File I/O & loop intro

Checked & unchecked exceptions

- There are two general types of exceptions that may occur within programs:
  - checked exceptions are those that must be accounted for or the program won’t compile
  - unchecked exceptions are those that the compiler doesn’t account for; hence the name
- All of the exceptions we have discussed so far have been unchecked exceptions

Not catching exceptions

- As stated on the previous slide, there are some types of exceptions (checked exceptions) that we must handle in order for our programs to compile
- In such a situation, a ***throws clause*** can be used instead of a try/catch block

Example

The code below contains a main method that would produce a compiler error because of a checked exception:

```
public class ThreadTest  {
        public static void main(String [] args) {
           System.out.println(“Wait for it…”);
           Thread.sleep(1000);
           System.out.println (“OK");
        }
}
```

One way to take care of the exception and get rid of the error message is shown on the following slide

Example

```
public class ThreadTest {
        public static void main(String [] args)
                                    throws InterruptedException {
            System.out.println (“Wait for it…");
            Thread.sleep(1000);
            System.out.println (“OK");
        }
}
```

The “throws clause,” which is part of the method heading, simply indicates that we acknowledge that an exception might occur, and that it will be handled elsewhere. This is the default behavior with unchecked exceptions; we only have to add a throws clause if our code could throw a checked exception.

Text file Output

1. **Import the necessary code: import java.io.*;**
2. **Declare & instantiate objects:**

```
FileOutputStream outs = null;
PrintWriter file = null;
```

3. **Open file (& deal with checked exception)**

```
try {
    outs = new FileOutputStream(“name.txt”);
} catch (FileNotFoundException e) {
    System.out.println(“Could not open file – ending program”);
    System.exit(1);
}
file = new PrintWriter(outs);
```

4. **Use PrintWriter object just as you would System.out**
5. **Close file when finished: file.close();**

Reading input from a text file

- Can use the same Scanner class we use for keyboard input to read from a text file
- To read from a text file, we construct the Scanner object by passing a FileInputStream object instead – in other words, we open an input file much the same way we open an output file
- When finished reading an input file, we close it

Syntax & example

- General syntax:

```
Scanner scanner = new Scanner
                    (new FileInputStream(name));
```

- Example:

```
Scanner input = new Scanner
        (new FileInputStream(“C:\\myfiles\\data.txt”));
```

Example – opening input file

```
FileInputStream in = null;
boolean fileNotOpen = true;
String filename;
Scanner kb = new Scanner(System.in);
Scanner file = null;
while (fileNotOpen) {
     System.out.print(“Enter name of input file: ”);
     filename = kb.nextLine();
     try {
         in = new FileInputStream(filename);
         fileNotOpen = false;
     }
     catch (FileNotFoundException e) {
         System.out.println(input + " is not valid.  Please try again.");
    } // end of catch block
} // end of loop
```

Scanner methods

- Once a Scanner object is open, you can call the same methods to read data from an input file as you have used to read data from the keyboard
  - for integers: nextInt()
  - for doubles: nextDouble()
  - for Strings: next() and nextLine()

More Scanner methods

- The Scanner class has several “look ahead” methods that can be used to determine:
  - whether or not there is more input to come
  - the nature of the input (number vs. text)
- All of the following methods return a boolean value:
  - hasNextInt()
  - hasNextDouble()
  - hasNext()