

The background features a white central area surrounded by abstract green shapes. On the left, a solid green triangle points towards the center. On the right, several overlapping, semi-transparent green triangles of varying shades (from light lime to dark forest green) create a layered effect. A large, bold, lime-green number '10' is positioned on the right side, partially overlapping the green shapes. A thin, light gray line extends diagonally from the bottom left towards the top right, passing behind the green shapes.

10

Recall

- ▶ *We already learnt how to get input from console. Do you remember the class we used for this?*

Console IO

- ▶ Java 6 introduces a new class called `Console` in `java.io` package.
- ▶ This class has convenient method that can prompt the user for the input and read input from the terminal at the same time.
- ▶ It also does have features that will not echo the password entry on the screen.
- ▶ This class is character based.
- ▶ `Console` is a `final` class with no public constructors. To obtain `Console` object
`System.console()` method is used which returns the only instance that JVM has for this class
- ▶ Read and write operations are `synchronized` methods.

JVM and Console

- ▶ JVM has a console that is dependent upon the underlying platform. This object can be obtained using the `System.console()` *method that returns Console object*
- ▶ If the JVM is started from command line then its console will exist.
- ▶ If the virtual machine is started automatically or from an IDE then console will not be available.
- ▶ Therefore before using the Console object, a check for null has to be done to ensure that the object exists.

```
if ((cons = System.console()) != null){  
...  
}
```

Console Methods

- ▶ `Console format(String fmt, Object... args)`
- ▶ `Console printf(String fmt, Object... Args)`
 - ▶ Used to write formatted data. The `fmt` represents format string which are same as the one that was used for `System.out.printf`.
- ▶ `String readLine(String fmt, Object... args0)`
 - ▶ Prompts and reads a single line of text
- ▶ `String readLine()`
 - ▶ reads a single line of text
- ▶ `char[] readPassword(String fmt, Object... args)`
 - ▶ Prompts and reads a password or passphrase from the console with echoing disabled
- ▶ `char[] readPassword()`
 - ▶ reads a password or passphrase from the console with echoing disabled

Example

```
import java.io.Console;
import java.util.Arrays;

public class Test {
    public static void main(String[] args) {
        String pass="abcd";
        Console c = System.console();
        if (c == null) {
            System.err.println("Console Object is not available.");
            System.exit(1);
        }
        String login = c.readLine("Login:");
        char [] pwd = c.readPassword("Password: ");
        String s=new String(pwd);
        if(s.equals(pass)) System.out.println("Right pwd");
        else System.out.println("Incorrect pwd");
    }
}
```

Note that this code throws an exception at runtime when run on eclipse. It prints "Console Object is not available".

But it works fine when you run from the command prompt.

Working with files at OS level

- ▶ `java.io.File` class can be used to work with system dependent commands for files and directories.
- ▶ The path name in the code hence will depend on the underlying OS in which JVM is installed.
- ▶ To make the code portable so that it works on all systems, **static** member **separator** defined in the **File** class can be used.
- ▶ The path name can be either *absolute* or *relative*.

Creating a file using `File`

- ▶ **`File(String pathname)`**
 - ▶ Creates a new **`File`** instance (if relative path is given then it converts it into abstract pathname)
- ▶ **`File(String parent, String child)`**
- ▶ **`File(File parent, String child)`**
 - ▶ Creates a new **`File`** instance from a parent pathname and a child pathname string. If “parent” is `null` then it behaves like the single-argument **`File`** constructor
- ▶ **`boolean createNewFile()` throws `IOException`**
 - ▶ creates a new, empty file if a file with the name (as specified in the constructor) does not exist and returns `true`; otherwise it returns `false`. Note that this method throws `IOException` if I/O error occurs
- ▶ So to create a file:
 - ▶ Create instance of **`File`** object
 - ▶ Call **`createNewFile`** method

Other members in File

- ▶ **static final String separator**
 - ▶ system-dependent separator character
- ▶ **boolean delete()**
 - ▶ Deletes the file or directory. Directory must be empty in order to be deleted. Returns **true** if the delete operation is successful.
- ▶ **void deleteOnExit()**
 - ▶ Deletes the file or directory when the virtual machine terminates. Deletion happen only for normal termination of the virtual machine. Once deletion has been requested, it is not possible to cancel the request
- ▶ **boolean renameTo(File dest)**
 - ▶ Renames the file . This is system dependent so return value should always be checked to make sure that the rename operation was successful

- ▶ **boolean mkdir()**
 - ▶ Creates the directory named by the pathname. Returns true if the directory was created; false otherwise
- ▶ **boolean mkdirs()**
 - ▶ Creates the directory named by this pathname, including any necessary but nonexistent parent directories. Note that if this operation fails it may have succeeded in creating some of the necessary parent directories.
- ▶ **boolean isDirectory()**
 - ▶ Returns true if the File object denotes a directory; false otherwise.
- ▶ **String getName()**
 - ▶ Returns the name of the file or directory . just the last name in the pathname's name sequence

- ▶ **String[] list()**
 - ▶ Returns list names files and directories in the directory
- ▶ **boolean isFile()**
 - ▶ Returns true if the File object denotes a file; false otherwise.
- ▶ **boolean isHidden()**
 - ▶ Returns true if the File object is a hidden file; false otherwise.
- ▶ **boolean exists()**
 - ▶ Returns true if the file or directory exists; false otherwise
- ▶ **String getAbsolutePath()**
 - ▶ Returns the absolute pathname string of this abstract pathname
- ▶ **long lastModified()**
 - ▶ Returns the time that the file object was last modified.
- ▶ **long length()**
 - ▶ Returns the length of the file, unspecified it is a directory.

New methods added in Java 6

- ▶ **boolean canExecute()**
- ▶ **boolean canRead()**
- ▶ **boolean canWrite()**
 - ▶ Returns true if the application can execute/read/write the file denoted; false otherwise
- ▶ **boolean setXxx(boolean permission)**
- ▶ **setXxx(boolean permission, boolean ownerOnly)**
 - ▶ Xxx could be **Readable**, **Executable** or **Writable**
 - ▶ Sets the read/execute/write permission if **permission** is true
 - ▶ **ownerOnly** is true then the read/execute/write permission applies only to the owner's execute permission provided underlying file system can distinguish between owner and others; otherwise, it applies to everybody.

Example: Creating a file

Creates a new file named `newFile.txt`. If file exists then it deletes the file and creates a new one

```
import java.io.*;
class FileOper{
public static void main(String str[]){
try{
File file = new File("newFile.txt");
if(file.exists())
file.delete();
boolean b=file.createNewFile();
System.out.println(b);
}catch(IOException e){ }
}}
```

What are streams

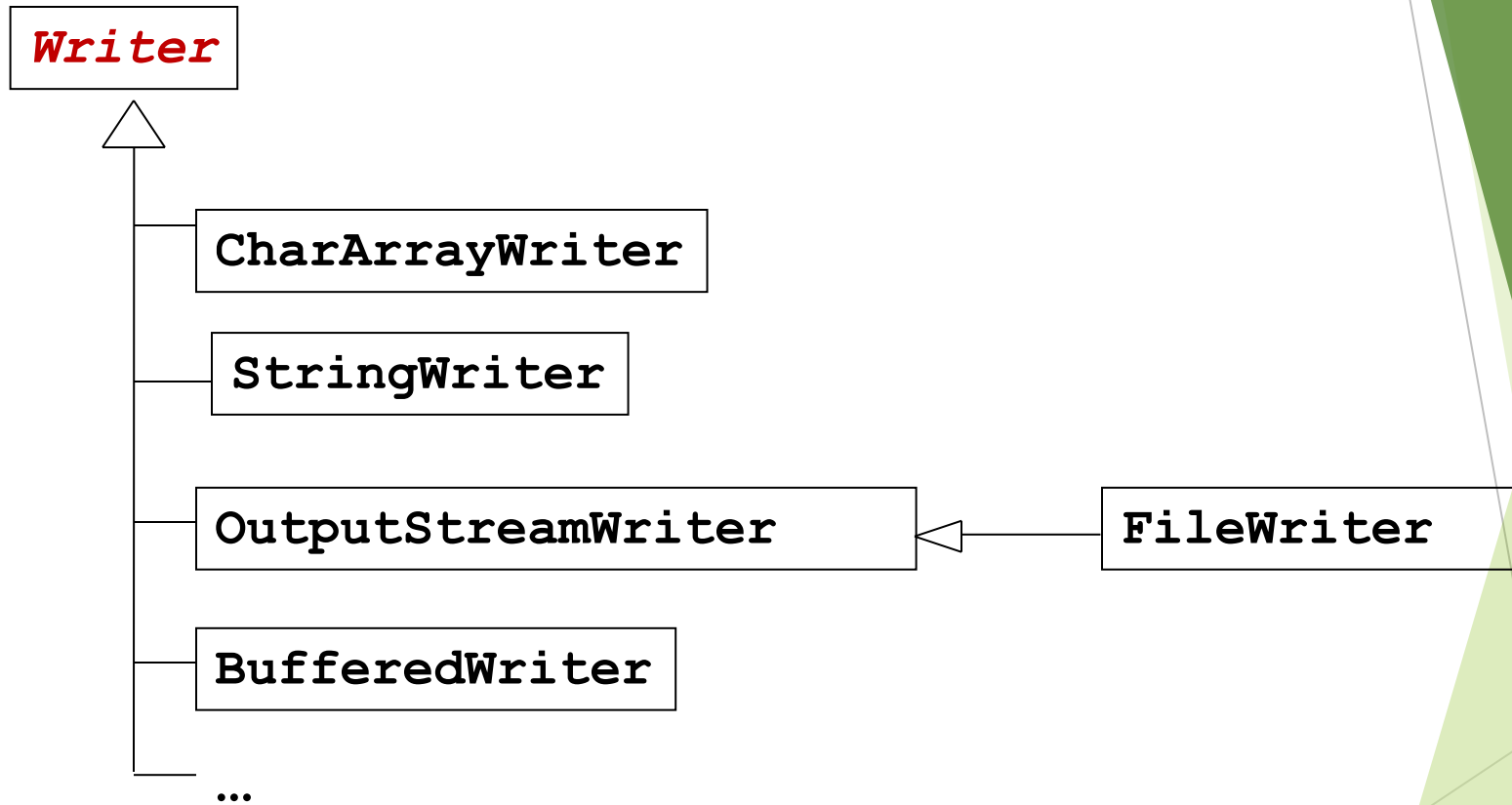
- ▶ An IO stream is an abstract term for any type of input or output device.
- ▶ There are 2 types of stream
 - ▶ Input stream to read data from a source. An input stream may be files, keyboard, console, other programs, a network, or an array!
 - ▶ Output stream to read data into a destination. An output stream may be disk files, monitor, a network, other programs, or an array
- ▶ Fundamentally stream may be
 - ▶ Byte stream : data read or written is in the form of byte
or
 - ▶ Character stream: data read or written is in the form of character
- ▶ Stream is a sequence of data

Stream types in Java

- ▶ Character stream
 - ▶ Character stream writer classes
 - ▶ Character stream reader classes
- ▶ Byte stream
 - ▶ Byte stream writer classes
 - ▶ Byte stream reader classes
 - ▶ **Supports Serialization**

Character stream

- ▶ As we are aware, the character in java is in the form of unicode.
- ▶ Character stream I/O automatically translates unicode to the local character set.
- ▶ At the top of the hierarchy we have **Reader** and **Writer** abstract classes are provided
- ▶ *First we will explore Writer classes*



Writer

```
void write(char[] cbuf)
void write(char[] cbuf, int off, int len)
void write(String str)
void write(String str, int off, int len)
void write(int c)
void close()
void flush()
```

- ▶ It is an abstract class for writing to character streams. Methods are to write or append a character or character array or strings and flush.
- ▶ All the methods throw **IOException**.

CharArrayWriter

- ▶ **CharArrayWriter** allows to write data into char array.
- ▶ Constructors:
`CharArrayWriter()`
`CharArrayWriter(int initialSize)`
- ▶ Methods
`char[] toCharArray()`
`int size()`
`String toString()`
`CharArrayWriter append(char c)`
`void writeTo(Writer out)`

And all the methods of the **Writer** class.

StringWriter

- ▶ **StringWriter** allows to write data string.
- ▶ Constructors:
 - ▶ **StringWriter()**
 - ▶ **StringWriter(int initialSize)**
- ▶ Methods
 - ▶ **StringWriter append(char c)**
 - ▶ **String toString()**
 - ▶ **StringBuffer getBuffer()**

OutputStreamWriter

An **OutputStreamWriter** is used as a bridge from character streams to byte streams:

Constructor:

```
OutputStreamWriter(OutputStream out)
```

- **OutputStream** is the top-most class in the byte stream (like **Writer** is top-most class in the character stream)

Example:

```
Writer out = new OutputStreamWriter(System.out) ;
```

If you remember **out** is a member of **System** class which is of type **PrintStream** which is subclass of **OutputStream**.

```
public void write(int c)
```

Example :

The code extracts part of string and displays on the console.
Note the way we have used finally to close the stream.

```
public class Test {  
    public static void main(String[] args) {  
        Writer out = new OutputStreamWriter(System.out);  
        String s="hello java";  
        try {  
            out.write(s, 3, 5);  
        } catch (IOException e) {e.printStackTrace();}  
        finally{  
            try {  
                out.close();  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

Result
lo ja

FileWriter

FileWriter inherits from **OutputStreamWriter**.

Constructors:

► **FileWriter(File file)**

► **FileWriter(String fileName)**

Creates an instance of **FileWriter** and also the file if it does not exist. If it exists it overwrites.

If the file exists but is a directory rather than a regular file **IOException** is thrown

► **FileWriter(File file, boolean append)**

► **FileWriter(String fileName, boolean append)**

Provide same functionalities as that of the previous constructor, if **append** is **true**, then data will be written to the end of the file rather than the beginning.

All constructors throw **IOException**

Example: Using CharArrayWriter & FileWriter

In this example we first get all the command-line strings in a `CharArrayWriter` and then copy the content of `CharArrayWriter` to a file using `FileWriter`

```
import java.io.*;
public class CharWriterMain {
    public static void main(String args[]) {
        FileWriter f2 =null;
        try{
            CharArrayWriter f= new CharArrayWriter();
            int i=1;
            for (String s:args){
                char buf[] = s.toCharArray() ;
                f.write(i++ +".");
                f.write(buf,0,s.length());
                f.write("\n");
            }
            f2 =new FileWriter("register.txt");
            f.writeTo(f2);
        } catch(IOException ioe){}
```



```
finally{  
try{if(f2!=null)f2.close();}  
catch(IOException e){}}  
}}
```

On executing with the following
command line arguments

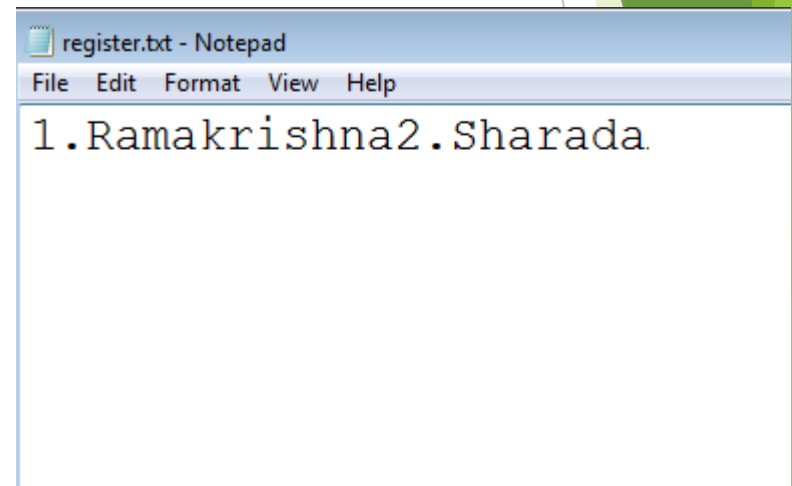
```
java CharWriterMain Ramakrishna  
Sharada
```

register.txt file gets created .

When we open the file we find that the
newline char is written into the file.

This happens because Notepad expects
"lines" to be separated by `\r\n`.

Since line separator is OS dependent best
way to code this would be to use
newline() discussed in the next slide.



BufferedWriter

- This class wraps the `Writer` class to provide additional functionality of buffering characters for the efficient writing of single characters, arrays, and strings.

Constructors:

`BufferedWriter(Writer out)`

`BufferedWriter(Writer out, int size)`

Creates a buffered character stream object. The default buffer size is large enough for most purposes. In cases where more is required size can be specified.

If `size < 0` _____ is thrown.

(Can you guess what exception it could be?)

Methods:

`void newLine() throws IOException`

Example : Creating a CSV file

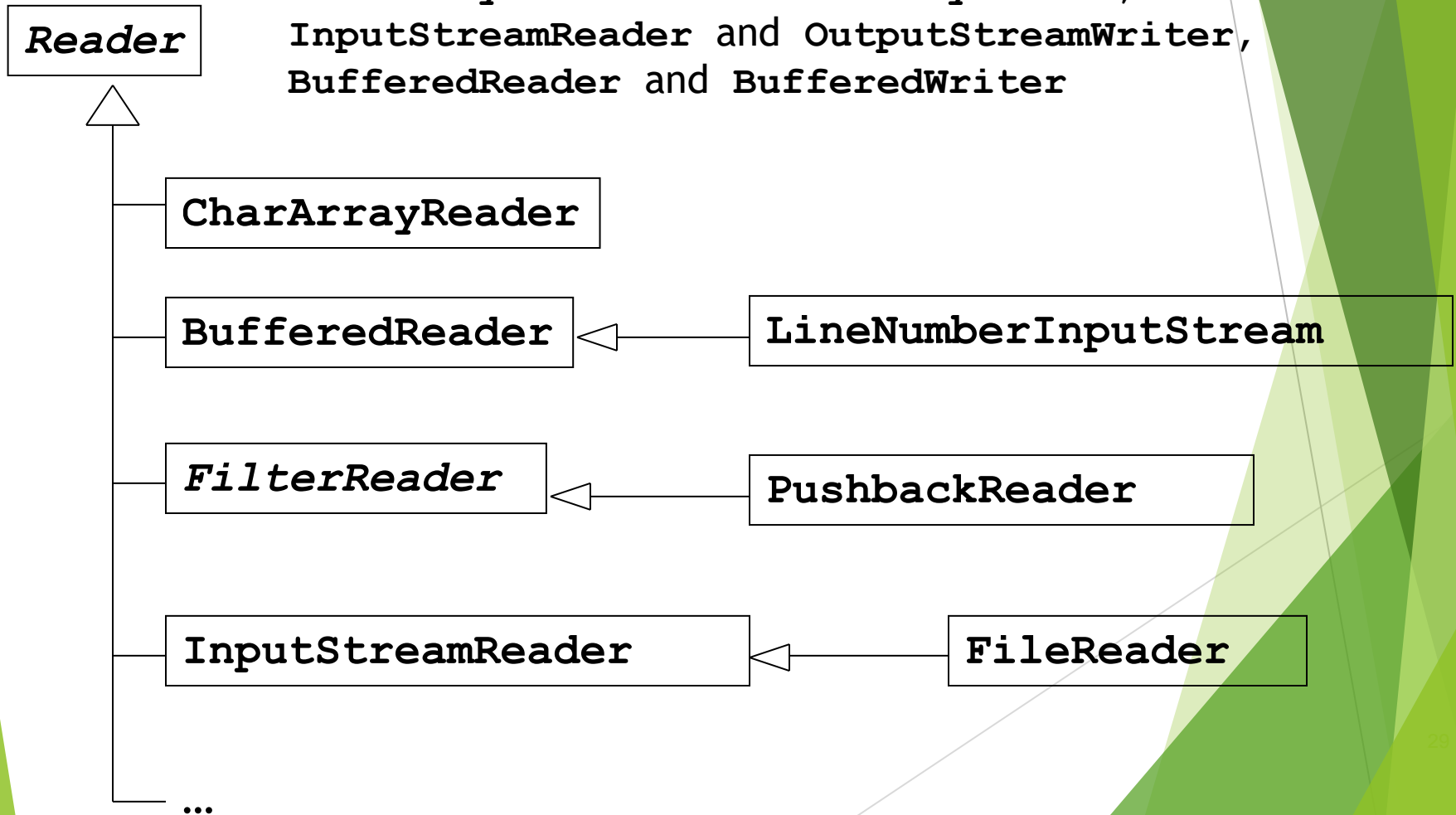
- ▶ This example creates a csv file by getting the inputs from the user.
- ▶ Note the use of separator while specifying absolute path.

```
import java.io.*;
public class CSVWrite {
public static void main(String args[]) {
    BufferedWriter f2=null;
try{
    f2= new BufferedWriter(new
    FileWriter("D:"+File.separator+"a.csv"));
    while(true){
        Console c= System.console();
        String s=c.readLine("Enter Name");
        f2.write(s+",");
    }
}
```

```
s=c.readLine("Enter ID");  
    f2.write(s+",");  
    f2.append(',');  
    s=c.readLine("Enter Degree");  
    f2.write(s+",");  
    f2.newLine();  
    s=c.readLine("do you want to continue(y/n)");  
    if(s.equalsIgnoreCase("n")) return;  
}    } catch(IOException ioe){}  
finally{  
    try{if(f2!=null)f2.close();}  
    catch(IOException e){}}  
}}
```

Hierarchy of character stream reader

Reader class hierarchy is very similar to that of the Writer hierarchy. `FileReader` and `FileWriter` form pairs where one can be used for reading the text which other has written. Similarly we have `CharArrayReader` and `CharArrayWriter`, `InputStreamReader` and `OutputStreamWriter`, `BufferedReader` and `BufferedWriter`



Reader

Reader is an abstract class for reading character streams.

Methods:

```
void close()
```

```
int read()
```

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

```
void mark(int readAheadLimit)
```

```
void reset()
```

- ▶ Marks the current position in the stream. When **reset()** is called after **mark()** the file pointer is positioned to the marked position.
- ▶ **readAheadLimit** is used to specify how many characters can be read further from the marked position so as to retain the marked position. If characters read is greater than what is specified in **readAheadLimit**, then calling reset does not position the file pointer in the marked position.

`long skip(long n)`

`boolean markSupported()`

`mark()` and `reset()` are optional methods that is not all implementing class need to provide the implementation for `mark()` and `reset()`. Therefore before they are used we must test if they are supported by the implementing class using `markSupported()`

All of the methods except `markSupported()` throw `IOException`.

CharArrayReader and

The data source for this class is a character array. Therefore reading happens from here.

CharArrayReader(char[] buf)

CharArrayReader(char[] buf, int offset, int length)

This class supports **mark()** and **reset()**

The data source for this class is a character array. Therefore reading happens from here.

StringReader(String s)

This class supports **mark()** and **reset()**

Example: using mark and reset

- ▶ This example reads from a string and prints “Right-Shift” when it encounters “>>” and “Greater-Than” when it encounters ‘>’.
- ▶ Note how we have read-ahead to see if the next symbol to > is again a >, used mark to go back and read if the symbol is not >.

```
import java.io.*;
class Expression {
public static void main(String[] s) throws IOException{
String s1="1>>2>3>4>>5;";
StringReader sw= new StringReader(s1);
int i;
while((i=sw.read())!=';'){
if(i=='>') {
sw.mark(1);
i=sw.read();
if(i=='>') System.out.print(" Right-Shift ");
else{ System.out.print(" Greater-Than ");
sw.reset();}}
else System.out.print((char)i);}}
```

1 Right-Shift 2 Greater-Than 3 Greater-Than 4 Right-Shift 5

PushbackReader

- ▶ This class allows characters to be pushed back into the stream. This is a wrapper class.
- ▶ This class supports **mark()** and **reset()**
- ▶ Constructor
 - ▶ **PushbackReader(Reader in)**
- ▶ Methods
 - ▶ **void unread(int c)**

Pushes back a character specified by **c** by copying it to the front of the pushback buffer. Next character that will be read is **c**.
 - ▶ **void unread(char[] cbuf)**
 - ▶ **void unread(char[] cbuf, int off, int len)**

Pushes back a char array or part of char array (of length **len** starting from **offset** off) by copying it to the front of the pushback
 - ▶ **long skip(long n)**

Places the file pointer after **n** characters.

Example:

► Same as the previous example using `PushbackReader`.

```
import java.io.*;
class TestWrite {
public static void main(String[] s) throws IOException{
String s1="1>>2>3>4>>5;";
StringReader sw= new StringReader(s1);
PushbackReader f = new PushbackReader(sw);
int i;
while((i=f.read())!=';'){
if(i=='>') {
i=f.read();
if(i=='>') System.out.print(" Right-Shift ");
else{ System.out.print(" Greater-Than ");
f.unread(i);}}
else System.out.print((char)i);}} }
```

Can you guess the result if `f.unread(i)` changed to `f.unread('#')`

InputStreamWriter

An **InputStreamWriter** is used as a bridge from character streams to byte streams:

Constructor:

InputStreamWriter(**InputStream** out)

- **InputStream** is the top-most class in the byte stream (like **Reader** is top-most class in the character stream)

Example:

```
Reader out = new new InputStreamReader(System.in) ;
```

System.in returns **InputStream** object.

FileReader

FileReader is subclass of **InputStreamWriter**

This class is used to read from a text file.

Constructors:

FileReader(File file) throws **FileNotFoundException**

FileReader(String fileName) throws **FileNotFoundException**

Either filename can be specified as a String or File object is passed to the **FileReader** constructor.

If the file specified by the name does not exist a **FileNotFoundException** is thrown

FileNotFoundException is a subclass of **IOException**

BufferedReader

- Reads text from a character-input stream by buffering characters for the efficient reading of characters, arrays, and lines.

Constructor:

BufferedReader(Reader in)

BufferedReader(Reader in, int sz)

The default buffer size is large enough for most purposes. In cases where more is required size can be specified.

Methods:

String readLine() throws IOException

This class supports **mark()** and **reset()**

LineNumberReader

This class is subclass of **BufferedReader**. So in addition to providing buffering it is also used to get and set the line number.

Constructor:

```
LineNumberReader(Reader in)
```

```
LineNumberReader(Reader in, int sz)
```

Methods:

```
int getLineNumber()
```

```
void setLineNumber(int lineNumber)
```

By default the line number begins from 0.

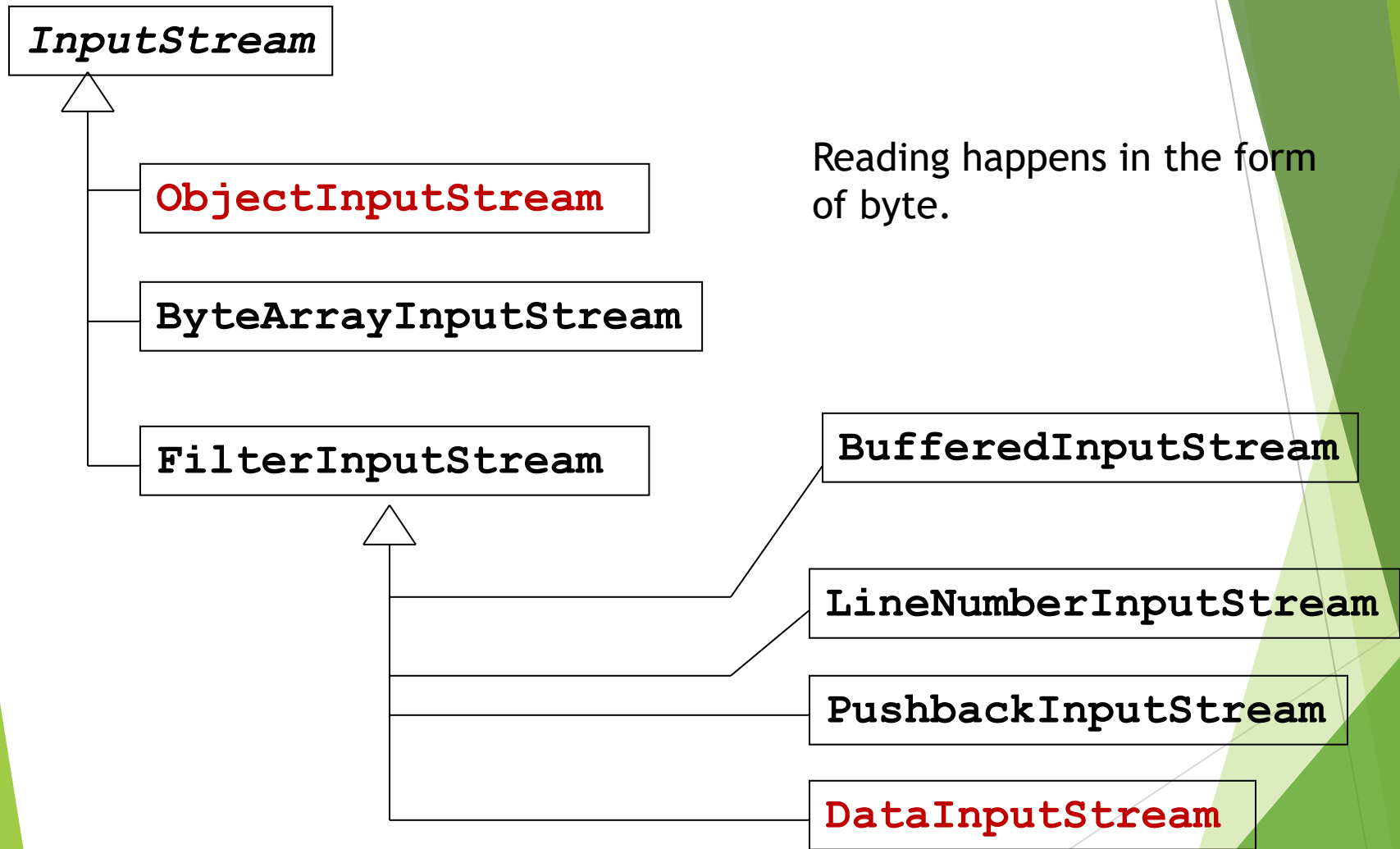
This class supports **mark()** and **reset()**

Example: Reading from a CSV file

Code reads from a CSV file and prints on the console

```
public static void main(String[] as) throws IOException{
    LineNumberReader f2=null;
    try{
        f2= new LineNumberReader(new
        FileReader("D:"+File.separator+"a.csv"));
        String s=null;
        while((s=f2.readLine())!=null){
            StringTokenizer st=new StringTokenizer(s,",");
            while(st.hasMoreElements()){
                System.out.print(f2.getLineNumber());
                System.out.print(" Name:" +st.nextElement());
                System.out.print("ID:" +st.nextElement());
                System.out.println("Degree:" +st.nextElement());
            } } catch(IOException ioe){}
        finally{
            try{if(f2!=null)f2.close();}
            catch(IOException e){}} } }
```


Hierarchy of byte stream



OutputStream



ObjectOutputStream

ByteArrayOutputStream

FilterOutputStream

DataOutput interface



BufferedOutputStream

ObjectOutputStream

PrintStream

Writing happens in the form of byte.

Example: using byte stream

- ▶ *Classes which are not in red in the last 2 slides are similar/parallel to character stream classes . Only difference is in place of char array we have byte array. So we end with an example for these classes.*
- ▶ *Example below copies the content of one file into another file.*

```
import java.io.*;
public class CopyFile {
public static void main(String[] args) throws
IOException {
File file1 = new File("D:"+File.separator+"read.txt");
File file2 = new
File("D:"+File.separator+"write.txt");
FileInputStream fin=null;
FileOutputStream fout=null;
```

```
try    {
fin = new FileInputStream(file1);
fout = new FileOutputStream (file2);
byte fileContent[] = new byte[(int)file1.length()];
fin.read(fileContent);
String strFileContent = new String(fileContent);
    fout.write(fileContent);
    System.out.println(strFileContent);
    }
    catch(FileNotFoundException e)    {
        System.out.println("File not found" + e);
    }
    catch(IOException ioe)    {
        System.out.println("Exception while reading the
file " + ioe);
    }
finally{
if(fin!=null)fin.close();
if(fout!=null)fout.close();}}}
```

DataInputStream and

- ▶ DataOutputStream
A data input stream and data output stream lets an application read and write primitive Java data types from an underlying input stream and output stream in a machine-independent way.
- ▶ An application uses a data output stream to write data that can later be read by a data input stream and vice versa.

DataOutputStream methods

Inherited from **OutputStream**

```
void write(int b)
void write(byte[] b, int off, int len)
void writeXxx(xxx v)
```

DataInputStream methods

Inherited from **InputStream**

```
int read(byte[] b)
int read(byte[] b, int off, int len)

xxx readXxx()
```

where xxx can be byte, short, int, long, char, float, double.

All the above methods throw **IOException**

Example: using DataInputStream and DataOutputStream

Example shows how primitive can be written and read using DataOutputStream and DataInputStream

```
import java.io.*;
class Test{
public static void main(String[] st) throws Exception{
    DataOutputStream out= new DataOutputStream(new
    FileOutputStream("a.txt"));
    int i=10;
    double d= 12.3;
    out.writeInt(i);
    out.writeDouble(d);
    out.close();
    DataInputStream in= new DataInputStream(new
    FileInputStream("a.txt"));
    System.out.println( in.readInt() );
    System.out.println( in.readDouble() );
    in.close();
} }
```

PrintStream

- ▶ **PrintStream** is a class that has functionality like the ability to print representations of various data values conveniently.
- ▶ **System.out** is an instance of **PrintStream**.
- ▶ Apart from this, two other functionalities that are provided here are:
 - A) Unlike other output streams, a **PrintStream** never throws an **IOException**
 - B) The **flush()** method can be made to automatically invoked after **println** method is invoked or newline (`'\n'`) is written.

PrintStream members

Constructors:

`PrintStream(File file)` throws `FileNotFoundException`

`PrintStream(OutputStream out, [boolean autoFlush])`

`PrintStream(String fileName)` throws `FileNotFoundException`

(The option in square brackets are optional)

`void print(XXX b)`

`void println(XXX b)`

where xxx is any primitive type, `String` or `Object`.

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

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

Both of the above methods have same functionality.

We have been using these methods extensively through `System.out`

Serialization

- ▶ The mechanism of storing the state of an object in the hard disk so that it can be restored later by your program.
- ▶ Serialization enables storing values of all instance variables which includes both primitives and **Serializable** objects.
- ▶ Serialization mechanism creates a file into which the state of the object is written.
- ▶ This file can later be read by the java program which can then restore the object's state.
- ▶ **ObjectOutputStream** and **ObjectInputStream** classes are used for these purposes. They are wrapper classes that take **OutputStream** and **InputStream** objects respectively

ObjectOutputStream & ObjectInputStream

► **ObjectOutputStream**

- `ObjectOutputStream(OutputStream out)` throws `IOException`
- `void writeXxx(XXX v)` where `xxx` is any primitive type, or `Object`
- `void write(int x)`
- And all the methods from `OutputStream`

► **ObjectInputStream**

- `ObjectInputStream(InputStream in)` throws `IOException`
- `XXX readXxx()` where `xxx` is any primitive type, or `Object`
- `int read()`
- And all the methods from `InputStream`

Steps to save and retrieve an object's state

Saving an object state

1. `FileOutputStream f= new
FileOutputStream("MySerFile.ser");`
2. `ObjectOutputStream obfile= new ObjectOutputStream(f);`
3. `obfile.writeObject(objectInstance);`
4. `Obfile.close();`

Retrieving an object state

1. `FileInputStream f= new FileInputStream("MySerFile.ser");`
2. `ObjectInputStream obfile= new ObjectInputStream(f);`
3. `Object o=obfile.readObject();`
4. `MyObject m=(MyObject)o;`
5. `Obfile.close();`

java.io.Serializable

- ▶ Only the objects which implement **Serializable** interface can be serialized.

```
class MyObject implements Serializable{... }
```

- ▶ **Serializable** is a marker interface.
- ▶ If object has references, then the references also must be either **Serializable** or should be marked **transient**.
- ▶ In JSE, some classes are not **Serializable**. For example **Thread** class, Subclasses of **Writer**, **Reader**, **InputStream**, **OutputStream**.
- ▶ All the collection classes, all primitive wrappers, **String**, **StringBuffer**, **StringBuilder** are **Serializable**
- ▶ If an attempt to serialize an object that does not implement **Serializable** is made, **NotSerializableException** is thrown.

transient

- ▶ Instance variables marked **transient** will not be saved.
- ▶ When object is de-serialized the **transient** variables are set to the default value based on their type.
- ▶ During serialization even the **private** state of the object is stored.
- ▶ Hence sensitive information like credit card number, password, a file descriptor contains a handle that provides access to an operating system resource must be marked transient.
- ▶ Also if a class contains references of object that cannot be serialized (like Thread), must be marked **Serializable** .

Example: Serialization

```
package general;

public abstract class Person
    implements Serializable{
    ...
}

import java.io.*;

public class SerializeP {

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

        Teacher f=new Teacher ("Tom");
```

```
//saving Teacher
```

```
ObjectOutputStream o=
```

```
    new    ObjectOutputStream(
```

```
    new    FileOutputStream("t.ser"));
```

```
o.writeObject(f);
```

```
o.close();
```

→ Could be any extension

```
// reloading the object state from file
```

```
ObjectInputStream in= new ObjectInputStream(
```

```
    new FileInputStream("t.ser"));
```

```
f=(Teacher )in.readObject();
```

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

```
in.close();
```

```
}}
```


Beware!

- ▶ You could save any number of object in a file
- ▶ Objects are read back in the same sequence as they are written.
- ▶ Care must be taken while de-serializing the objects.
 1. The objects must be cast into its correct type otherwise an **ClassCastException** will be thrown at runtime
 2. The objects must be retrieved in the same way as they are saved. For instance, if you save an integer using **writeInt()** then you must retrieve using **readInt()** method. Using **readObject()** and casting it back to **int** will not work(an **java.io.OptionalDataException** will be thrown at runtime)
- ▶ Safest and more common way to save and retrieved is to use **writeObject()** and **readObject()** methods

Properties

- ▶ Properties are strings stored as key-value pairs that are stored in a file.
- ▶ These values are generally used for configuration purpose like application startup parameter values, database configuration values or can be even used to standardize error messages
- ▶ The application reads the value of a property based on the key.
- ▶ For example

username **scott**

password **tiger**

could represent key-value pair for a database configuration file.

- ▶ Invariably most of the application need to read from this kind of file.

java.util.Properties

- ▶ Java provides a simple class called **Properties** that helps reading from and writing into property file.
- ▶ **Properties** class inherits from **Hashtable** class.
- ▶ *Call you recall methods of **Hashtable** class?*
- ▶ Using the methods of **Hashtable** is not advisable as they allow the insertions of key-value that are not strings.
- ▶ Instead methods provided in **Properties** class like **setProperty()**, **getProperty()** are to be used.
- ▶ The class has methods to load data from XML file as well. At this point we are not going to look at these methods.

Members of Properties

- ▶ `Properties()`
- ▶ `Properties(Properties defaults)`
- ▶ `void load(InputStream inStream)` throws `IOException`
- ▶ `void load(Reader reader)` throws `IOException`

Reads a key and element pairs from a character stream or byte stream.

- ▶ `Object setProperty(String key, String value)`
- ▶ `String getProperty(String key)`
- ▶ `String getProperty(String key, String defaultValue)`

`setProperty()` calls sets the key-value pair in the

`getProperty()` looks for the property with the specified key in this property list. If the key is not found in this file, the default property list, and its defaults, recursively, are then checked. The method returns null in case of the 1st `getProperty()` method and

- ▶ `void store(OutputStream out, String comments) throws IOException`
- ▶ `void store(Writer writer, String comments) throws IOException`

Write the key-values pairs in the property list to the byte/character stream.

Structure of a property file

- Properties are processed in terms of lines. There are 2 types of line (as specified by the API)
 - Natural line:
 - Line of characters terminated by `\n` or `\r` or `\r\n`
 - Example: blank line, comment line
 - Comment line begins within `#` or `!`
 - `# This is a property file`
 - Logical line:
 - Line that holds all the data of a key-element pair or
 - Multiple (natural) lines by escaping the line terminator sequence with a backslash character `\`. Any white space at the start of subsequent line is ignored. (example in next slide)

Key-Value pair in a logical line

- ▶ The key contains all of the characters in the line starting with the first non-white space character and up to, but not including, the first = or : or blank spaces.
- ▶ Examples of acceptable key-value pairs:
 - ▶ `flowers rose`
 - ▶ `flowers:rose, lily`
 - ▶ `flowers:rose,lily`
 - ▶ `flowers : rose`
 - ▶ `flowers=rose`
 - ▶ `flowers = rose`
 - ▶ `flowers rose, lily, \`
`lotus, orchid`

Example: Property file

```
import java.io.*;
import java.util.Properties;

public class PtyFile {
    public static void writeProperties() {
        FileWriter fileWriter = null;
        try{
            Properties props = new Properties();
            fileWriter = new FileWriter("D:"+
File.separator+"db.properties");
            props.setProperty("uname", "scott");
            props.setProperty("pwd", "tiger");
            props.store(fileWriter,"Database
credentials");
        }catch(IOException ioe){ ioe.printStackTrace(); }
        finally{ try{
            fileWriter.close();
        }catch(IOException ioe1){} }
    }
}
```

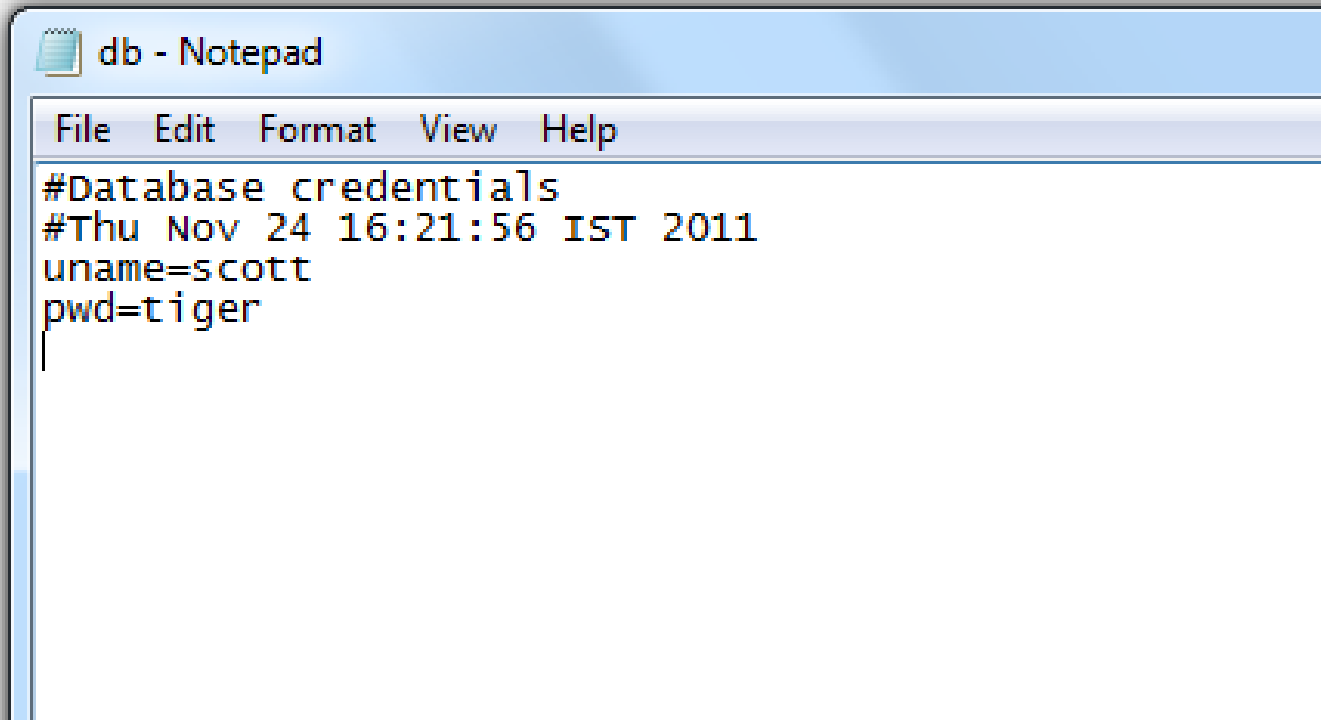


```
public static void main(String[] args) {
    //writeProperties();
    readProperties();
}

public static void readProperties() {
    FileReader fileReader = null;
    try{
        Properties props = new Properties();
        fileReader = new FileReader("D:"+
File.separator+"db.properties");
        props.load(fileReader);

System.out.println(props.getProperty("uname"));
System.out.println(props.getProperty("pwd"));
}catch(IOException ioe){ ioe.printStackTrace();}
finally{
try{    fileReader.close();
}catch(IOException ioe1){
    }
    }
}
```

Result



A screenshot of a Notepad window titled "db - Notepad". The window has a menu bar with "File", "Edit", "Format", "View", and "Help". The text inside the window is as follows:

```
#Database credentials
#Thu Nov 24 16:21:56 IST 2011
uname=scott
pwd=tiger
```

Result of execution of the code

```
scott
tiger
```

Resource bundles

- ▶ Resource bundle is also like properties files.
- ▶ But these are specifically used to store locale-specific information usually used for internationalization/globalization of messages.
- ▶ For every locale there could be separate resource bundle.
- ▶ For example, we could have 2 different files one for US (English) as `ResBun_en_US.properties` and another for France(french) as `ResBun_fr_FR.properties`.
- ▶ Two java classes are of help here
 - ▶ **ResourceBundle**
 - ▶ **Locale**

java.util.Locale

- ▶ This class is used to represents a specific geographical, political, or cultural region.
- ▶ The date format, for instance are different in different regions. In India, we have dd/mm/yyyy where as in US they follow mm/dd/yyyy.
- ▶ Application must be locale sensitive when it displays data so that users are comfortable.
- ▶ Constructor
 - ▶ **Locale(String language)**
 - ▶ **Locale(String language, String country)**
 - ▶ **Locale(String language, String country, String variant)**
- ▶ Language argument should be ISO Language Code(<http://www.loc.gov/standards/iso639-2/englangn.html>)
- ▶ Country argument should be ISO Country (<http://www.iso.ch/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html>)

Members of Locale

- ▶ Static constant that return local for many language and countries (some of them are listed here)
 - ▶ `static final Locale ENGLISH`
 - ▶ `static final Locale FRENCH`
 - ▶ `static final Locale GERMAN`
 - ▶ `static final Locale CHINESE`
 - ▶ `static final Locale JAPANESE`
 - ▶ `static final Locale US`
 - ▶ `static final Locale UK`
 - ▶ `static final Locale GERMANY`
 - ▶ `static final Locale CHINA`
 - ▶ `static final Locale JAPAN`
- ▶ Get methods like
 - ▶ `String getCountry()`
 - ▶ `String getLanguage()`
 - ▶ `static Locale getDefault()` which gets the current value of the default locale for this instance of the Java Virtual Machine.

ResourceBundle

- ▶ This is **abstract** class.
- ▶ This class allows working with the resource bundle files with the help of **Locale** class.
- ▶ The resource bundle files are to be named in specific manner.
 - ▶ `baseName + "_" + language1 + "_" + country1`
 - ▶ `baseName + "_" + language1`
 - ▶ `baseName`
 - ▶ Extension could be either `.properties` or `.class`.
- ▶ Example:
 - ▶ `ResBun_en_US.properties`
 - ▶ `ResBun_fr_FR.properties`
 - ▶ `ResBun_en_US.class`

Members of

- ▶ There are many overloaded **getBundle()** methods in this class of which we are looking at only two in this session:

ResourceBundle

- ▶ **static final ResourceBundle getBundle(String baseName)**
 - ▶ Gets the default locale specific resource bundle using the specified base name
- ▶ **static final ResourceBundle getBundle(String baseName, Locale locale)**
 - ▶ Gets the **locale** specific resource bundle using the specified base name

These methods throw a **MissingResourceException** when the file is not found

- ▶ **Locale getLocale()**
 - ▶ This method is usually called after **getBundle()** to make sure that the resource bundle returned was really corresponds to the requested locale or is a fallback.

Example:

ResourceBundle

- ▶ GreetResourceBundle.properties

GoodMorning=Good Morning

Goodbye=Good Bye

- ▶ GreetResourceBundle fr FR.properties

GoodMorning=Bonjour

Goodbye=Au revoir


```
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.MissingResourceException;
public class ResBundle {
    public static void main(String [] argv) {
        try {
            ResourceBundle rb1 =
ResourceBundle.getBundle("GreetResourceBundle");
            System.out.println(rb1.getLocale());
            System.out.println(rb1.getString("GoodMorning"));
            System.out.println(rb1.getString("Goodbye"));
            Locale frenchLocale = new Locale("fr", "FR");
            ResourceBundle rb =
ResourceBundle.getBundle("GreetResourceBundle",
frenchLocale);
            System.out.println(rb.getString("GoodMorning"));
            System.out.println(rb.getString("Goodbye"));
        } catch (MissingResourceException mre) {
            mre.printStackTrace();    }    }}
```

RandomAccessFile

- ▶ This class supports both reading and writing to a file simultaneously.
- ▶ A file pointer is maintained which can be read by the **getFilePointer()** method and set by the **seek()** method.
- ▶ Constructor:
 - ▶ **RandomAccessFile(File file, String mode)**
- ▶ Methods:
 - ▶ **String readLine()**
 - ▶ **void writeBytes(String s)**
 - ▶ **XXX readXXX()** where **XXX** represents all primitive type.
 - ▶ **XXX writeXXX()** where **XXX** represents all primitive type.
 - ▶ **void seek(long pos)**
 - ▶ **long length()**
 - ▶ **long getFilePointer()**

Modes

- ▶ **r**: Open for reading only. Invoking any of the write methods of the resulting object will cause an **IOException** to be thrown.
- ▶ **rw**: Open for reading and writing. If the file does not already exist then an attempt will be made to create it.
- ▶ **rws**: Same as **rw**, and also require that every update to the file's content or metadata be written synchronously to the underlying storage device.
- ▶ **rwd**: Same as **rw**, and also require that every update to the file's content be written

Example:

Code that replaces : by ;

```
import java.io.*;
public class Semi {
    public static void main(String str[]) {
        try {
            File file = new File("Test1.java");
            RandomAccessFile raf = new RandomAccessFile(file, "rw");
            String s="";
            long fp=raf.getFilePointer();
            while((s= raf.readLine())!=null){
                System.out.println("line: " +s);
                if(s.contains(":")){
                    s= s.replace(':', ';');
                    raf.seek(fp);
                    raf.writeBytes(s); }
                fp=raf.getFilePointer();
            } catch (IOException e){
                e.printStackTrace();}
        }
    }
}
```