

Exploring Java.io

Java io classes

BufferedInputStream	FileWriter	PipedInputStream
BufferedOutputStream	FilterInputStream	PipedOutputStream
BufferedReader	FilterOutputStream	PipedReader
BufferedWriter	FilterReader	PipedWriter
ByteArrayInputStream	FilterWriter	PrintStream
ByteArrayOutputStream	InputStream	PrintWriter
CharArrayReader	InputStreamReader	PushbackInputStream
CharArrayWriter	LineNumberReader	PushbackReader
DataInputStream	ObjectInputStream	RandomAccessFile
DataOutputStream	ObjectInputStream.GetField	Reader
File	ObjectOutputStream	SequenceInputStream
FileDescriptor	ObjectOutputStream.PutField	SerializablePermission
FileInputStream	ObjectStreamClass	StreamTokenizer
FileOutputStream	ObjectStreamField	StringReader
FilePermission	OutputStream	StringWriter
FileReader	OutputStreamWriter	Writer



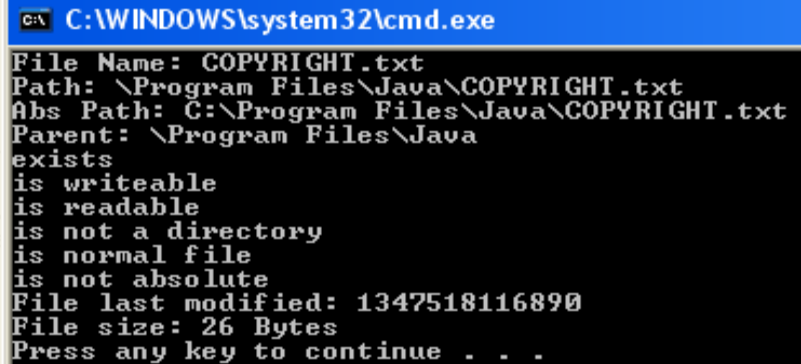
Java io interfaces

Closeable	FileFilter	ObjectInputValidation
DataInput	FilenameFilter	ObjectOutput
DataOutput	Flushable	ObjectStreamConstants
Externalizable	ObjectInput	Serializable



File

```
import java.io.File;
class FileDemo {
static void p(String s) {
System.out.println(s); }
public static void main(String args[]) {
    File f1 = new File("/Program Files/Java/COPYRIGHT.txt");
    p("File Name: " + f1.getName());
    p("Path: " + f1.getPath());
    p("Abs Path: " + f1.getAbsolutePath());
    p("Parent: " + f1.getParent());
    p(f1.exists() ? "exists" : "does not exist");
    p(f1.canWrite() ? "is writeable" : "is not writeable");
    p(f1.canRead() ? "is readable" : "is not readable");
    p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
    p(f1.isFile() ? "is normal file" : "might be a named pipe");
    p(f1.isAbsolute() ? "is absolute" : "is not absolute");
    p("File last modified: " + f1.lastModified());
    p("File size: " + f1.length() + " Bytes");
    }
}
```



```
C:\WINDOWS\system32\cmd.exe
File Name: COPYRIGHT.txt
Path: \Program Files\Java\COPYRIGHT.txt
Abs Path: C:\Program Files\Java\COPYRIGHT.txt
Parent: \Program Files\Java
exists
is writeable
is readable
is not a directory
is normal file
is not absolute
File last modified: 1347518116890
File size: 26 Bytes
Press any key to continue . . .
```



// Using directories.

```
import java.io.File;
class DirList {
public static void main(String args[]) {
    String dirname = "/package";
    File f1 = new File(dirname);
    if (f1.isDirectory()) {
        System.out.println("Directory of " + dirname);
        String s[] = f1.list();
        for (int i=0; i < s.length; i++) {

            File f = new File(dirname + "/" + s[i]);
            if (f.isDirectory()) {
                System.out.println(s[i] + " is a directory");
            } else {
                System.out.println(s[i] + " is a file");
            }
        }
    } else {
        System.out.println(dirname + " is not a directory");
    }
}
```

C:\WINDOWS\system32\cmd.exe

```
Directory of /package
Demo.java is a file
p2 is a directory
Press any key to continue . . .
```



java.io package(1)

Java has two separate interfaces in java.io package.

Closeable

A Closeable is a source or destination of data that can be closed. The close method is invoked to release resources that the object is holding.

Flushable

A Flushable is a destination of data that can be flushed. The flush method is invoked to write any buffered output to the underlying stream.

Closeable and Flushable interfaces

The Closeable and Flushable interfaces define a uniform way for specifying that a stream can be closed or flushed.

Closeable: The objects that implement the Closeable interface can be closed.

This interface defines the close() method.

void close() throws IOException

It closes the invoking stream and releases all the resources. It is implemented by all the I/O classes that open a stream.



java.io package(2)

Flushable: The objects that implement the Flushable interface can force buffered output to be written to a stream to which the objects are attached.

This interface defines the flush() method.

void flush() throws IOException

1. Closeable interface

- The Closeable interface includes only one abstract method, close().
- When close() method is called, the system resources held by the stream object are released and can be used by other part of the program.
- Many stream classes implement this interface and overrides the close() method.
- Any class that implements this interface can use close() method to close the stream handle.
- Also if the super class implements this interface, the sub class can use this method.
- For example, the InputStream implements this method and its subclass FileInputStream can use close() method.
- For that matter, all the streams can use close() method as the super classes of all streams, InputStream, OutputStream, Reader and Writer, implement Closeable interface.



java.io package(3)

2. Flushable interface

- The Flushable interface includes only one method – flush().
- Many destination streams implement this interface and overrides the flush() method.
- When this method is called, the data held in the buffers is flushed out to the destination file to write.

Following are a few classes ' signature that implements Closeable or Flushable and some times both interfaces.

1. `public abstract class InputStream extends Object implements Closeable`
2. `public abstract class OutputStream extends Object implements Closeable, Flushable`
3. `public abstract class Reader extends Object implements Readable, Closeable`
4. `public abstract class Writer extends Object implements Appendable, Closeable, Flushable`
5. `public class PrintStream extends FilterOutputStream implements Appendable, Closeable`



A decorative blue wavy line with a light blue shadow, flowing from the top left towards the bottom left of the slide.

Stream classes

Stream Classes

- ❖ Java's stream-based I/O is built upon four abstract classes: `InputStream`, `OutputStream`, `Reader`, and `Writer`.
 - They are used to create several concrete stream subclasses.
 - User programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream classes.
 - `InputStream` and `OutputStream` are designed for byte streams.
 - `Reader` and `Writer` are designed for character streams.
 - The byte stream classes and the character stream classes form separate hierarchies.
 - We use the character stream classes when working with characters or strings, and use the byte stream classes when working with bytes or other binary objects.



Byte Streams

- ❖ The byte stream classes provide a rich environment for handling byte-oriented I/O.
 - A byte stream can be used with any type of object, including binary data.
 - This versatility makes byte streams important to many types of programs.
- The byte stream classes are topped by `InputStream` and `OutputStream`.

InputStream

`InputStream` is an abstract class that defines Java's model of streaming byte input. All of the methods in this class will throw an `IOException` on error conditions.

OutputStream

`OutputStream` is an abstract class that defines streaming byte output. All of the methods in this class return a void value and throw an `IOException` in the case of errors.



Method	Description
<code>int available()</code>	Returns the number of bytes of input currently available for reading.
<code>void close()</code>	Closes the input source. Further read attempts will generate an IOException .
<code>void mark(int <i>numBytes</i>)</code>	Places a mark at the current point in the input stream that will remain valid until <i>numBytes</i> bytes are read.
<code>boolean markSupported()</code>	Returns true if <code>mark()</code> / <code>reset()</code> are supported by the invoking stream.
<code>int read()</code>	Returns an integer representation of the next available byte of input. -1 is returned when the end of the file is encountered.
<code>int read(byte <i>buffer</i>[])</code>	Attempts to read up to <i>buffer.length</i> bytes into <i>buffer</i> and returns the actual number of bytes that were successfully read. -1 is returned when the end of the file is encountered.
<code>int read(byte <i>buffer</i>[], int <i>offset</i>, int <i>numBytes</i>)</code>	Attempts to read up to <i>numBytes</i> bytes into <i>buffer</i> starting at <i>buffer[offset]</i> , returning the number of bytes successfully read. -1 is returned when the end of the file is encountered.
<code>void reset()</code>	Resets the input pointer to the previously set mark.
<code>long skip(long <i>numBytes</i>)</code>	Ignores (that is, skips) <i>numBytes</i> bytes of input, returning the number of bytes actually ignored.

The Methods Defined by InputStream



Method

Description

`void close()`

Closes the output stream. Further write attempts will generate an `IOException`.

`void flush()`

Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.

`void write(int b)`

Writes a single byte to an output stream. Note that the parameter is an `int`, which allows you to call `write()` with expressions without having to cast them back to `byte`.

`void write(byte buffer[])`

Writes a complete array of bytes to an output stream.

`void write(byte buffer[], int offset,
 int numBytes)`

Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer[offset]*.

The Methods Defined by OutputStream




```
import java.io.*;
class FileInputStreamDemo {
public static void main(String args[]) {
int size;
try ( FileInputStream f = new
FileInputStream("FileInputStreamDemo.j
ava") ) {
System.out.println("Total Available Bytes:
" + (size = f.available()));
int n = size/40;
System.out.println("First " + n +
" bytes of the file one read() at a time");
for (int i=0; i < n; i++) {
System.out.print((char) f.read()); }
System.out.println("\nStill Available: " +
f.available());
System.out.println("Reading the next " +
n + " with one read(b[])");
byte b[] = new byte[n];
if (f.read(b) != n) {
System.err.println("couldn't read " + n + "
bytes "); }
}
```

```
System.out.println(new String(b, 0, n));
System.out.println("\nStill Available: "
+ (size = f.available()));
System.out.println("Skipping half of
remaining bytes with skip()");
f.skip(size/2);
System.out.println("Still Available: " +
f.available());
System.out.println("Reading " + n/2 + "
into the end of array");
if (f.read(b, n/2, n/2) != n/2) {
System.err.println("couldn't read " +
n/2 + " bytes."); }
System.out.println(new String(b, 0,
b.length));
System.out.println("\nStill Available: "
+ f.available());
} catch(IOException e) {
System.out.println("I/O Error: " + e); }
}
```




```
C:\Windows\system32\cmd.exe

Total Available Bytes: 1411
First 35 bytes of the file one read() at a time

// Demonstrate FileInputStream.

Still Available: 1376
Reading the next 35 with one read(b[])
// This program uses try-with-resou

Still Available: 1341
Skipping half of remaining bytes with skip()
Still Available: 671
Reading 17 into the end of array
// This program uad(b) != n) {
$yu

Still Available: 654
Press any key to continue . . . _
```

FileOutputStream

```
import java.io.*;
```

```
public class FileOutputStreamDemo {
```

```
public static void main(String[ ] args) {
```

```
FileOutputStream out; // declare a file output object
```

```
PrintStream p; // declare a print stream object
```

```
try {
```

```
// Create a new file output stream,connected to "File.txt"
```

```
out = new FileOutputStream("File.txt");
```

```
// Connect print stream to the output stream
```

```
p = new PrintStream(out);
```

```
p.println("The text shown here will write to a file after run");
```

```
System.out.println("The Text is written to File.txt");
```

```
p.close();
```

```
}
```

```
catch (Exception e) {
```

```
System.err.println("Error writing to file");
```

```
}
```

```
}
```

```
}
```



C:\WINDOWS\system32\cmd.exe

The Text is written to File.txt
Press any key to continue . . .



Byte Array input/output stream

```
import java.io.*;

class ByteStreamTest{

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

```
    ByteArrayOutputStream bOutput = new
ByteArrayOutputStream(12);
```

```
    while( bOutput.size() != 10 ){
        // Gets the inputs from the user
        bOutput.write(System.in.read());
    }
    byte b [ ] = bOutput.toByteArray();
    System.out.println("Print the content");
    for(int x= 0 ; x < b.length; x++){
        //printing the characters
        System.out.print((char)b[x] + " ");
    }
```

```
        System.out.println(" ");
```

```
        int c;
```

```
        ByteArrayInputStream bInput = new
ByteArrayInputStream(b);
```

```
        System.out.println("Converting characters to
Upper case " );
```

```
        for(int y = 0 ; y < 1; y++ ){
            while(( c = bInput.read()) != -1){
```

```
                System.out.println(Character.toUpperCase((char)c)
);
```

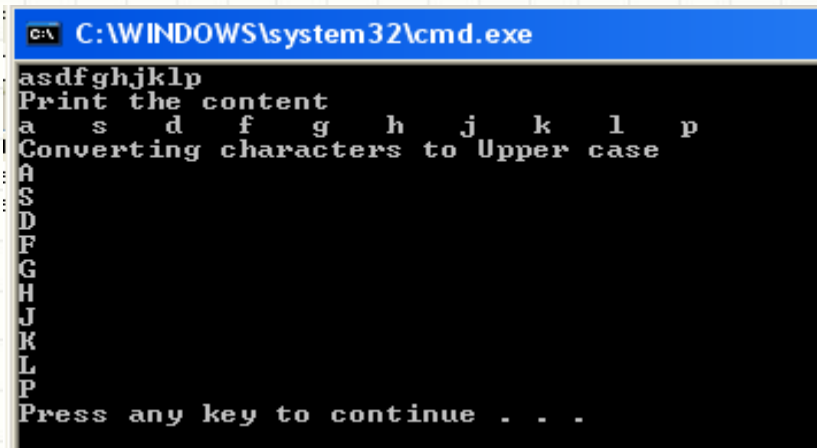
```
            }
```

```
            bInput.reset();
```

```
        }
```

```
    }
```

```
}
```

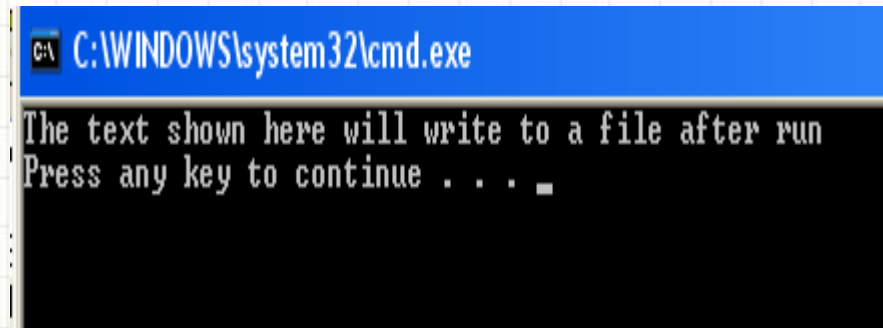


```
C:\WINDOWS\system32\cmd.exe
asdfghjklp
Print the content
a s d f g h j k l p
Converting characters to Upper case
A S D F G H J K L P
Press any key to continue . . .
```



Buffered byte streams

```
import java.io.*;
class ReadFilter
{
    public static void main(String args[])
    {
        try
        {
            FileInputStream fin = new FileInputStream("File.txt");
            BufferedInputStream bis = new BufferedInputStream(fin);
            // Now read the buffered stream.
            while (bis.available() > 0)
            {
                System.out.print((char)bis.read());
            }
        }
        catch (Exception e)
        {
            System.err.println("Error reading file: " + e);
        }
    }
}
```



BufferedInputStream

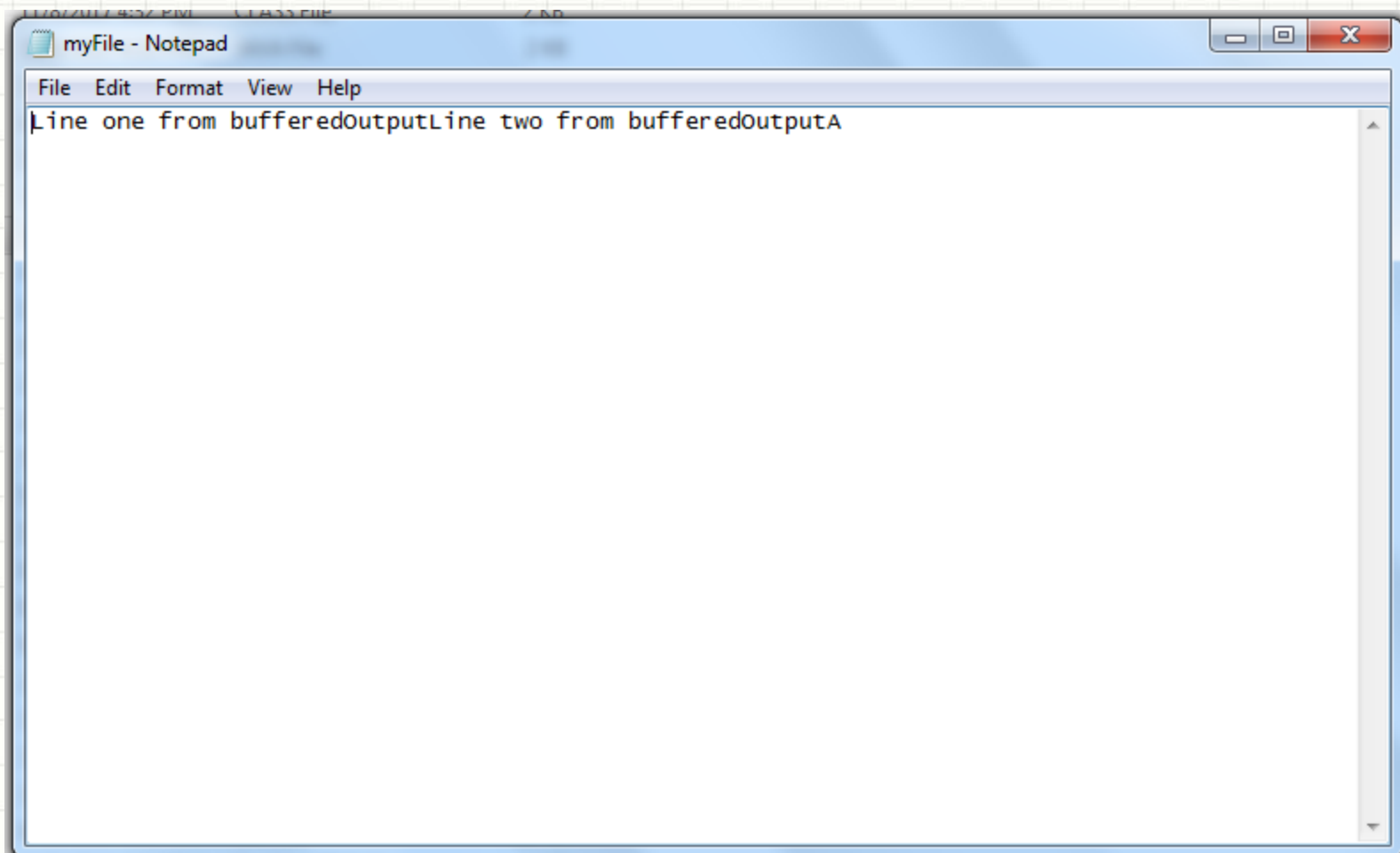
```
import java.io.BufferedOutputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class writeMain {
    public void writeToFile(String filename) {
        BufferedOutputStream bufferedOutput = null;
        try {
            //Construct the BufferedOutputStream object
            bufferedOutput = new
            BufferedOutputStream(new
            FileOutputStream(filename));
            //Start writing to the output stream
            bufferedOutput.write("Line one".getBytes());
            bufferedOutput.write("\n".getBytes());
            //new line, you might want to use \r\n if
            you're on Windows
            bufferedOutput.write("Line two".getBytes());
            bufferedOutput.write("\n".getBytes());

            //prints the character that has the decimal value
            of 65
            bufferedOutput.write(65);
        } catch (FileNotFoundException ex) {
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        } finally {
            //Close the BufferedOutputStream
            try {
                if (bufferedOutput != null) {
                    bufferedOutput.flush();
                    bufferedOutput.close();
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        new writeMain().writeToFile("myFile.txt");
    }
}
```





5/25/2016 2:57 PM Firefox HTML Doc... 1 KB





Character Streams

Character Streams

- byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters.
- Since one of the main purposes of Java is to support the “write once, run anywhere” philosophy, it was necessary to include direct I/O support for characters.
- ✓ At the top of the character stream hierarchies are the Reader and Writer abstract classes.

Reader

Reader is an abstract class that defines Java’s model of streaming character input. All of the methods in this class will throw an IOException on error conditions.

Writer

Writer is an abstract class that defines streaming character output. All of the methods in this class return a void value and throw an IOException in the case of errors .



Method

`abstract void close()`

`void mark(int numChars)`

`boolean markSupported()`

`int read()`

`int read(char buffer[])`

`abstract int read(char buffer[],
 int offset,
 int numChars)`

`boolean ready()`

`void reset()`

`long skip(long numChars)`

Description

Closes the input source. Further read attempts will generate an **IOException**.

Places a mark at the current point in the input stream that will remain valid until *numChars* characters are read.

Returns **true** if **mark()**/**reset()** are supported on this stream.

Returns an integer representation of the next available character from the invoking input stream. -1 is returned when the end of the file is encountered.

Attempts to read up to *buffer.length* characters into *buffer* and returns the actual number of characters that were successfully read. -1 is returned when the end of the file is encountered.

Attempts to read up to *numChars* characters into *buffer* starting at *buffer[offset]*, returning the number of characters successfully read. -1 is returned when the end of the file is encountered.

Returns **true** if the next input request will not wait. Otherwise, it returns **false**.

Resets the input pointer to the previously set mark.

Skips over *numChars* characters of input, returning the number of characters actually skipped.



Method	Description
Writer append(char <i>ch</i>)	Appends <i>ch</i> to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence <i>chars</i>)	Appends <i>chars</i> to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence <i>chars</i> , int <i>begin</i> , int <i>end</i>)	Appends the subrange of <i>chars</i> specified by <i>begin</i> and <i>end</i> -1 to the end of the invoking output stream. Returns a reference to the invoking stream.
abstract void close()	Closes the output stream. Further write attempts will generate an IOException .
abstract void flush()	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.

Method	Description
void write(int <i>ch</i>)	Writes a single character to the invoking output stream. Note that the parameter is an int , which allows you to call write with an expression without having to cast it back to char . However, only the low-order 16 bits are written.
void write(char <i>buffer</i> [])	Writes a complete array of characters to the invoking output stream.
abstract void write(char <i>buffer</i> [], int <i>offset</i> , int <i>numChars</i>)	Writes a subrange of <i>numChars</i> characters from the array <i>buffer</i> , beginning at <i>buffer[offset]</i> to the invoking output stream.
void write(String <i>str</i>)	Writes <i>str</i> to the invoking output stream.
void write(String <i>str</i> , int <i>offset</i> , int <i>numChars</i>)	Writes a subrange of <i>numChars</i> characters from the string <i>str</i> , beginning at the specified <i>offset</i> .

The Methods Defined by Writer



Buffered Reader

- ❖ `BufferedReader` improves performance by buffering input. It has two constructors:
 1. `BufferedReader(Reader inputStream)`
 2. `BufferedReader(Reader inputStream, int bufSize)`
- The first form creates a buffered character stream using a default buffer size.
- In the second, the size of the buffer is passed in `bufSize`.
- As is the case with the byte-oriented stream, buffering an input character stream also provides the foundation required to support moving backward in the stream within the available buffer.
- To support this, `BufferedReader` implements the `mark()` and `reset()` methods, and `BufferedReader.markSupported()` returns `true`.



```
import java.io.*;

public class BufferedReaderDemo {
    public static void main(String[] args) {
        try {
            BufferedReader br = new BufferedReader(new FileReader("BMS.txt"));
            String bms;

            while ((bms = br.readLine()) != null) {
                System.out.println(bms);
            }
        } catch (IOException e) {
        }
    }
}
```

C:\WINDOWS\system32\cmd.exe

Hi This is demo program..
Press any key to continue . . .



Buffered Writer

- ❖ A `BufferedWriter` is a `Writer` that adds a `flush()` method that can be used to ensure that data buffers are physically written to the actual output stream.
- Using a `BufferedWriter` can increase performance by reducing the number of times data is actually physically written to the output stream.

A `BufferedWriter` has these two constructors:

1. `BufferedWriter(Writer outputStream)`
 2. `BufferedWriter(Writer outputStream, int bufSize)`
- The first form creates a buffered stream using a buffer with a default size.
 - In the second, the size of the buffer is passed in `bufSize`.



```
import java.io.*;
```

```
public class FileWriteBufferedWriter {
```

```
public static void main(String[] args) {
```

```
try {
```

```
BufferedWriter out = new BufferedWriter(new FileWriter("BMS1.txt"));
```

```
out.write("Welcome to BMS MCA Java Class");
```

```
out.close();
```

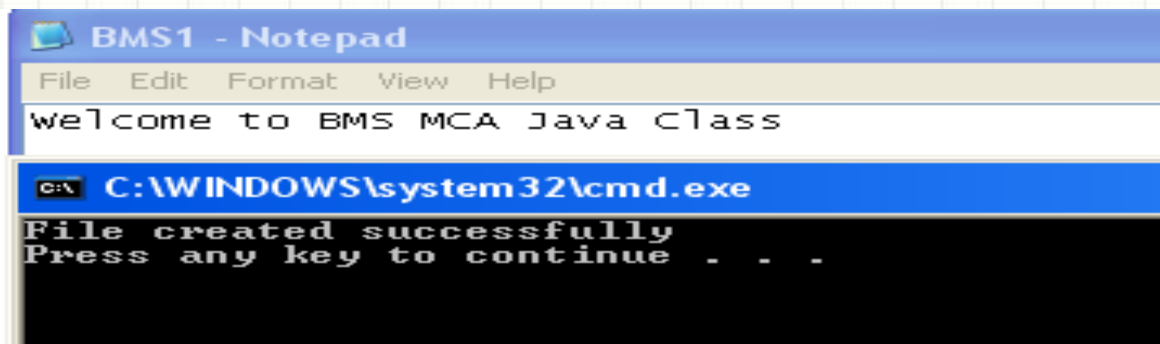
```
System.out.println("File created successfully");
```

```
} catch (IOException e) {
```

```
}
```

```
}
```

```
}
```



Print Writer

- ❖ `PrintWriter` is essentially a character-oriented version of `PrintStream`.
- It provides the formatted output methods `print()` and `println()`.

`PrintWriter` has four constructors:

1. `PrintWriter(OutputStream outputStream)`
 2. `PrintWriter(OutputStream outputStream, boolean flushOnNewline)`
 3. `PrintWriter(Writer outputStream)`
 4. `PrintWriter(Writer outputStream, boolean flushOnNewline)`
- where `flushOnNewline` controls whether Java flushes the output stream every time `println()` is called. If `flushOnNewline` is true, flushing automatically takes place. If false, flushing is not automatic.
 - The first and third constructors do not automatically flush.
 - Java's `PrintWriter` objects support the `print()` and `println()` methods for all types, including `Object`.
 - If an argument is not a simple type, the `PrintWriter` methods will call the object's `toString()` method and then print out the result.



```
import java.io.*;

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

        PrintWriter pw = new PrintWriter(System.out, true);
        pw.println("This is a string");

        int i = -7;
        pw.println(i);
        double d = 4.5e-7;
        pw.println(d);
    }
}
```



C:\WINDOWS\system32\cmd.exe

```
This is a string
-7
4.5E-7
Press any key to continue . . .
```



Java console(1)

The `java.util.Console` class provides methods to access the character-based console device, if any, associated with the current Java virtual machine.

Following is the declaration for `java.util.Console` class:

```
public final class Console extends Object implements Flushable
```

Class methods

1. `void flush()`

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

2. `Console format(String fmt, Object... args)`

This method writes a formatted string to this console's output stream using the specified format string and arguments.

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

This method is used to write a formatted string to this console's output stream using the specified format string and arguments.

4. `Reader reader()`

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



Java console(2)

5. **String readLine()**

This method reads a single line of text from the console.

6. **String readLine(String fmt, Object... args)**

This method provides a formatted prompt, then reads a single line of text from the console.

7. **char[] readPassword()**

This method reads a password or passphrase from the console with echoing disabled.

8. **char[] readPassword(String fmt, Object... args)**

This method provides a formatted prompt, then reads a password or passphrase from the console with echoing disabled.

9. **PrintWriter writer()**

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




```

import java.io.Console;
import java.util.Arrays;
public class ConsoleDemo {
    public static void main(String[] args) {
        Console console = System.console();

        if (console != null) {
            String username = console.readLine("Username: ");
            char[] password = console.readPassword("Password: ");

            if (username.equals("BMS Admin")
                && String.valueOf(password).equals("BMSMCA")) {
                console.printf("Welcome to Java Application %1$s.\n",username);
                Arrays.fill(password, ' ');
            } else {
                console.printf("Invalid username or password.\n");
            }
        } else {
            System.out.println("Console is not available.");
        }
    }
}

```

```

C:\WINDOWS\system32\cmd.exe
Username: admin
Password:
Invalid username or password.
Press any key to continue . . .

```

```

C:\WINDOWS\system32\cmd.exe
Username: BMS Admin
Password:
Welcome to Java Lab BMS Admin.
Press any key to continue . . .

```





Generics

- It is possible to create classes, interfaces, and methods that will work with various kinds of data.

Eg: Collections Framework.

- Generics means parameterized types.



- Using generics, it is possible to create a single class, that automatically works with different types of data.
- A class, interface, or method that operates on a parameterized type is called generic.



A Simple Generics Example

// A simple generic class.

// Here, T is a type parameter that

// will be replaced by a real type

// when an object of type Gen is created.

```
class Gen<T> {
```

```
    T ob; // declare an object of type T
```

```
    // Pass the constructor a reference to
```

```
    // an object of type T.
```

```
    Gen(T o) {
```

```
        ob = o;
```

```
    }
```



```
// Return ob.
```

```
T getob() {  
    return ob;  
}
```

```
// Show type of T.
```

```
void showType() {  
    System.out.println("Type of T is " +  
        ob.getClass().getName());  
}
```

```
}
```



// Demonstrate the generic class.

```
class GenDemo {  
    public static void main(String args[]) {  
        // Create a Gen reference for Integers.  
        Gen<Integer> iOb;  
        // Create a Gen<Integer> object and assign its  
        // reference to iOb. Notice the use of autoboxing  
        // to encapsulate the value 88 within an Integer  
        object.  
  
        iOb = new Gen<Integer>(88);  
        // Show the type of data used by iOb.  
        iOb.showType();  
    }  
}
```



// Get the value in iOb. Notice that
// no cast is needed.

```
int v = iOb.getob();
```

```
System.out.println("value: " + v);
```

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

// Create a Gen object for Strings.

```
Gen<String> strOb = new  
    Gen<String>("Generics Test");
```

// Show the type of data used by strOb.

```
strOb.showType();
```



// Get the value of strOb. Again, notice
// that no cast is needed.

```
String str = strOb.getob();
```

```
System.out.println("value: " + str);
```

```
}
```

```
}
```



Output:

Type of T is java.lang.Integer
value: 88

Type of T is java.lang.String
value: Generics Test



A Generic Class with Two Type Parameters

- We can declare more than one type parameter in a generic type.
- To specify two or more type parameters, simply use a comma-separated list.



// A simple generic class with two type
// parameters: T and V.

```
class TwoGen<T, V> {  
    T ob1;  
    V ob2;  
    // Pass the constructor a reference to  
    // an object of type T and an object of type V.  
    TwoGen(T o1, V o2) {  
        ob1 = o1;  
        ob2 = o2;  
    }  
}
```



// Show types of T and V.

```
void showTypes() {  
    System.out.println("Type of T is " +  
                        ob1.getClass().getName());  
    System.out.println("Type of V is " +  
                        ob2.getClass().getName());  
}  
T getob1() {  
    return ob1;  
}  
V getob2() {  
    return ob2;  
}  
}
```



// Demonstrate TwoGen.

```
class SimpGen {
```

```
    public static void main(String args[]) {
```

```
        TwoGen<Integer, String> tgObj =  
        new TwoGen<Integer, String>(88, "Generics");
```

```
        // Show the types.
```

```
        tgObj.showTypes();
```

```
        // Obtain and show values.
```

```
        int v = tgObj.getob1();
```

```
        System.out.println("value: " + v);
```

```
        String str = tgObj.getob2();
```

```
        System.out.println("value: " + str);
```

```
    }
```

```
}
```



Output:

Type of T is java.lang.Integer

Type of V is java.lang.String

value: 88

value: Generics



The General Form of a Generic Class

- The syntax for declaring a generic class:
class class-name<type-param-list> { // ...
- The syntax for declaring a reference to a generic class:

**class-name<type-arg-list> var-name =
new class-name<type-arg-list>(cons-arg-list);**



UNIT – 5 ..

