Lecture - 17

Character Streams

Arpit Rana

12th April 2022

Character Encoding

A character encoding is a scheme for representing characters.

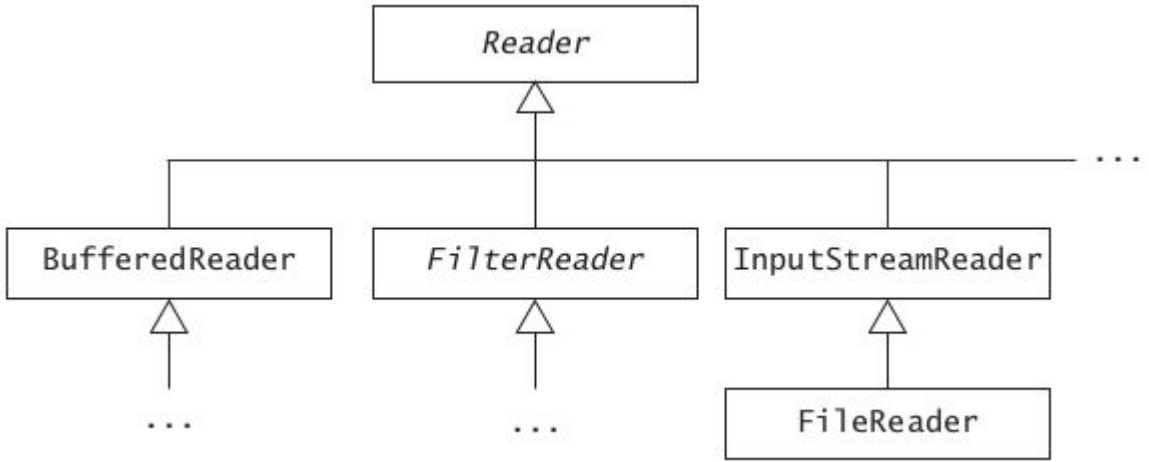
Java programs represent values of the char type internally in the 16-bit Unicode character encoding, but the host platform might use another character encoding to represent and store characters externally.

The abstract classes **`Reader`** and **`Writer`** are the roots of the inheritance hierarchies for streams that read and write Unicode characters using a specific character encoding.

Inheritance Hierarchy for Character Streams

A reader is an input character stream that reads a sequence of Unicode characters.

Character encodings are used by readers and writers to convert between external encoding and internal Unicode characters.



Inheritance Hierarchy for Character Streams

A reader is an input character stream that reads a sequence of Unicode characters.

Character encodings are used by readers and writers to convert between external encoding and internal Unicode characters.

`BufferedReader`	A reader that buffers the characters read from an underlying reader. The underlying reader must be specified and an optional buffer size can be given.
`InputStreamReader`	Characters are read from a byte input stream which must be specified. The default character encoding is used if no character encoding is explicitly specified.
`FileReader`	Reads characters from a file, using the default character encoding. The file can be specified by a `File` object, a `FileDescriptor`, or a `String` file name. It automatically creates a `FileInputStream` that is associated with the file.

Inheritance Hierarchy for Character Streams

Readers use the following methods for reading Unicode characters:

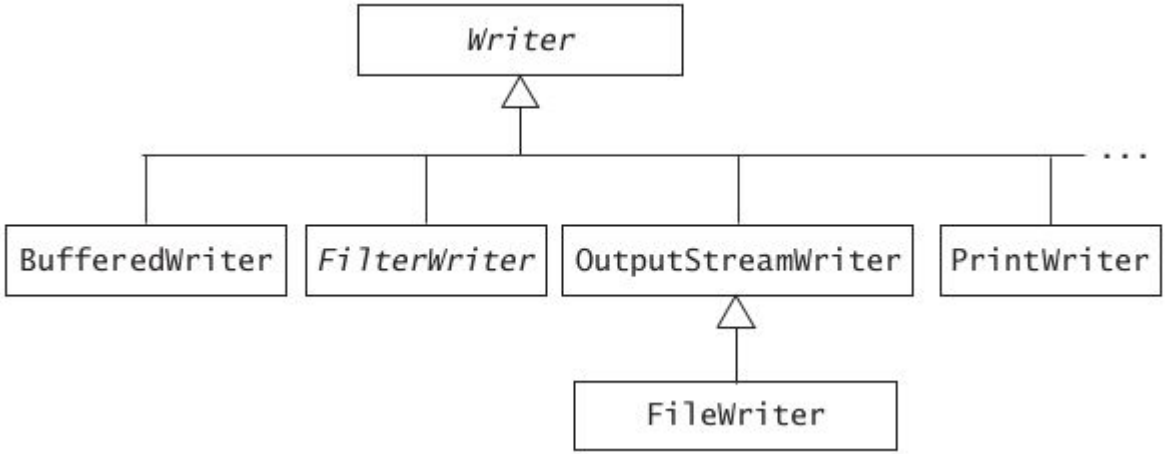
```
int read() throws IOException
int read(char cbuf[]) throws IOException
int read(char cbuf[], int off, int len) throws IOException
```

Note that the read() methods read the character as an int in the range 0 to 65535 *(0x0000–0xFFFF)*.

The value –1 is returned if the end of the stream has been reached.

Inheritance Hierarchy for Character Streams

A writer is an output character stream that writes a sequence of Unicode characters.



Inheritance Hierarchy for Character Streams

A writer is an output character stream that writes a sequence of Unicode characters.

`BufferedWriter`	A writer that buffers the characters before writing them to an underlying writer. The underlying writer must be specified, and an optional buffer size can be specified.
`OutputStreamWriter`	Characters are written to a byte output stream which must be specified. The default character encoding is used if no explicit character encoding is specified.
`FileWriter`	Writes characters to a file, using the default character encoding. The file can be specified by a `File` object, a `FileDescriptor`, or a `String` file name. It automatically creates a `FileOutputStream` that is associated with the file.
`PrintWriter`	A filter that allows *text* representation of Java objects and Java primitive values to be written to an underlying output stream or writer. The underlying output stream or writer must be specified.

Inheritance Hierarchy for Character Streams

Writers use the following methods for writing Unicode characters:

```
void write(int c) throws IOException
```

The `write()` method takes an `int` as argument, but writes only the least significant 16 bits.

```
void write(char[] cbuf) throws IOException
void write(String str) throws IOException
void write(char[] cbuf, int off, int length) throws IOException
void write(String str, int off, int length) throws IOException
```

These methods write the characters from an array of characters or a string.

Like byte streams, a character stream should be closed when no longer needed to free system resources.

```
void close() throws IOException
void flush() throws IOException
```

Print Writers

The capabilities of the OutputStreamWriter and the InputStreamReader classes are limited, as they primarily write and read characters.

In order to write a text representation of Java primitive values and objects, a PrintWriter should be chained to either

- a writer,
- a byte output stream,
- a File, or
- a String file name

Print Writers

It uses one of the following constructors:

```
PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoFlush)
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
PrintWriter(File file)
PrintWriter(File file, String charsetName)
PrintWriter(String fileName)
PrintWriter(String fileName, String charsetName)
```

The autoFlush argument specifies whether the PrintWriter should be flushed when any println() method of the PrintWriter class is called.

When supplying the File object or the file name, the character encoding can be specified explicitly.

Print Writers

The PrintWriter class provides the following methods for writing text representation of Java primitive values and objects

print()-methods	*println*-methods
	println()
print(boolean b)	println(boolean b)
print(char c)	println(char c)
print(int i)	println(int i)
print(long l)	println(long l)
print(float f)	println(float f)
print(double d)	println(double d)
print(char[] s)	println(char[] ca)
print(String s)	println(String str)
print(Object obj)	println(Object obj)

Print Writers

The println() methods write the text representation of their argument to the underlying stream, and then append a line-separator.

The println() methods use the correct platform-dependent line-separator. For example,

- on Unix platforms the line-separator is '\n' (newline),
- on Windows platforms it is "\r\n" (carriage return + newline), and
- on the Macintosh it is '\r' (carriage return).

The print() methods create a text representation of an object by calling the toString() method on the object. The print() methods do not throw any IOException.

Writing Text Files

When writing text to a file using the default character encoding, the following four procedures for setting up a PrintWriter can be used.

Setting up a PrintWriter based on an OutputStreamWriter which is chained to a FileOutputStream

1. Create a FileOutputStream:
   FileOutputStream outputFile = new FileOutputStream("info.txt");
2. Create an OutputStreamWriter which is chained to the FileOutputStream:
   OutputStreamWriter outputStream = new OutputStreamWriter(outputFile);

   The OutputStreamWriter uses the default character encoding for writing the characters to the file.
3. Create a PrintWriter which is chained to the OutputStreamWriter:
   PrintWriter printWriter1 = new PrintWriter(outputStream, true);

Writing Text Files

Setting up a PrintWriter based on a FileOutputStream:

1. Create a FileOutputStream:

   FileOutputStream outputFile = new FileOutputStream("info.txt");

2. Create a PrintWriter which is chained to the FileOutputStream:

   PrintWriter printWriter2 = new PrintWriter(outputFile, true);

The intermediate OutputStreamWriter to convert the characters using the default encoding is automatically supplied.

Writing Text Files

Setting up a PrintWriter based on a FileWriter:

1. Create a FileWriter which is a subclass of OutputStreamWriter:

   FileWriter fileWriter = new FileWriter("info.txt");

This is equivalent to having an OutputStreamWriter chained to a FileOutputStream for writing the characters to the file

2. Create a PrintWriter which is chained to the FileWriter:

   PrintWriter printWriter3 = new PrintWriter(fileWriter, true);

Writing Text Files

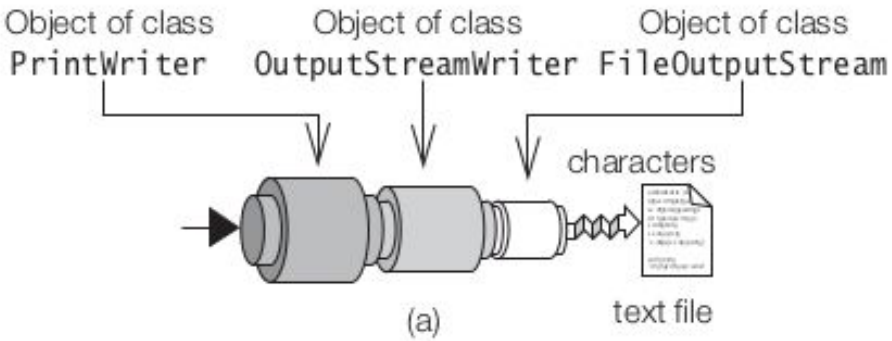
Setting up a PrintWriter, given the file name:

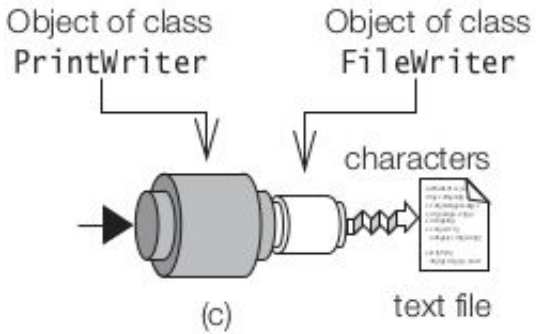
1. Create a PrintWriter, supplying the file name:

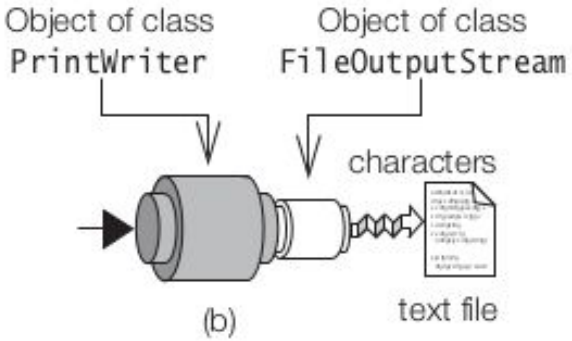
   ```
   PrintWriter printWriter3 = new PrintWriter("info.txt");
   ```

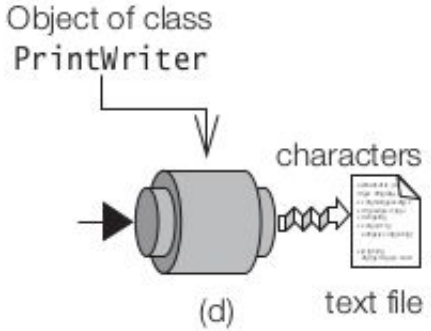
The underlying OutputStreamWriter is created to write the characters to the file in the default encoding. In this case, there is no automatic flushing.

Writing Text Files









Reading Text Files

Java primitive values and objects cannot be read directly from their text representation.

Characters must be read and converted to the relevant values explicitly.

One common strategy is to write lines of text and tokenize the characters as they are read, a line at a time (e.g. using java.util.Scanner Class).

Reading Text Files

When reading characters from a file using the default character encoding, the following two procedures for setting up an InputStreamReader can be used.

Setting up an InputStreamReader which is chained to a FileInputStream

1. Create a FileInputStream :

   ```
   FileInputStream inputFile = new FileInputStream("info.txt");
   ```

2. Create an InputStreamReader which is chained to the FileInputStream :

   ```
   InputStreamReader reader = new InputStreamReader(inputFile);
   ```

The InputStreamReader uses the default character encoding for reading the characters from the file.

Reading Text Files

When reading characters from a file using the default character encoding, the following two procedures for setting up an InputStreamReader can be used.

Setting up a FileReader which is a subclass of InputStreamReader

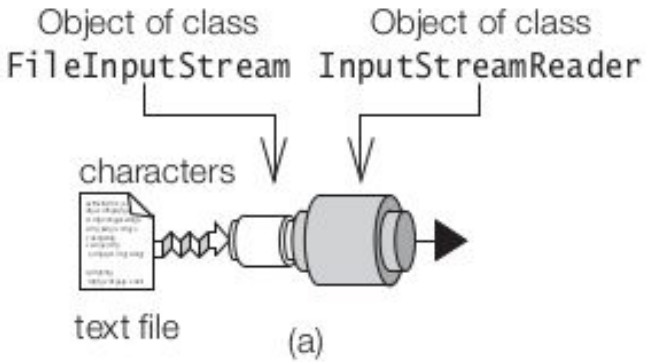
1. Create a FileReader :

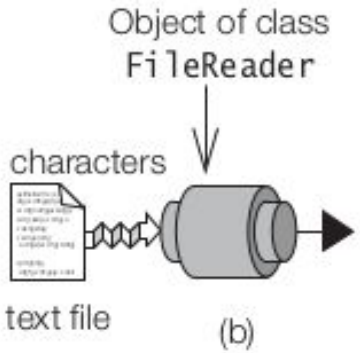
   FileReader fileReader = new FileReader("info.txt");

This is equivalent to having an InputStreamReader chained to a FileInputStream for reading the characters from the file, using the default character encoding.

Other constructors of the FileReader class accept a File or a FileDescriptor .

Reading Text Files





Using Buffered Writers

A BufferedWriter can be chained to the underlying writer by using one of the following constructors:

```
BufferedWriter(Writer out)
BufferedWriter(Writer out, int size)
```

- The default buffer size is used, unless the buffer size is explicitly specified.
- Characters, strings, and arrays of characters can be written using the methods for a Writer, but these now use buffering to provide efficient writing of characters.
- The BufferedWriter class also provides the method newLine() for writing the platform-dependent line-separator.

Using Buffered Writers

The following code creates a PrintWriter whose output is buffered and the characters are written using the 8859_1 character encoding

```
FileOutputStream   outputFile      = new FileOutputStream("info.txt");
OutputStreamWriter outputStream    = new OutputStreamWriter(outputFile, "8859_1");
BufferedWriter     bufferedWriter1 = new BufferedWriter(outputStream);
PrintWriter        printWriter1    = new PrintWriter(bufferedWriter1, true);
```

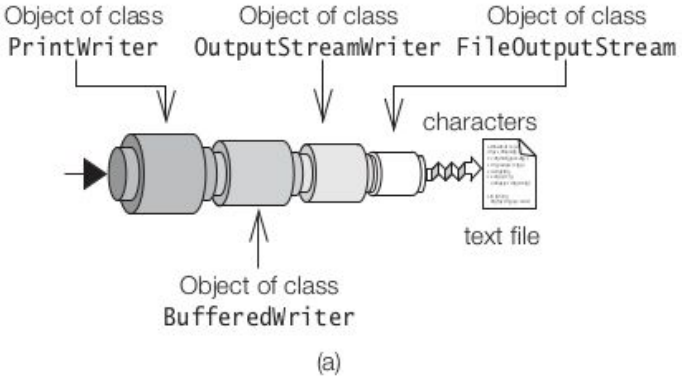
The following code creates a PrintWriter whose output is buffered, and the characters are written using the default character encoding

```
FileWriter     fileWriter      = new FileWriter("info.txt");
BufferedWriter bufferedWriter2 = new BufferedWriter(fileWriter);
PrintWriter    printWriter2    = new PrintWriter(bufferedWriter2, true);
```

Using Buffered Writers

Note that in both cases, the PrintWriter is used to write the characters.

The BufferedWriter is sandwiched between the PrintWriter and the underlying OutputStream Writer.



Using Buffered Readers

A BufferedReader can be chained to the underlying reader by using one of the following constructors:

```
BufferedReader(Reader in)
BufferedReader(Reader in, int size)
```

- The default buffer size is used, unless the buffer size is explicitly specified.
- In addition to the methods of the Reader class, the BufferedReader class provides the method readLine() to read a line of text from the underlying reader:

```
String readLine() throws IOException
```

  - The null value is returned when the end of the stream is reached. The returned string must explicitly be converted to other values.

Using Buffered Writers

The following code creates a BufferedReader that can be used to read text lines from a file, using the 8859_1 character encoding

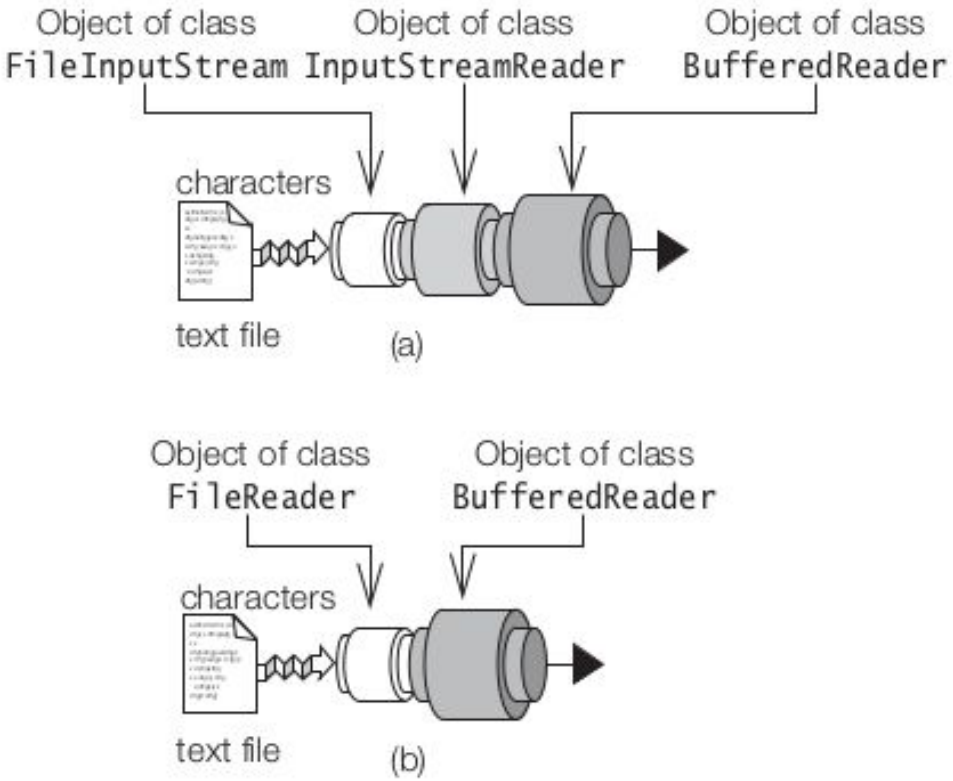
```
FileInputStream   inputFile       = new FileInputStream("info.txt");
InputStreamReader reader          = new InputStreamReader(inputFile, "8859_1");
BufferedReader    bufferedReader1 = new BufferedReader(reader);
```

The following code creates a BufferedReader that can be used to read text lines from a file, using the default character encoding.

```
FileReader     fileReader      = new FileReader("lines.txt");
BufferedReader bufferedReader2 = new BufferedReader(fileReader);
```

Note that in both cases the BufferedReader object is used to read the text lines.

Using Buffered Writers



Demonstrating Readers and Writers, and Character Encoding

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;

public class CharEncodingDemo {
  public static void main(String[] args)
                    throws IOException, NumberFormatException {

    // Character encoding.                                          (1)
    FileOutputStream outputFile = new FileOutputStream("info.txt");
    OutputStreamWriter writer = new OutputStreamWriter(outputFile, "8859_1");
    BufferedWriter bufferedWriter1 = new BufferedWriter(writer);
    PrintWriter printWriter = new PrintWriter(bufferedWriter1, true);
    System.out.println("Writing using encoding: " + writer.getEncoding());

    // Print Java primitive values, one on each line.               (2)
    printWriter.println(true);
    printWriter.println('A');
    printWriter.println(Byte.MAX_VALUE);
    printWriter.println(Short.MIN_VALUE);
    printWriter.println(Integer.MAX_VALUE);
    printWriter.println(Long.MIN_VALUE);
    printWriter.println(Float.MAX_VALUE);
    printWriter.println(Math.PI);
```

Demonstrating Readers and Writers, and Character Encoding

```
// Close the writer, which also closes the underlying stream      (3)
printWriter.flush();
printWriter.close();

// Create a BufferedReader which uses 8859_1 character encoding   (4)
FileInputStream inputFile = new FileInputStream("info.txt");
InputStreamReader reader = new InputStreamReader(inputFile, "8859_1");
BufferedReader bufferedReader = new BufferedReader(reader);
System.out.println("Reading using encoding: " + reader.getEncoding());

// Read the (exact number of) Java primitive values               (5)
// in the same order they were written out, one on each line
boolean v = bufferedReader.readLine().equals("true")? true : false;
char    c = bufferedReader.readLine().charAt(0);
byte    b = (byte) Integer.parseInt(bufferedReader.readLine());
short   s = (short) Integer.parseInt(bufferedReader.readLine());
int     i = Integer.parseInt(bufferedReader.readLine());
long    l = Long.parseLong(bufferedReader.readLine());
float   f = Float.parseFloat(bufferedReader.readLine());
double  d = Double.parseDouble(bufferedReader.readLine());
```

Demonstrating Readers and Writers, and Character Encoding

```
        // Check for end of stream:                                    (6)
        String line = bufferedReader.readLine();
        if (line != null ) {
          System.out.println("More input: " + line);
        } else {
          System.out.println("End of stream");
        }

        // Close the reader, which also closes the underlying stream   (7)
        bufferedReader.close();

        // Write the values read on the terminal                       (8)
        System.out.println("Values read:");
        System.out.println(v);
        System.out.println(c);
        System.out.println(b);
        System.out.println(s);
        System.out.println(i);
        System.out.println(l);
        System.out.println(f);
        System.out.println(d);
    }
}
```

Demonstrating Readers and Writers, and Character Encoding

Output from the program:

```
Writing using encoding: ISO8859_1
Reading using encoding: ISO8859_1
End of stream
Values read:
true
A
127
-32768
2147483647
-9223372036854775808
3.4028235E38
3.141592653589793
```

The Standard Input, Output, and Error Streams

- The standard output stream (usually the display) is represented by the PrintStream object System.out.
- The standard input stream (usually the keyboard) is represented by the InputStream object System.in . In other words, it is a byte input stream.
- The standard error stream (also usually the display) is represented by System.err which is another object of the PrintStream class.
- The PrintStream class offers print() methods which act as corresponding print() methods from the PrintWriter class.
- These methods can be used to write output to System.out and System.err.
- In other words, both System.out and System.err act like PrintWriter, but in addition they have write() methods for writing bytes.

The Console Class

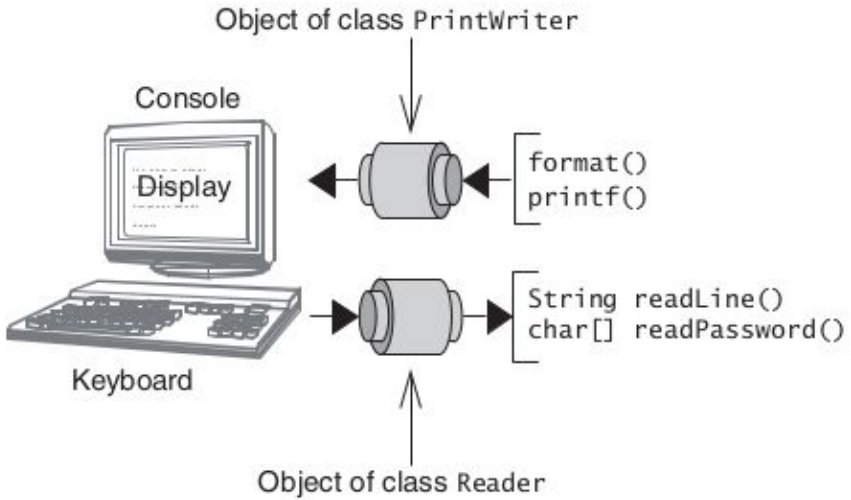
A console is a unique character-based device associated with a JVM. Whether a JVM has a console depends on the platform, and also on the manner in which the JVM is invoked.

- When the JVM is started from a command line, and the standard input and output streams have not been redirected, the console will normally correspond to the keyboard and the display.
- In any case, the console will be represented by an instance of the class Console .
- This Console instance is obtained by calling the static method console() of the System class.
- If there is no console associated with the JVM, the null value is returned by this method.

The Console Class

```
// Obtaining the console:
Console console = System.console();

if (console == null) {
  System.err.println("No console available.");
  return;
}
// Continue ...
```



The Console Class

For creating dialog for console-based applications, the Console class provides the following functionality:

- Prompt and read a line of character-based response.

```
String username = console.readLine("Enter the user name (%d chars): ", 4);
```

The readLine() method first prints the formatted prompt on the console, and then returns the characters typed at the console when the line is terminated by the ENTER key.

The Console Class

- Prompt and read passwords without echoing the characters on the console.

```
char[] password;
do {
    password = console.readPassword("Enter password (min. %d chars): ", 6);
} while (password.length < 6);
```

The readPassword() method first prints the formatted prompt, and returns the password characters typed by the user in an array of char when the line is terminated by the ENTER key.

The password characters are not echoed on the display.

- Print formatted strings to the console.

The Console class provides the format() and the printf() methods for this purpose.

The Console Class

```
String readLine()
String readLine(String format, Object... args)
```

The first method reads a single line of text from the console. The second method prints a formatted prompt first, then reads a single line of text from the console. The prompt is constructed by formatting the specified `args` according to the specified `format`.

```
char[] readPassword()
char[] readPassword(String format, Object... args)
```

The first method reads a password or a password phrase from the console with echoing disabled. The second method does the same, but first prints a formatted prompt.

```
Reader reader()
```

This retrieves the unique `Reader` object associated with this console.

The Console Class

```
Console format(String format, Object... args)
Console printf(String format, Object... args)
```

These methods write a formatted string to this console's output stream using the specified format string and arguments, according to the default locale.

```
PrintWriter writer()
```

The method retrieves the unique `PrintWriter` object associated with this console.

```
void flush()
```

This method flushes the console and forces any buffered output to be written immediately.

The Console Class: Changing Password Example

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;

/** Class to create a password file */
public final class MakePasswordFile {

  public static void main (String[] args) throws IOException {
    Map<String, Integer> pwStore = new TreeMap<String, Integer>();
    pwStore.put("tom", "123".hashCode());
    pwStore.put("dick", "456".hashCode());
    pwStore.put("harry", "789".hashCode());

    PrintWriter destination = new PrintWriter(new FileWriter("pws.txt"));
    Set<Map.Entry<String, Integer>> pwSet = pwStore.entrySet();
    for (Map.Entry<String, Integer> entry : pwSet) {
      // Format: login password
      destination.printf("%s %s%n", entry.getKey(), entry.getValue());
    }
    destination.flush();
    destination.close();
  }
}
```

The Console Class: Changing Password Example

```
import java.io.BufferedReader;
import java.io.Console;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.Map;
import java.util.Scanner;
import java.util.Set;
import java.util.TreeMap;

/** Class to change the password of a user */
public class ChangePassword {

  // Map for storing login/password info.                          (1)
  private static Map<String, Integer> pwStore;

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

    // Obtain the console:                                         (2)
    Console console = System.console();
    if (console == null) {
      System.err.println("No console available.");
      return;
    }

    // Read the login/password info from a file:                   (3)
    readPWStore();
```

The Console Class: Changing Password Example

```
// Verify user:                                                    (4)
String login;
char[] oldPassword;
do {
  login = console.readLine("Enter your login: ");
  oldPassword = console.readPassword("Enter your current password: ");
} while (login.length() == 0 || oldPassword.length == 0 ||
         !verifyPassword(login, oldPassword));
Arrays.fill(oldPassword, '0');

// Changing the password:                                          (5)
boolean noMatch = false;
do {
  // Read the new password and its confirmation:
  char[] newPasswordSelected
         = console.readPassword("Enter your new password: ");
  char[] newPasswordConfirmed
         = console.readPassword("Confirm your new password: ");
```

The Console Class: Changing Password Example

```
    // Compare the supplied passwords:
    noMatch = newPasswordSelected.length == 0  ||
              newPasswordConfirmed.length == 0 ||
              !Arrays.equals(newPasswordSelected, newPasswordConfirmed);
    if (noMatch) {
      console.format("Passwords don't match. Please try again.%n");
    } else {
      changePassword(login, newPasswordSelected);
      console.format("Password changed for %s.%n", login);
    }
    // Zero-fill the password arrays:
    Arrays.fill(newPasswordSelected, '0');
    Arrays.fill(newPasswordConfirmed, '0');
  } while (noMatch);

  // Save the login/password info to a file:                    (6)
  writePWStore();
}
```

```
/** Verifies the password. */                                    // (7)
private static boolean verifyPassword(String login, char[] password) {
  Integer suppliedPassword = String.copyValueOf(password).hashCode();
  Integer storedPassword = pwStore.get(login);
  return storedPassword != null && storedPassword.equals(suppliedPassword);
}

/** Changes the password for the user. */                        // (8)
private static void changePassword(String login, char[] password) {
  Integer newPassword = String.copyValueOf(password).hashCode();
  pwStore.put(login, newPassword);
}

/** Reads login/password from a file */                          // (9)
private static void readPWStore() throws IOException {
  pwStore = new TreeMap<String, Integer>();
  BufferedReader source = new BufferedReader(new FileReader("pws.txt"));
  while (true) {
    String txtLine = source.readLine();
    if (txtLine == null) break;              // EOF?
    Scanner scanner = new Scanner(txtLine);
    // Format: <login string> <password int hash value>
    String login = scanner.next();
    Integer password = scanner.nextInt();
    pwStore.put(login, password);
  }
  source.close();
}
```

The Console Class: Changing Password Example

```
    /** Writes login/password to a file */                          // (10)
    private static void writePWStore() throws IOException {
      PrintWriter destination = new PrintWriter(new FileWriter("pws.txt"));
      Set<Map.Entry<String, Integer>> pwSet = pwStore.entrySet();
      for (Map.Entry<String, Integer> entry : pwSet) {
      // Format: <login string> <password int hash value>
      destination.printf("%s %s%n", entry.getKey(), entry.getValue());
    }
    destination.close();
  }
}
```

Running the program:

```
>java ChangePassword
Enter your login: tom
Enter your current password:
Enter your new password:
Confirm your new password:
Password changed for tom
```

Next lecture -
Object Serialization