the import statements have two classes with same name from different packages, then full package name must be used each time the class name is used.

```
import mypack.first.HelloWorld;
import mypackage.pack1.HelloWorld;


class Test{
     public static void main(String []args){
          mypack.first.HelloWorld obj = new mypack.first.HelloWorld();
          mypackage.pack1.HelloWorld obj1 = new mypackage.pack1.HelloWorld();
     }
}
```

static import

- A form of the import statement permits the importing of static methods and fields, not just classes.
- For example, if you add this

  import static java.lang.System.*;

  to the top of your source file, then you can use the static methods and fields of the System class without the class name prefix:

  out.println("Goodbye, World!"); // i.e., System.out

  exit(0); // i.e., System.exit
- You can also import a specific method or field:

  import static java.lang.System.out;

Access Modifiers for access protection

	(non-nested)class	constructor
private	• Not allowed	• Allowed but not used as it allows object creation in same class only. • Object creation is not allowed in different class even in same package.
default (no modifier)	• Can be accessed in same package. • This class can be written in any .java file	• Can create object in same package • Does not allow inheritance into subclass outside package.
protected	• Not allowed	• Can create object in same package • Allows inheritance into subclass outside package • Does not allow object creation outside package
public	• Can be accessed anywhere • This class must be written in .java file with same name as class name	• Allows everything

```
package pack1;
public class MyClass {
    private int pri;
    int def;
    protected int pro;
    public int pub;
    private void pri_method() {
/*
* this method can be accessed only in
    same class. for simplification
* private methods can be written which
    are helper methods
* they cannot be called out the class
    using object.
*/
    }
    void default_method() {
    }
    protected void pro_method() {
    }
    public void pub_method() {
    }

    public static void main(String args[]) {
        MyClass obj = new MyClass();
        obj.pri = 1;
        obj.def = 1;
        obj.pro = 1;
        obj.pub = 1;
        obj.pri_method();
        obj.default_method();
        obj.pro_method();
        obj.pub_method();
    }
}
```

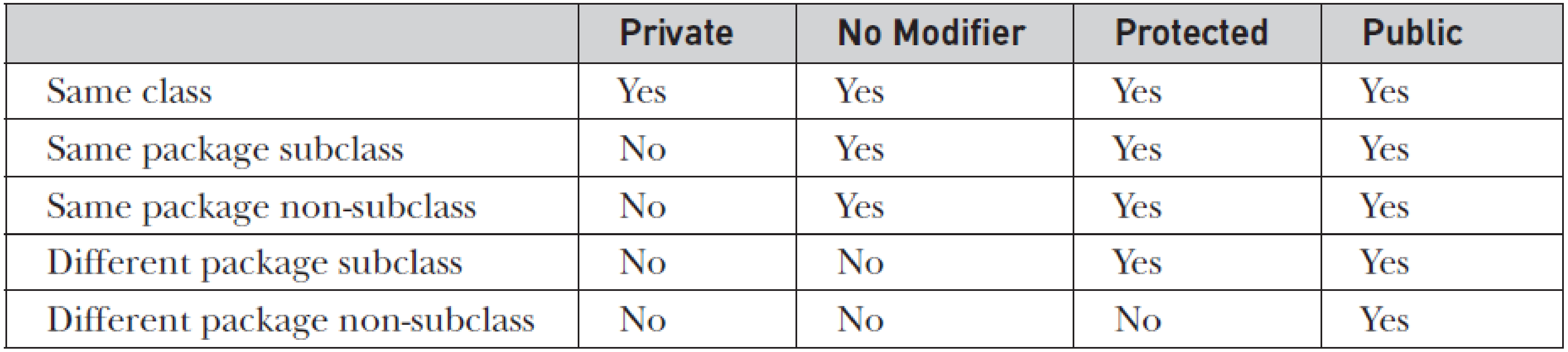
```
package pack1;
public class SamePack {
    public static void main(String args[]){
            MyClass obj = new MyClass();
            //obj.pri=1; only in same class
            obj.def=1;
            obj.pro=1;
            obj.pub=1;
            //obj.pri_method(); only in same class
            obj.default_method();
            obj.pro_method();
            obj.pub_method();
    }
}
```

```
package pack1;
public class SubClassSamePack extends MyClass {
    public static void main(String args[]){
        SubClassSamePack obj = new SubClassSamePack();
        //obj.pri=1; //only in same class
        obj.def=1;
        obj.pro=1;
        obj.pub=1;
        //obj.pri_method(); only in same class
        obj.default_method();
        obj.pro_method();
        obj.pub_method();
    }
}
```

```
package pack2;
public class DiffPack {
    public static void main(String args[]){
        pack1.MyClass obj = new pack1.MyClass();
        //obj.pri=1; only in same class
        //obj.def=1; only in same package
        //obj.pro=1; only in same package or sub class
        obj.pub=1;
        //obj.pri_method(); only in same class
        //obj.default_method(); only in same package
        //obj.pro_method(); only in same package or sub class
        obj.pub_method();
    }
}
```

```
package pack2;
public class SubClassDiffPack extends pack1.MyClass{
    public static void main(String args[]){
            SubClassDiffPack obj= new SubClassDiffPack();
            //obj.pri=1; only in same class
            //obj.def=1; only in same package
            obj.pro=1;
            obj.pub=1;
            //obj.pri_method(); only in same class
            //obj.default_method(); only in same package
            obj.pro_method();
            obj.pub_method();
    }
}
```

Class Member Access



	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

JAR

- JAR stands for Java ARchive.
- It's a file format based on the popular ZIP file format and is used for aggregating many files into one.
- Although JAR can be used as a general archiving tool, the primary motivation for its development was so that Java applets and their requisite components (.class files, images and sounds) can be downloaded to a browser in a single HTTP transaction, rather than opening a new connection for each piece.
- JAR files can be used for tasks such as lossless data compression, archiving, decompression, and archive unpacking.
- To perform basic tasks with JAR files, you use the Java Archive Tool provided as part of the Java Development Kit (JDK).

https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html

Common JAR file operations

jar command options	Use
-c	Creates a new archive file
-v	Generates verbose output. It displays the included or extracted resource on the standard output.
-m	Includes manifest information from the given mf file.
-f	Specifies the archive file name
-x	Extracts files from the archive file

Operation	Command
To run an application packaged as a JAR file (requires the Main-class manifest header)	java -jar app.jar

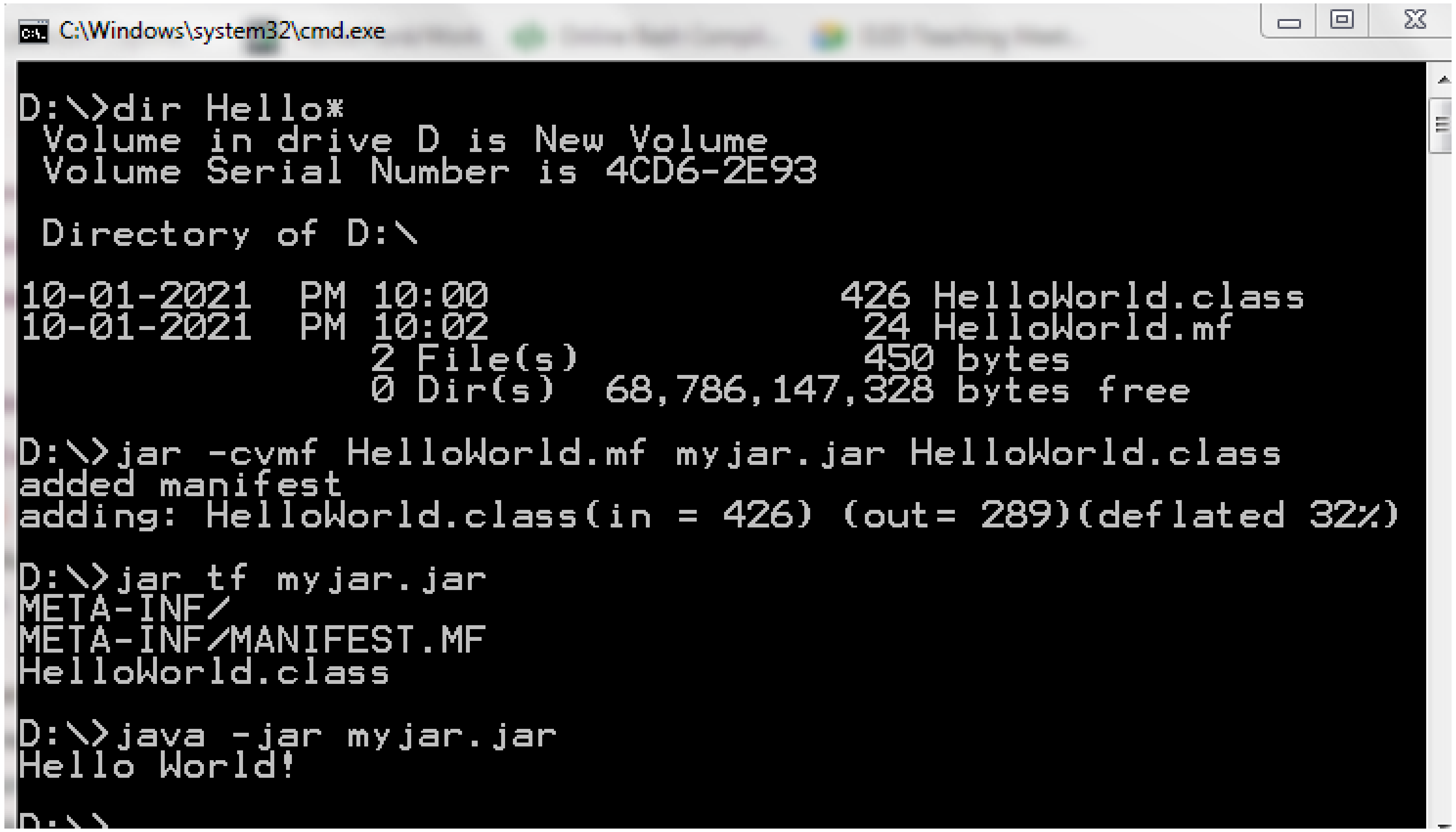
HelloWorld.java

```
class HelloWorld {
            public static void main(String args[]) {
                        System.out.println("Hello World!");

            }
}
```

HelloWorld.mf

```
Main-Class: HelloWorld
```



```
D:\>dir Hello*
 Volume in drive D is New Volume
 Volume Serial Number is 4CD6-2E93

 Directory of D:\

10-01-2021  PM 10:00               426 HelloWorld.class
10-01-2021  PM 10:02                24 HelloWorld.mf
               2 File(s)            450 bytes
               0 Dir(s)  68,786,147,328 bytes free

D:\>jar -cvmf HelloWorld.mf myjar.jar HelloWorld.class
added manifest
adding: HelloWorld.class(in = 426) (out= 289)(deflated 32%)

D:\>jar tf myjar.jar
META-INF/
META-INF/MANIFEST.MF
HelloWorld.class

D:\>java -jar myjar.jar
Hello World!

D:\>
```

To create the manifest file, you need to write: Main-Class, then colon, then space, then classname then enter.

Person.java

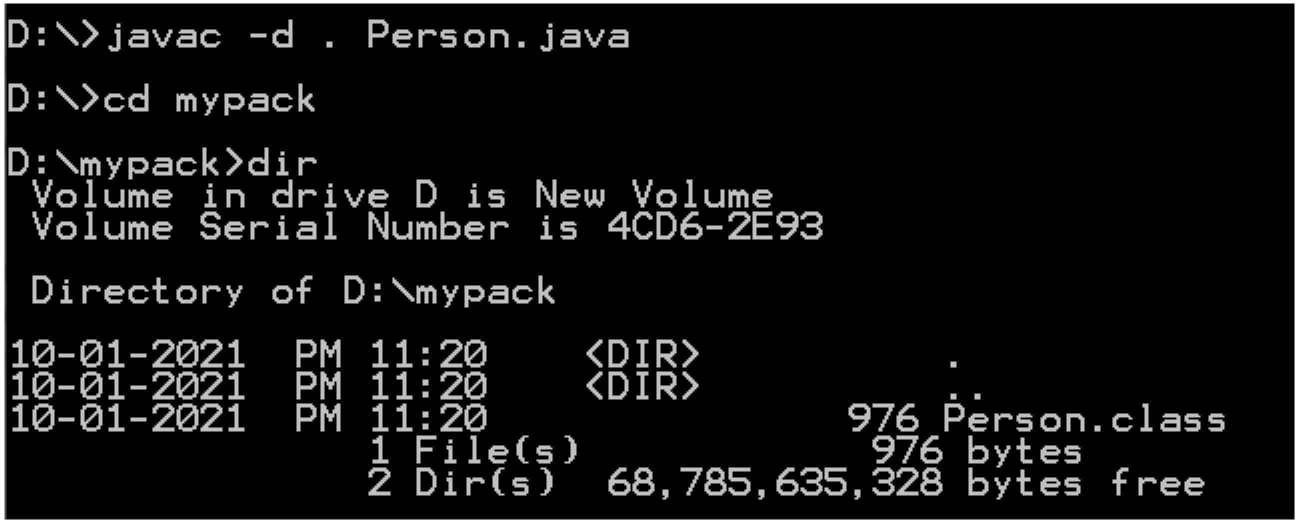
```
package mypack;
public class Person {

    private String name;

    public Person() {
    }
    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Person{" + "name=" + name + '}';
    }
}
```



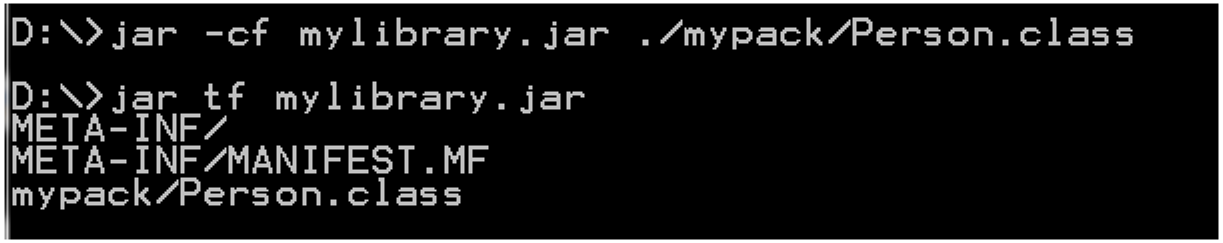
```
D:\>javac -d . Person.java

D:\>cd mypack

D:\mypack>dir
 Volume in drive D is New Volume
 Volume Serial Number is 4CD6-2E93

 Directory of D:\mypack

10-01-2021  PM 11:20    <DIR>          .
10-01-2021  PM 11:20    <DIR>          ..
10-01-2021  PM 11:20               976 Person.class
               1 File(s)            976 bytes
               2 Dir(s)  68,785,635,328 bytes free
```



```
D:\>jar -cf mylibrary.jar ./mypack/Person.class

D:\>jar tf mylibrary.jar
META-INF/
META-INF/MANIFEST.MF
mypack/Person.class
```

File class

- The File class deals directly with files and the file system.
- It is an abstract representation of file and directory pathnames.
- It describes the properties of a file itself.
- A File object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.

Creating File object

- **File(File parent, String child)**
  - Creates a new File instance from a parent abstract pathname and a child pathname string.
- **File(String pathname)**
  - Creates a new File instance by converting the given pathname string into an abstract pathname.
- **File(String parent, String child)**
  - Creates a new File instance from a parent pathname string and a child pathname string.
- **File(URI uri)**
  - Creates a new File instance by converting the given file: URI into an abstract pathname.

Some useful methods

1	boolean canExecute()	10	boolean isAbsolute()
2	boolean canRead()	11	boolean isDirectory()
3	boolean canWrite()	12	boolean isFile()
4	boolean exists()	13	boolean isHidden()
5	File getAbsoluteFile()	14	long length()
6	String getAbsolutePath()	15	String[] list()
7	String getName()	16	File[] listFiles()
8	String getParent()	17	boolean mkdir()
9	File getParentFile()	18	boolean renameTo(File dest)

I/O Streams

- **Byte Streams** handle I/O of raw binary data.
- **Character Streams** handle I/O of character data, automatically handling translation to and from the local character set.
- **Buffered Streams** optimize input and output by reducing the number of calls to the native API.
- **Data Streams** handle binary I/O of primitive data type and String values.
- **Object Streams** handle binary I/O of objects.

Byte Stream

- Programs use *byte streams* to perform input and output of 8-bit bytes.
- All byte stream classes are descended from **InputStream** and **OutputStream**.

```
/*Program to copy a file using FileInputStream and FileOutputStream classes*/
import java.io.*;
public class FileCopyByteStream {
   public static void main(String args[]) throws IOException  {
      FileInputStream in = null;
      FileOutputStream out = null;
      try {
         in = new FileInputStream("f1.txt");
         out = new FileOutputStream("f1_copy.txt");
         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      } finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

```
/*Program to append a file using FileInputStream and FileOutputStream classes*/
import java.io.*;
public class FileCopyByteStream {
    public static void main(String args[]) throws IOException  {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("f1.txt");
            out = new FileOutputStream("f2.txt", true);
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Character Stream

- The Java platform stores character values using Unicode conventions.
- Character stream I/O automatically translates this internal format to and from the local character set.
- All character stream classes are descended from **Reader** and **Writer**.
- As with byte streams, there are character stream classes that specialize in file I/O: FileReader and FileWriter.

```
/*Program to append a file using FileReader and FileWriter classes*/
import java.io.*;
public class FileCopyByteStream {
    public static void main(String args[]) throws IOException  {
        FileReader fr = null;
        FileWriter fw = null;
        try {
            fr = new FileReader("f1.txt");
            fw = new FileWriter("f1_1.txt", true);
            int c;
            while ((c = fr.read()) != -1) {
                fw.write(c);
            }
        } finally {
            if (fr != null) {
                fr.close();
            }
            if (fw != null) {
                fw.close();
            }
        }
    }
}
```

Buffered Byte Streams

- There are four buffered stream classes used to wrap unbuffered streams:
- **BufferedInputStream** and **BufferedOutputStream**
  - create buffered byte streams,
- **BufferedReader** and **BufferedWriter**
  - create buffered character streams.

```
import java.io.*;
public class FileCopyWithBuffer {
    public static void main(String args[]) throws IOException {
        String src = "f1.txt";
        String dest1 = "copyfile1.txt";
        String dest2 = "copy2file1.txt";
        long t1, t2;
        FileInputStream fis = null;
        FileOutputStream fos = null;
        try {
            fis = new FileInputStream(src);
            fos = new FileOutputStream(dest1);
            int i;
            t1 = System.nanoTime();
            while ((i = fis.read()) != -1)
                fos.write(i);
            t2 = System.nanoTime();
            System.out.println(t2 - t1);
        } finally {
            if (fis != null)
                fis.close();
            if (fos != null)
                fos.close();
        }
        BufferedInputStream bis = null;
        BufferedOutputStream bos = null;
        try {
            bis = new BufferedInputStream(new FileInputStream(src));
            bos = new BufferedOutputStream(new FileOutputStream(dest2));
            int i;
            t1 = System.nanoTime();
            while ((i = bis.read()) != -1) {
                bos.write(i);
            }
            t2 = System.nanoTime();
            System.out.println(t2 - t1);
        } finally {
            if (bis != null) {
                bis.close();
            }
            if (bos != null) {
                bos.close();
            }

        }
    }
}
```

Programs to practice

- Merge all the files in a given directory into a single file.

- Split a given file into n parts.

- Hint:

```
byte[] b = new byte[1024];
int x = bis.read(b);
if (x != -1) bos.write(b);
```

Scanner class

- Scanner sc = new Scanner(System.in);
  - This scans from standard input.
- Instead, Scanner class can be used to scan from a file too.

```
import java.io.*;
import java.util.Scanner;
public class ScanWords {
    public static void main(String[] args) throws IOException {
        try (Scanner s = new Scanner(new BufferedReader(new FileReader("f1.txt")))) {
            while (s.hasNext()) {
                System.out.println(s.next());
            }
        }
    }
}
```

Scanning a .csv

```
import java.io.*;
import java.util.Scanner;
public class ScanCSV {
    public static void main(String[] args) throws IOException {
        try ( Scanner s = new Scanner(new BufferedReader(new FileReader("addresses.csv")))) {
            s.useDelimiter(",");
            while (s.hasNext()) {
                System.out.print(s.next() + “ ");
            }
        }
    }
}
```

PrintStream and PrintWriter

- **print** and **println** format individual values in a standard way.
- **format** formats almost any number of values based on a format string, with many options for precise formatting.

```
public class Formatting {
    public static void main(String[] args) {
        int i = 3;
        double r = Math.sqrt(i);
        System.out.format("The square root of %d is %f.%n", i, r);
    }
}
```

```
OUTPUT
The square root of 3 is 1.732051.
```

d formats an integer value as a decimal value.
f formats a floating point value as a decimal value.
n outputs a platform-specific line terminator.

RandomAccessFile

- Instances of this class support **both reading and writing** to a random access file.
- A random access file behaves like a large array of bytes stored in the file system.
- There is a kind of cursor, or index into the implied array, called the *file pointer*; input operations read bytes starting at the file pointer and advance the file pointer past the bytes read.
- The file pointer can be read by the getFilePointer() method and set by the seek method.

```
import java.io.*;
public class RandomAccessFileDemo {
    public static void main(String args[]) throws IOException {
        RandomAccessFile raf = null;
        try {
            String path = "myfile.txt";
            raf = new RandomAccessFile(path, "rw");
            String s[] = {"abc", "def", "ghi"};
            for (String a : s) {
                byte b[] = a.getBytes();
                raf.write(b);
                raf.write(' ');
            }
            raf.seek(0);
            int i;
            while ((i = raf.read()) != -1) {
                System.out.print((char) i);
            }
        } finally {
            if (raf != null)
                raf.close();
        }
    }
}
```

```
OUTPUT
abc def ghi
```

Serialization and Deserialization

- Serializability of a class is enabled by the class implementing the java.io.Serializable interface.
- All subtypes of a serializable class are themselves serializable.
- The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

```
public class Student implements java.io.Serializable{
…
…
}
```

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/Serializable.html

```
Import java.io.*;
public class SerializationDemo {
   public static void main(String args[]) throws IOException, ClassNotFoundException {
      Student s1 = new Student();
      s1.setId(1);
      s1.setName("abc");
      String filename = "StudentRecord.txt";
      FileOutputStream fos = null;
      ObjectOutputStream oos = null;
      try {
         fos = new FileOutputStream(filename);
         oos = new ObjectOutputStream(fos);
         oos.writeObject(s1);
         oos.writeObject(s1);
      } finally {
         if (fos != null) {
            fos.close();
         }
         if (oos != null) {
            oos.close();
         }
      }
```

```
        FileInputStream fis = null;
        ObjectInputStream ois = null;
        try {
            fis = new FileInputStream(filename);
            ois = new ObjectInputStream(fis);
            Object obj;

            while ((obj = ois.readObject()) != null) {
                if (obj.getClass().getSimpleName().equals("Student")) {
                    System.out.println(((Student) obj).getName());
                }
            }
        } catch (EOFException ex) {
        } finally {
            if (fis != null) {
                fis.close();
            }
            if (ois != null) {
                ois.close();
            }
        }
    }
}
```