



# Input and Output

---

## Input and Output



# File Input and Output

## The File Class

---

- **The File Class**
- The `java.io.File` class represent the name of a file or directory that might exist on the host machine's file system. For example, you use the File class to find out when a file was last modified or to remove or rename the file.
- The stream classes are concerned with the contents of the file, whereas the File class is concerned with the storage if the file on a disk.
- **constructors**
- `File (String pathname)`
- `File(String dir, String subpath);`
- `File(File dir, String subpath)`



# File Input and Output

## The File Class

---

```
■ import java.io.*;
■ public class FileNav{
■     public static void main(String argv[]){
■         String[] filenames;
■         File f = new File(".");
■         filenames = f.list();
■         for(int i=0; i< filenames.length; i++)
■             System.out.println(filenames[i]);
■     }
■ }
```



# File Input and Output

## The File Class

---

- Another example is File



# File Input and Output

## The File Class

---

- It is important to know that constructing an instance of File does not create a file on the local file system. Remember that the File class can represent a directory as well as a file.



# File Input and Output

## The File Class

---

- The program listed below uses some of methods to create a recursive listing of a directory. The application expects the directory to be specified in the command line.
- `import java.io.*;`
- `public class FindDirectories`
- `{`
- `public static void main(String[] args)`
- `{`
- `// if no arguments provided, start at the`  
`parent directory`
- `if (args.length == 0) args = new String[]`  
`{ ".." };`
- `try`
- `{`



# File Input and Output

## The File Class

---

- `//Make sure path exists and is a directory`
- `File f=new File(path);`
- `if(!f.isDirectory()) {`
- `System.out.println(path+"doesn't exist or not dir");`
- `System.exit(0);`
- `}`
- `//Recursively list contents`
- `Lister lister=new Lister(f);`
- `lister.setVisible(true);`
- `}`



# File Input and Output

## The File Class

```
File pathName = new File(args[0]);  
String[] fileNames = pathName.list();  
// enumerate all files in the directory  
for (int i = 0; i < fileNames.length; i++)  
{  
    File f = new File(pathName.getPath(),  
        fileNames[i]);  
    // if the file is again a directory, call  
    // the main method recursively  
    if (f.isDirectory())  
    {  
        System.out.println(f.getCanonicalPath());  
        main(new String [] { f.getPath() });  
    }  
}
```

Written by Paul Pu All Rights  
Reservedwww.torontocollege.com



# File Input and Output

## The File Class

---

```
■ catch (IOException e)
■     {
■         e.printStackTrace();
■     }
■ }
■ }
```



# File Input and Output

## The File Class

---

- `import java.io.*;`
- `public class CreateNewFile {`
- `public static void main(String args[]){`
- `File f1=new File("IamnewFile");`
- `System.out.println(f1.getAbsolutePath());`
- `System.out.println("File exists:"+f1.exists());`
- `try {`
- `f1.createNewFile();`
- `} catch(IOException e) { }`
- `System.out.println("File exists:"+f1.exists());`
- `System.out.println("Is a directory"+f1.isDirectory());`
- `System.out.println("Is a file"+f1.isFile());`
- `System.out.println("Last modify:"+f1.lastModified());`
- `}`



# File Input and Output

## The File Class

---

- `f1.createNewFile();`
- `} catch(IOException e) { }`
- `System.out.println("File exists:"+f1.exists());`
- `System.out.println("Is a directory"+f1.isDirectory());`
- `System.out.println("Is a file"+f1.isFile());`
- `System.out.println(" Last modify:"+f1.lastModified());`
- `f1.setReadOnly();`
- `File f2=new File("IamnewDIR");`
- `System.out.println(f2.getAbsolutePath());`
- `System.out.println("File exists:"+f2.exists());`
- 
- `f2.mkdir();`



# File Input and Output

## The File Class

---

- `System.out.println("File exists:"+f2.exists());`
- `System.out.println("Is a directory"+f2.isDirectory());`
- `System.out.println("Is a file"+f2.isFile());`
- `System.out.println(" Last modify:"+f2.lastModified());`
  
- `}`
- `}`



## File Input and Output

# The RandomAccessFile

---

- One way to read or modify a file is to use the java.io. RandomAccessFile class. This class presents a model of files that is incompatible with the stream/reader/writer model.
- With a random-access file, you can seek to a desired position within a file, and then read or write a desired amount of data. The RandomAccessFile class provides methods that support seeking, reading, and writing.
- Two constructors :
- RandomAccessFile(String file,String mode)
- RandomAccessFile(File file,String mode)
- The mode string should be either "r" or "rw"
-



## File Input and Output

# The RandomAccessFile

---

- After a random-access file is constructed, you can seek to any byte position within the file and then read or write.
- The methods that support seeking are:
- `long getFilePointer()` throws `IOException`” This returns the current position within the file, in bytes.
- `Long length()` throws `IOException`: This returns the length of the file, in bytes.
- `Void seek(long position)` throws `IOException`: This sets the current position within the file, in bytes. File start at position 0.



## File Input and Output

# The RandomAccessFile

---

- The common methods are:
- `int read()` throws `IOException`: This returns the next byte from the file or `-1` if at end of file.
- `int read(byte dest[])` throws `IOException`: This attempts to read enough bytes to fill array `dest[]`. It returns the number of bytes read, or `-1` if the file was at end of file
- `int read(byte dest[],int offset,int len)` throws `IOException`
- This attempts to read `len` bytes into array `dest[]`, starting at `offset`. It returns the number of bytes read, or `0` if the file was at end of file.
- It has same write methods too.



File Input and Output

# The RandomAccessFile

---

- RandomAccessFile has methods for Primitive Types, please check java API.



File Input and Output

# The RandomAccessFile

---

- Example: RandomFileTest.java



## File Input and Output Streams, Readers, and Writers

---

- Java's stream, reader, and writer classes view input and output as ordered sequences of bytes.
- A low-level output stream receives bytes and writes bytes to an output device
- A high-level filter output stream receives general-format data, such as primitives, and writes bytes to a low-level output stream or to another filter output stream
- A writer is similar to a filter output stream but is specialized for writing Java strings in units of Unicode characters.



## File Input and Output Streams, Readers, and Writers

---

- A low-level input stream reads bytes from an input device and returns bytes to its caller
- A high-level filter input stream reads bytes from a low-level input stream, or from another filter input stream, and returns general format data to its caller



## File Input and Output

# Low-Level Streams

---

- Low-Level input Streams have methods that read input and return the input as bytes. Low-Level output Streams have methods that are passed bytes, and write thee bytes as output. The FileInputStream and FileOutputStream classes are excellent examples.



# File Input and Output

## High-Level Streams

---

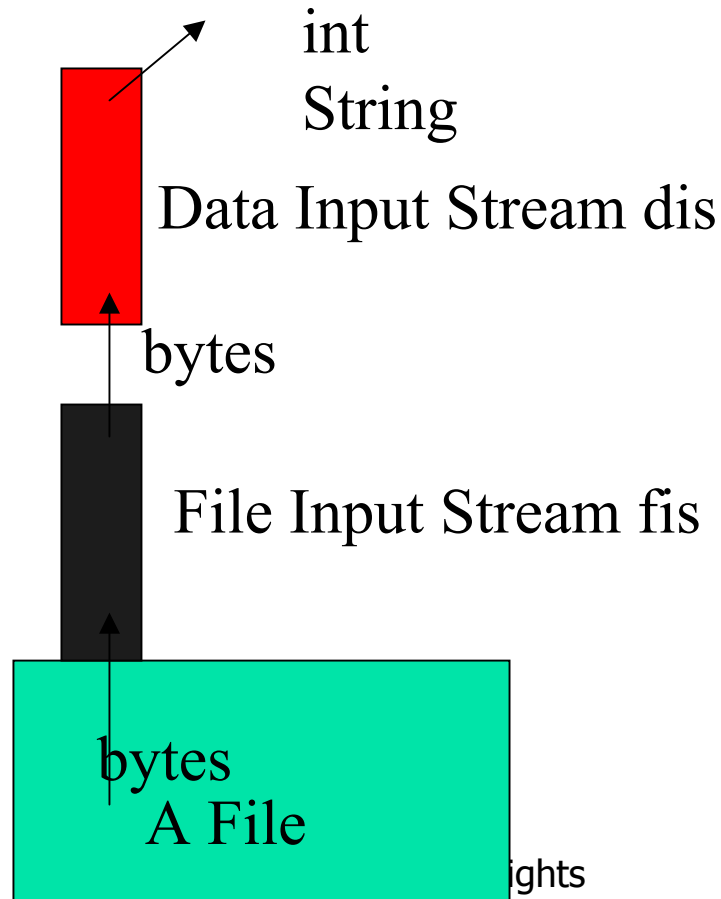
- It is all well to read bytes from input devices and write bytes to output devices. However, more often than not the bytes to be read or written constitute higher-level information such as an int or a string
- Java supports high-level I/O with high-level streams. A good example of a high-level input stream is the data input stream.
- `DataInputStream(InputStream instrem)`
- `DataOutputStream(OutputStream outputStream)`



# File Input and Output

## High-Level Streams

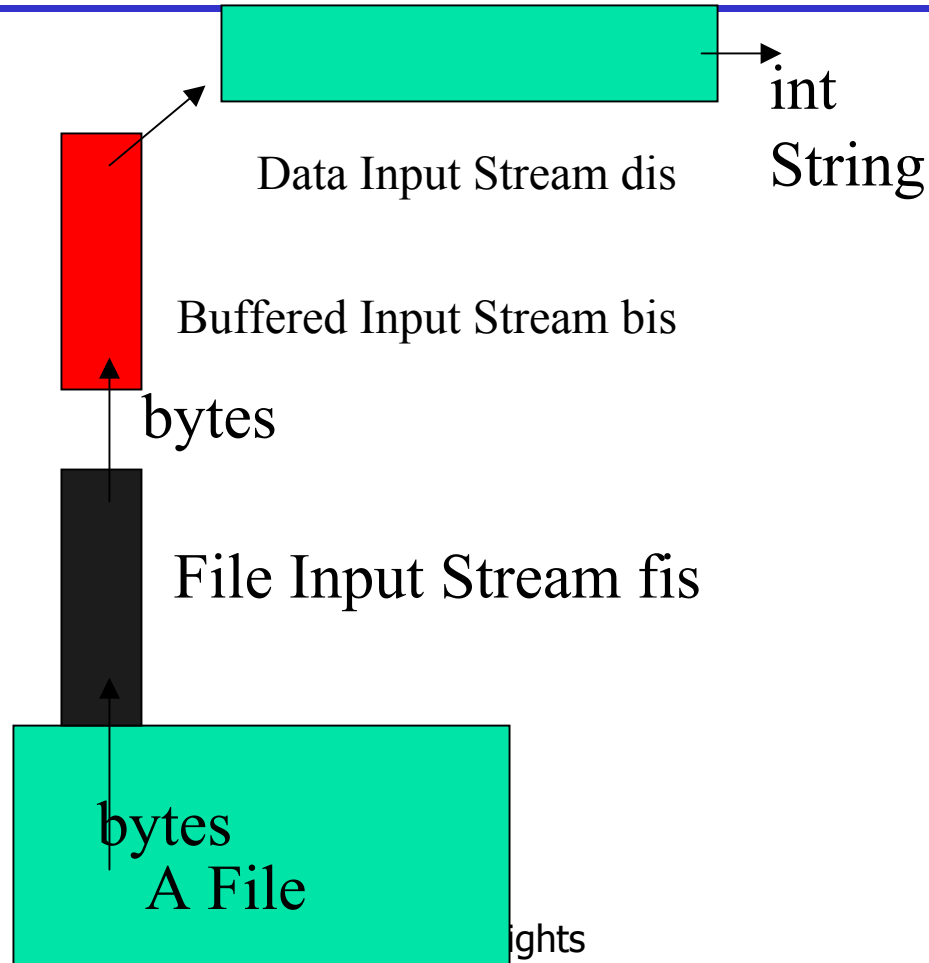
---





# File Input and Output

## High-Level Streams





# File Input and Output

## High-Level Streams

---

- `import java.io.*;`
- `public class BufIn{`
- `public static void main(String`  
`argv[]) {`
- `try{`
- `FileInputStream`  
`fin = new`  
`FileInputStream("BufIn.java");`
- `BufferedInputStream bin = new`  
`BufferedInputStream(fin);`



# File Input and Output

## High-Level Streams

---

- `//Read the file`
- `import java.io.*;`
- `public class Dis{`
- `public static void main(String argv[]){`
- `try{`
- `FileInputStream fis= new`  
`FileInputStream("fos.dat");`
- `DataInputStream dis = new`  
`DataInputStream(fis);`
- `System.out.println(dis.readChar());`
- `}catch (IOException`  
`e) {System.out.println(e.getMessage());}`
- `} Written by Paul Pu All Rights`  
`Reservedwww.torontocollege.com`
- `}`



# File Input and Output

## High-Level Streams

---

- `//Write the file`
- `import java.io.*;`
- `public class Dos{`
- `public static void main(String argv[]){`
- `try{`
- `FileOutputStream fos = new`
- `FileOutputStream("fos.dat");`
- `DataOutputStream dos = new`
- `DataOutputStream(fos);`
- `dos.writeChar('J');`
- `}catch(IOException`
- `e){System.out.println(e.getMessage());}`
- `}`
- `}`



# File Input and Output

## High-Level Streams

---

```
■ import java.io.*;
■ public class BufIn{
■     public static void main(String argv[]){
■         try{
■             FileInputStream fin = new
FileInputStream("BufIn.java");
■             BufferedInputStream bin = new
BufferedInputStream(fin);
■             int ch=0;
■             while((ch=bin.read())> -1){
■                 StringBuffer buf = new
StringBuffer();
■                 buf.append((char)ch);
■             }
■         }
■     }
```





# Reading from standard input

---

- `// How to read from standard input.`
- `import java.io.*;`
- `public class Echo {`
- `public static void main(String[] args) {`
- `DataInputStream in =`
- `new DataInputStream(`
- `new BufferedInputStream(System.in));`
- `String s;`
- `try {`
- `while((s = in.readLine()).length() != 0)`
- `System.out.println(s);`
- `// An empty line terminates the program`
- `} catch(IOException e) {`
- `e.printStackTrace();`
- `}`
- `}`
- `} ///:~`

Written by Paul Pu All Rights Reserved  
www.torontocollege.com



# Readers and Writers

---

- Readers and Writers are like input and output streams. What makes readers and writers different is that they are exclusively oriented to Unicode characters.
- The low-level classes communicate with I/O devices, while the high-level classes communicate with low-level classes.
- Low-level reader and writer class:
  - FileReader
  - FileWriter
  - PipedReader: These classes provide a mechanism for thread communication
  - StringReader and StringWriter: These classes read and write strings



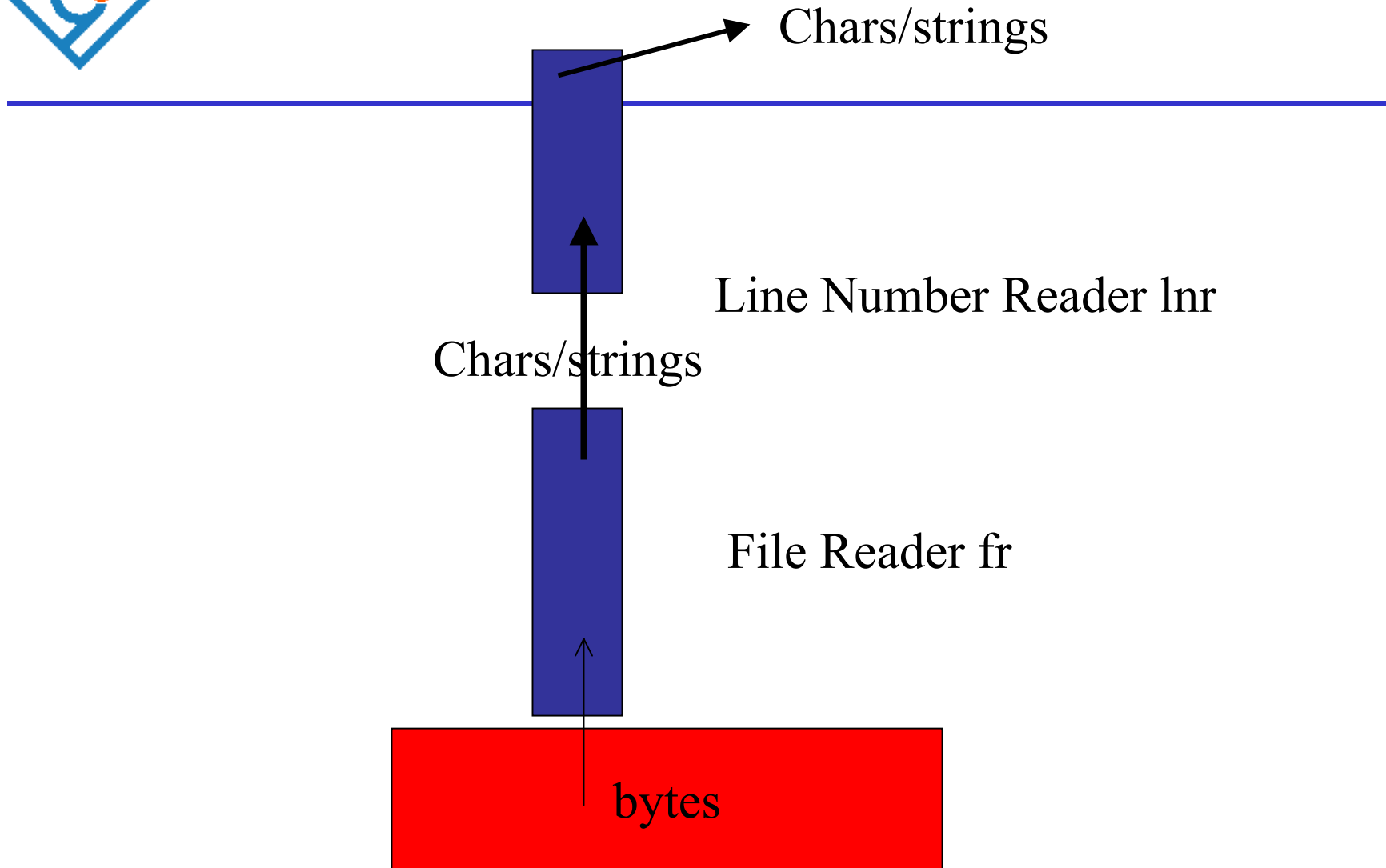
# Readers and Writers

---

- The high-level readers and writers all inherit from the Reader or Writer superclass. As with high-level streams, when you construct a high-level reader or writer you pass in the next-lower object in the chain.
- The high-level classes:
  - `BufferedReader` and `BufferWriter`: These classes have internal buffers so that data can be read or written in large blocks. They are similar to buffered input streams and buffered output streams.
  - `PrintWriter`: This class is similar to `PrintStream`, but it writes chars rather than bytes.
  - `InputStreamReader` and `OutputStreamWriter`: These classes convert between streams of bytes and sequences of Unicode characters.



# Readers and Writers





# Readers and Writers example

---

- `import java.io.*;`
- `public class TestWriter {`
- `public static void main(String[] args) {`
- `PrintWriter out;`
- `String name="Harry Hacker";`
- `double salary=75000;`
- `String name1="Tom Avon";`
- `double salary1=80000;`



# Readers and Writers example

---

- try{
- out= new PrintWriter(new FileWriter("employee.txt"));
  
- out.print(name);
- out.print(' ');
- out.println(salary);
- out.print(name1);
- out.print(' ');
- out.println(salary1);
- out.close();
- } catch(IOException e){}
- }
- }



# Readers and Writers example

---

- `import java.io.*;`
- `public class TestReader {`
  
- `public static void main(String[] args) {`
- `String line;`
- `BufferedReader in;`
- `try{`
- `in= new BufferedReader(new FileReader("employee.txt"));`
  
- `while ((line=in.readLine()) !=null){`
- `System.out.println(line);`
- `}`
- `} catch(IOException e){}`
- `}`
- `}`



## StringTokenizer and Delimited Text

---

- The **StreamTokenizer** class is used to break any **InputStream** into a sequence of “tokens,” which are bits of text delimited by whatever you choose. For example, your tokens could be words, and then they would be delimited by white space and punctuation.



# StringTokenizer and Delimited Text

---

- `public void readStudentFile(String fileName) throws IOException{`
- `Student student;`
- `BufferedReader in=new BufferedReader(new FileReader("data.txt"));`
- `int c;`
- `while((c = in.read()) != -1) {`
-



# StringTokenizer and Delimited Text

- `String s=in.readLine();`

---

- `StringTokenizer t=new StringTokenizer(s,"/");`
- `while (t.hasMoreTokens()) {`
- `String firstName=t.nextToken();`
- `String lastName=t.nextToken();`
- `String major=t.nextToken();`
- `student=new`  
`Student(firstName,lastName,major,Double.parseDouble(t`  
`.nextToken()),Double.parseDouble(t.nextTok`  
`en()));`
- `putStudent(student);`
- `}`
- `}`



# StringTokenizer and Delimited Text

---

- Example: DataFileTest.java



# Object Stream

---

Low-level streams, whether they are connected to disk files or to networks, provide byte-oriented I/O. What is missing is a way to read and write general Java Objects. This functionality is provided by object streams.

## **Write Object:**

```
fos=new FileOutputStream(file);  
os=new ObjectOutputStream(fos);  
os.writeObject(o);  
os.close();
```



# Object Stream

---

- Read Object:
- `fis=new FileInputStream(file);`
- `ois=new ObjectInputStream(fis);`
- `o=ois.readObject();`
- `ois.close();`



# Object Stream

```
■ import java.io.*;
■ public class Saver {
    ■ public boolean save(Object o, String file) {
    ■     boolean status=true;
    ■     FileOutputStream fos;
    ■     ObjectOutputStream os;
    ■     try {
    ■         fos=new FileOutputStream(file);
    ■         os=new ObjectOutputStream(fos);
    ■         os.writeObject(o);
    ■         os.close();
    ■     } catch (Exception ex) {
    ■         status=false;
    ■         System.out.println("Save Error" +ex);
    ■     }
    ■     return status;
    ■ }
```



# Object Stream

- public Object load(String file) {

---

- FileInputStream fis;
- ObjectInputStream ois;
- Object o=null;
- try {
- fis=new   FileInputStream(file);
- ois=new   ObjectInputStream(fis);
- o=ois.readObject();
- ois.close();
- } catch (Exception ex) {
  
- System.out.println("Load Error:"+ex);
- }
- return o;
  
- }
  
- }

Written by Paul Pu All Rights  
Reservedwww.torontocollege.com



# Object serialization

---

- Java 1.1 has added an interesting feature called *object serialization* that allows you to take any object that implements the **Serializable** interface and turn it into a sequence of bytes that can later be restored fully into the original object. This is even true across a network, which means that the serialization mechanism automatically compensates for differences in operating systems. That is, you can create an object on a Windows machine, serialize it, and send it across the network to a Unix machine where it will be correctly reconstructed. You don't have to worry about the data representations on the different machines, the byte ordering, or any other details.



# Object serialization

---

- By itself, object serialization is interesting because it allows you to implement *lightweight persistence*. Remember that persistence means an object's lifetime is not determined by whether a program is executing—the object lives *in between* invocations of the program. By taking a serializable object and writing it to disk, then restoring that object when the program is reinvoked, you're able to produce the effect of persistence. The reason it's called "lightweight" is that you can't simply define an object using some kind of "persistent" keyword and let the system take care of the details (although this might happen in the future). Instead, you must explicitly serialize and de-serialize the objects in your program.



# Object serialization

---

- Object serialization was added to the language to support two major features. Java 1.1's *remote method invocation* (RMI) allows objects that live on other machines to behave as if they live on your machine. When sending messages to remote objects, object serialization is necessary to transport the arguments and return values.



# Object serialization

---

- Object serialization is also necessary for Java Beans, introduced in Java 1.1. When a Bean is used, its state information is generally configured at design time. This state information must be stored and later recovered when the program is started; object serialization performs this task.



# Object serialization

---

- Serializing an object is quite simple, as long as the object implements the **Serializable** interface (this interface is just a flag and has no methods). In Java 1.1, many standard library classes have been changed so they're serializable, including all of the wrappers for the primitive types, all of the container classes, and many others.



# Object serialization

---

```
■ class Employee implements Serializable
■ {
■     public Employee() {}

■     public Employee(String n, double s,
■         int year, int month, int day)
■     {
■         name = n;
■         salary = s;
■         GregorianCalendar calendar
■             = new GregorianCalendar(year, month - 1, day);
■         // GregorianCalendar uses 0 for January
■         hireDay = calendar.getTime();
■     }
```

■  
Written by Paul Pu All Rights Reserved  
www.torontocollege.com



# Object serialization

---

```
■ public String getName()  
■     {  
■         return name;  
■     }  
  
■ public double getSalary()  
■     {  
■         return salary;  
■     }  
  
■ public Date getHireDay()  
■     {  
■         return hireDay;  
■     }
```



# Object serialization

```
■ public void raiseSalary(double byPercent)
■     {
■         double raise = salary * byPercent / 100;
■         salary += raise;
■     }

■ public String toString()
■     {
■         return getClass().getName()
■             + "[name=" + name
■             + ",salary=" + salary
■             + ",hireDay=" + hireDay
■             + "];"
■     }

■ private String name;
■ private double salary;
■ private Date hireDay;
■ }
```



# Object serialization

---

- Example: ObjectRefTest.java
- Example: ObjectFileTest.java