Programming in Java

Lecture 17: I/O and Applets

Mahesh Kumar
Assistant Professor (Adhoc)

Department of Computer Science
Acharya Narendra Dev College
University of Delhi

Course webpage
[http://www.mkbhandari.com/mkwiki]

Outline

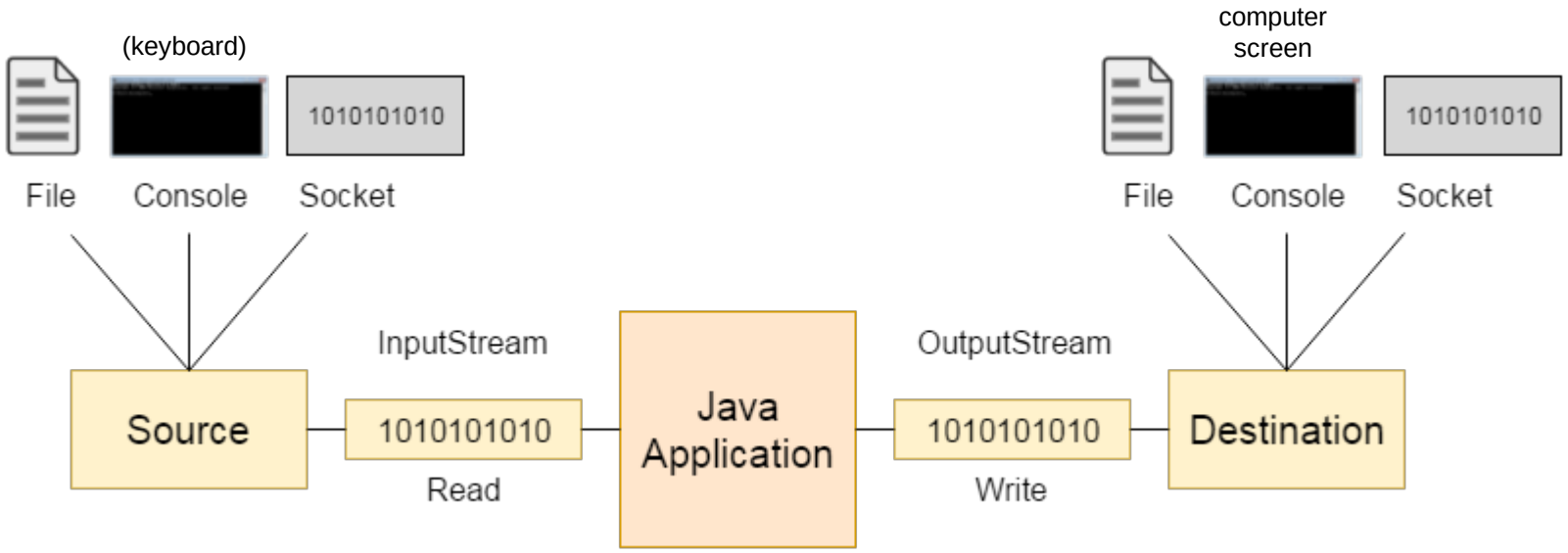
1. I/O Stream Basics
2. Byte Stream and Character Stream
3. The Predefined Stream
4. Reading Console Input
5. Writing Console Output
6. Reading and Writing Files
7. Applet Fundamentals

I/O Basics - Streams

- Java ***I/O*** (Input and Output) is used to *process the input and produce the output*. Java programs perform I/O through ***streams***.
- A ***stream*** is a sequence of data. A ***stream*** is an **abstraction** that either produces or consumes information.
- A ***stream*** is linked to a **physical device** by the Java I/O system.
- All ***streams*** behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to different types of devices.
- An ***input stream*** can abstract many different kinds of input:
  - *Disk file*
  - *Keyboard*
  - *Network socket*
- An ***output stream*** may refer to:
  - *Disk file*
  - *Console (Monitor)*
  - *Network connection*

I/O Basics - Streams

- Java application uses an ***input stream*** to *read data from a source*



- Java application uses an ***output stream*** to *write data to a destination*

[Source: (3)]

I/O Basics - Streams

- ***Streams*** are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network

- The ***java.io*** package contains all the classes required for input and output operations.

- We can perform ***file handling*** in Java by Java I/O API.

- In addition to the stream-based I/O defined in ***java.io***, Java also provides buffer- and channel- based I/O, which is defined in ***java.nio*** and its subpackages.

- Java defines two types of ***streams***:
  - *byte*
  - *character*

Byte Streams and Character Streams

- ***Byte streams***
  - *Provide a convenient means for handling input and output of **bytes*** (8-bits).
  - *Used, for example, when reading or writing binary data.*
  - *Using these you can store characters, videos, audios, images etc.*
  - *Byte-Stream classes are topped by **InputStream** and **OutputStream** classes*
- ***Character streams***
  - *Provide a convenient means for handling input and output of **characters**.*
  - *They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.*
  - *Using these you can read and write text data only.*
  - *Character Stream classes are topped by **Reader** and **Writer** class*
- At the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters.

The Byte Stream Classes

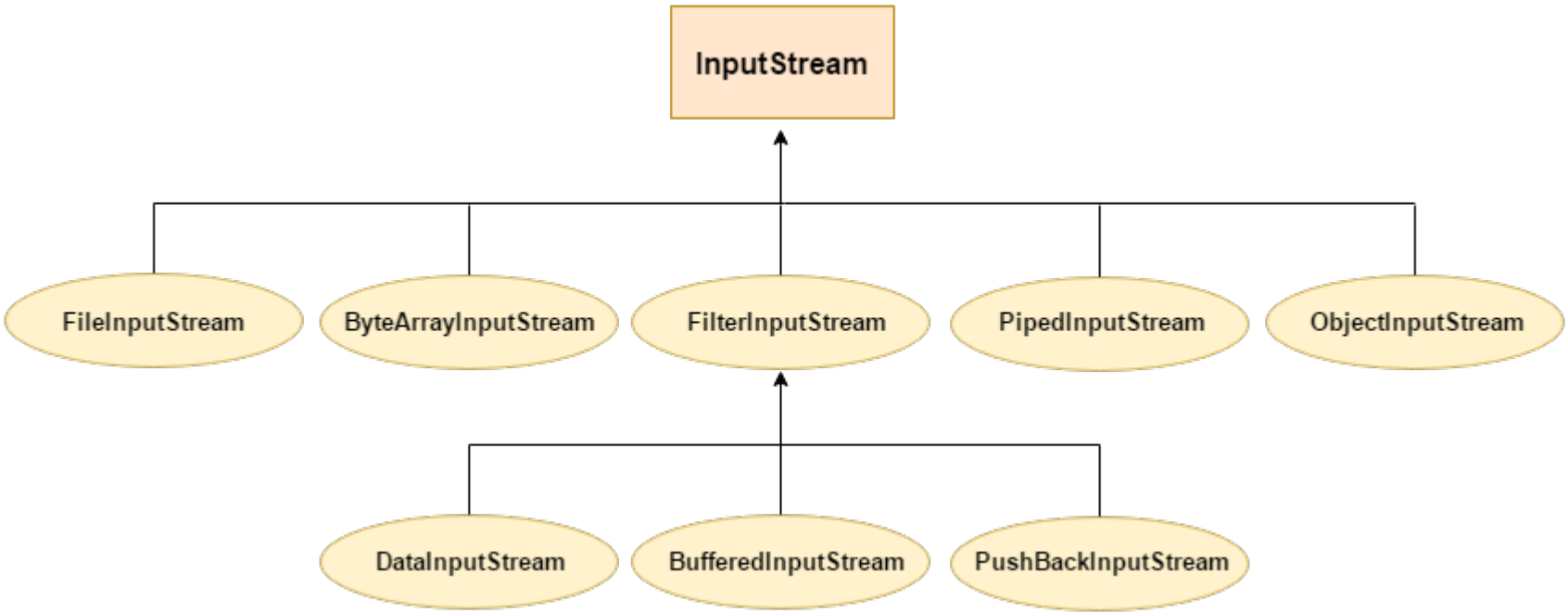
- Byte streams are defined by using two class hierarchies. At the top are two abstract classes:
  - *InputStream*
  - *OutputStream*

- Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers.

- The abstract classes *InputStream* and *OutputStream* define several key methods that the other stream classes implement.

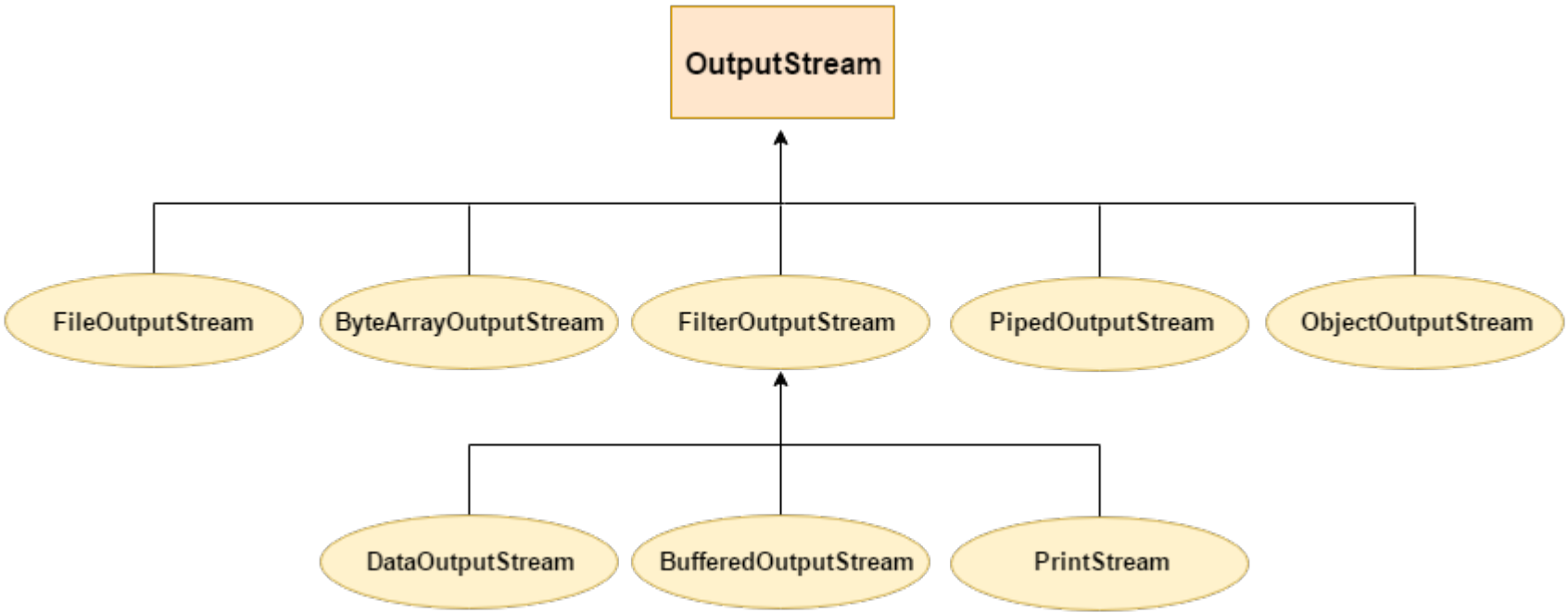
- Two of the most important are *read( )* and *write( )*, which, respectively, read and write bytes of data. Each has a form that is abstract and must be overridden by derived stream classes.

The Byte Stream Classes



[Source: (3)]

The Byte Stream Classes



[Source: (3)]

The Byte Stream Classes

- The byte stream classes in ***java.io***

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

The Character Stream Classes

- Character streams are defined by using two class hierarchies. At the top are two abstract classes:
  - *Reader*
  - *Writer*
- These abstract classes handle Unicode character streams.
- The abstract classes *Reader* and *Writer* define several key methods that the other stream classes implement.
- Two of the most important methods are *read( )* and *write( )*, which read and write characters of data, respectively. Each has a form that is abstract and must be overridden by derived stream classes.

The Character Stream Classes

- The Character Stream I/O Classes in ***java.io***

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

The Predefined Streams

- All Java programs automatically import the ***java.lang*** package, which defines a class called ***System***, which encapsulates several aspects of the run-time environment.

- For example, using some of its methods, you can obtain the current time and the settings of various properties associated with the system.

- System also contains three ***predefined stream variables***:
  - ***in -*** System.in refers to standard input, which is the keyboard by default.
  - ***out -*** System.out refers to the standard output stream, which is the console by default.
  - ***err -*** System.err refers to the standard error stream, which also is the console by default.

- These fields are declared as public, static, and final within ***System***. This means that they can be used by any other part of your program and without reference to a specific ***System*** object.

The Predefined Streams

- However, these *streams* may be redirected to any compatible I/O device.

- System.in is an object of type ***InputStream***; System.out and System.err are objects of type ***PrintStream.***

- These are *byte streams*, even though they are typically used to read and write *characters* from and to the console.

Reading Console Input

- In Java, console input is accomplished by reading from System.in.
- To obtain a character-based stream that is attached to the console, wrap System.in in a ***BufferedReader*** object.
- ***BufferedReader*** supports a buffered input stream. A commonly used constructor is shown here:
  - BufferedReader(Reader *inputReader*)
- Here, *inputReader* is the stream that is linked to the instance of ***BufferedReader*** that is being created.
- *Reader* is an abstract class. One of its concrete subclasses is ***InputStreamReader***, which converts bytes to characters.

Reading Console Input

- To obtain an ***InputStreamReader*** object that is linked to System.in, use the following constructor:
  - InputStreamReader(InputStream *inputStream*)
- Because System.in refers to an object of type ***InputStream***, it can be used for *inputStream.*
- Putting it all together, the following line of code creates a ***BufferedReader*** that is connected to the keyboard:
  - BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
- After this statement executes, *br* is a character-based stream that is linked to the console through System.in.

Reading Characters

```
// Use a BufferedReader to read characters from the console.

import java.io.*;
class BRRead {
        public static void main(String args[ ]) throws IOException{
                char c;
                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

                // read a character
                System.out.println("Enter a character");
                c = (char) br.read( );  /* read the byte as integer
                                and convert the integer to character */
                System.out.println(c);

                // read characters
                System.out.println("Enter characters, 'q' to quit.");
                do {
                        c = (char) br.read( );
                        System.out.println(c);
                } while(c != 'q');
        }
}
```

The output from the program is shown here:

```
Enter a character
a
a
Enter characters, 'q' to quit.
b
b
c
c
q
q
```

Reading Characters

```
// Read a string from console using a BufferedReader.

import java.io.*;

class BRReadLines {

        public static void main(String args[ ]) throws IOException{

                // create a BufferedReader using System.in
                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

                String str;

                System.out.println("Enter lines of text:");
                str = br.readLine( );

                System.out.println("Entered text is:");
                System.out.println(str);

        }
}
```

The output from the program is shown here:

```
Enter lines of text:
Hello World!

Entered text is:
Hello World!
```

Writing Console Output

- Console output is most easily accomplished with print( ) and println( ).
- These methods are defined by the class ***PrintStream*** (which is the type of object referenced by System.out).
- Even though System.out is a byte stream, using it for simple program output is still acceptable.
- Because ***PrintStream*** is an output stream derived from ***OutputStream***, it also implements the low-level method *write( ).*
- Thus, *write( )* can be used to write to the console. The simplest form of *write( )* defined by ***PrintStream*** is shown here:
  - void write(int byteval)
- Although **byteval** is declared as an **integer**, only the low-order eight bits are written.

Writing Console Output

```
// Demonstrate System.out.write().

class WriteDemo {

        public static void main(String args[ ]) {

                int b;

                b = 'A';

                System.out.write(b);

                System.out.write('\n');
        }
}
```

The output from the program is shown here:

A

// Self Study from Page Nos. 308-309

- The PrintWriter Class

Reading and Writing Files

- Java provides a number of classes and methods that allow you to read and write files.
- Two of the most often-used stream classes are ***FileInputStream*** and ***FileOutputStream***, which create byte streams linked to files.

1. To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor.
   - FileInputStream(String *fileName*) throws FileNotFoundException
   - FileOutputStream(String *fileName*) throws FileNotFoundException

   1. Here, *fileName* specifies the name of the file that you want to open.
   2. When you create an input stream, if the file does not exist, then FileNotFoundException is thrown.

Reading and Writing Files

3. For output streams, if the file cannot be opened or created, then FileNotFoundException is thrown.

4. FileNotFoundException is a subclass of ***IOException***. When an output file is opened, any preexisting file by the same name is destroyed.

2. When you are done with a file, you must close it. This is done by calling the *close( )* method, which is implemented by both ***FileInputStream*** and ***FileOutputStream***.

   - void close( ) throws IOException

1. Closing a file releases the system resources allocated to the file, allowing them to be used by another file.

2. Failure to close a file can result in "memory leaks" because of unused resources remaining allocated.

Reading and Writing Files

3. To read from a file, you can use a version of *read( )* that is defined within ***FileInputStream***. The one that we will use is shown here:

   - int read( ) throws IOException

   1. It reads a single byte from the file and returns the byte as an integer value.
   2. *read( )* returns –1 when the end of the file is encountered. It can throw an ***IOException.***

4. To write to a file, you can use the *write( )* method defined by ***FileOutputStream***. Its simplest form is shown here:

   - void write(int *byteval*) throws IOException

   1. This method writes the byte specified by *byteval* to the file. Although *byteval* is declared as an integer, only the low-order eight bits are written to the file.
   1. If an error occurs during writing, an ***IOException*** is thrown.

Example Reading from a File

/ The following program uses **read( )** to input and display the contents of a file that contains ASCII text. The name of the file is specified as a command-line argument. */*

```
import java.io.*;
class ShowFile {
        public static void main(String args[ ]) {
                int i;
                FileInputStream fin;
                // First, confirm that a filename has been specified.
                if(args.length != 1) {
                        System.out.println("Usage: ShowFile filename");
                        return;     // exit from main( ) method
                }
                // Attempt to open the file.
                try {
                        fin = new FileInputStream(args[0]);
                } catch(FileNotFoundException e) {
                        System.out.println("Cannot Open File");
                        return;
                }
```

Example Reading from a File

```
        // At this point, the file is open and can be read.
        // The following reads characters until EOF is encountered.
        try {
                do {
                        i = fin.read( );
                        if(i != -1)
                                System.out.print((char) i);
                } while(i != -1);
        } catch(IOException e) {
                System.out.println("Error Reading File");
        }
        // Close the file.
        try {
                fin.close( );
        } catch(IOException e) {
                System.out.println("Error Closing File");
        }
    }
}
```

How to run this program?

1. Create a text file named TEXT.TXT
2. Put some contents in TEXT.TXT

3. Compile your program as usual
4. Run as:
   java ShowFile TEXT.TXT

5. Try running your program without command line arguments
6. Try running your program with no contents in TEXT.TXT file

Example Writing to a File

```
/* Copy a file. To use this program, specify the name of the source file and the destination file. For example, to
copy a file called FIRST.TXT to a file called SECOND.TXT, use the following command line.
java CopyFile FIRST.TXT SECOND.TXT */

import java.io.*;
class CopyFile {
        public static void main(String args[ ]) throws IOException{
                int i;
                FileInputStream fin = null;
                FileOutputStream fout = null;
                // First, confirm that both files have been specified at command line.
                if(args.length != 2) {
                        System.out.println("Usage: CopyFile from to");
                        Return;
                }
                // Copy a File.
                try {
                        // Attempt to open the files.
                        fin = new FileInputStream(args[0]);
                        fout = new FileOutputStream(args[1]);
```

Example Writing to a File

```
                do {
                        i = fin.read( );
                        if(i != -1)
                                fout.write(i);
                } while(i != -1);
        } catch(IOException e) {
                System.out.println("I/O Error: " + e);
        } finally {
                try {
                        if(fin != null)
                                fin.close( );
                } catch(IOException e2) {
                        System.out.println("Error Closing Input File");
                }
                try {
                        if(fout != null)
                                fout.close( );
                } catch(IOException e2) {
                        System.out.println("Error Closing Output File");
                }
        }
    }
}
```

How to run this program?

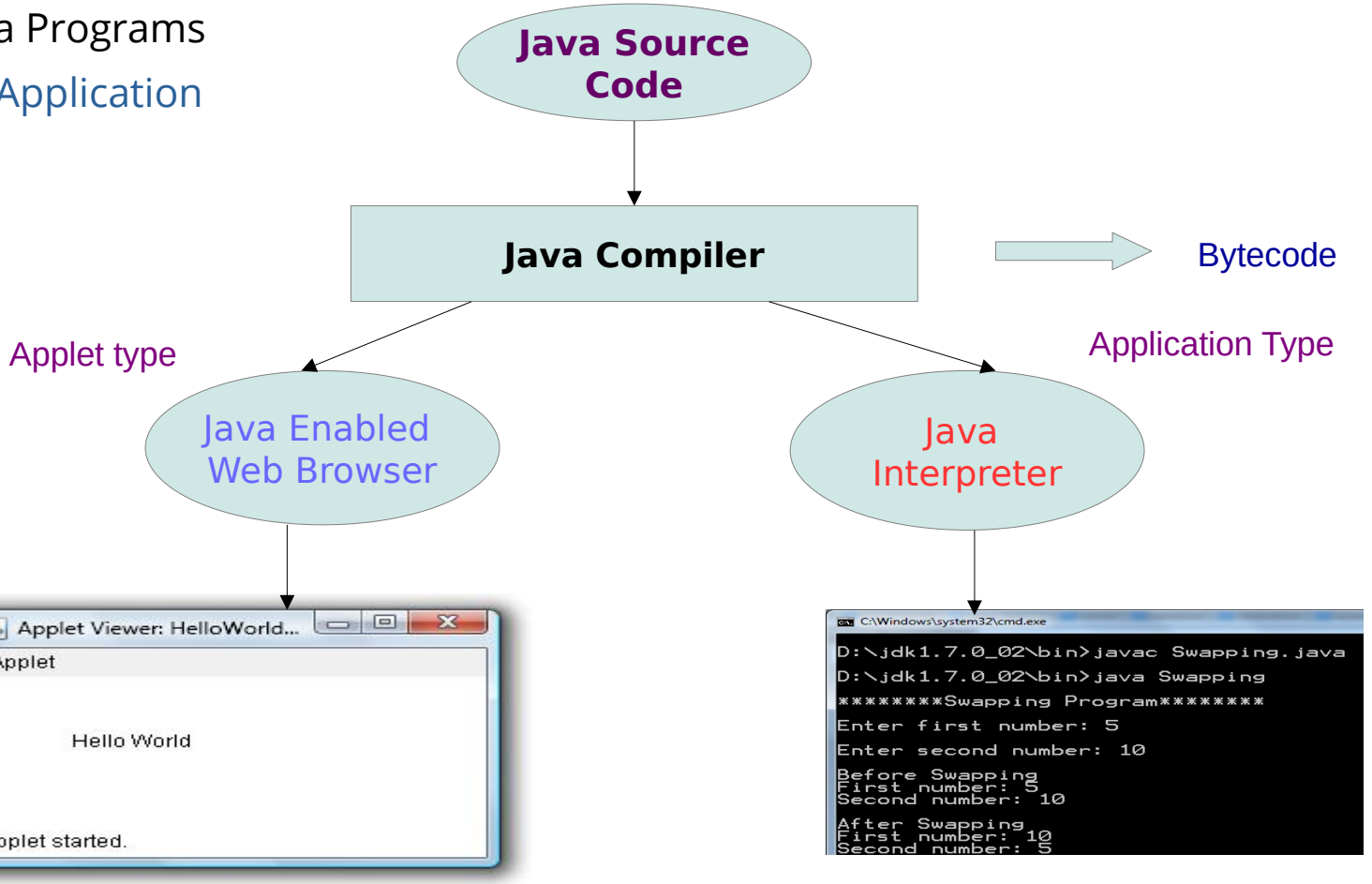
1. Create two text files FIRST.TXT & SECOND.TXT
2. Put some contents in FIRST.TXT

3. Compile your program as usual
4. Run as:
   java CopyFile FIRST.TXT SECOND.TXT

5. Try running your program without command line arguments
6. Try running your program with no contents in FIRST.TXT file

Applet Fundamentals

- Types of Java Programs
  - Console Application
  - Applets



Applet Fundamentals

- ***Applets*** are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document.
- After an applet arrives on the client, it has limited access to resources so that it can produce a **graphical user interface** and run various computations without introducing the risk of viruses or breaching data integrity.
- ***Applets*** differ from console-based applications in several key areas.
- A simple ***applet*** is shown below:

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
        public void paint(Graphics g) {
                g.drawString("A Simple Applet", 20, 20);
        }
}
```

Applet Fundamentals

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
        public void paint(Graphics g) {
                g.drawString("A Simple Applet", 20, 20);
        }
}
```

1. This applet begins with two *import* statements.
2. The first imports the Abstract Window Toolkit (**AWT**) classes. Applets interact with the user through a GUI framework, not through the console-based I/O classes.
3. The **AWT** contains very basic support for a window-based, graphical user interface.
4. The second import statement imports the applet package, which contains the class ***Applet***.
5. Every **AWT**-based applet that you create must be a subclass (either directly or indirectly) of ***Applet***.
6. SimpleApplet class must be declared as public, because it will be accessed by code that is outside the program.

Applet Fundamentals

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
        public void paint(Graphics g) {
                g.drawString("A Simple Applet", 20, 20);
        }
}
```

7. Inside SimpleApplet, **paint( )** is declared which is defined by the **AWT** and must be overridden by the applet.

8. **paint( )** is called each time that the applet must redisplay its output.

9. The **paint( )** method has one parameter of type ***Graphics*** which ontains the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

10. **drawString( )**, which is a member of the ***Graphics*** class, outputs a string beginning at the specified X,Y location.

Applet Fundamentals

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
        public void paint(Graphics g) {
                g.drawString("A Simple Applet", 20, 20);
        }
}
```

11. Notice that the applet does not have a **main( )** method. An applet begins execution when the name of its class is passed to an applet viewer or to a network browser.

12. **Compiling** is same as in console applications. However, **running *Applet*** involves a different process. In fact, there are two ways in which you can run an applet:

    a. Executing the applet within a Java-compatible web browser**(by html file)**

    b. Using an applet viewer, such as the standard tool, **appletviewer**.

References

Reference for this topic

- [Book: Java: The Complete Reference, Ninth Edition: Herbert Schildt]
  https://www.amazon.in/Java-Complete-Reference-Herbert-Schildt/dp/0071808558

- [Web: GeeksforGeeks]
  https://www.geeksforgeeks.org/java/

- [Web: Java T Point tutorial]
  https://www.javatpoint.com/java-tutorial