CISC 3120
C17: File I/O and Exception Handling

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Outline

- Recap and issues
- Exception Handling
- File I/O

Exceptions and Exception Handling

- Exceptions
- Catch exceptions (try/catch)
- Specify requirement for methods (throws)
- Declare and throw exceptions (throw)
- Checked and unchecked exception
- Some Best Practice

What may can go wrong?

- A simple Java expression evaluator

```
public class SimpleExpr {
   public static void main(String[] args) {
     int d1 = Integer.parseInt(args[0]);  int d2 = Integer.parseInt(args[2]);  int result;
     switch(args[1]) {
     case "+":
        result = d1 + d2;  System.out.println(String.join(" ",  args) + " = " + result);  break;
     case "-":
        result = d1 - d2;  System.out.println(String.join(" ",  args) + " = " + result);  break;
     case "*":
        result = d1 * d2;  System.out.println(String.join(" ",  args) + " = " + result);  break;
     case "/":
        result = d1 / d2;  System.out.println(String.join(" ",  args) + " = " + result);  break;
     }
   }
}
```

Things can go wrong

```
$ java SimpleExpr ↵
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at edu.cuny.brooklyn.cisc3120.simpleexpr.SimpleExpr.main(SimpleExpr.java:3)
$ java SimpleExpr  1.0 + 2.0 ↵
Exception in thread "main" java.lang.NumberFormatException: For input string: "1.0"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at edu.cuny.brooklyn.cisc3120.simpleexpr.SimpleExpr.main(SimpleExpr.java:3)
$ java SimpleExpr  3 / 0 ↵
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at edu.cuny.brooklyn.cisc3120.simpleexpr.SimpleExpr.main(SimpleExpr.java:12)
```

Exceptions

- Java uses *exceptions* to handle errors and other exceptional events.
- Exception: exceptional situations at runtime
  - An event that occurs during the execution of a program that disrupts the normal flow of instructions.
  - Examples
    - Division-by-zero
    - Access an array element that does not exist
- When the exception situation occurs, Java throws an "exception".

Throws an Exception

- The method where an error occurs creates an "Exception" object and hands it off to the Java runtime
- Exception object, contains
  - Information about error
    - Its type and the state of the program when the error occurred..

Call Stack

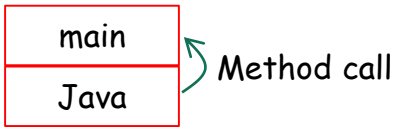
- Java runtime attempts to find objects and methods to handle it
  - An ordered list of methods that had been called to get to the method where the error occurred.
- The list of methods is known as the call stack

Example Exception

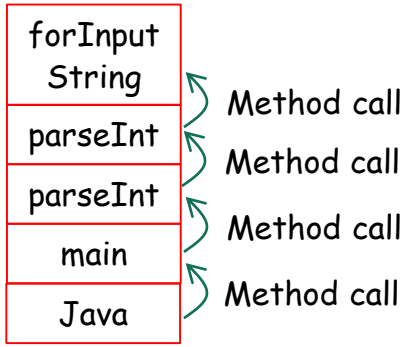
```
$ java SimpleExpr ↵
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at edu.cuny.brooklyn.cisc3120.simpleexpr.SimpleExpr.main(SimpleExpr.java:3)
$ java SimpleExpr  1.0 + 2.0 ↵
Exception in thread "main" java.lang.NumberFormatException: For input string: "1.0"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at edu.cuny.brooklyn.cisc3120.simpleexpr.SimpleExpr.main(SimpleExpr.java:3)
$ java SimpleExpr  3 / 0 ↵
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at edu.cuny.brooklyn.cisc3120.simpleexpr.SimpleExpr.main(SimpleExpr.java:12)
```

Example Call Stack

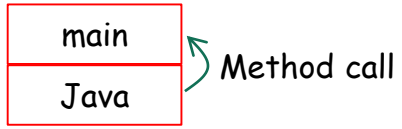
```
$ java SimpleExpr ↵
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at edu.cuny.brooklyn.cisc3120.simpleexpr.SimpleExpr.main(SimpleExpr.java:3)
```



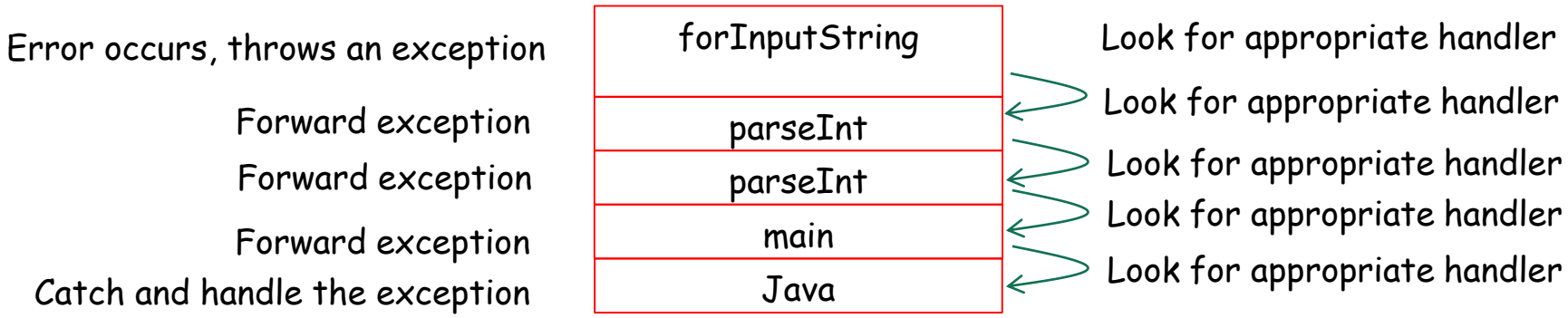

```
$ java SimpleExpr  1.0 + 2.0 ↵
Exception in thread "main" java.lang.NumberFormatException: For input string: "1.0"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at edu.cuny.brooklyn.cisc3120.simpleexpr.SimpleExpr.main(SimpleExpr.java:3)
```




```
$ java SimpleExpr  3 / 0 ↵
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at edu.cuny.brooklyn.cisc3120.simpleexpr.SimpleExpr.main(SimpleExpr.java:12)
```



Example Exception Throws and Catch



Catch or Specify Requirement

- Java code must either to catch or forward exceptions
  - A try statement that catches the exception.
    - The try must provide a handler for the exception.
  - A method that specifies that it can throw the exception.
    - The method must provide a throws clause that lists the exception.

Catching and Handling Exceptions

- Use the try-catch blocks
- Use the try-catch-finally blocks
  - To be discussed when we discuss File I/O
- Use the try with resource blocks
  - To be discussed with we discuss File I/O

Try-catch Example: One Exception

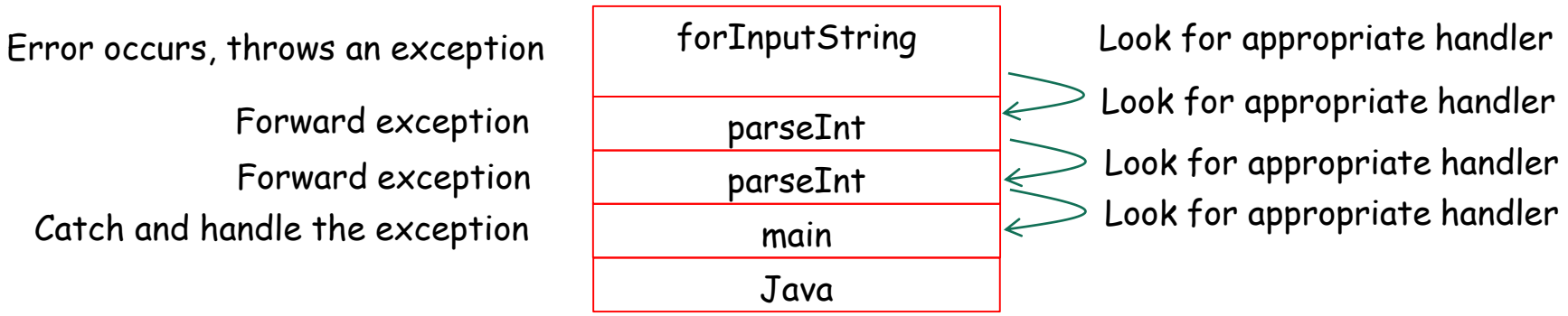
- Try and catch a single exception

```
try {
    d1 = Integer.parseInt(args[0]);
} catch (NumberFormatException e) {
    System.out.println("SimpleExprImproved must take two integer operants.“ + args[0] + " is not an integer.");
    System.exit(-1);
}
```

```
try {
    result = d1 / d2;
    System.out.println(String.join(" ",  args) + " = " + result);
} catch (ArithmeticException e) {
    System.out.println("The divisisor cannot be zero.");
    System.exit(-1);
}
```

Try-catch Example: One Exception: Call Stack

```
try {
    d1 = Integer.parseInt(args[0]);
} catch (NumberFormatException e) {
    System.out.println("SimpleExprImproved must take two integer operants." + args[0] + " is not an integer.");
    System.exit(-1);
}
```



Try-Catch Example: More Exception

- Try and catch more than one exceptions

```
try {
    d1 = Integer.parseInt(args[0]);
} catch (NumberFormatException e) {
    System.out.println("SimpleExprImproved must take two integer operants.“ + args[0] + " is not an integer.");
    System.exit(-1);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Usage: SimpleExprImproved <integer> <+|-|*|/> <integer>"
    return;
}
```

```
try {
    d1 = Integer.parseInt(args[0]);
} catch (NumberFormatException | ArrayIndexOutOfBoundsException  e) {
    System.out.println(“Either “ + args[0] + " is not an integer.“
        + “ or use it correctly: Usage: SimpleExprImproved <integer> <+|-|*|/> <integer>");
    System.exit(-1);
}
```

Specify Requirements

- What if we do want to let a method further up the call stack handle the exception?
- Specifying the Exceptions thrown by a method

Let Method Throw an Exception: Example

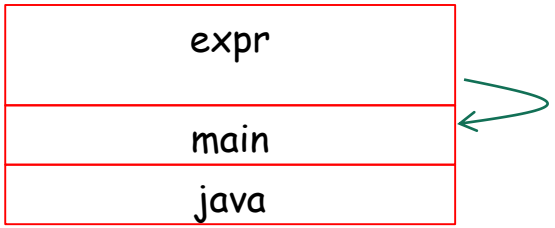
```
public class SimpleExprThrows {
    public static void main(String[] args) {
        ......
        try {
            result = expr(d1, d2, args[1]); System.out.println(d1 + args[1] + d2 + "=" + result);
        } catch(ArithmeticException e) {
            System.out.println("The divisor is 0.");
        }
    }
    public static int expr(int d1, int d2, String operator) throws ArithmeticException {
        int result = 0;
        switch(operator) {
        case "+":  result = d1 + d2; break;
        case "-":  result = d1 - d2; break;
        case "*":  result = d1 * d2; break;
        case "/":  result = d1 / d2;
        }
        return result;
    }
}
```

Let Method Throw an Exception: Example: Call Stack

```
public class SimpleExprThrows {
    public static void main(String[] args) {
        ……
        try { result = expr(d1, d2, args[1]); ….} catch(ArithmeticException e) { … }
    }
    public static int expr(int d1, int d2, String operator) throws ArithmeticException {
        int result = 0;
        switch(operator) {
         ……
        case "/":  result = d1 / d2;
        }
        return result;
    }
}
```

Error occurs, throws an exception

Catch and handle the exception



Look for appropriate handler

Look for appropriate handler

How do you dislike this code snippet?

- Can anything go wrong in addition to ArithmeticException?

```
public class SimpleExprThrows {
    ……
    public static int expr(int d1, int d2, String operator) throws ArithmeticException {
        int result = 0;
        switch(operator) {
        case "+":  result = d1 + d2; break;
        case "-":  result = d1 - d2; break;
        case "*":  result = d1 * d2; break;
        case "/":  result = d1 / d2;
        }
        return result;
    }
}
```

How do you dislike this code snippet?

- How about we do this? What's wrong?

```
$ java SimpleExpr 2 ^ 3 ↵
2^3=0
```

```
public class SimpleExprThrows {
    ……
    public static int expr(int d1, int d2, String operator) throws ArithmeticException {
        int result = 0;
        switch(operator) {
        case "+":  result = d1 + d2; break;
        case "-":  result = d1 - d2; break;
        case "*":  result = d1 * d2; break;
        case "/":  result = d1 / d2;
        }
        return result;
    }
}
```

How to Throw an Exception

- Any code can throw an exception.
- Use the throw statement.

Throw an Exception: Example

```
public class SimpleExprThrows {
   public static void main(String[] args) { ...
      try {
         result = expr(d1, d2, args[1]);  System.out.println(d1 + args[1] + d2 + "=" + result);
      } catch(ArithmeticException e) {  System.out.println("The divisor is 0.");
      } catch(IllegalArgumentException e) { System.out.println(e.getMessage()); }
   }

   public static int expr(int d1, int d2, String operator) throws ArithmeticException, IllegalArgumentException {
      int result = 0;
      switch(operator) {
      case "+":  result = d1 + d2;  break;
      case "-":  result = d1 - d2;  break;
      case "*":  result = d1 * d2; break;
      case "/":  result = d1 / d2;  break;
      default:
         throw new IllegalArgumentException("Operator " + operator + " is not supported.");
      }
      return result;
   }
}
```

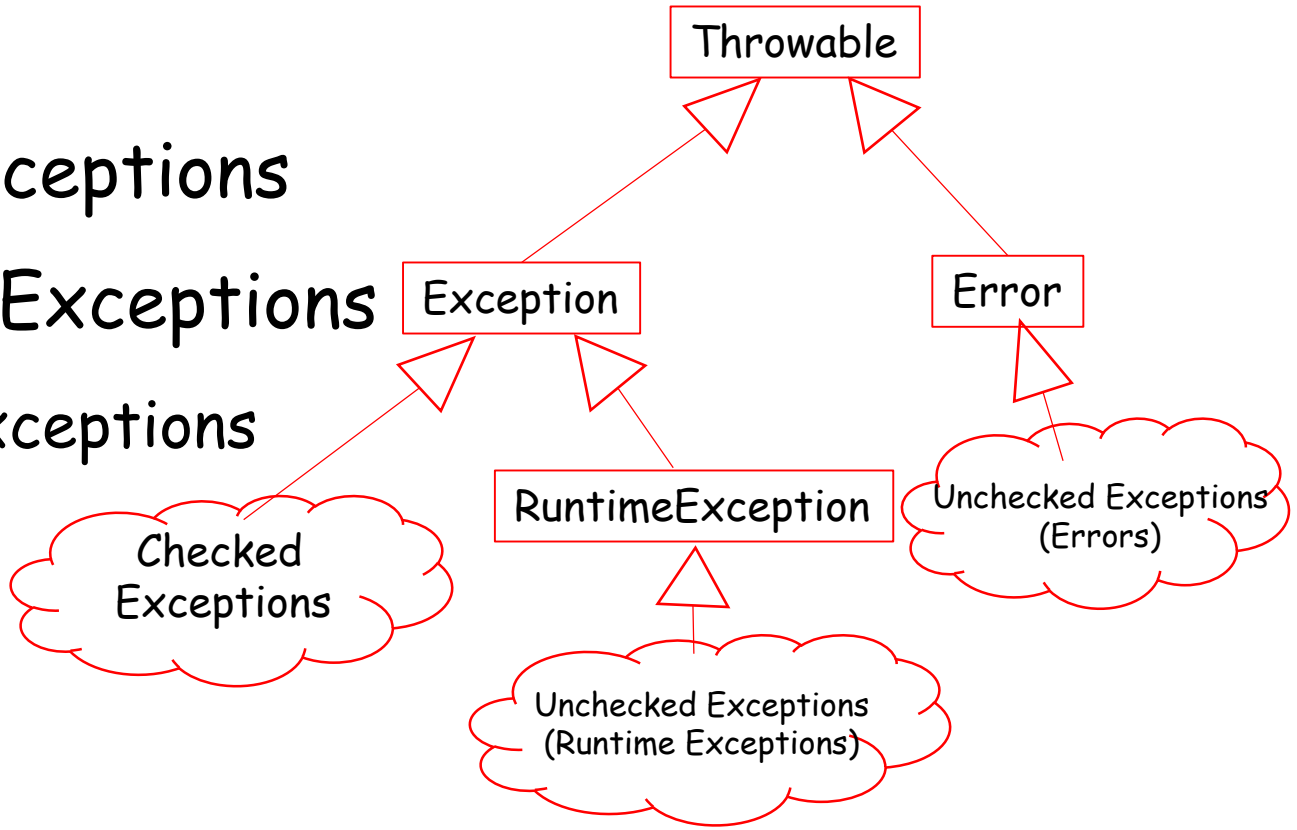
Commonly Reused Exceptions

- Use of standard exceptions are generally preferred (Bloch, J., 2008)

Exception	Occasion for Use
IllegalArgumentException	Non-null parameter value is inappropriate
IllegalStateException	Object state is inappropriate for method invocation
NullPointerException	Parameter value is null where prohibited
IndexOutOfBoundsException	Index parameter value is out of range
ConcurrentModificationException	Concurrent modification of an object has been detected where it is prohibited
UnsupportedOperationException	Object does not support method

Checked and Unchecked Exceptions, and Errors

- Java has three types of exceptions and errors
- Checked Exceptions
- Unchecked Exceptions
  - RuntimeExceptions
  - Errors



Checked Exceptions

- Semantics
  - Exceptional conditions that a well-written application should anticipate and recover from.
- Subject to the Catch or Specify Requirement.
- Checked exceptions if not Error, RuntimeException, and their subclasses.

Unchecked Exceptions

- RuntimeException and Error and their subclasses
  - Not subject to the Catch or Specify Requirement.
  - The application usually cannot anticipate or recover from.
  - Errors
    - Exceptional conditions that are external to the application.
    - Example: hardware failure: disk I/O error after a file is open, and read will fail and cannot recover from it.
  - Runtime exceptions
    - Exceptional conditions that are internal to the application.
    - Example: logical error: passed a string in a non-integer form to Integer.parseInt() that cannot recover from it

Some Best Practices

- Do throw specific Exceptions

```
throw new RunTimeException("Exception at runtime");
```

- Throw early, catch late.
  - better to throws a checked exception than to handle the exception poorly.
- Use exception only for exception situations

```
if (args.length != 3) {
   System.out.println("Usage …");
}
```

```
try {
  d1 = Integer.parseInt(args[2]);
} catch (ArrayIndexOutOfBoundsException e) {
  System.out.println("Usage …");
}
```

Questions

- Murphy's law
  - Anything that can go wrong will go wrong.

- Exceptions
- Catch exceptions (try/catch)
- Specify requirement for methods (throws)
- Declare and throw exceptions (throw)
- Checked and unchecked exception
- Some Best Practice

File I/O

- I/O streams
- File I/O (featuring NIO.2)
  - Path
  - Path Operations
  - File Operations
  - File and directory operations

I/O Streams

- A stream is a sequence of data associated with an input source or an output destination.
  - A program uses an *input stream* to read data from a source, one item at a time
  - A program uses an *output stream* to write data to a destination, one item at time

Many Streams in Java

- Byte streams
- Character streams
- Buffered streams
- Data streams
- Object streams
- Scanner and Formatter
- Standard streams and Console object

Streams

- Programs use *byte streams* to perform input and output of 8-bit bytes.
- Subclasses of
  - InputStream, an abstract class (byte stream)
  - OutputStream, an abstract class (byte stream)
  - Reader, an abstract class (character stream)
  - Writer, an abstract class (character stream)
- Classes of RandomAccessFile, Scanner, Console
- Classes implement DataInput and DataOutput interfaces

Byte Stream

- Programs use *byte streams* to perform input and output of 8-bit bytes.
- Subclasses of
  - InputStream, an abstract class
  - OutputStream, an abstract class

Byte Stream: Example

- See InputOutputStreamLesson in the sampleprograms repository
- Always close streams
- Low-level, you may have better options

Character Streams

- Java characters are in Unicode
- Character streams automatically translate Unicode to and from the local character set
- Subclasses of
  - Reader, an abstract class (character stream)
  - Writer, an abstract class (character stream)

Character Streams: Example

- See InputOutputStreamLesson in the sampleprograms repository
- Always close streams
- Deal with internationalization

Buffered Streams

- *Unbuffered* I/O
  - Each read or write request is handled directly by the underlying OS.
  - Each request often triggers disk access, network activity, or others
- Buffered I/O
  - *Buffered* I/O streams.
    - Buffered input streams read data from a memory area known as a *buffer;* the native input API is called only when the buffer is empty.
    - Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full
  - Buffered I/O streams are generally more efficient.

Buffered Streams

- Byte streams
  - BufferedInputStream
  - BufferedOutputStream
- Character streams
  - BufferedReader
  - BufferedWriter

Use Buffered Streams

- Wrap an unbuffered streams with a buffered stream
  - Example
    - inputStream = new BufferedReader(new FileReader("kaynmay.txt"));
    - outputStream = new BufferedWriter(new FileWriter(" kaynmay.txt "));
- Flushing Buffered Streams
  - Write out a buffer at critical points, without waiting for it to fill.
  - To flush a stream manually, invoke its flush method.
  - Some buffered output classes support autoflush
    - Example: an autoflush PrintWriter object flushes the buffer on every invocation of println or format.

Scanner

- Scanner breaks down inputs into tokens using a delimiter pattern
  - Delimiter pattern is expressed in Regular Expressions
  - Default delimiter pattern is whitespace
- The tokens may then be converted into values of primitive types or Strings using the various next methods.

Formatting

- PrintWriter and PrintStream support formatting
  - System.out and System.err are PrintStream objects
  - Similar to C/C++'s printf-family functions
  - More examples later

Standard Streams

- Many operating systems have Standard Streams.
- By default, they read input from the keyboard and write output to the display.
- Standard Output Streams
  - Standard output: System.out, a PrintStream object
  - Standard error: System.err, a PrintStream object
- Standard Input Streams
  - Standard input: System.in, a byte stream

The Console Class

- Access the character-based console device associated with current JVM
- Not every JVM has a console
  - If it has one, obtain it via System.console()
  - If it doesn't, System.console() returns null
- A few read and write methods

Data Streams

- Data streams support binary I/O of primitive data type values and String values
  - boolean, char, byte, short, int, long, float, and double as well as String values
- Implement either of the interfaces
  - the DataInput interface
  - the DataOutput interface

Formatted and Unformatted I/O

- Unformatted I/O
  - Transfers the internal binary representation of the data directory between memory and the file
  - Example: binary files
- Formatted I/O
  - Converts the internal binary representation to encode characters before transferring to file
  - Converts to the internal binary representation from encode characters when transferring from a file
  - Example: text files

Formatted or Unformatted?

	Formatted	Unformatted
Example	Text files	Binary files
Efficiency	Slower	Faster
Space	Larger	Smaller
Fidelity	Not exact	Exact
Portability	More	Less
Human Readability	More	Less

Examples: Formatted and Unformatted IO

- Two examples in the fileio directory of the sampleprograms repository
  - ScannerDemo
  - FormattedUnformattedFilesDemo

Object Streams

- Support I/O of objects.
- ObjectInputStream, ObjectOutputStream
  - They implement ObjectInput and ObjectOutput
  - ObjectInput is a sub-interface of DataInput
  - ObjectOutput is a sub-interface of DataOutput

File I/O

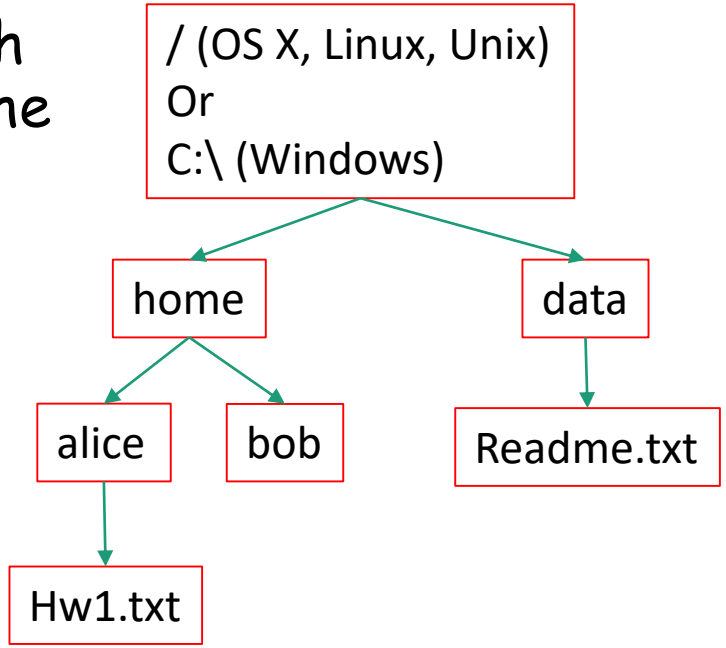
- Concept of path in OS
- The Path interface and Paths helper class
- The Files class

File System Trees

- A file system stores and organizes files on some form of media allowing easy retrieval
- Most file systems in use store the files in a tree (or hierarchical) structure.
  - Root node at the top
  - Children are files or directories (or folders in Microsoft Windows)
  - Each directory/folder can contain files and subdirectories

Path

- Identify a file by its *path* through the file system, beginning from the root node
  - Example: identify Hw1.txt
  - OS X
    - /home/alice/Hw1.txt
  - Windows
    - C:\home\alice\Hw1.txt
  - Delimiter
    - Windows: "\"
    - Unix-like: "/"



Relative and Absolute Path

- Absolute path
  - Contains the root element and the complete directory list required to locate the file
    - Example: /home/alice/Hw1.txt or C:\home\alice\Hw1.txt
- Relative path
  - Needs to be combined with another path in order to access a file.
  - Example
    - alice/Hw1.txt or alice\Hw1.txt, without knowing where alice is, a program cannot locate the file
  - "." is the path representing the current working directory
  - ".." is the path representing the parent of the current working directory

Symbolic Link

- A file-system object (source) that points to another file system object (target).
- Symbolic links are transparent to users
  - The links appear as normal files or directories, and can be acted upon by the user or application in exactly the same manner.
- Create symbolic links from the Command Line
  - Unix-like: link
  - Windows: mklink

The Path Interface

- A programmatic representation of a path in the file system.
- Use a Path object to examine, locate, manipulate files
  - It contains the file name and directory list used to construct the path.
- Reflect underlying file systems, is system-dependent.
- The file or directory corresponding to the Path might not exist.

Path Operations

- Creating a Path
- Retrieving information about a Path
- Removing redundancies from a Path
- Converting a Path
- Joining two Paths
- Creating a relative Path of two Paths
- Comparing two Paths

Path Operations: Example

- The example program is in the sampleprograms repository
  - FileInputOutLesson
    - PathDemo.java

Obtain an Instance of Path

- The Paths helper class

Modifier and Type	Method and Description
`static Path`	`get(String first, String... more)` Converts a path string, or a sequence of strings that when joined form a path string, to a `Path`.
`static Path`	`get(URI uri)` Converts the given URI to a `Path` object.

- Examples
  - Path p1 = Paths.get('alice/hw1.txt');
  - Path p1 = Paths.get('alice', 'hw1.txt');
  - Path p3 = Paths.get(URI.create("file://C:\\home\\alice\\Hw1.txt"));
- Paths.get methods is equivalent to FileSystems.getDefault().getPath methods

Retrieve Information about a Path

- Use various methods of the Path interface

```
// On Microsoft Windows use:
Path path = Paths.get("C:\\home\\alice\\hw1.txt");
// On Unix-like OS (Mac OS X) use:
// Path path = Paths.get("/home/alice/hw1.txt");
System.out.format("toString: %s%n", path.toString());
System.out.format("getFileName: %s%n", path.getFileName());
System.out.format("getName(0): %s%n", path.getName(0));
System.out.format("getNameCount: %d%n", path.getNameCount());
System.out.format("subpath(0,2): %s%n", path.subpath(0,2));
System.out.format("getParent: %s%n", path.getParent());
System.out.format("getRoot: %s%n", path.getRoot());
```

More about Path

- Normalize a Path and remove redundancy
- Convert a Path
  - To a URI
  - To absolute Path
  - To real Path
- Join two Paths
- Creating a relative Path of two Paths
- Compare two Paths, iterate Path

Convert to Real Path

- The Path.toRealPath method returns the real path of an existing file.
  - If true is passed to this method and the file system supports symbolic links, this method resolves any symbolic links in the path (thus, the real path)
  - If the Path is relative, it returns an absolute path.
  - If the Path contains any redundant elements, it returns a path with those elements removed.
- The method checks the existence of the Path
  - It throws an exception if it does not exist or cannot be accessed.

Compare Two Paths, Iterate Path

- Equals: test two paths for equality
- startsWith and endsWith: test whether a path begins or ends with a particular string
- Iterator: iterate over names of a Path
- Comparable: compare Path, e.g., for sorting

Questions?

- Concept of path
- Creating a Path
- Retrieving information about a Path
- Removing redundancies from a Path
- Converting a Path
- Joining two Paths
- Creating a relative Path of two Paths
- Comparing two Paths

File Operations

- Static methods that operate on files, directories, or other types of files.
  - Check a file or a directory
    - Does that file exist on the file system? Is it readable? Writable? Executable?
  - Delete, copy, or move a file or directory
  - Manage metadata, i.e., file and file store attributes
  - Read, write, and create files; create and read directories; read, write, and create random access file
  - Create and read directories
  - Deal with Links
  - Walk a file tree, find files, watch a directory for changes

Random Access and Sequential Access

- Sequential access
  - Read sequentially, byte or character or other units are read sequentially one after another
  - Generally, streams must be read sequentially
- Random access
  - Behaves like a large array of bytes stored in the file system.
  - Any byte or character or other units can be read without having to read anything before it first
  - RandomAccessFile

Random Access File

- Generally, provides
  - Length
    - the size of the file
  - File pointer/Cursor
    - an index into the implied array, pointing to the byte next read reads from or next write writes to.
    - Each read or write results an advancement of the pointer
    - The file pointer can be obtained
  - Seek:
    - Set the file pointer
  - Generally, unformatted files (binary files)

When to use Random Access Files?

- For applications
  - only interested in a portion of the file.
  - read and write large files that cannot be load into the memory

Example: Random Access File

- RandomAccessFileDemo in the sampleprograms repository
- RandomAccessFile
  - Must be opened in a mode
    - "r": read-only
    - "rw": read & write
    - "rws": read, and write content & metadata synchronously
    - "rwd": read, and write content synchronously

A Few Common Concepts

- File operations use system resources
  - Release resources, one of the two, if permissible
    - close() method to release resources explicitly
    - Try-with-resources blocks (if implemented Autocloseable)
- File operations often throws exceptions
  - Catch or specify requirement for the exceptions
- Method chaining
- Link awareness
- Variable arguments (varargs)
- Some file operations are "atomic"
- Two methods of Files accept Glob parameter

Questions

- Check a file or a directory
- Delete, copy, or move a file or directory
- Manage metadata, i.e., file and file store attributes
- Read, write, and create files; create and read directories; read, write, and create random access file
- Create and read directories
- Deal with Links
- Walk a file tree, find files, watch a directory for changes

Assignments

- Practice assignment to be available