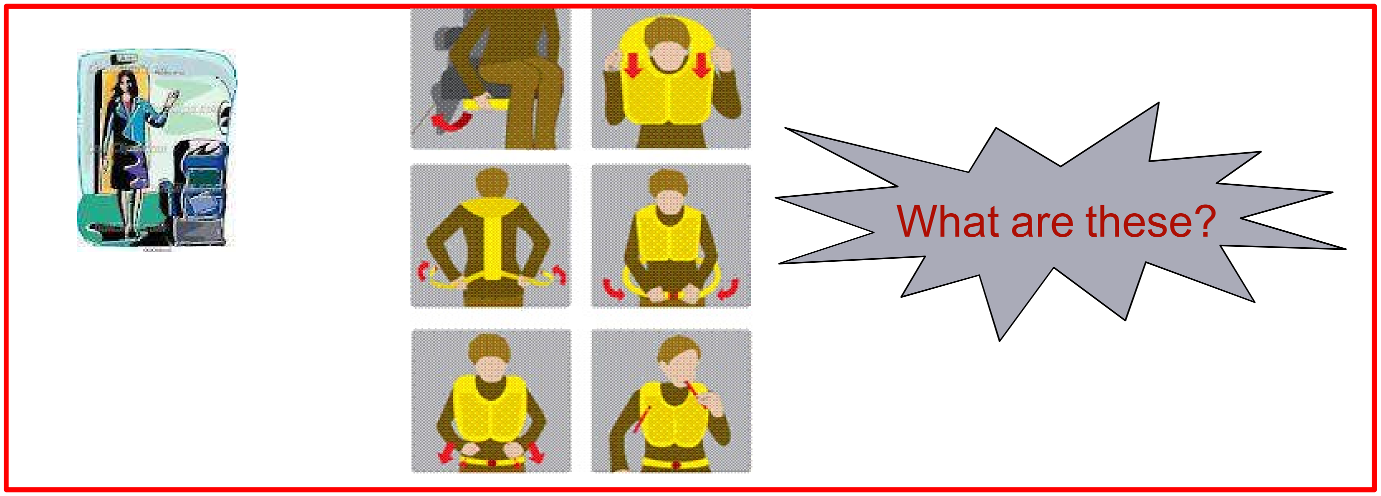
Exception Handling

Meera is flying to NewYork



This example demonstrates how you have to think in advance the many possibilities of mishaps that can occur and what are the preventive measures that can be taken.

Exception Handling

Similarly, when we write programs as part of an application, we may have to visualize the challenges that can disrupt the normal flow of execution of the code.

Once we know what are the different situations that can disrupt the flow of execution, we can take preventive measures to overcome these disruptions.

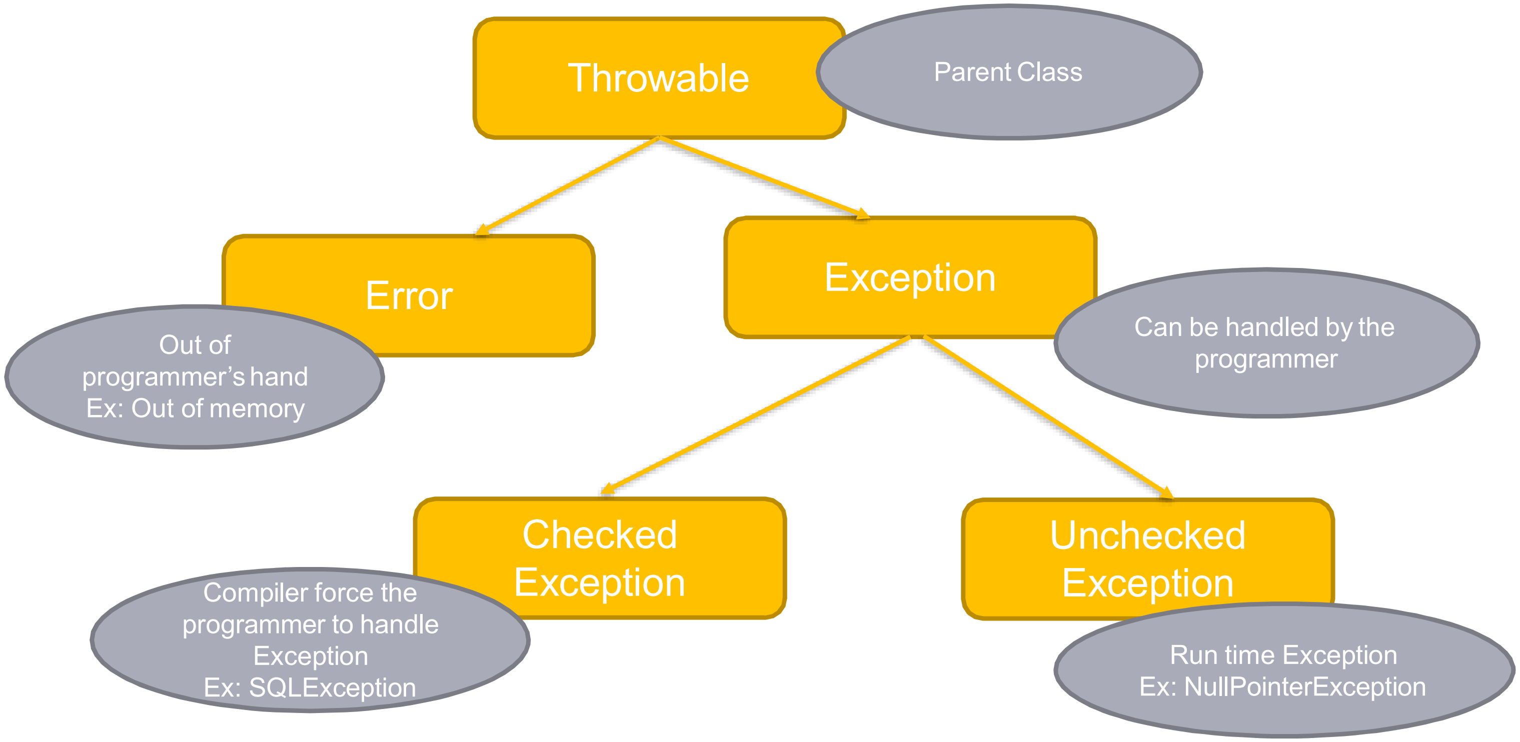
In java, this mechanism comes in the form of **Exception Handling.**

What is Exception?

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions

- The ability of a program to intercept run-time errors, take corrective measures and continue execution is referred to as exception handling
- Example
  - Attempting to access a file that does not exist
  - Inserting an element into an array at a position that is not in its bounds
  - Performing some mathematical operation that is not permitted
  - Declaring an array using negative values

Types of Exception



Errors

- Error is not considered as an Exception
- Errors are problems that arise beyond the control of the programmer or the user
- A programmer can rarely do anything about an Error that occurs during the execution of a program
- This is the precise reason Errors are typically ignored in the code
- Errors are also ignored by the compiler
- Ex : Stack Overflow

Try it

```
public class Tester {

public static void recursivePrint(int num) {
System.out.println("Number: " + num);

if(num == 0)  return;  else
recursivePrint(++num);
}

public static void main(String[] args) {
  Tester.recursivePrint(1);
}
}
```

Checked Exception

- A checked exception is an exception that usually happens due to user error, or it is an error situation that cannot be foreseen by the programmer
- A checked exception must be handled - Non compliance of this rule results in a compilation error

  Ex: FileNotFoundException

  If you try to open a file using

  ```
  FileInputStream fx = new FileInputStream(“A1.txt”);
  ```
- During execution, the system will throw a FileNotFoundException, if the file A1.txt is not located, which may be beyond the control of a programmer.

Try it

```
import java.io.*;
class Main
  {
    public static void main(String args[])
     {
          FileInputStream fx = new FileInputStream("A1.txt");
     }
  }
```

```
Main.java:6: error: unreported exception FileNotFoundException; must be caught or declared to
be thrown
        FileInputStream fx = new FileInputStream("A1.txt");
                             ^
1 error
```

UnChecked Exception

- An unchecked exception is an exception, which could have been avoided by the programmer
- If there is any chance of an unchecked exception occurring in the code, it is ignored during compilation
- Example
  - ArithmeticException
  - NumberFormatException
  - NullPointerException
  - ClassCastException

Try it

```
class    Demo
 {
   public static void main(String args[])
     {
          int x = 0;
          int y = 50/x;
          System.out.println("y     = " +y);
      }
 }
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Demo.main(Demo.java:3)
```

Exception Handling Techniques

- There are several built-in exception classes that are used to handle the very fundamental errors that may occur in your programs
- You can create your own exceptions also by extending the **Exception** class. These are called **user-defined exceptions**, and will be used in situations that are unique to your applications.
- Java's exception handling is managed using the following keywords:
  - **try, catch,**
  - **throw,**
  - **throws**
  - **finally.**

Try Catch Block

- Any part of the code that can generate an error should be put in the **try** block.
- Any error should be handled in the c**atch** block defined by the **catch** clause. This block is also called the **catch block**, or the **exception handler.**
- The corrective action to handle the exception should be put in the **catch** block.

```
try {
// block of code to monitor for errors
// the code you think can raise an exception
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
```

Try it

```
class Demo
 {
   public static void main(String args[])
    {
      try{
        int x = 0;
        int y = 50/x;
        System.out.println("y = " +y);
       }
      catch (ArithmeticException e){
         System.out.println("Division by zero.");
       }
      System.out.println("After catch statement.");
    }
 }
```

Quiz

- What will be the result, if we try to compile and execute the following code as

  java Ex1 Wipro Bangalore

```
class Ex1
  {
    public static void main(String[] xyz)
      {
        for(int i=0;i<=xyz.length;i++)
          System.out.println(xyz[i]);
        System.out.println("Thank you");
      }
  }
```

Quiz

```
class Ex1
  {
    public static void main(String[] xyz)
      {
        for(int i=0;i<=xyz.length;i++)
          {
            try{
             System.out.println(xyz[i]);
              }catch(ArrayIndexOutOfBoundsException ar)  {
               System.out.println("Array Out of Bounds Error : Rectify");
             }
          }
   System.out.println("Thank you");
      }
  }
```

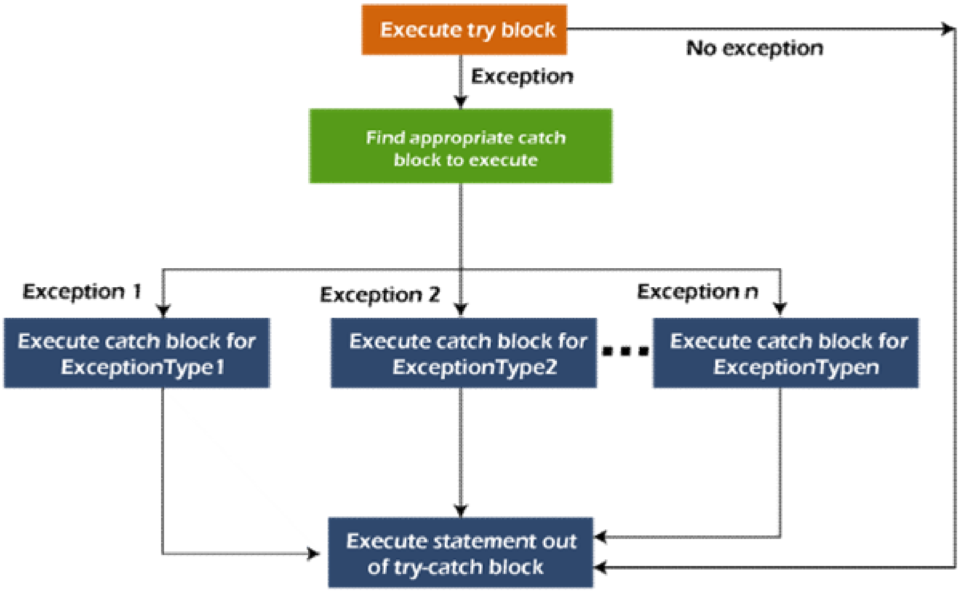
Quiz

```
class Ex1
 {
   public static void main(String args[])
     {
           int x=100;
          int y=0;
          int z=x/y;
          System.out.println(args[1]);
     }
 }
```

Java Ex1

Multiple Try Catch Block

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.


Execute try block
No exception
Exception
Find appropriate catch block to execute
Exception 1
Exception 2
Exception n
Execute catch block for ExceptionType1
Execute catch block for ExceptionType2
...
Execute catch block for ExceptionTypen
Execute statement out of try-catch block

Quiz

```
class Ex1
 {
  public static void main(String args[])
   {
     try{
       int x=100;
       int y=0;
       int z=x/y;
       System.out.println(args[1]);
     }catch(Exception e)
       {
         System.out.println("Ooooopssss");
        }
   }
 }
```

Nested Try Catch Block

- The **try** statement can be nested
- If an inner **try** statement does not have a **catch** handler for a particular exception, the outer block's catch handler will handle the exception
- This continues until one of the **catch** statement succeeds, or until all of the nested **try** statements are exhausted
- If no catch statement matches, then the Java runtime system will handle the exception

```
try
{
    statement 1;
    statement 2;
    try
    {
        statement 1;
        statement 2;
    }
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
```

throws

- The **Java throws keyword** is used to declare an exception.
- It gives an information to the programmer that there may occur an exception.
- So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.
- Exception Handling is mainly used to handle the **checked exceptions**.

```
return_type method_name() throws exception_class_name{
//method code
}
```

Try it

```
import java.io.*;
class Main
  {
    public static void main(String args[])
     {
          FileInputStream fx = new FileInputStream("A1.txt");
     }
  }
```

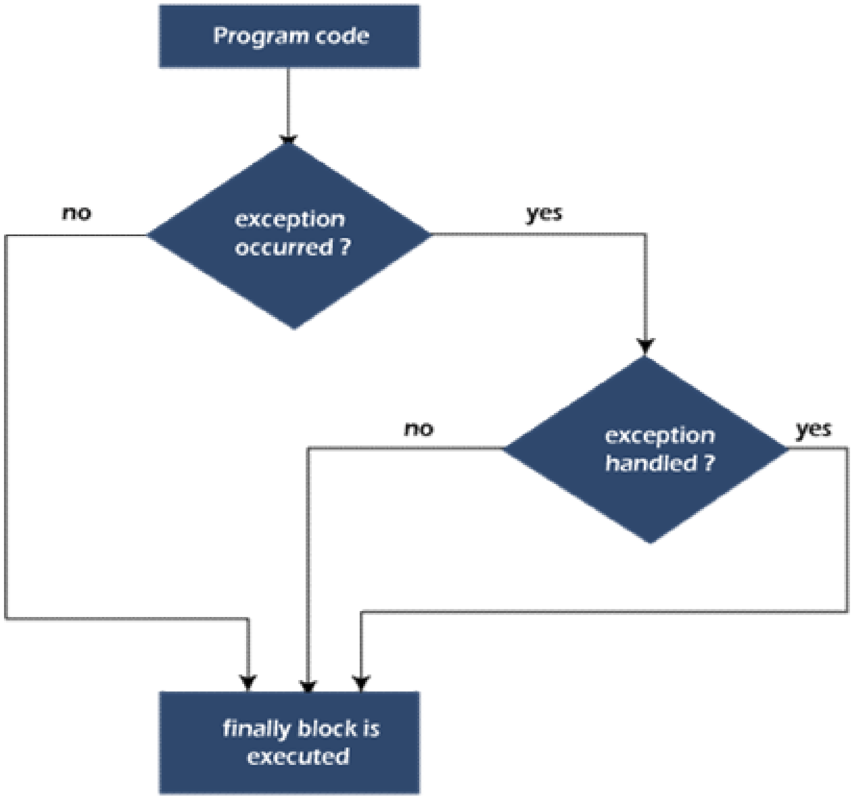
```
Main.java:6: error: unreported exception FileNotFoundException; must be caught or declared to
be thrown
      FileInputStream fx = new FileInputStream("A1.txt");
                      ^
1 error
```

Try it

```
import java.io.*;
class Main
  {
    public static void main(String args[]) throws FileNotFoundException
     {
          FileInputStream fx = new FileInputStream("A1.txt");
     }
  }
```

Finally block

- **Java finally block** is a block used to execute important code such as closing the connection, etc.
- Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.
- The finally block follows the try-catch block.



Try it

TRY IT NOW!

```
class TestFinallyBlock
 {
   public static void main(String args[])
    {
           try{
             int a=25/5;
            System.out.println("Value of a is" +a);
           }catch(NullPointerException e){System.out.println("Null ");}
           finally
             {
               System.out.println("Finally Block");
             }
           System.out.println("Rest of Prog");
    }
 }
```

Try it

TRY IT NOW!

```
class TestFinallyBlock
 {
   public static void main(String args[])
    {
            try{
              int a=25/0;
             System.out.println("Value of a is" +a);
            }catch(NullPointerException e){System.out.println("Null ");}
            finally
              {
                System.out.println("Finally Block");
              }
            System.out.println("Rest of Prog");
    }
 }
```

Try it

TRY IT NOW!

```
class TestFinallyBlock
 {
   public static void main(String args[])
    {
           try{
             int a=25/0;
            System.out.println("Value of a is" +a);
           }catch(ArithmeticException e){System.out.println("Null ");}
           finally
             {
               System.out.println("Finally Block");
             }
           System.out.println("Rest of Prog");
    }
 }
```

Throw keyword

- The Java throw keyword is used to throw an exception explicitly.
- We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.
- We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a **custom exception**.
- The syntax of the Java throw keyword is given below.

  throw new exception_class("error message");

Try it

TRY IT NOW!

```
class TestFinallyBlock
 {
   public static void main(String args[])
    {
           int a=Integer.parseInt(args[0]);
           if(a<18)
             throw new ArithmeticException(" Age is less than 18");
           System.out.println("Age above 18");
    }
 }
```

Throw vs Throws

Throw	throws
Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.
The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
throw is used within the method.	throws is used with the method signature.
We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Custom Exceptions

- Java exceptions cover almost all the general type of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.
- Following are few of the reasons to use custom exceptions:
  1. To catch and provide specific treatment to a subset of existing Java exceptions.
  2. Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

In order to create custom exception, we need to extend Exception class that belongs to java.lang package.

Try it

TRY IT NOW!

```
class ageException extends Exception
 {
  ageException(String str)
   {
       System.out.println(str);
   }
  ageException()
   {
       System.out.println("Age is less than 18");
   }
 }
```

Try it

TRY IT NOW!

```
class TestFinallyBlock
  {
    public static void main(String args[])
     {
            int a=Integer.parseInt(args[0]);
            try{
            if(a<10)
              throw new ageException("Age is too Less");
            if(a<18)
              throw new ageException();
            }catch(ageException e)
              {System.out.println("Exception Handled");}
            System.out.println("Age above 18");
     }
  }
```

DIY

Problem statement:

Get the input String from user and parse it to integer, if it is not a number it will throw number format exception Catch it and print "Entered input is not a valid format for an integer." or else print the square of that number. (Refer Sample Input and Output).

Sample input and output 1:

```
Enter an integer: 12
The square value is 144
The work has been done successfully
```

Sample input and output 2:

```
Enter an integer: Java
Entered input is not a valid format for an integer.
```

DIY

Write a program that takes as input the size of the array and the elements in the array. The program then asks the user to enter a particular index and prints the element at that index.

This program may generate Array Index Out Of Bounds Exception. Use exception handling mechanisms to handle this exception. In the catch block, print the class name of the exception thrown.

Sample Input and Output:

```
Enter the number of elements in the array 3

Enter the elements in the array

20  90  4

Enter the index of the array element you want to access

6

java.lang.ArrayIndexOutOfBoundsException
```

DIY

Write a class MathOperation which accepts integers from command line. Create an array using these parameters. Loop through the array and obtain the sum and average of all the elements.

Display the result.

Check for various exceptions that may arise like ArithmeticException, NumberFormatException, and so on.

For example: The class would be invoked as follows:

```
C:>java MathOperation 1900, 4560, 0, 32500
```

DIY

Write a Program to take care of Number Format Exception if user enters values other than integer for calculating average marks of 2 students. The name of the students and marks in 3 subjects are taken from the user while executing the program.

In the same Program write your own Exception classes to take care of Negative values and values out of range (i.e. other than in the range of 0-100)

DIY

A student portal provides user to register their profile. During registration the system needs to validate the user should be located in India. If not the system should throw an exception.

Step 1: Create a user defined exception class named “InvalidCountryException”.

Step 2: Overload the respective constructors.

Step 3: Create a main class “UserRegistration”, add the following method,

registerUser– The parameter are String username,String userCountry and add the following logic,

• if userCountry is not equal to “India” throw a InvalidCountryException with the message “User Outside India cannot be registered”

• if userCountry is equal to “India”, print the message “User registration done successfully”

Invoke the method registerUser from the main method with the data specified ar
program behaves,

Name Country Expected Output

Mickey US InvalidCountryException should be thrown.

The message should be “User Outside India cannot be registered”

DIY

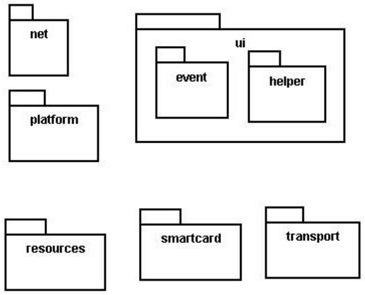
Write a program to accept name and age of a person from the command prompt(passed as arguments when you execute the class) and ensure that the age entered is >=18 and < 60.

Display proper error messages.

The program must exit gracefully after displaying the error message in case the arguments passed are not proper. (Hint : Create a user defined exception class for handling errors.)

Java packages

- **package**: A collection of related classes.
  - Can also "contain" sub-packages.
  - *Sub-packages* can have similar names, but are not actually contained inside.
    - `java.awt` does not contain `java.awt.event`



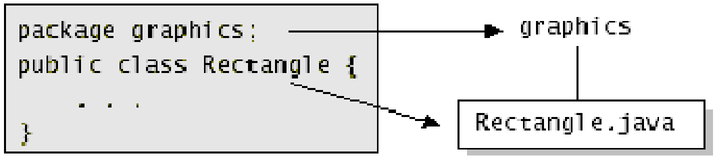
- Uses of Java packages:
  - group related classes together
  - as a *namespace* to avoid name collisions
  - provide a layer of access / protection
  - keep pieces of a project down to a manageable size

Packages and directories

- package ←→ directory (folder)
- class ←→ file

- A class named `D` in package `a.b.c` should reside in this file:

  `a/b/c/D.class`



```
package graphics;
public class Rectangle {
    . . .
}
```

  - (relative to the root of your project)

- The "root" directory of the package hierarchy is determined by your *class path* or the directory from which `java` was run.

Classpath

- **class path**: The location(s) in which Java looks for class files.

- Can include:
  - the current "working directory" from which you ran javac / java
  - other folders
  - JAR archives
  - URLs
  - ...

- Can set class path manually when running java at command line:
  - `java -cp /home/stepp/libs:/foo/bar/jbl MyClass`

A package declaration

```
package name;

public class name { ...
```

Example:

```
package pacman.model;

public class Ghost extends Sprite {
    ...
}
```

- File `Sprite.java` should go in folder `pacman/model` .

Importing a package

```
import packageName.*;            // all classes
```

Example:

```
package pacman.gui;
import pacman.model.*;

public class PacManGui {
    ...
    Ghost blinky = new Ghost();
}
```

- `PacManGui` must import the model package in order to use it.

Importing a class

```
import packageName.className;  // one class
```

Example:

```
package pacman.gui;
import pacman.model.Sprite;

public class PacManGui {
    Ghost blinky = new Ghost();
}
```

- Importing single classes has high precedence:
  - if you import `.*`, a same-named class in the current dir will override
  - if you import **`.className`**, it will not

Static import

```
import static packageName.className.*;
```

Example:

```
import static java.lang.Math.*;

...
double angle = sin(PI / 2) + ln(E * E);
```

- Static import allows you to refer to the members of another class without writing that class's name.
- Should be used rarely and only with classes whose contents are entirely static "utility" code.

Referring to packages

packageName . className

Example:

```
java.util.Scanner console =
    new java.util.Scanner(java.lang.System.in);
```

- You can use a type from any package without importing it if you write its full name.
- Sometimes this is useful to disambiguate similar names.
  - Example: `java.awt.List` and `java.util.List`
  - Or, explicitly import one of the classes.

The default package

- Compilation units (files) that do not declare a package are put into a default, unnamed, package.

- Classes in the default package:
  - Cannot be imported
  - Cannot be used by classes in other packages

- Many editors discourage the use of the default package.

- Package java.lang is implicitly imported in all programs by default.
  - `import java.lang.*;`

Package access

- Java provides the following access modifiers:
  - `public` : Visible to all other classes.
  - `private` : Visible only to the current class (and any nested types).
  - `protected` : Visible to the current class, any of its subclasses, and any other types within the same package.
  - default (package): Visible to the current class and any other types within the same package.

- To give a member default scope, do not write a modifier:

```
package pacman.model;
public class Sprite {
    int points;      // visible to pacman.model.*
    String name;     // visible to pacman.model.*
```

Package exercise

- Add packages to the Rock-Paper-Scissors game.
  - Create a package for core "model" data.
  - Create a package for graphical "view" classes.

  - Any general utility code can go into a default package or into another named utility (util) package.
  - Add appropriate package and import statements so that the types can use each other properly.

JAR Files (yousa likey!)

- **JAR**: **J**ava **AR**chive. A group of Java classes and supporting files combined into a single file compressed with ZIP format, and given .JAR extension.

- Advantages of JAR files:
  - compressed; quicker download
  - just one file; less mess
  - can be executable

- The closest you can get to having a .exe file for your Java application.

Creating a JAR archive

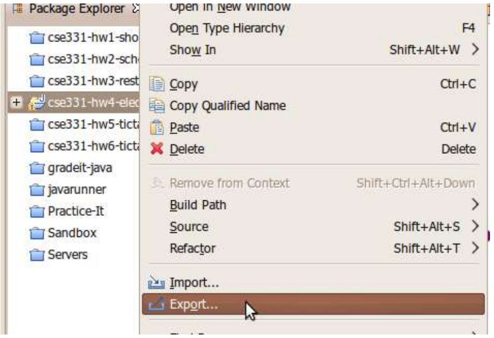
- from the command line:

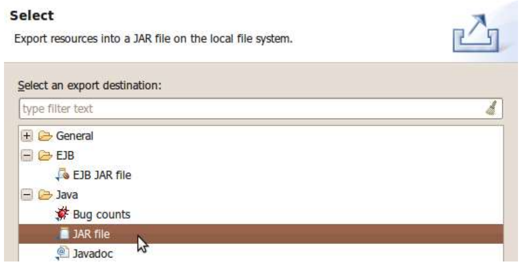
  `jar -cvf` **filename**`.jar` **files**

  - Example:

    ```
    jar -cvf MyProgram.jar *.class *.gif *.jpg
    ```

- some IDEs (e.g. Eclipse) can create JARs automatically
  - File → Export... → JAR file




Select

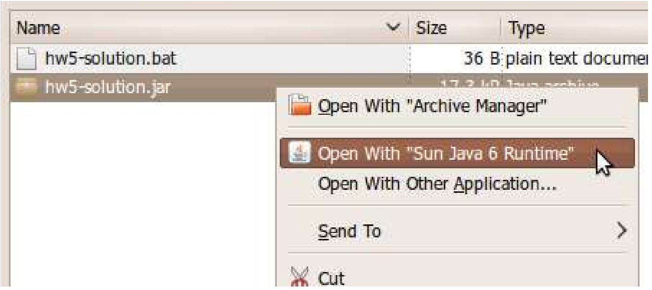
Export resources into a JAR file on the local file system.

Select an export destination:

Running a JAR

- Running a JAR from the command line:
  - `java -jar` **filename**`.jar`

- Most OSes can run JARs directly by double-clicking them:



Name	Size	Type
hw5-solution.bat	36 B	plain text documer
hw5-solution.jar	17.3 kB	Java archive

Open With "Archive Manager"
Open With "Sun Java 6 Runtime"
Open With Other Application...
Send To
Cut

Making a runnable JAR

- **manifest file**: Used to create a JAR runnable as a program.

```
jar -cvmf manifestFile MyAppletJar.jar
          mypackage/*.class *.gif
```

Contents of MANIFEST file:

```
Main-Class: MainClassName
```

- Eclipse will automatically generate and insert a proper manifest file into your JAR if you specify the main-class to use.

Resources inside a JAR

- You can embed external resources inside your JAR:
  - images (GIF, JPG, PNG, etc.)
  - audio files (WAV, MP3)
  - input data files (TXT, DAT, etc.)
  - ...

- But code for opening files will look outside your JAR, not inside it.

  ```
  Scanner in = new Scanner(new File("data.txt"));   // fail
  ImageIcon icon = new ImageIcon("pony.png");       // fail
  Toolkit.getDefaultToolkit().getImage("cat.jpg");  // fail
  ```

Accessing JAR resources

- Every class has an associated .class object with these methods:
  - `public URL getResource(String filename)`
  - `public InputStream getResourceAsStream(String name)`

- If a class named `Example` wants to load resources from within a JAR, its code to do so should be the following:

  -
    ```
    Scanner in = new Scanner(
        Example.class.getResourceAsStream("/data.txt"));
    ```

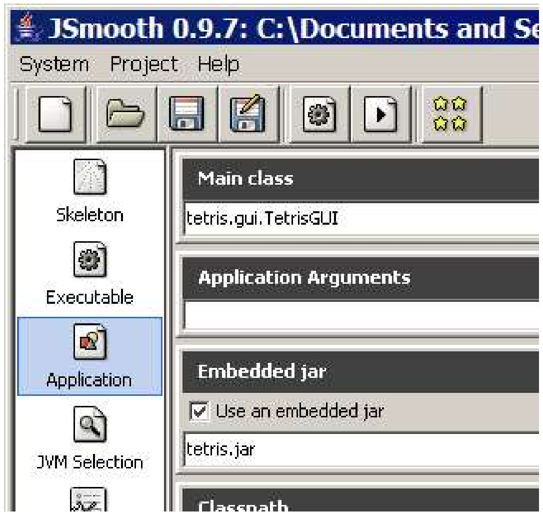
  -
    ```
    ImageIcon icon = new ImageIcon(
        Example.class.getResource("/pony.png"));
    ```

  -
    ```
    Toolkit.getDefaultToolkit().getImage(
        Example.class.getResource("/images/cat.jpg"));
    ```

  - (Some classes like Scanner read from streams; some like Toolkit read from URLs.)
  - NOTE the very important leading `/` character; without it, you will get a `null` result

JAR to EXE (JSmooth)

- *JSmooth* is a free program that converts JARs into Windows EXE files.
  - http://jsmooth.sourceforge.net/
  - If the machine does not have Java installed, your EXE will help the user to download and install Java.
  - A bit of a hack; not generally needed.



JSmooth 0.9.7: C:\Documents and Se

Main class

tetris.gui.TetrisGUI

Application Arguments

Embedded jar

☑ Use an embedded jar

tetris.jar

Classpath

- Using JSmooth:
  - choose Skeleton → Windowed Wrapper
  - name your .exe under Executable → Executable Binary
  - browse to your .jar under Application → Embedded JAR
  - select the main class under Application → Main class

Streams and File I/O

Objectives

- become familiar with the concept of an I/O stream
- understand the difference between binary files and text files
- learn how to save data in a file
- learn how to read data from a file

Outline

- Overview of Streams and File I/O
- Text-File I/O
- Using the `File` Class
- Basic Binary-File I/O
- Object I/O with Object Streams
- (optional) Graphics Supplement

Objectives, cont.

- learn how use the classes `ObjectOutputStream` and `ObjectInputStream` to read and write class objects with binary files

I/O Overview

- *I/O* = Input/Output
- In this context it is input to and output from programs
- Input can be from keyboard or a file
- Output can be to display (screen) or a file
- Advantages of file I/O
  - permanent copy
  - output from one program can be input to another
  - input can be automated (rather than entered manually)

Note: Since the sections on text file I/O and binary file I/O have some similar information, some duplicate (or nearly duplicate) slides are included.

Streams

- ***Stream***: an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.)
  - it acts as a buffer between the data source and destination
- ***Input stream***: a stream that provides input to a program
  - `System.in` is an input stream
- ***Output stream***: a stream that accepts output from a program
  - `System.out` is an output stream
- A stream connects a program to an I/O object
  - `System.out` connects a program to the screen
  - `System.in` connects a program to the keyboard

Binary Versus Text Files

- *All* data and programs are ultimately just zeros and ones
  - each digit can have one of two values, hence *binary*
  - *bit* is one binary digit
  - *byte* is a group of eight bits
- *Text files*: the bits represent printable characters
  - one byte per character for ASCII, the most common code
  - for example, Java source files are text files
  - so is any file created with a "text editor"
- *Binary files*: the bits represent other types of encoded information, such as executable instructions or numeric data
  - these files are easily read by the computer but not humans
  - they are *not* "printable" files
    - actually, you *can* print them, but they will be unintelligible
    - "printable" means "easily readable by humans when printed"

Java: Text Versus Binary Files

- Text files are more readable by humans
- Binary files are more efficient
  - computers read and write binary files more easily than text
- Java binary files are portable
  - they can be used by Java on different machines
  - Reading and writing binary files is normally done by a program
  - text files are used only to communicate with humans

Java Text Files

- Source files
- Occasionally input files
- Occasionally output files

Java Binary Files

- Executable files (created by compiling source files)
- Usually input files
- Usually output files

Text Files vs. Binary Files

- Number: 127 (decimal)
    - Text file
        - Three bytes: “1”, “2”, “7”
        - ASCII (decimal): 49, 50, 55
        - ASCII (octal): 61, 62, 67
        - ASCII (binary): 00110001, 00110010, 00110111
    - Binary file:
        - One byte (`byte`): 01111110
        - Two bytes (`short`): 00000000 01111110
        - Four bytes (`int`): 00000000 00000000 00000000 01111110

Text file: an example

[unix: od –w8 –bc <file>]

[http://www.muquit.com/muquit/software/hod/hod.html for a Windows tool]

```
127     smiley
faces
```

```
0000000 061 062 067 011 163 155 151 154
          1   2   7  \t   s   m   i   l
0000010 145 171 012 146 141 143 145 163
          e   y  \n   f   a   c   e   s
0000020 012
         \n
```

Binary file: an example [a .class file]

```
0000000 312 376 272 276 000 000 000 061
        312 376 272 276  \0  \0  \0   1
0000010 000 164 012 000 051 000 062 007
         \0   t  \n  \0   )  \0   2  \a
0000020 000 063 007 000 064 010 000 065
         \0   3  \a  \0   4  \b  \0   5
0000030 012 000 003 000 066 012 000 002
         \n  \0 003  \0   6  \n  \0 002

...
0000630 000 145 000 146 001 000 027 152
         \0   e  \0   f 001  \0 027   j
0000640 141 166 141 057 154 141 156 147
          a   v   a   /   l   a   n   g
0000650 057 123 164 162 151 156 147 102
          /   S   t   r   i   n   g   B
0000660 165 151 154 144 145 162 014 000
          u   i   l   d   e   r  \f  \0
```

Text File I/O

- Important classes for text file **output** (to the file)
  - **`PrintWriter`**
  - **`FileOutputStream`** [or **`FileWriter]`**
- Important classes for text file **input** (from the file):
  - **`BufferedReader`**
  - **`FileReader`**
- **`FileOutputStream`** and **`FileReader`** take file names as arguments.
- **`PrintWriter`** and **`BufferedReader`** provide useful methods for easier writing and reading.
- Usually need a combination of two classes
- To use these classes your program needs a line like the following:

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

Buffering

- Not buffered: each byte is read/written from/to disk as soon as possible
  - “little” delay for each byte
  - A disk operation per byte---higher overhead
- Buffered: reading/writing in “chunks”
  - Some delay for some bytes
    - Assume 16-byte buffers
    - Reading: access the first 4 bytes, need to wait for all 16 bytes are read from disk to memory
    - Writing: save the first 4 bytes, need to wait for all 16 bytes before writing from memory to disk
  - A disk operation per a buffer of bytes---lower overhead

Every File Has Two Names

1. the stream name used by Java
    - `outputStream` in the example
2. the name used by the operating system
    - `out.txt` in the example

Text File Output

- To open a text file for output: connect a text file to a stream for writing

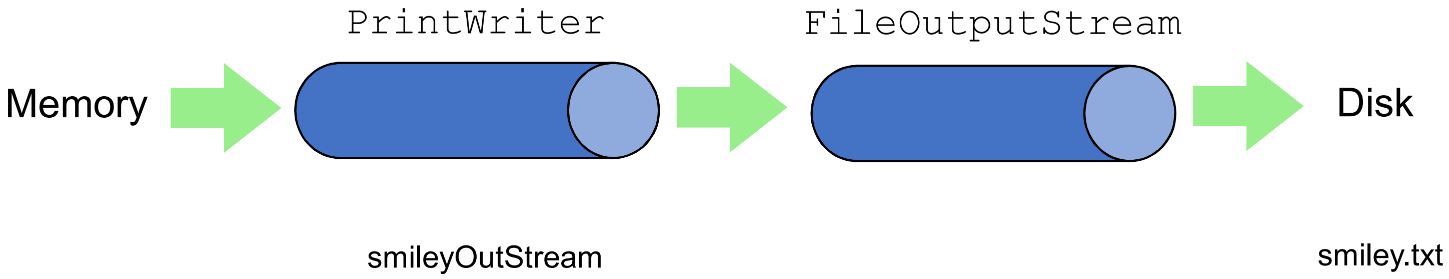
```
PrintWriter outputStream =
 new PrintWriter(new FileOutputStream("out.txt"));
```

- Similar to the long way:

```
FileOutputStream s = new FileOutputStream("out.txt");
PrintWriter outputStream = new PrintWriter(s);
```

- Goal: create a `PrintWriter` object
  - which uses `FileOutputStream` to open a text file
- `FileOutputStream` "connects" `PrintWriter` to a text file.

Output File Streams



```
PrintWriter smileyOutStream = new PrintWriter( new FileOutputStream(“smiley.txt”) );
```

Methods for `PrintWriter`

- Similar to methods for `System.out`
- `println`

```
outputStream.println(count + " " + line);
```

- `print`
- `format`
- `flush`: write buffered output to disk
- `close`: close the `PrintWriter` stream (and file)

TextFileOutputDemo

Part 1

```
public static void main(String[] args)
{
   PrintWriter outputStream = null;
   try
   {
       outputStream =
            new PrintWriter(new FileO
   }
   catch(FileNotFoundException e)
   {
       System.out.println("Error opening the file out.txt. ‟
                          + e.getMessage());
       System.exit(0);
   }
```

Opening the file

<u>A `try`-block is a block:</u> `outputStream` would not be accessible to the rest of the method if it were declared inside the `try`-block

Creating a file can cause the `FileNotFound-Exception` if the new file cannot be made.

TextFileOutputDemo

Part 2

```
System.out.println("Enter three lines of text:");
String line = null;
int count;
   for (count = 1; count <= 3; count++)
   {
       line = keyboard.nextLine();
       outputStream.println(count + " " + line);
   }
   outputStream.close();
   System.out.println("... written to out.txt.");
}
```

Writing to the file

Closing the file

The `println` method is used with two different streams: `outputStream` and `System.out`

Gotcha: Overwriting a File

- Opening an output file creates an empty file

- Opening an output file creates a new file if it does not already exist

- Opening an output file that already exists eliminates the old file and creates a new, empty one
    - data in the original file is lost

- To see how to check for existence of a file, see the section of the text that discusses the `File` class (later slides).

Java Tip: Appending to a Text File

- To add/append to a file instead of replacing it, use a different constructor for **`FileOutputStream`**:

```
outputStream =
  new PrintWriter(new FileOutputStream("out.txt", true));
```

- Second parameter: append to the end of the file if it exists?
- Sample code for letting user tell whether to replace or append:

```
System.out.println("A for append or N for new file:");
char ans = keyboard.next().charAt(0);
boolean append = (ans == 'A' || ans == 'a');
outputStream = new PrintWriter(
        new FileOutputStream("out.txt", append));
```

true if user enters 'A'

Closing a File

- An output file should be closed when you are done writing to it (and an input file should be closed when you are done reading from it).

- Use the `close` method of the class `PrintWriter` `(BufferedReader` also has a `close` method).

- For example, to close the file opened in the previous example:

```
outputStream.close();
```

- If a program ends normally it will close any files that are open.

FAQ: Why Bother to Close a File?

If a program automatically closes files when it ends normally, why close them with explicit calls to `close`?

Two reasons:

1. To make sure it is closed if a program ends abnormally (it could get damaged if it is left open).

2. A file opened for writing must be closed before it can be opened for reading.
   - Although Java does have a class that opens a file for both reading and writing, it is not used in this text.

Text File Input

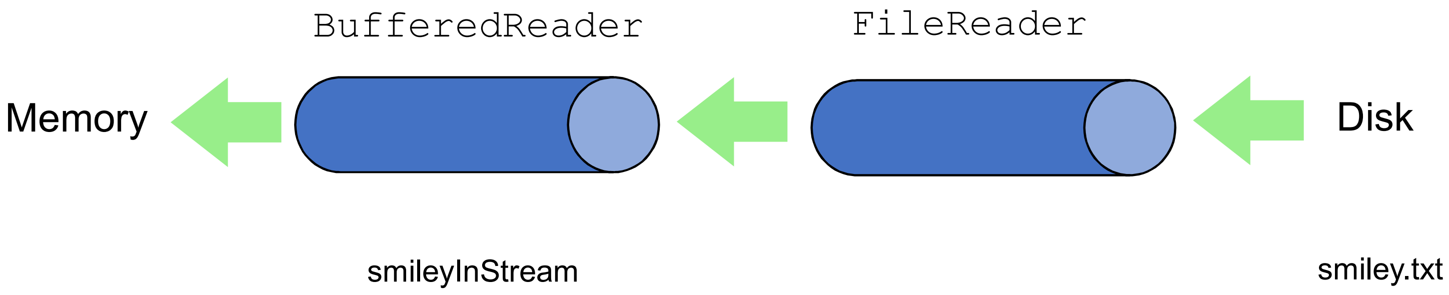
- To open a text file for input: connect a text file to a stream for reading
  - Goal: a `BufferedReader` object,
    - which uses `FileReader` to open a text file
  - `FileReader` “connects” `BufferedReader` to the text file
- For example:

```
BufferedReader smileyInStream =
  new BufferedReader(new FileReader(“smiley.txt"));
```

- Similarly, the long way :

```
FileReader s = new FileReader(“smiley.txt");
BufferedReader smileyInStream = new
 BufferedReader(s);
```

Input File Streams



BufferedReader smileyInStream = new BufferedReader(new FileReader(“smiley.txt”));

Methods for `BufferedReader`

- `readLine`: read a line into a `String`
- no methods to read numbers directly, so read numbers as `String`s and then convert them (`StringTokenizer` later)
- `read`: read a `char` at a time
- `close`: close `BufferedReader` stream

Exception Handling with File I/O

<u>Catching IOExceptions</u>

- `IOException` is a predefined class
- File I/O might throw an `IOException`
- catch the exception in a catch block that at least prints an error message and ends the program
- `FileNotFoundException` is derived from `IOException`
  - therefor any catch block that catches `IOException`s also catches `FileNotFoundException`s
  - put the more specific one first (the derived one) so it catches specifically file-not-found exceptions
  - then you will know that an I/O error is something other than file-not-found

Example: Reading a File Name from the Keyboard

```
public static void main(String[] args)
  {
    String fileName = null;  // outside try block, can be used in catch
    try
    { Scanner keyboard = new Scanner(System.in);
      System.out.println("Enter file name:");
      fileName = keyboard.next();
      BufferedReader inputStream =
        new BufferedReader(new FileReader(fileName));
      String line = null;
      line = inputStream.readLine();
      System.out.println("The first line in " + filename + " is:");
      System.out.println(line);
      // . . . code for reading second line not shown here . . .
      inputStream.close();
    }
    catch(FileNotFoundException e)
    {
      System.out.println("File " + filename + " not found.");
    }
    catch(IOException e)
    {
      System.out.println("Error reading from file " + fileName);
    }
  }
```

reading a file name from the keyboard

using the file name read from the keyboard

reading data from the file

closing the file

Exception.getMessage()

```
try
{
   …
}
catch (FileNotFoundException e)
{
   System.out.println(filename + “ not found”);
   System.out.println(“Exception: “ +
                       e.getMessage());
   System.exit(-1);
}
```

Reading Words in a String: Using **`StringTokenizer`** Class

- There are `BufferedReader` methods to read a line and a character, but not just a single word

- `StringTokenizer` can be used to parse a line into words
  - import `java.util.*`
  - some of its useful methods are shown in the text
    - e.g. test if there are more tokens
  - you can specify *delimiters* (the character or characters that separate words)
    - the default delimiters are "white space" (space, tab, and newline)

Example: **StringTokenizer**

- Display the words separated by any of the following characters: space, new line (\n), period (.) or comma (,).

```
String inputLine = keyboard.nextLine();
StringTokenizer wordFinder =
            new StringTokenizer(inputLine, " \n.,");
//the second argument is a string of the 4 delimiters
while(wordFinder.hasMoreTokens())
{
   System.out.println(wordFinder.nextToken());
}
```

Entering "`Question,2b.or !tooBee.`" gives this output:

```
Question
2b
or
!tooBee
```

Testing for End of File in a Text File

- When `readLine` tries to read beyond the end of a text file it returns the special value *`null`*
  - so you can test for `null` to stop processing a text file

- `read` returns -1 when it tries to read beyond the end of a text file
  - the `int` value of all ordinary characters is nonnegative

- Neither of these two methods (`read` and `readLine`) will throw an `EOFException`.

Example: Using Null to Test for End-of-File in a Text File

When using **`readLine`** test for `null`

Excerpt from `TextEOFDemo`

```
int count = 0;
String line = inputStream.readLine();
while (line != null)
{
   count++;
   outputStream.println(count + " " + line);
   line = inputStream.readLine();
}
```

When using **`read`** test for -1

File I/O example

- http://www.cs.fit.edu/~pkc/classes/cse1001/FileIO/FileIO.java

Using Path Names

- ***Path name***—gives name of file and tells which directory the file is in
- ***Relative path name***—gives the path starting with the directory that the program is in
- Typical UNIX path name:

```
/user/smith/home.work/java/FileClassDemo.java
```

- Typical Windows path name:

```
D:\Work\Java\Programs\FileClassDemo.java
```

- When a backslash is used in a quoted string it must be written as two backslashes since backslash is the escape character:

```
"D:\\Work\\Java\\Programs\\FileClassDemo.java"
```

- Java will accept path names in UNIX or Windows format, regardless of which operating system it is actually running on.

File Class [java.io]

- Acts like a wrapper class for file names
- A file name like "numbers.txt" has only String properties
- File has some very useful methods
  - exists: tests if a file already exists
  - canRead: tests if the OS will let you read a file
  - canWrite: tests if the OS will let you write to a file
  - delete: deletes the file, returns true if successful
  - length: returns the number of bytes in the file
  - getName: returns file name, excluding the preceding path
  - getPath: returns the path name—the full name

```
File numFile = new File(“numbers.txt”);
if (numFile.exists())
  System.out.println(numfile.length());
```

File Objects and Filenames

- FileInputStream and FileOutputStream have constructors that take a File argument as well as constructors that take a String argument

```
PrintWriter smileyOutStream = new PrintWriter(new
 FileOutputStream(“smiley.txt”));
```

```
File smileyFile = new File(“smiley.txt”);
if (smileyFile.canWrite())
  PrintWriter smileyOutStream = new PrintWriter(new
 FileOutputStream(smileyFile));
```

Alternative with Scanner

- Instead of `BufferedReader` with `FileReader`, then `StringTokenizer`
- Use `Scanner` with `File`:

```
Scanner inFile =
   new Scanner(new File(“in.txt”));
```

- Similar to `Scanner` with `System.in`:

```
Scanner keyboard =
   new Scanner(System.in);
```

Reading in `int`’s

```
Scanner inFile = new Scanner(new File(“in.txt"));
int number;
while (inFile.hasInt())
  {
    number = inFile.nextInt();
    // …
  }
```

Reading in lines of characters

```
Scanner inFile = new Scanner(new File(“in.txt"));
String line;
while (inFile.hasNextLine())
  {
    line = inFile.nextLine();
    // …
  }
```

Multiple types on one line

```
// Name, id, balance
Scanner inFile = new Scanner(new File(“in.txt"));
while (inFile.hasNext())
  {
    name = inFile.next();
    id = inFile.nextInt();
    balance = inFile.nextFloat();
    // …  new Account(name, id, balance);
  }
--------------------
String line;
while (inFile.hasNextLine())
  {
    line = inFile.nextLine();
    Scanner parseLine = new Scanner(line) // Scanner again!
    name = parseLine.next();
    id = parseLine.nextInt();
    balance = parseLine.nextFloat();
    // …  new Account(name, id, balance);
  }
```

Multiple types on one line

```
// Name, id, balance
Scanner inFile = new Scanner(new File(“in.txt"));
String line;
while (inFile.hasNextLine())
  {
   line = inFile.nextLine();
   Account account = new Account(line);
  }
--------------------
public Account(String line) // constructor
{
  Scanner accountLine = new Scanner(line);
  _name = accountLine.next();
  _id = accountLine.nextInt();
  _balance = accountLine.nextFloat();
}
```

`BufferedReader` vs `Scanner` (parsing primitive types)

- `Scanner`
  - `nextInt(), nextFloat(),` … for parsing types
- `BufferedReader`
  - `read(), readLine(), …` none for parsing types
  - needs `StringTokenizer` then wrapper class methods like `Integer.parseInt(token)`

`BufferedReader` vs `Scanner`
(Checking End of File/Stream (EOF))

- `BufferedReader`
  - `readLine()` returns `null`
  - `read()` returns -1
- `Scanner`
  - `nextLine()` throws exception
  - needs `hasNextLine()` to check first
  - `nextInt(), hasNextInt(), …`

```
BufferedReader inFile = …
line = inFile.readline();
while (line != null)
{
  // …
  line = inFile.readline();
}

-------------------

Scanner inFile = …
while (inFile.hasNextLine())
{
  line = infile.nextLine();
  // …
}
```

```
BufferedReader inFile = …
line = inFile.readline();
while (line != null)
{
  // …
  line = inFile.readline();
}

-------------------

BufferedReader inFile = …
while ((line = inFile.readline()) != null)
{
  // …
}
```

My suggestion

- Use `Scanner` with `File`
  - `new Scanner(new File(“in.txt”))`
- Use `hasNext…()` to check for EOF
  - `while (inFile.hasNext…())`
- Use `next…()` to read
  - `inFile.next…()`
- Simpler and you are familiar with methods for `Scanner`

My suggestion cont…

- File input

  -
    ```
    Scanner inFile =
      new Scanner(new File(“in.txt”));
    ```
- File output

  -
    ```
    PrintWriter outFile =
      new PrintWriter(new File(“out.txt”));
    ```
  - `outFile.print(), println(), format(), flush(), close(), …`
- http://www.cs.fit.edu/~pkc/classes/cse1001/FileIO/FileIONew.java

Skipping binary file I/O for now;
if we have time, we’ll come back

Basic Binary File I/O

- Important classes for binary file **output** (to the file)
  - **`ObjectOutputStream`**
  - **`FileOutputStream`**
- Important classes for binary file **input** (from the file):
  - **`ObjectInputStream`**
  - **`FileInputStream`**
- Note that **`FileOutputStream`** and **`FileInputStream`** are used only for their constructors, which can take file names as arguments.
  - **`ObjectOutputStream`** and **`ObjectInputStream`** cannot take file names as arguments for their constructors.
- To use these classes your program needs a line like the following:

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

Java File I/O: Stream Classes

- `ObjectInputStream` and `ObjectOutputStream`:
  - have methods to either read or write data one byte at a time
  - automatically convert numbers and characters into binary
    - binary-encoded numeric files (files with numbers) are not readable by a text editor, but store data more efficiently
- Remember:
  - *input* means data into a <u>program</u>, not the file
  - similarly, *output* means data out of a program, not the file

When Using **`ObjectOutputStream`** to Output Data to Files:

- The output files are binary and can store any of the primitive data types (`int`, `char`, `double`, etc.) and the `String` type

- The files created can be read by other Java programs but are not printable

- The Java I/O library must be imported by including the line: `import java.io.*;`
  - it contains `ObjectOutputStream` and other useful class definitions

- An `IOException` might be thrown

Handling **`IOException`**

- `IOException` cannot be ignored
    - either handle it with a catch block
    - or defer it with a `throws`-clause

We will put code to open the file and write to it in a `try`-block and write a `catch`-block for this exception :

```
catch(IOException e)
{
  System.out.println("Problem with output...";
}
```

Opening a New Output File

- The file name is given as a `String`
    - file name rules are determined by your operating system

- Opening an output file takes two steps
    1. Create a `FileOutputStream` object associated with the file name `String`
    2. Connect the `FileOutputStream` to an `ObjectOutputStream` object

    This can be done in one line of code

Example: Opening an Output File

To open a file named `numbers.dat`:

```
ObjectOutputStream outputStream =
  new ObjectOutputStream(
  new FileOutputStream("numbers.dat"));
```

- The constructor for `ObjectOutputStream` requires a `FileOutputStream` argument
- The constructor for `FileOutputStream` requires a `String` argument
  - the `String` argument is the output file name
- The following two statements are equivalent to the single statement above:

```
FileOutputStream middleman =
 new FileOutputStream("numbers.dat");
ObjectOutputStream outputStream =
 new ObjectOutputSteam(middleman);
```

Some **`ObjectOutputStream`** Methods

- You can write data to an output file after it is connected to a stream class
  - Use methods defined in `ObjectOutputStream`
    - `writeInt(int n)`
    - `writeDouble(double x)`
    - `writeBoolean(boolean b)`
    - etc.
    - See the text for more

- Note that each write method throws `IOException`
  - eventually we will have to write a catch block for it

- Also note that each write method includes the modifier `final`
  - `final` methods cannot be redefined in derived classes

Closing a File

- An Output file should be closed when you are done writing to it
- Use the `close` method of the class `ObjectOutputStream`
- For example, to close the file opened in the previous example:

```
outputStream.close();
```

- If a program ends normally it will close any files that are open

Writing a Character to a File: an Unexpected Little Complexity

- The method `writeChar` has an annoying property:
  - it takes an `int`, not a `char`, argument
- But it is easy to fix:
  - just cast the character to an int
- For example, to write the character 'A' to the file opened previously:

```
outputStream.writeChar((int) 'A');
```

- Or, just use the automatic conversion from `char` to `int`

Writing a **`boolean`** Value to a File

- `boolean` values can be either of two values, `true` or `false`

- `true` and `false` are not just names for the values, they actually are of type `boolean`

- For example, to write the `boolean` value `false` to the output file:

```
outputStream.writeBoolean(false);
```

Writing Strings to a File:
Another Little Unexpected Complexity

- Use the `writeUTF` method to output a value of type `String`
  - there is no `writeString` method
- UTF stands for Unicode Text Format
  - a special version of Unicode
- Unicode: a text (printable) code that uses 2 bytes per character
  - designed to accommodate languages with a different alphabet or no alphabet (such as Chinese and Japanese)
- ASCII: also a text (printable) code, but it uses just 1 byte per character
  - the most common code for English and languages with a similar alphabet
- UTF is a modification of Unicode that uses just one byte for ASCII characters
  - allows other languages without sacrificing efficiency for ASCII files

When Using `ObjectInputStream` to Read Data from Files:

- Input files are binary and contain any of the primitive data types (`int`, `char`, `double`, etc.) and the `String` type

- The files can be read by Java programs but are not printable

- The Java I/O library must be imported including the line:
  `import java.io.*;`
  - it contains `ObjectInputStream` and other useful class definitions

- An `IOException` might be thrown

Opening a New Input File

- Similar to opening an output file, but replace "output" with "input"

- The file name is given as a `String`
  - file name rules are determined by your operating system

- Opening a file takes two steps
  1. Creating a `FileInputStream` object associated with the file name `String`
  2. Connecting the `FileInputStream` to an `ObjectInputStream` object

- This can be done in one line of code

Example: Opening an Input File

To open a file named `numbers.dat`:

```
ObjectInputStream inStream =
  new ObjectInputStream (new
  FileInputStream("numbers.dat"));
```

- The constructor for `ObjectInputStream` requires a `FileInputStream` argument
- The constructor for `FileInputStream` requires a `String` argument
    - the `String` argument is the input file name
- The following two statements are equivalent to the statement at the top of this slide:

```
 FileInputStream middleman =
  new FileInputStream("numbers.dat");
ObjectInputStream inputStream =
  new ObjectInputStream (middleman);
```

Some **ObjectInputStream** Methods

- For every output file method there is a corresponding input file method

- You can read data from an input file after it is connected to a stream class
  - Use methods defined in `ObjectInputStream`
    - `readInt()`
    - `readDouble()`
    - `readBoolean()`
    - etc.
    - See the text for more

- Note that each write method throws `IOException`

- Also note that each write method includes the modifier `final`

Input File Exceptions

- A `FileNotFoundException` is thrown if the file is not found when an attempt is made to open a file
- Each read method throws `IOException`
  - we still have to write a catch block for it
- If a read goes beyond the end of the file an `EOFException` is thrown

Avoiding Common **`ObjectInputStream`** File Errors

There is no error message (or exception)

if you read the wrong data type!

- Input files can contain a mix of data types
  - it is up to the programmer to know their order and use the correct read method
- `ObjectInputStream` works with binary, not text files
- As with an output file, close the input file when you are done with it

Common Methods to Test for the End of an Input File

- A common programming situation is to read data from an input file but not know how much data the file contains

- In these situations you need to check for the end of the file

- There are three common ways to test for the end of a file:
  1. Put a sentinel value at the end of the file and test for it.
  2. Throw and catch an end-of-file exception.
  3. Test for a special character that signals the end of the file (text files often have such a character).

The **EOFException** Class

- Many (but not all) methods that read from a file throw an end-of-file exception (`EOFException`) when they try to read beyond the file
  - all the `ObjectInputStream` methods in Display 9.3 do throw it
- The end-of-file exception can be used in an "infinite" (`while(true)`) loop that reads and processes data from the file
  - the loop terminates when an `EOFException` is thrown
- The program is written to continue normally after the `EOFException` has been caught

Using **EOFException**

`main` method from `EOFExceptionDemo`

```
try
{
    ObjectInputStream inputStream =
      new ObjectInputStream(new FileInputStream("numbers.dat"));
    int n;

    System.out.println("Reading ALL the integers");
    System.out.println("in the file numbers.dat.");
    try
    {
        while (true)
        {
            n = inputStream.readInt();
            System.out.println(n);
        }
    }
    catch(EOFException e)
    {
        System.out.println("End of reading from file.");
    }
    inputStream.close();
}
catch(FileNotFoundException e)
{
    System.out.println("Cannot find file numbers.dat.");
}
catch(IOException e)
{
    System.out.println("Problem with input from file numbers.dat.");
}
```

Intentional "infinite" loop to process data from input file

Loop exits when end-of-file exception is thrown

Processing continues after `EOFException`: the input file is closed

Note order of catch blocks: the most specific is first and the most general last

Binary I/O of Class Objects

- read and write class objects in binary file

- class must be ***serializable***
  - **`import java.io.*`**
  - implement **`Serializable`** interface
  - add **`implements Serializable`** to heading of class definition

```
public class Species implements Serializable
```

- **`methods used:`**

  to **write** object to file:
  `writeObject` method in
  `ObjectOutputStream`

  to **read** object from file:
  `readObject` method in
  `ObjectInputStream`

```
outputStream = new ObjectOutputStream(
       new FileOutputStream("species.records"));
...
Species oneRecord =
       new Species("Calif. Condor, 27, 0.02);
...
outputStream.writeObject(oneRecord);
```

ClassIODemo Excerpts

```
inputStream = new ObjectInputStream(
       new FileInputStream("species.records"));
...
Species readOne = null;
...
readOne = (Species)inputStream.readObject(oneRecord);
```

readObject returns a reference to type Object so it must be cast to Species before assigning to readOne

The **`Serializable`** Interface

- Java assigns a serial number to each object written out.
  - If the same object is written out more than once, after the first write only the serial number will be written.
  - When an object is read in more than once, then there will be more than one reference to the same object.
- If a serializable class has class instance variables then they should also be serializable.
- Why aren't all classes made serializable?
  - security issues: serial number system can make it easier for programmers to get access to object data
  - doesn't make sense in all cases, e.g., system-dependent data