Files

- File input and output
  - Storing data in files
  - Reading data from files

Files

- It's useful to store data between applications
- Consider an application that deal with student data
- Typing in this data each time the program starts is clearly not ideal!
- Similarly, writing the data in the .java file means that the end user cannot change the data

Files

- Instead, data can be stored in files
- There are two uses for files:
  - Input files that are read by your program
  - Output files that have data written by your program
  - A file can be both an input file and an output file
- There are two types of file
  - Binary Files
  - Text files

Writing data to a file

- To write data to a file, you must create an instance of the ***PrintWriter*** class
- To use the PrintWriter import *java.io.PrintWriter.*
- *Create an instance of PrintWriter*
- The file name is the first constructor argument

```
PrintWriter writer = new PrintWriter("File.txt");
```

Writing data to a file

- When opening a file, there is potential for problems:
  - The file may not be accessible due to permissions
  - The file name/location may be invalid. e.g. reading from a disk which doesn't exist

```
PrintWriter writer = new PrintWriter("#fg://///:~");
```

Writing to a file

- Because of this, you must construct the object inside a try/catch block

```
try {
    PrintWriter writer = new PrintWriter("File.txt");
}
catch (Exception e) {
        System.out.println("The file could not be opened");
}
```

Writing to a file

- Warning: If you supply a file name that already exists it will be erased and replaced with a new file

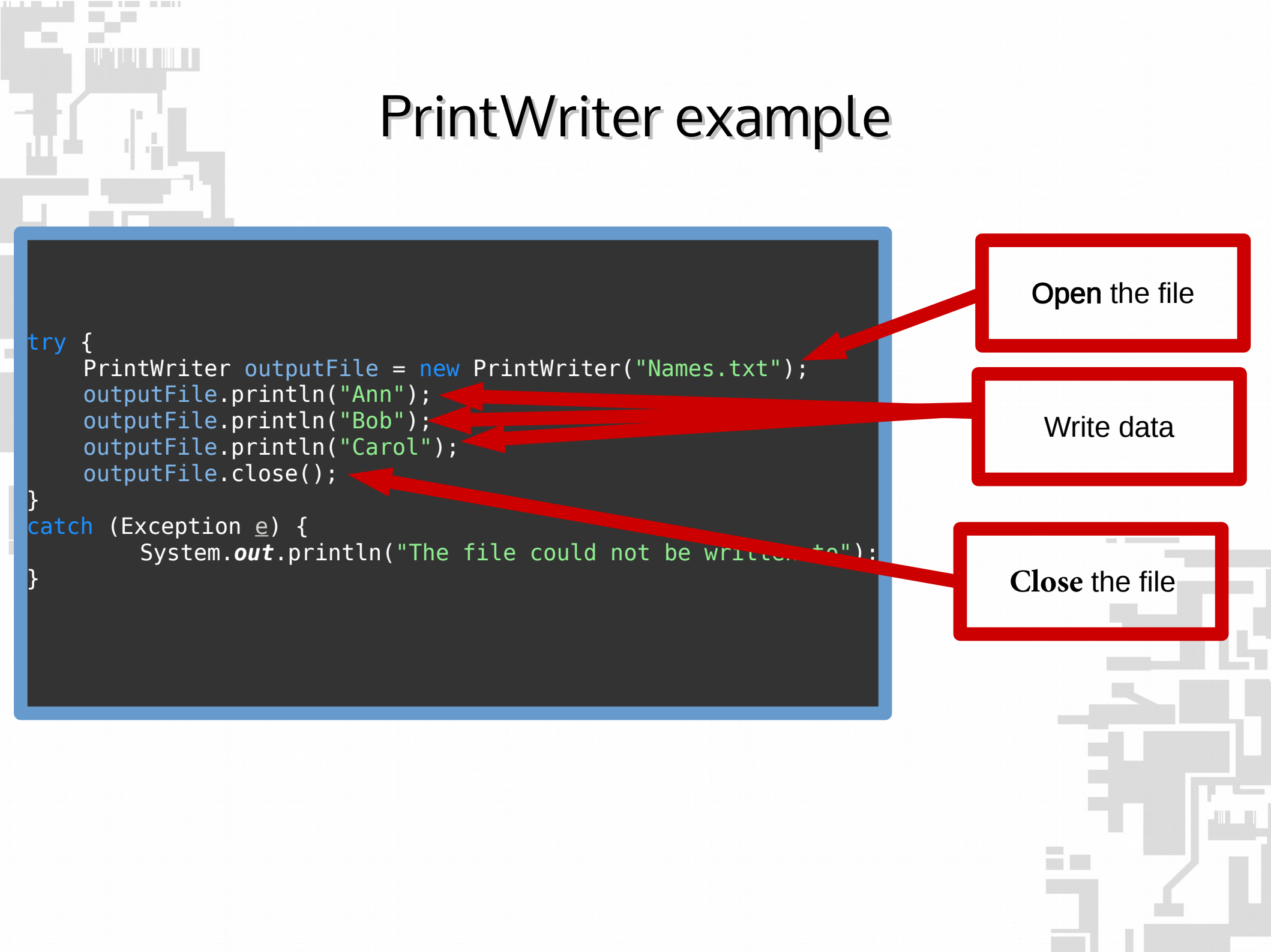
```
try {
    PrintWriter writer = new PrintWriter("File.txt");
}
catch (Exception e) {
        System.out.println("The file could not be opened");
}
```

The PrintWriter class

- The PrintWriter class allows you to write data to a file using the **print(str)** and **println(str)** methods that you have also been using with **System.out**
- Just as with System.out, the **println()** method of the **PrintWriter** class will places a newline character after the written data
- The **print()** method writes data without writing the newline character

PrintWriter example

```
try {
    PrintWriter outputFile = new PrintWriter("Names.txt");
    outputFile.println("Ann");
    outputFile.println("Bob");
    outputFile.println("Carol");
    outputFile.close();
}
catch (Exception e) {
        System.out.println("The file could not be written to");
}
```



PrintWriter example

- You must encase all file operations in the try/catch block as the file may be readable but not writeable so may fail when close
- The data is not written to the file until close is called

```
try {
    PrintWriter outputFile = new PrintWriter("Names.txt");
    outputFile.println("Ann");
    outputFile.println("Bob");
    outputFile.println("Carol");
    outputFile.close();
}
catch (Exception e) {
        System.out.println("The file could not be written to");
}
```

PrintWriter example

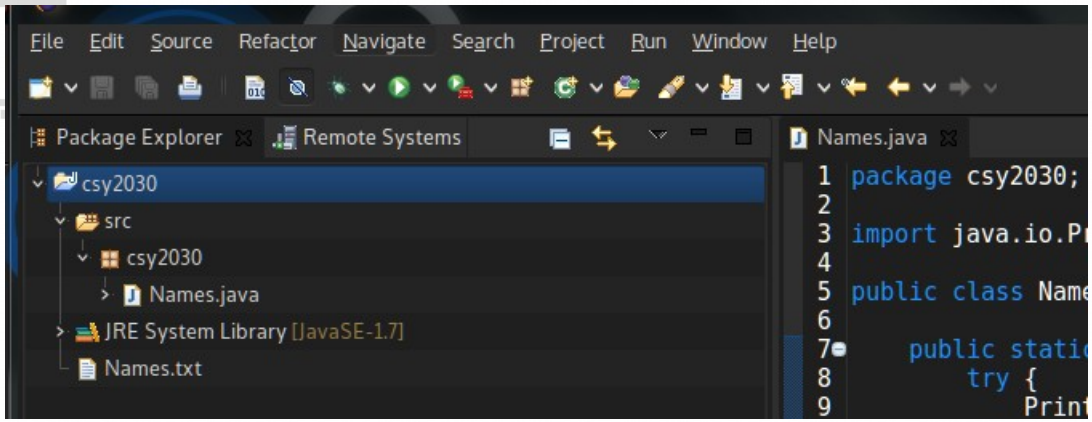
- If close() is never called, the data will not be saved!

```
try {
    PrintWriter outputFile = new PrintWriter("Names.txt");
    outputFile.println("Ann");
    outputFile.println("Bob");
    outputFile.println("Carol");
}
catch (Exception e) {
        System.out.println("The file could not be written to");
}
```

- When this runs, the file will be created but no data will be stored inside it

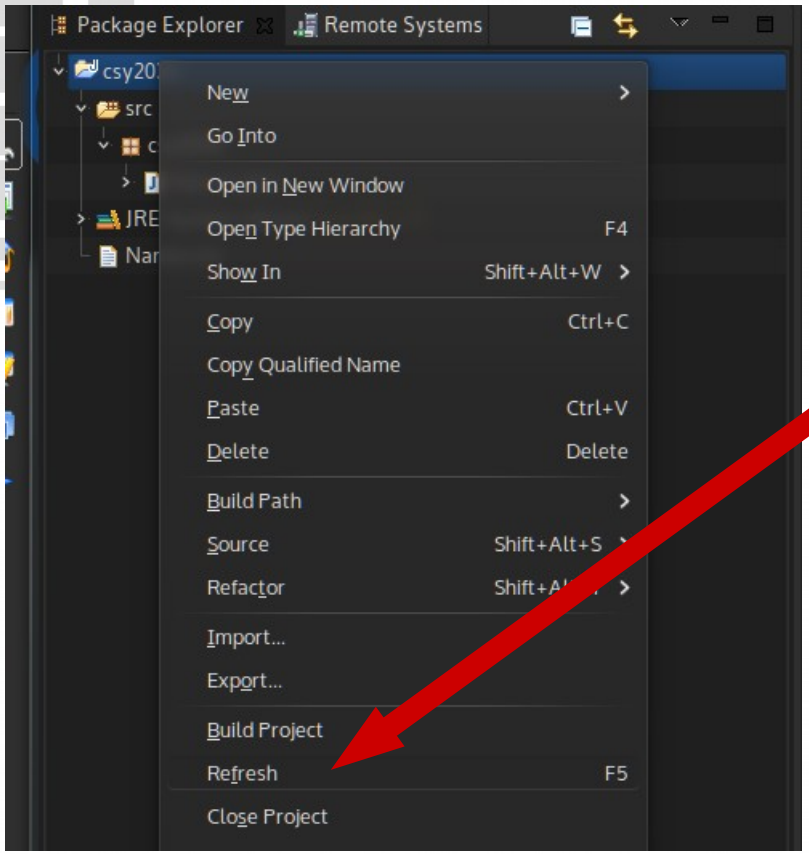
PrintWriter class

- When using Eclipse, files will get created at the top level (above the /src/ directory) of the project. For example after running the names example:



PrintWriter

- In Eclipse, the newly created file does not appear immediately even though the file has been created
- To show the file in Eclipse, right click on the project and select Refresh



PrintWriter

- The PrintWriter class **always** overwrites a file that already exists.
- Running the program multiple times will cause the data to get overwritten each time the program is run
- To avoid this, Java provides a **FileWriter** class
- This is used *alongside* the **PrintWriter** class. It is not a direct replacement!

Appending text to a file

- To append data to a file:
  - Create an instance of **FileWriter**
  - The first argument is the filename. The second argument is true/false for whether to append or overwrite:

```
FileWriter fw = new FileWriter("Names.txt", true);
```

  - Pass the **FileWriter** instance into the constructor for the PrintWriter

```
PrintWriter outputFile = new PrintWriter(fw);
```

Example 1

```
FileWriter fw = new FileWriter("Names.txt", true);
PrintWriter outputFile = new PrintWriter(fw);
outputFile.println("Ann");
outputFile.println("Bob");
outputFile.println("Carol");
outputFile.close();
```

```
Names.txt content after first run
Ann
Bob
Carol
```

Example 1

```
FileWriter fw = new FileWriter("Names.txt", true);
PrintWriter outputFile = new PrintWriter(fw);
outputFile.println("Ann");
outputFile.println("Bob");
outputFile.println("Carol");
outputFile.close();
```

```
Names.txt content after second run
Ann
Bob
Carol
Ann
Bob
Carol
```

Example 1

```
FileWriter fw = new FileWriter("Names.txt", true);
PrintWriter outputFile = new PrintWriter(fw);
outputFile.println("Ann");
outputFile.println("Bob");
outputFile.println("Carol");
outputFile.close();
```

```
Names.txt content after third run
Ann
Bob
Carol
Ann
Bob
Carol
Ann
Bob
Carol
```

File locations

- If you provide a filename it will get written to the project's directory.
- As with any other file system operation you can write to a specific location by specifying the full path. E.g.:

```
PrintWriter outFile =   new PrintWriter("C:\\Users\\Tom\\PriceList.txt");
```

File locations

- On Windows file locations use a backslash (\) to separate directories. Note that in the example, \\ has to be used.
- This is because \ has special meaning in a Java string and must be escaped!

```
PrintWriter outFile =   new PrintWriter("C:\\Users\\Tom\\PriceList.txt");
```

File locations

- On Unix based systems (Linux, BSD, Apple OSX) the separator is a forward slash (/) so this is not an issue:

```
PrintWriter outFile =   new PrintWriter("/home/tom/PriceList.txt");
```

- You can also use forward slash on Windows and don't need to worry about escaping it

```
PrintWriter outFile =   new PrintWriter("C:/Users/Tom/PriceList.txt");
```

Ensuring Portability

- Java can be run on any operating system
- Because of this, using a full path name that is operating system specific (E.g. c:\Users\Tom\PriceList.txt) is not a good idea as the location will not exist on all Operating Systems
- You're best off writing to the application directory and using forward slashes

Case sensitivity

- On Windows, file names are not case sensitive. Opening PriceList.txt, PRICELIST.TXT or pricelist.txt will all open the same file
- On most other filesystems, this is not the case and PriceList.txt and pricelist.txt will be **different files**
- It's very easy to accidentally make a program that works on Windows but not on Linux or OSX by referencing a file name in a non-case-sensitive manner.

Reading data from a file

- You can use the File class and the Scanner class to read data from a file:

```
File myFile = new File("file.txt");
Scanner inputFile = new Scanner(myFile);
```

Pass the name of the file an argument to the file class constructor

Pass the file object as an argument to the Scanner class constructor

Reading data from a file

- Once an instance of Scanner is created, data can be read using the same methods that you have used to read keyboard input (nextLine, nextInt, nextDouble, etc)

```
// Open the file.
File file = new File("Names.txt");
Scanner inputFile = new Scanner(file);
// Read a line from the file.
String str = inputFile.nextLine();
//Print out the first line of the file
System.out.println(str);
// Close the file.
inputFile.close();
```

Detecting the end of the file

- The Scanner has a method called ***hasNext()*** which returns true if there is more of the file which has yet to be read
- This can be used with a **while** loop to loop through the entire file:

```
File file = new File("Names.txt");
Scanner inputFile = new Scanner(file);

while (inputFile.hasNext()) {
    System.out.println(inputFile.nextLine());
}

inputFile.close();
```

Text files

- Files which are stored line by line are text files
- These are useful for when you want to read data in a very specific order
- Most of the time, however you will want to store something more complex such as an entire object

Binary Files

- The way data is stored in memory is sometimes called the *raw binary format*
- Data can be stored in a file in its raw binary format.
- A file that contains binary data is often called a binary file
- Storing data in its binary format is a lot more efficinet than storing it as text

Text vs Binary

- Text files make sense if you open them in an external program such as notepad
- Binary files won't be a readable format
- To write binary data you must create objects from the following classes:
  - **FileOutputStream –** allows you to open a file for writing binary data.
  - **DataOutputStream –** Allows you to write data of any primitive type or String objects to a binary file. Cannot directly access a file and must be used with a FileOutputStream object to write to a file

Binary Files

- A DataOutputStream is wrapped around a FileOutputStream object to write data to a binary file:

```
FileOutputStream fstream = new FileOutputStream("MyInfo.dat");
DataOutputStream outputFile = new DataOutputStream(fstream);
```

- To simplify the code, this can be combined onto one line:

```
DataOutputStream outputFile = new DataOutputStream(new FileOutputStream("MyInfo.dat"));
```

- The **DataOutputStream** has methods for writing all the primitives and strings:
  - writeInt();
  - writeChar();
  - WriteDouble();
- The string method is
  - writeUTF()
- This uses a string encoding format called UTF-8

Writing a file

- To write the number 5 to a file:

```
FileOutputStream fstream = new   FileOutputStream("MyInfo.dat");
DataOutputStream outputFile = new  DataOutputStream(fstream);

outputFile.writeInt(5);

outputFile.close();
```

Reading from a binary file

- Just being able to read data isn't very useful.
- To read data back out of the written file, you can use the complementary classes:
  - FileInputStream
  - DataInputStream
- They are used the same way as writing only for reading:

Reading from a binary file

```
FileInputStream fstream = new FileInputStream("MyInfo.dat");
DataInputStream inputFile = new DataInputStream(fstream);

int myInt = inputFile.readInt();

System.out.println("The number stored in the file is " + myInt);

inputFile.close();
```

Combining reading and writing

- You can combine reading and writing to load and save from the same file:

```
int num = 0;
try {

     FileInputStream fstream = new FileInputStream("MyInfo.dat");
     DataInputStream inputFile = new DataInputStream(fstream);

     num = inputFile.readInt();

     System.out.println("The program has been run " + num + " times");

     inputFile.close();
}
catch (Exception e) {
     System.out.println("The program has not been run yet");
}

try {
     FileOutputStream outfstream = new   FileOutputStream("MyInfo.dat");
     DataOutputStream outputFile = new  DataOutputStream(outfstream);

     outputFile.writeInt(num+1);

     outputFile.close();
}
catch (Exception e) {
}
```

Storing objects

- Rather than storing primitives it's often useful to store whole objects in the exact state they were in when you left them
- To enable this, you have to design your classes in such a way that this is possible

Object Serialzation

- The String class, and many others in the Java API implements the ***Serializable*** interface
- If an object implements the ***Serializable*** interface it can be written to a file using a **ObjectOutputStream**

Object Serialization

```
public class Person implements Serializable {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }


    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

Object Serialzation

- Once a class has implemented the Serialzable interface it can be written to a file use ObjectOutPutStream's writeObject method:

```
Person p = new Person("Dave", 22);

try {
    FileOutputStream outfstream = new   FileOutputStream("person.dat");
    ObjectOutputStream outputFile = new  ObjectOutputStream(outfstream);

    outputFile.writeObject(p);

    outputFile.close();
}
catch (Exception e) {

}
```

Reading Objects

- To read the object back out, you can use ObjectInputStream:

```
try {
    FileInputStream fstream = new   FileInputStream("person.dat");
    ObjectInputStream inputFile = new  ObjectInputStream(fstream);

    Person p = (Person) inputFile.readObject();

    System.out.println(p.getName() + " is " +  p.getAge() + " years old");

    inputFile.close();
}
catch (Exception e) {

}
```

Reading Objects

- To read the object back out, you can use ObjectInputStream:

Note the need to cast the object after reading it!

```
try {
    FileInputStream fstream = new FileInputStream("person.dat");
    ObjectInputStream inputFile = new ObjectInputStream(fstream);

    Person p = (Person) inputFile.readObject();

    System.out.println(p.getName() + " is " +  p.getAge() + " years old");

    inputFile.close();
}
catch (Exception e) {

}
```

Writing objects

- Reading and writing lots of data to the file can be difficult to manage
- However, to make your life easier collections:
  - Lists (e.g. ArrayList)
  - Sets
  - Maps
- Are all serialzable
- To make it easier you can write a single list to the file and read it back using the collection to easily extract the data you want