Input/Output (IO)
I/O

- So far we have looked at modeling classes
I/O

- So far we have looked at modeling classes
- Not much in the way of Input...
Input

- 3 ways of providing input to the program:
Input

- 3 ways of providing input to the program:
  - Pass parameters directly to the program
Input

- 3 ways of providing input to the program:
  - Pass parameters directly to the program
  - Command-line input from the user
Input

- 3 ways of providing input to the program:
  - Pass parameters directly to the program
  - Command-line input from the user
  - Reading in files
Passing parameters

- When running the program can directly pass parameters to it
Passing parameters

When running the program can directly pass parameters to it

```
java ProgramName parameter1 parameter2 ...
```
public static void main(String[] args) {

    // args is the array of all parameters
    // args[0] would be the first parameter

}
Passing parameters

public static void main(String[] args) {

    // args is the array of all parameters
    // args[0] would be the first parameter

}

Let’s look at an example
Command-line input

- Receive input from the console during execution
Command-line input

- Receive input from the console during execution
- Use the `Scanner` class
The Scanner class

- Used for reading data
The Scanner class

- Used for reading data

- Constructors:
  - `Scanner(InputStream)`
    - `InputStream = System.in`
The Scanner class

- Used for reading data

- Constructors:
  - `Scanner(InputStream)`
    - `InputStream = System.in`
  - `Scanner(String)`
The Scanner class

- Used for reading data

- Constructors:
  - Scanner(InputStream)
    - InputStream = System.in
  - Scanner(String)
  - Scanner(File)
The Scanner class

- **Methods:**
  - `boolean hasNext()`: If scanner has more tokens
The Scanner class

- **Methods:**
  - `boolean hasNext()`: If scanner has more tokens
  - `String next()`: Returns the next String
  - `int nextInt()`: Returns the next int
  - `double nextDouble()`: Returns the next double
The Scanner class

- **Methods:**
  - **boolean hasNext():** If scanner has more tokens
  - **String next():** Returns the next String
  - **int nextInt():** Returns the next int
  - **double nextDouble():** Returns the next double
  - **void useDelimiter(pattern: String):** Set's the delimiting pattern (" ") by default
The Scanner class

- **Methods:**
  - `boolean hasNext()`: If scanner has more tokens
  - `String next()`: Returns the next String
  - `int nextInt()`: Returns the next int
  - `double nextDouble()`: Returns the next double
  - `void useDelimiter(pattern: String)`: Set’s the delimiting pattern (" " by default)
  - `void close()`: Closes the Scanner
Command-line input

- Use the “next..” methods to read from the standard input
Use the “next..” methods to read from the standard input

```java
import java.util.Scanner;

Scanner scanner = new Scanner(System.in);
System.out.print(“Enter number1: “);
double number1 = scanner.nextDouble();

System.out.print(“Enter number2: “);
double number2 = scanner.nextDouble();

System.out.println(“The addition of the two numbers: “ + (number1 + number2));
```
File Input

- Ability to read files essential to any language.
File Input

- Ability to read files essential to any language.

- Two ways to store data:
  - Text format:
File Input

- Ability to read files essential to any language.

- Two ways to store data:
  - Text format:
    - Human-readable form
    - Can be read by text editors
File Input

- Ability to read files essential to any language.

- Two ways to store data:
  - Text format:
    - Human-readable form
    - Can be read by text editors
  - Binary format:
    - Used for executable programs
    - Cannot be read by text editors
The File class

- java.io package
- Represents a “file” object
- Used for input/output through data streams, the file system and serialization.
The File class

- **Constructors:**
  - `File(pathname: String)`: Creates a File object for the specified pathname. `pathname` = directory or file
The File class

- **Constructors:**
  - `File(pathname: String)`: Creates a File object for the specified pathname. `pathname` = directory or file
  - `File(parent: String, child: String)`: Creates a File object for the child under the directory `parent`. `child` may be a filename or subdirectory.
The File class

- **Methods:**
  - `boolean exists()`: If the file exists
The File class

Methods:

- `boolean exists()`: If the file exists
- `boolean canRead()`: If the file exists and we can read it
- `boolean canWrite()`: If the file exists and we can write to it
The File class

Methods:

- boolean exists() : If the file exists
- boolean canRead() : If the file exists and we can read it
- boolean canWrite() : If the file exists and we can write to it
- void isDirectory() : if the object is a directory
- void isFile() : if the object is a file
The File class

Methods:

- `boolean exists()`: If the file exists
- `boolean canRead()`: If the file exists and we can read it
- `boolean canWrite()`: If the file exists and we can write to it
- `void isDirectory()`: if the object is a directory
- `void isFile()`: if the object is a file
- `String getName()`: Returns the name of the file
The File class

Methods:

- `boolean exists()`: If the file exists
- `boolean canRead()`: If the file exists and we can read it
- `boolean canWrite()`: If the file exists and we can write to it
- `void isDirectory()`: If the object is a directory
- `void isFile()`: If the object is a file
- `String getName()`: Returns the name of the file
- `boolean delete()`: Deletes the file and returns true if succeeded
- `renameTo (dest: File)`: Tries to rename the file and returns true if succeeded
Reading Files

- Use the **Scanner** class

- `new Scanner(File)`
Reading Files

- How does Scanner really work?
Reading Files

- How does **Scanner** really work?

- Breaks file contents into *tokens*
  - Uses a delimiter
Reading Files

- How does `Scanner` really work?

- Breaks file contents into `tokens`
  - Uses a delimiter
  - Delimiter by default is whitespace
Reading Files

- How does **Scanner** really work?
  - Breaks file contents into *tokens*
    - Uses a delimiter
    - Delimiter by default is whitespace
  - Reads a token, converts it to the required type
Reading Files

- How does `Scanner` really work?

- Breaks file contents into `tokens`
  - Uses a delimiter
  - Delimiter by default is whitespace

- Reads a token, converts it to the required type

- Can change the delimiter – use `useDelimiter()` method
Reading Files

// Reads in the file and outputs all the tokens

Scanner input = new Scanner(new File("test.txt"));

while (input.hasNext()) {
    System.out.println(input.next());
}
Reading Files

// Reads in the file and outputs all the tokens

Scanner input = new Scanner(new File("test.txt"));

while (input.hasNext()) {
    System.out.println(input.next());
}

ERROR – WON’T COMPILE
Reading Files

// Reads in the file and outputs all the tokens

Scanner input = new Scanner(new File("test.txt"));

while (input.hasNext()) {
    System.out.println(input.next());
}

ERROR – WON’T COMPILE
The constructor throws a FileNotFoundException
Reading Files

// Reads in the file and outputs all the tokens

try {
    Scanner input = new Scanner(new File("test.txt"));

    while (input.hasNext()) {
        System.out.println(input.next());
    }
} catch (FileNotFoundException fe) {
    fe.printStackTrace();
}
Reading files

- Have to be careful. Suppose a file contains the line:
  - 34  567
Have to be careful. Suppose a file contains the line:

34 567

What will be the contents of `intValue` and `line` after the following code is executed?

```java
Scanner in = new Scanner(new File("test.txt"));
int intValue = in.nextInt();
String line = in.nextLine();
```
Scanner scanner = new Scanner("file.txt");

Treats the String “file.txt” as the source, NOT the file “file.txt”
Writing Files

- Use the PrintWriter class
Writing Files

- Use the PrintWriter class

- Constructors:
  - PrintWriter(File file): Creates a PrintWriter for the specified File
  - PrintWriter(String name): Creates a PrintWriter for the specified File with the name
The PrintWriter class

- **Methods:**
  - `void print(String)`: Writes a String
The PrintWriter class

- **Methods:**
  - `void print(String)`: Writes a String
  - `void print(int)`: Writes an int
  - `void print(float)`: Writes a float
The PrintWriter class

- **Methods:**
  - `void print(String)`: Writes a String
  - `void print(int)`: Writes an int
  - `void print(float)`: Writes a float
  - `void println(String)`: Writes a String but also adds a line separator
The PrintWriter class

- **Methods:**
  - `void print(String)`: Writes a String
  - `void print(int)`: Writes an int
  - `void print(float)`: Writes a float
  - `void println(String)`: Writes a String but also adds a line separator
  - `void flush()`: Flushes the output stream. Ensures writing to the file
The PrintWriter class

- **Methods:**
  - **void print(String) :** Writes a String
  - **void print(int) :** Writes an int
  - **void print(float) :** Writes a float
  - **void println(String) :** Writes a String but also adds a line separator
  - **void flush() :** Flushes the output stream. Ensures writing to the file
  - **void close() :** Closes the output stream.
Writing Files

```java
PrintWriter output = null;
try {
    output = new PrintWriter(new File("test"));
    // creates a file if it does not exist;
    // discards the current content if the file exists

    output.print("John T Smith "); output.println(90);
    output.print("Eric K Jones "); output.println(85);
    output.flush();
}
catch (IOException ioe) {
    System.out.println(ioe.toString());
}
finally {
    if (output != null) output.close();
}
```
Problem: What if you want to *append* to the file not replace it?
Writing Files

- **Problem**: What if you want to *append* to the file not replace it?

- **Solution 1**: Read the whole file, then write it back.
Writing Files

- **Problem:** What if you want to *append* to the file not replace it?

- **Solution 1:** Read the whole file, then write it back.
  - Cumbersome and too much work
Writing Files

- **Problem**: What if you want to *append* to the file not replace it?

- **Solution 1**: Read the whole file, then write it back.
  - Cumbersome and too much work

- **Solution 2**: Use the `FileWriter` class
Writing Files

- **Problem**: What if you want to *append* to the file not replace it?

- **Solution 1**: Read the whole file, then write it back.
  - Cumbersome and too much work

- **Solution 2**: Use the `FileWriter` class
  - `FileWriter(File file, boolean append)`
Writing Files

- **Problem:** What if you want to *append* to the file not replace it?

- **Solution 1:** Read the whole file, then write it back.
  - Cumbersome and too much work

- **Solution 2:** Use the **FileWriter** class
  - `FileWriter(File file, boolean append)`
  - `PrintWriter pw = new PrintWriter(new FileWriter(file, true))`
Writing Files

```java
PrintWriter output = null;
try {
    output = new PrintWriter(new File("test"));
    // creates a file if it does not exist;
    // discards the current content if the file exists

    output.print("John T Smith "); output.println(90);
    output.print("Eric K Jones "); output.println(85);
    output.flush();
}
catch (IOException ioe) {
    System.out.println(ioe.toString());
}
finally {
    if (output != null) output.close();
}
```
Writing Files – Append

```java
PrintWriter output = null;
try {
    output = new PrintWriter(new FileWriter(new File("test"), true));
    // creates a file if it does not exist;
    // discards the current content if the file exists

    output.print("John T Smith ");
    output.println(90);
    output.print("Eric K Jones ");
    output.println(85);
    output.flush();
}
catch(IOException ioe) {
    System.out.println(ioe.toString());
}
finally {
    if (output != null) output.close();
}
```
Summary

- Use **Scanner** for **reading** from command-line and files.
  - Based on delimiters

- Use **PrintWriter** for **writing** to files