Java and Streams

61FIT3NPR -Network Programming

Faculty of Information Technology
Hanoi University
Fall 2020

Network programming

- Computer network have parts:
  - (Physical) Hardware
  - (Logical) Software
- Physical communication
  - Physical communication protocol
  - Driver intergrated in BIOS/OS
  - Programming: Drivers, APIs

Network programming

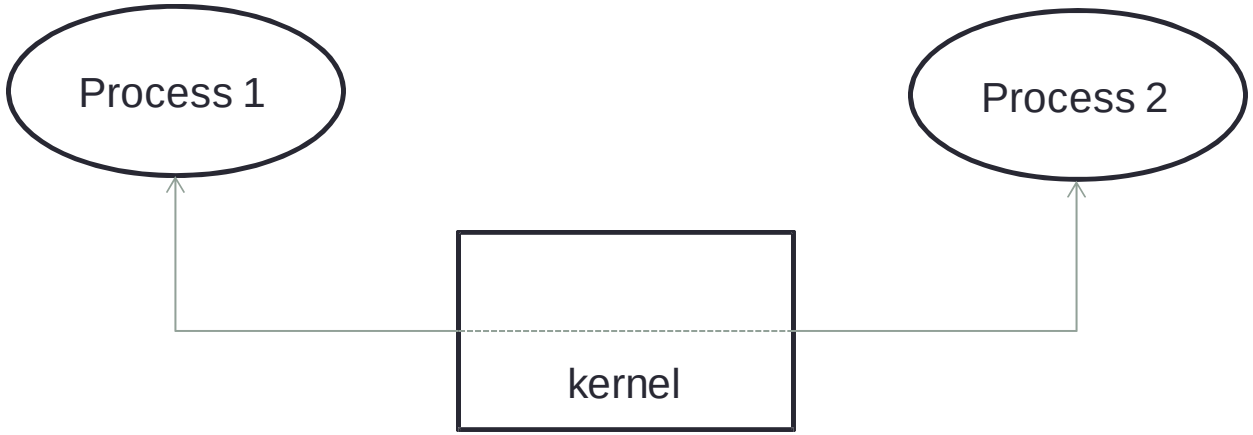
- Logical communication
  - Low protocol: spectified in RFCs, protocol programming
  - High protocol: spectified in RFCs, protocol programming.
  - Network application: User's application which communicates to another across the network, we can use API to do the programming.
  -

Interprocess communication

- Process
- Multitasking
- IPC (Inter-Process Communication)

Two types of IPC

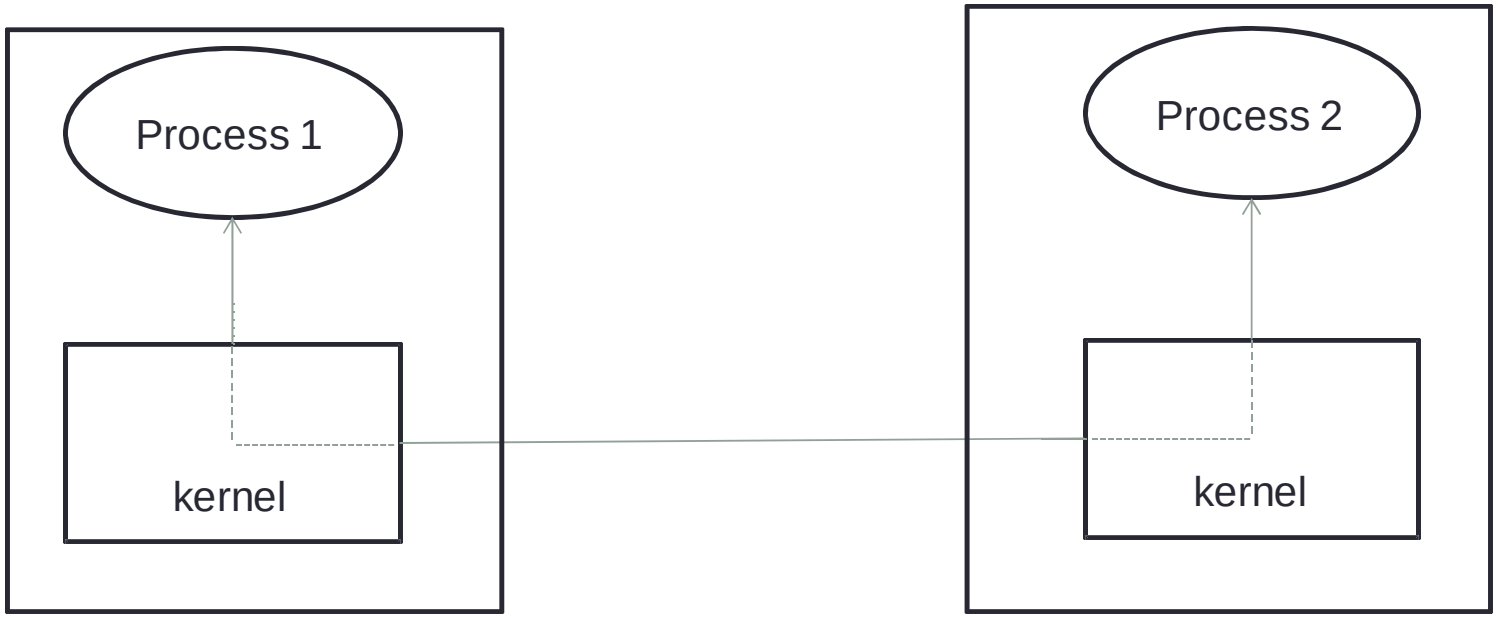
- Processes in the same computer



Two types of IPC

- 2 processes in the different computers

Kernel use somes data communication rules. Rules are a part of protocol.

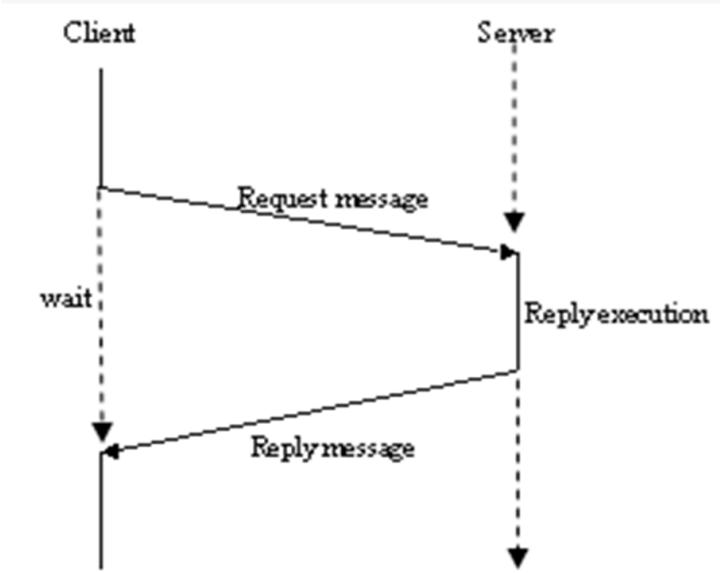


Network application

- Application provides services to the users:
  - Storage and access files (ftp)
  - Remote access (telnet)
  - Printer
  - Web browser
- Network application have 3 types:
  - peer-to-peer,
  - client-server,
  - Hybrid model.

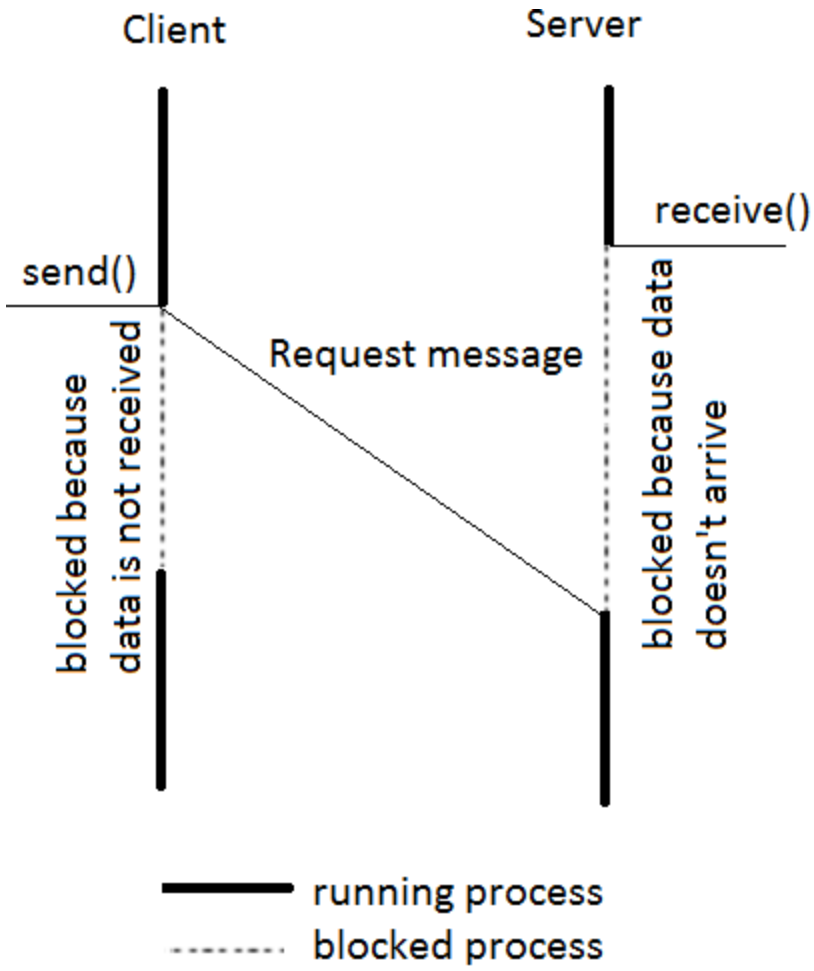
Client-server model

- There are 2 sides:
  - Server: which provider services to clients
  - Client: which request the services from server



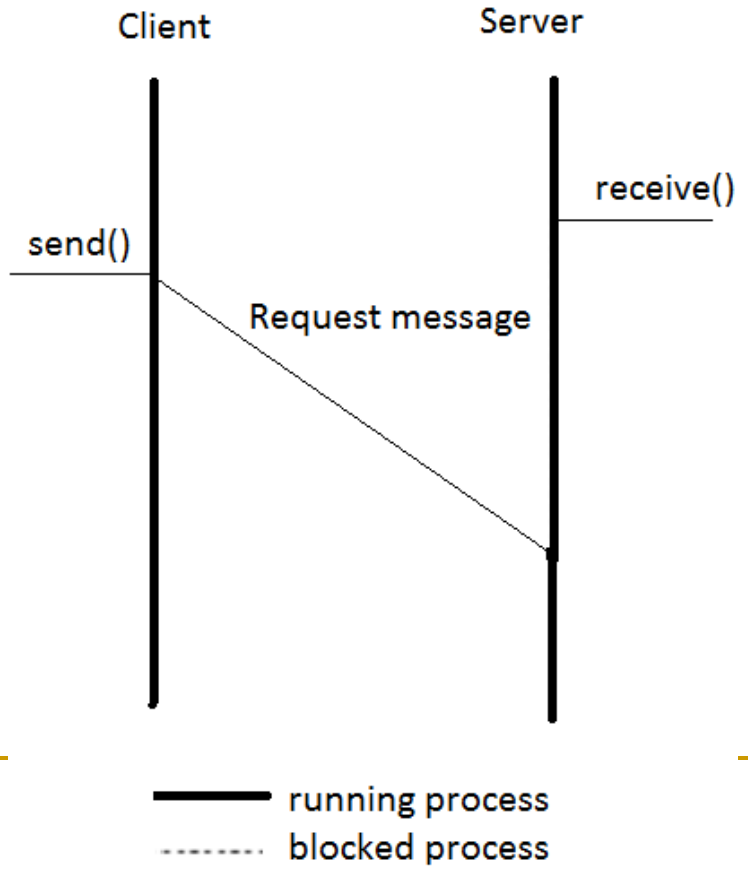
2 communication modes

- Blocking mode:
  - when client or server "send" a request, client and server are blocked until the receiver send the "receive" command to receive data.
  - Or calling a "receive" command when "send" command does not come to the server, server is blocked until data was sent to.



2 communication modes

- Nonblocking modes: Client and server process continue to run, non blocked when client or server “send” a request.



Program/application architecture

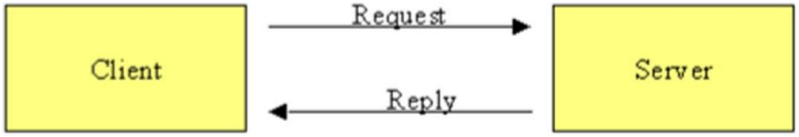
- There are 3 services/functions of a program/application:
  - user interface service
  - business rule service
  - data storage service
- At physical level, these functions/ services are coded in 1 or many code file depending on program architecture:
  - Single/1-tier architecture
  - 2-tier architecture
  - 3-tier architecture

Single-tier architecture

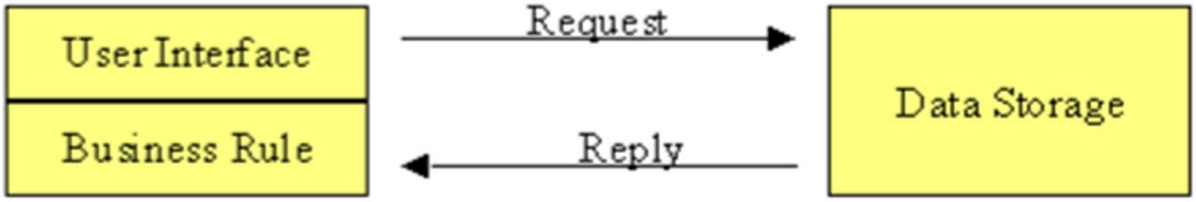
- All 3 functions of the program are coded in only ONE code file.

Two-tier architecture

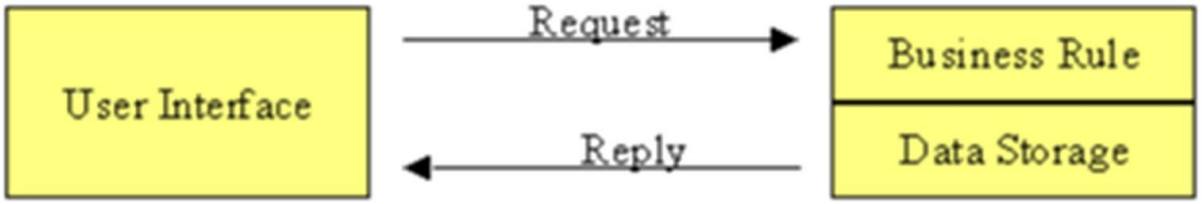
- So called client-server
- Is 2 processes running on 2 different machines or on the same machine.
- Each process must have a protocol for communication



- Client usually focus on the user interface
- Server do the data storage
- Business rule service may be on
  - client Fat client
  - or server Fat server



Fat Client



Fat Server

3-tier architecture

JAVA
Review

Topics of the Review

- Essentials of *object-oriented programming,* ***in Java***
- Java primitive data types, control structures, and arrays
- Using some predefined classes:
  - **`Math`**
  - **`JOptionPane`**, I/O streams
  - **`String`**, **`StringBuffer`**, **`StringBuilder`**
  - **`StringTokenizer`**
- Writing *and documenting* your own Java classes

Some Characteristics of Java

- Java is ***platform independent:*** the same program can run on any correctly implemented Java system
- Java is ***object-oriented:***
  - Structured in terms of ***classes***, which group data with operations on that data
  - Can construct new classes by ***extending*** existing ones
- Java designed as
  - A ***core language*** plus
  - A rich collection of ***commonly available packages***
- Java can be embedded in Web pages

Java Processing and Execution

- Begin with Java ***source code*** in text files: **`Model.java`**
- A Java source code compiler produces Java ***byte code***
  - Outputs one file per class: **`Model.class`**
  - May be standalone or part of an IDE
- A ***Java Virtual Machine*** loads and executes class files
  - May compile them to native code (e.g., x86) internally

Classes and Objects

- The ***class*** is the unit of programming
- A Java program is a ***collection of classes***
  - Each class definition (usually) in its own **`.java`** file
  - *The file name must match the class name*
- A class describes ***objects (instances)***
  - Describes their common characteristics: is a *blueprint*
  - Thus all the instances have these same characteristics
- These characteristics are:
  - ***Data fields*** for each object
  - ***Methods*** (operations) that do work on the objects

Grouping Classes: The Java API

- API = *Application Programming Interface*
- Java = small core + extensive collection of packages
- A ***package*** consists of some related Java classes:
  - Swing: a GUI (graphical user interface) package
  - AWT: Application Window Toolkit (more GUI)
  - util: utility data structures
- The ***import*** statement tells the compiler to make available classes and methods of another package
- A ***main*** method indicates where to begin executing a class (if it is designed to be run as a program)

A Little Example of **`import`** and **`main`**

```
import javax.swing.*;
    // all classes from javax.swing
public class HelloWorld {  // starts a class
  public static void main (String[] args) {
  // starts a main method
  // in: array of String; out: none (void)
  }
}
```

- **`public`** = can be seen from any package
- **`static`** = not “part of” an object

Processing and Running **`HelloWorld`**

- **`javac HelloWorld.java`**
  - Produces **`HelloWorld.class`** (byte code)
- **`java HelloWorld`**
  - Starts the JVM and runs the **`main`** method

Name principle

- Name with uppercase and lowercase are different
- Name can include characters, numbers, _ and $.
- Name can not start with a number.
- Class name: The first character of the word must be uppercase
  - Example: InputStream, WhileDemo
- Varial, constant or method name: The first word is lowercase, the first character of the remaining words are uppercase.
  - Example: maLop, giaTriDau, getInputStream()

References and Primitive Data Types

- Java distinguishes two kinds of entities
  - Primitive types
  - Objects
- Primitive-type data is stored in primitive-type variables
- Reference variables store the *address of* an object
  - No notion of “object (physically) in the stack”
  - No notion of “object (physically) within an object”

Primitive Data Types

- Represent numbers, characters, boolean values
- Integers: byte, short, int, and long
- Real numbers: float and double
- Characters: char

Primitive Data Types

Data type	Range of values
`byte`	-128 .. 127 (8 bits)
`short`	-32,768 .. 32,767 (16 bits)
`int`	-2,147,483,648 .. 2,147,483,647 (32 bits)
`long`	-9,223,372,036,854,775,808 .. ... (64 bits)
`float`	$+/-10^{-38}$ to $+/-10^{+38}$ and 0, about 6 digits precision
`double`	$+/-10^{-308}$ to $+/-10^{+308}$ and 0, about 15 digits precision
`char`	Unicode characters (generally 16 bits per char)
`boolean`	True or false

Operators

1. subscript `[ ]`, call `( )`, member access `.`
2. pre/post-increment `++ --`, boolean complement `!`, bitwise complement `~`, unary `+ -`, type cast `(type)`, object creation `new`
3. `* / %`
4. binary `+ -` (`+` also concatenates strings)
5. signed shift `<< >>`, unsigned shift `>>>`
6. comparison `< <= > >=`, class test `instanceof`
7. equality comparison `== !=`
8. bitwise and `&`
9. bitwise or `|`

Operators

11. logical (sequential) and **`&&`**
12. logical (sequential) or **`||`**
13. conditional **`cond ? true-expr : false-expr`**
14. assignment **`=`**, compound assignment **`+= -= *= /= <<= >>= >>>= &= |=`**

Type Compatibility and Conversion

- **Widening conversion:**
  - In operations on mixed-type operands, the numeric type of the smaller range is converted to the numeric type of the larger range
  - In an assignment, a numeric type of smaller range can be assigned to a numeric type of larger range
- **`byte`** to **`short`** to **`int`** to **`long`**
- **`int`** kind to **`float`** to **`double`**

Declaring and Setting Variables

- `int square;`

  `square = n * n;`
- `double cube = n * (double)square;`
  - Can generally declare local variables where they are initialized
  - All variables get a safe initial value anyway (zero/null)

Referencing and Creating Objects

- You can **declare reference variables**
  - They reference objects of **specified types**
- Two reference variables can reference **the same object**
- The **`new`** operator creates an instance of a class
- A **constructor** executes when a new object is created
- Examp

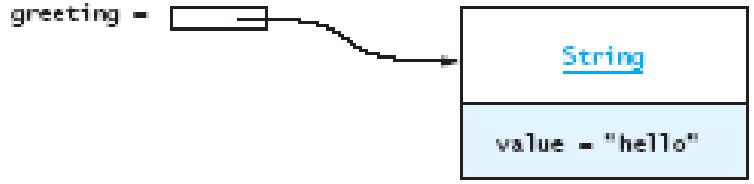


FIGURE A.2
Variable greeting References a String Object

```
lo";
```

Java Control Statements

- A group of statements executed in order is written
  - `{ stmt1; stmt2; ...; stmtN; }`
- The statements execute in the order 1, 2, ..., N
- Control statements alter this sequential flow of execution

Java Control Statements (continued)

TABLE A.4
Java Control Statements

Control Structure	Purpose	Syntax

`if ... else`	Used to write a decision with *conditions* that select the alternative to be executed. Executes the first (second) alternative if the *condition* is true (false).	`if (condition) {`
		`  ...`
		`} else {`
		`  ...`
		`}`

`switch`	Used to write a decision with scalar values (integers, characters) that select the alternative to be executed. Executes the *statements* following the *label* that is the *selector* value. Execution falls through to the next `case` if there is no `return` or `break`. Executes the statements following `default` if the *selector* value does not match any *label*.	`switch (selector) {`
		`  case label : statements; break;`
		`  case label : statements; break;`
		`  ...`
		`  default : statements;`
		`}`

`while`	Used to write a loop that specifies the repetition *condition* in the loop header. The *condition* is tested before each iteration of the loop and, if it is true, the loop body executes; otherwise, the loop is exited.	`while (condition) {`
		`  ...`
		`}`

`for`	Used to write a loop that specifies the *initialization*, repetition *condition*, and *update* steps in the loop header. The *initialization* statements execute before loop repetition begins; the *condition* is tested before each iteration of the loop and, if it is true, the loop body executes; otherwise, the loop is exited. The *update* statements execute after each iteration.	`for (initialization; condition; update) {`
		`  ...`
		`}`

Java Control Statements (continued)

TABLE A.4 (continued)

Control Structure	Purpose	Syntax

`do ... while`	Used to write a loop that specifies the repetition *condition* after the loop body. The *condition* is tested after each iteration of the loop and, if it is true, the loop body is repeated; otherwise, the loop is exited. The loop body always executes at least one time.	`do {`
		`  ...`
		`while (`*condition*`) ;`

Methods

- A Java method defines a group of statements as performing a particular operation
- **`static`** indicates a ***static*** or ***class*** method
- A method that is not **`static`** is an ***instance*** method
- All method arguments are ***call-by-value***
  - Primitive type: *value* is passed to the method
  - Method may modify local copy ***but*** will not affect caller's value
  - Object reference: *address of object* is passed
  - Change to reference variable does not affect caller
  - ***But*** operations can affect the object, visible to caller

The Class **Math**

TABLE A.5
Class `Math` Methods

Method	Behavior
`static` *numeric* `abs(`*numeric*`)`	Returns the absolute value of its *numeric* argument (the result type is the same as the argument type).
`static double ceil(double)`	Returns the smallest whole number that is not less than its argument.
`static double cos(double)`	Returns the trigonometric cosine of its argument (an angle in radians).
`static double exp(double)`	Returns the exponential number *e* (i.e., 2.718 ...) raised to the power of its argument.
`static double floor(double)`	Returns the largest whole number that is not greater than its argument.
`static double log(double)`	Returns the natural logarithm of its argument.
`static` *numeric* `max(`*numeric*`,` *numeric*`)`	Returns the larger of its *numeric* arguments (the result type is the same as the argument types).
`static` *numeric* `min(`*numeric*`,` *numeric*`)`	Returns the smaller of its *numeric* arguments (the result type is the same as the argument type).
`static double pow(double, double)`	Returns the value of the first argument raised to the power of the second argument.
`static double random()`	Returns a random number greater than or equal to 0.0 and less than 1.0.
`static double rint(double)`	Returns the closest whole number to its argument.
`static long round(double)`	Returns the closest `long` to its argument.
`static int round(float)`	Returns the closest `int` to its argument.
`static double sin(double)`	Returns the trigonometric sine of its argument (an angle in radians).
`static double sqrt(double)`	Returns the square root of its argument.
`static double tan(double)`	Returns the trigonometric tangent of its argument (an angle in radians).
`static double toDegrees(double)`	Converts its argument (in radians) to degrees.
`static double toRadians(double)`	Converts its argument (in degrees) to radians.

Escape Sequences

- An escape sequence is a sequence of two characters beginning with the character \
- A way to represents special characters/symbols

TABLE A.6
Escape Sequences

Sequence	Meaning
`\n`	Start a new output line
`\t`	Tab character
`\\`	Backslash character
`\"`	Double quote
`\'`	Single quote or apostrophe
`\u`*dddd*	The Unicode character whose code is *dddd* where each digit *d* is a hexadecimal digit in the range `0` to `F` (`0`–`9`, `A`–`F`)

The `String` Class

- The **`String`** class defines a data type that is used to store a sequence of characters
- You cannot modify a **`String`** object
  - If you attempt to do so, Java will create a new object

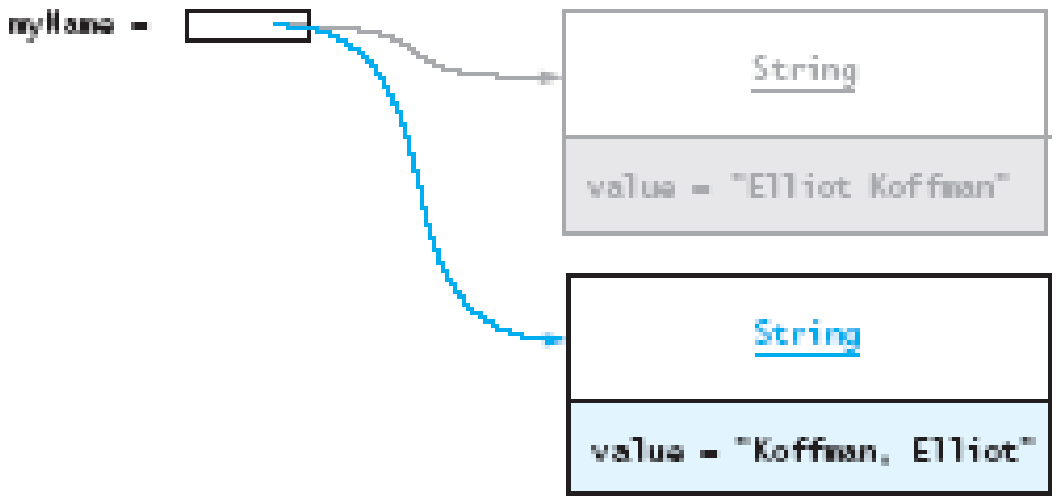


FIGURE A.4
Old and New Strings Referenced by myName

Comparing Objects

- You ***can't use the relational or equality operators*** to compare the values stored in strings (or other objects)

  (You will compare the *pointers*, not the *objects*!)

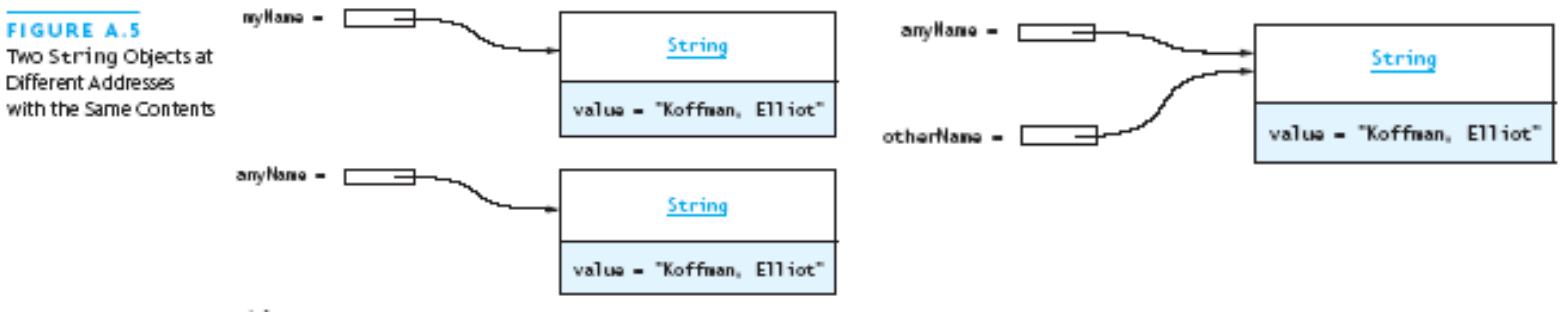


FIGURE A.5
Two String Objects at Different Addresses with the Same Contents

The `StringBuffer` Class

- Stores character sequences
- Unlike a **`String`** object, you ***can*** change the contents of a **`StringBuffer`** object

TABLE A.8
`StringBuffer` Methods in `java.lang.StringBuffer`

Method	Behavior
`void StringBuffer append(`*anyType*`)`	Appends the string representation of the argument to this `StringBuffer`. The argument can be of any data type.
`int capacity()`	Returns the current capacity of this `StringBuffer`.
`void StringBuffer delete(int start, int end)`	Removes the characters in a substring of this `StringBuffer`, starting at position `start` and ending with the character at position `end - 1`.
`void StringBuffer insert(int offset,` *anyType* `data)`	Inserts the argument `data` (any data type) into this `StringBuffer` at position `offset`, shifting the characters that started at `offset` to the right.
`int length()`	Returns the length (character count) of this `StringBuffer`.
`StringBuffer replace(int start, int end, String str)`	Replaces the characters in a substring of this `StringBuffer` (from position `start` through position `end - 1`) with characters in the argument `str`. Returns this `StringBuffer`.
`String substring(int start)`	Returns a new string containing the substring that begins at the specified index `start` and extends to the end of this `StringBuffer`.
`String substring(int start, int end)`	Return a new string containing the substring in this `StringBuffer` from position `start` through position `end - 1`.
`String toString()`	Returns a new string that contains the same characters as this `StringBuffer` object.

`StringTokenizer` Class

- We often need to process individual pieces, or *tokens*, of a **`String`**

TABLE A.9
`StringTokenizer` Methods in `java.util.StringTokenizer`

Method	Behavior
`StringTokenizer(String str)`	Constructs a new `StringTokenizer` object for the string specified by `str`. The delimiters are "whitespace" characters (space, newline, tab, and so on).
`StringTokenizer(String str, String delim)`	Constructs a new `StringTokenizer` object for the string specified by `str`. The delimiters are the characters specified in `delim`.
`boolean hasMoreTokens()`	Returns `true` if this tokenizer's string has more tokens; otherwise, returns `false`.
`String nextToken()`	Returns the next token of this tokenizer's string if there is one; otherwise, a run-time error will occur.

Wrapper Classes for Primitive Types

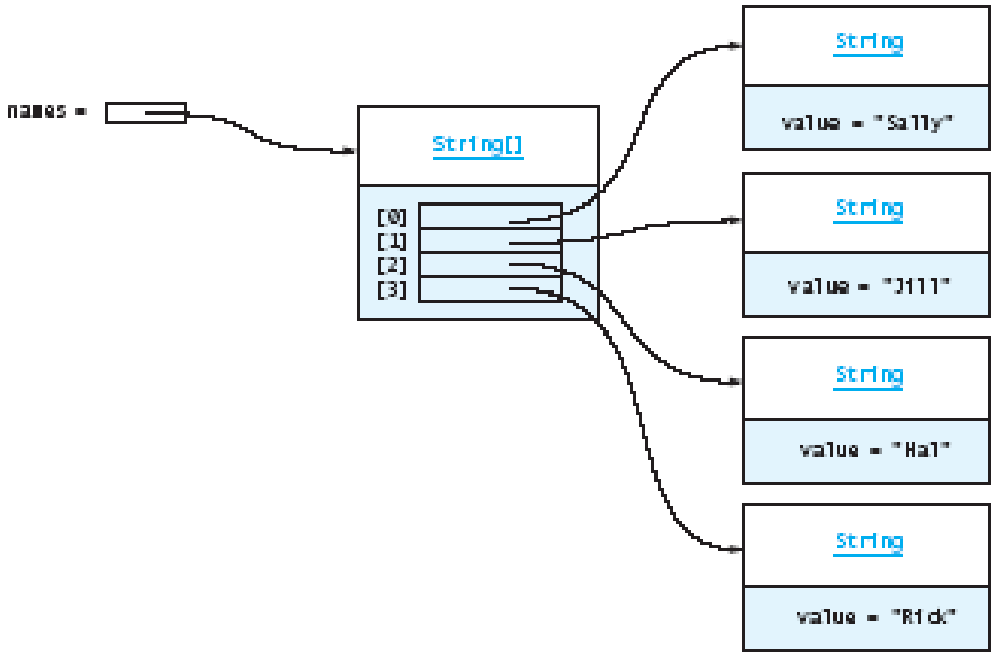
- Sometimes we need to process primitive-type data as objects
- Java provides a set of classes called wrapper classes whose objects contain primitive-type

TABLE A.10
Methods for Class `Integer`

Method	Behavior
`int compareTo(Integer anInt)`	Compares two `Integers` numerically.
`double doubleValue()`	Returns the value of this `Integer` as a `double`.
`boolean equals(Object obj)`	Returns `true` if the value of this `Integer` is equal to its argument's value; returns `false` otherwise.
`int intValue()`	Returns the value of this `Integer` as an `int`.
`static int parseInt(String s)`	Parses the string argument as a signed integer.
`String toString()`	Returns a `String` object representing this `Integer`'s value.

Arrays

- In Java, an array is also an object
- The elements are indexes and are referenced using the form **`arrayvar[subscript]`**



Array Example

```
float grades[] = new float[numStudents];
... grades[student] = something; ...

float total = 0.0;
for (int i = 0; i < grades.length; ++i) {
  total += grades[i];
}
System.out.printf("Average = %6.2f%n",
                  total / numStudents);
```

Array Example Variations

```
// possibly more efficient
for (int i = grades.length; --i >= 0; ) {
  total += grades[i];
}

// uses Java 5.0 "for each" looping
for (float grade : grades) {
  total += grade;
}
```

Input/Output using Class `JOptionPane`

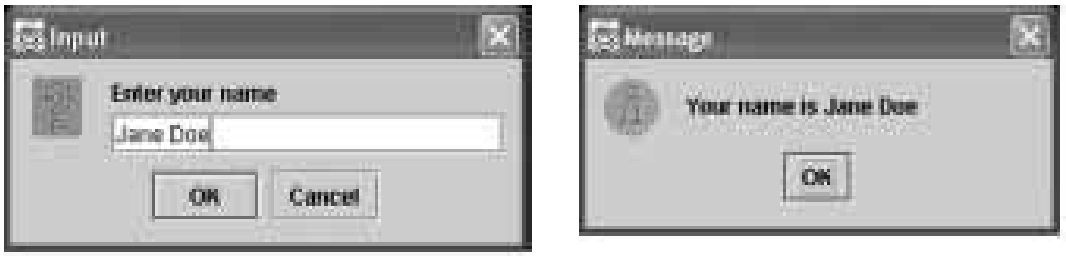
- Java 1.2 and higher provide class **`JOptionPane`**, which facilitates display
  - Dialog windows for input
  - Message windows for output

Input/Output using Class **`JOptionPane`** (continued)

TABLE A.13
Methods from Class `JOptionPane`

Method	Behavior
`static String showInputDialog(String prompt)`	Displays a dialog window that displays the argument as a prompt and returns the character sequence typed by the user.
`static void showMessageDialog(Object parent, String message)`	Displays a window containing a `message` string (the second argument) inside the specified container (the first argument).

FIGURE A.15
A Dialog Window (Left) and Message Window (Right)



Converting Numeric Strings to Numbers

- A dialog window always returns a reference to a **`String`**
- Therefore, a conversion is required, using **`static`** methods of class **`String`**:

TABLE A.14
Methods for Converting Strings to Numbers

Method	Behavior
`static int parseInt(String)`	Returns an `int` value corresponding to its argument string. A `NumberFormatException` occurs if its argument string contains characters other than digits.
`static double parseDouble(String)`	Returns a `double` value corresponding to its argument string. A `NumberFormatException` occurs if its argument string does not represent a real number.

Input/Output using Streams

- An **`InputStream`** is a sequence of characters representing program input data
- An **`OutputStream`** is a sequence of characters representing program output
- The console keyboard stream is **`System.in`**
- The console window is associated with **`System.out`**

Opening and Using Files: Reading Input

```
import java.io.*;
public static void main (String[] args) {
  // open an input stream   (**exceptions!)
  BufferedReader rdr =
    new BufferedReader(
      new FileReader(args[0]));
  // read a line of input
  String line = rdr.readLine();
  // see if at end of file
  if (line == null) { ... }
```

Opening and Using Files: Reading Input (2)

```
// using input with StringTokenizer
StringTokenizer sTok =
  new StringTokenizer (line);
while (sTok.hasMoreElements()) {
  String token = sTok.nextToken();
  ...;
}
// when done, always close a stream/reader
rdr.close();
```

Alternate Ways to Split a `String`

- Use the **`split`** method of **`String`**:

```
String[] = s.split("\\s");
// see class Pattern in java.util.regex
```

- Use a **`StreamTokenizer`** (in `java.io`)

Opening and Using Files: Writing Output

```
// open a print stream   (**exceptions!)
PrintStream ps = new PrintStream(args[0]);
// ways to write output
ps.print("Hello");  // a string
ps.print(i+3);      // an integer
ps.println(" and goodbye."); // with NL
ps.printf("%2d %12d%n", i, 1<<i); // like C
ps.format("%2d %12d%n", i, 1<<i); // same
// closing output streams is very important!
ps.close();
```

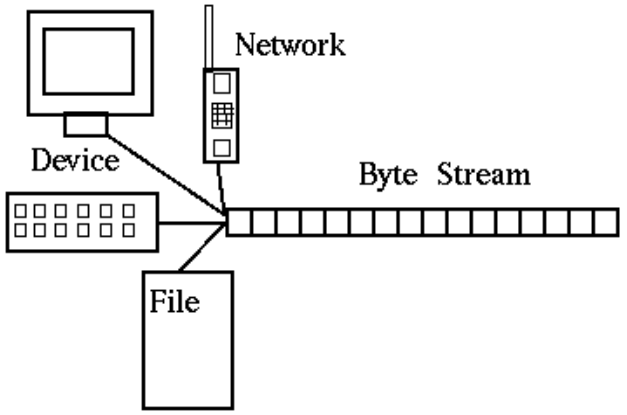
JAVA

Streams

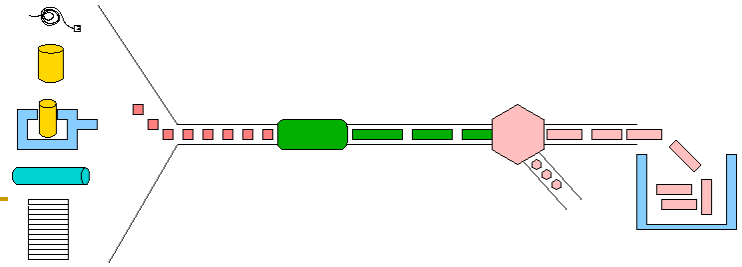
Input and output streams

- **stream**: an abstraction of a source or target of data
  - 8-bit bytes flow to (output) and from (input) streams

- can represent many data sources:
  - files on hard disk
  - another computer on network
  - web page
  - input device (keyboard, mouse, etc.)

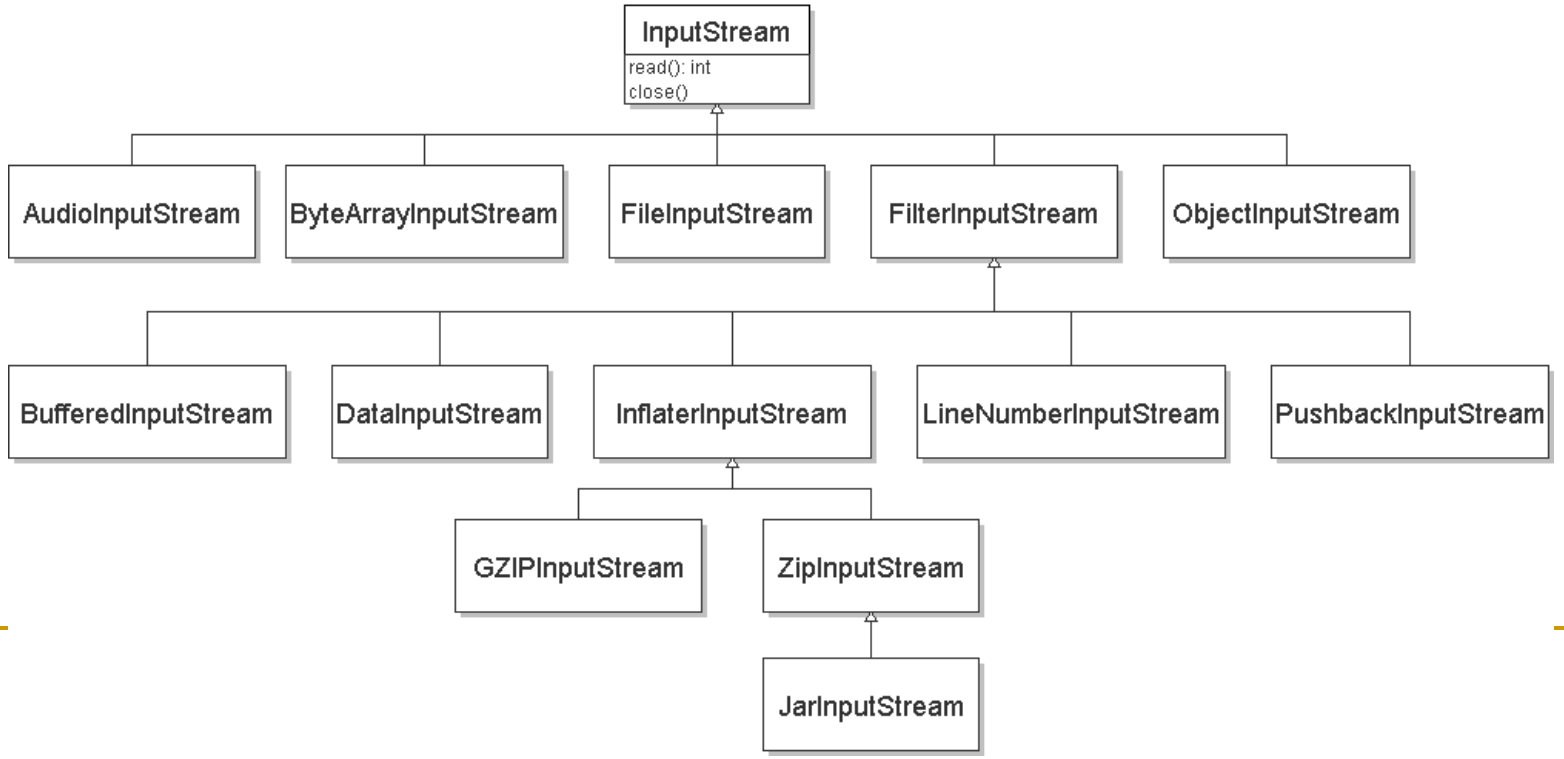


- represented by `java.io` classes
  - `InputStream`
  - `OutputStream`



Streams and inheritance

- all input streams extend common superclass `InputStream`;
  all output streams extend common superclass `OutputStream`
  - guarantees that all sources of data have the same methods
  - provides minimal ability to read/write one byte at a time



Input streams

- constructing an input stream:

Constructor
`public FileInputStream(String name) throws IOException`
`public ByteArrayInputStream(byte[] bytes)`
`public SequenceInputStream(InputStream a, InputStream b)`

(various objects also have methods to get streams to read them)

- methods common to all input streams:

Method	Description
`public int` **`read`**`() throws IOException`	reads/returns a byte (-1 if no bytes remain)
`public void` **`close`**`() throws IOException`	stops reading

Output streams

- constructing an output stream:

Constructor
`public FileOutputStream(String name) throws IOException`
`public ByteArrayOutputStream()`
`public PrintStream(File file)` `public PrintStream(String fileName)`

- methods common to all output streams:

Method	Description
`public void` **`write`**`(int b) throws IOException`	writes a byte
`public void` **`close`**`() throws IOException`	stops writing (also flushes)
`public void flush() throws IOException`	forces any writes in buffers to be written

int vs. char

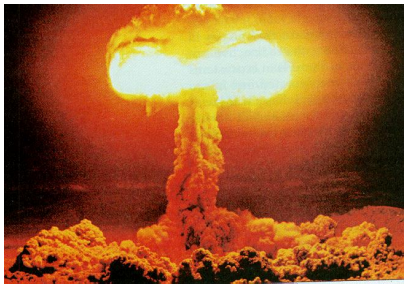
- The `read` and `write` methods work an `int` (byte) at a time.
- For text files, each byte is just an ASCII text character.
  - an `int` can be cast to `char` as needed:

```
FileInputStream in = new FileInputStream("myfile.txt");
int n = in.read();     // 81
char ch = (char) n;    // 'Q'
```

  - a `char` can be passed where an `int` is wanted without casting:

```
FileOutputStream out = new FileOutputStream("outfile.txt");
char ch = 'Q';
out.write(ch);
```

I/O and exceptions



- **exception**: An object representing an error.
  - **checked exception**: One that must be handled for the program to compile.
- Many I/O tasks throw exceptions.
  - Why?
- When you perform I/O, you must either:
  - also **throw** that exception yourself
  - **catch** (handle) the exception

Throwing an exception

```
public type name(params) throws type {
```

- **`throws` clause**: Keywords on a method's header that state that it may generate an exception.
  - Example:

```
public class ReadFile {
    public static void main(String[] args)
            throws FileNotFoundException {
```

"I hereby announce that this method might throw an exception, and I accept the consequences if it happens."

Catching an exception

```
try {
    statement(s);
} catch (type name) {
    code to handle the exception
}
```

- The `try` code executes. If the given exception occurs, the `try` block stops running; it jumps to the `catch` block and runs that.

```
try {
    Scanner input = new Scanner(new File("foo"));
    System.out.println(input.nextLine());
} catch (FileNotFoundException e) {
    System.out.println("File was not found.");
}
```

Dealing with an exception

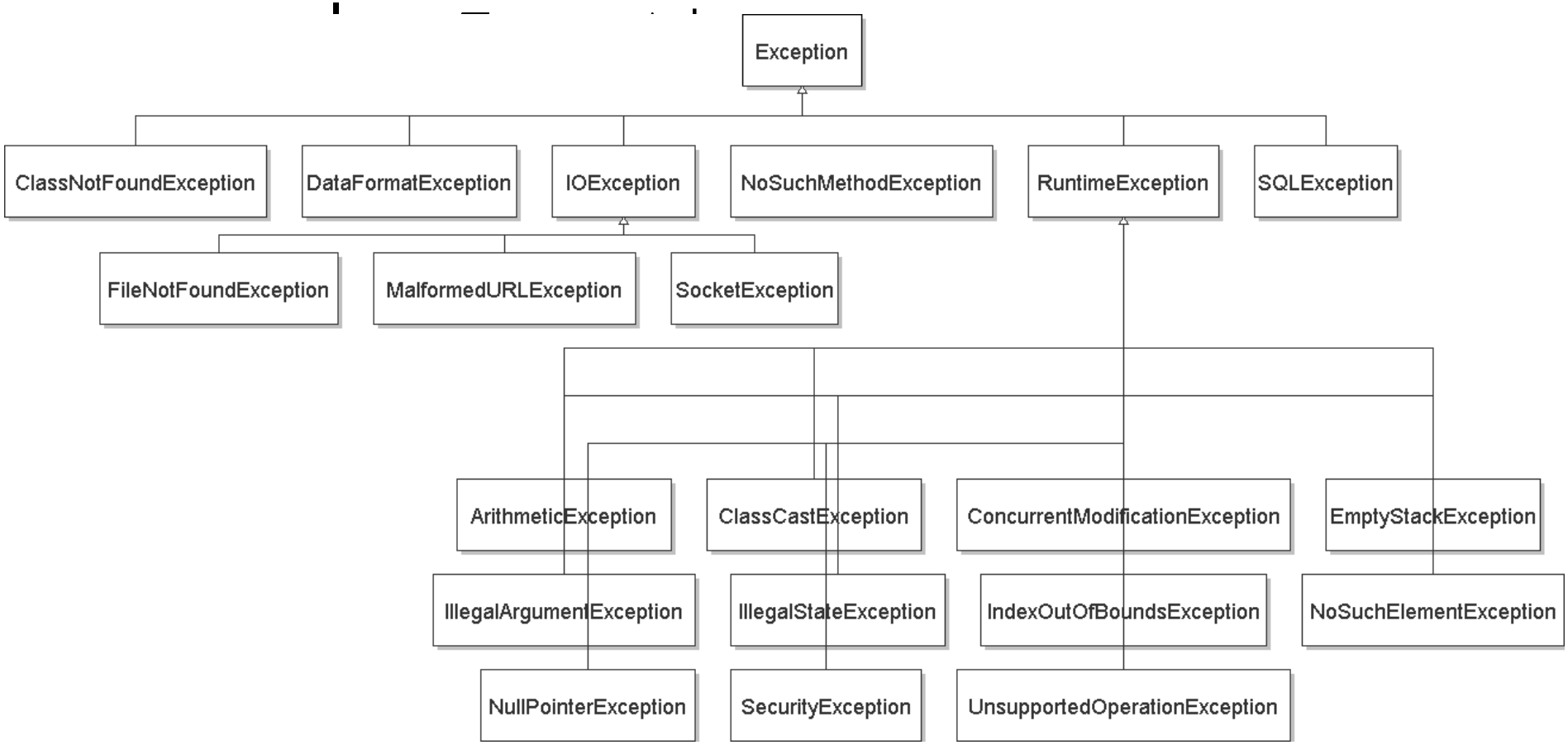
- All exception objects have these methods:

Method	Description
`public String getMessage()`	text describing the error
`public String toString()`	a stack trace of the line numbers where error occurred
`public InputStream openStream() throws IOException`	opens a stream for reading data from the document at this URL
`getCause`, `getStackTrace`, `printStackTrace`	other methods

- Some reasonable ways to handle an exception:
  - try again; re-prompt user; print a nice error message; quit the program; do nothing (!)

Exception inheritance

- All exceptions extend from a common



Inheritance and exceptions

- You can catch a general exception to handle any subclass:

```
try {
    Scanner input = new Scanner(new File("foo"));
    System.out.println(input.nextLine());
} catch (Exception e) {
    System.out.println("File was not found.");
}
```

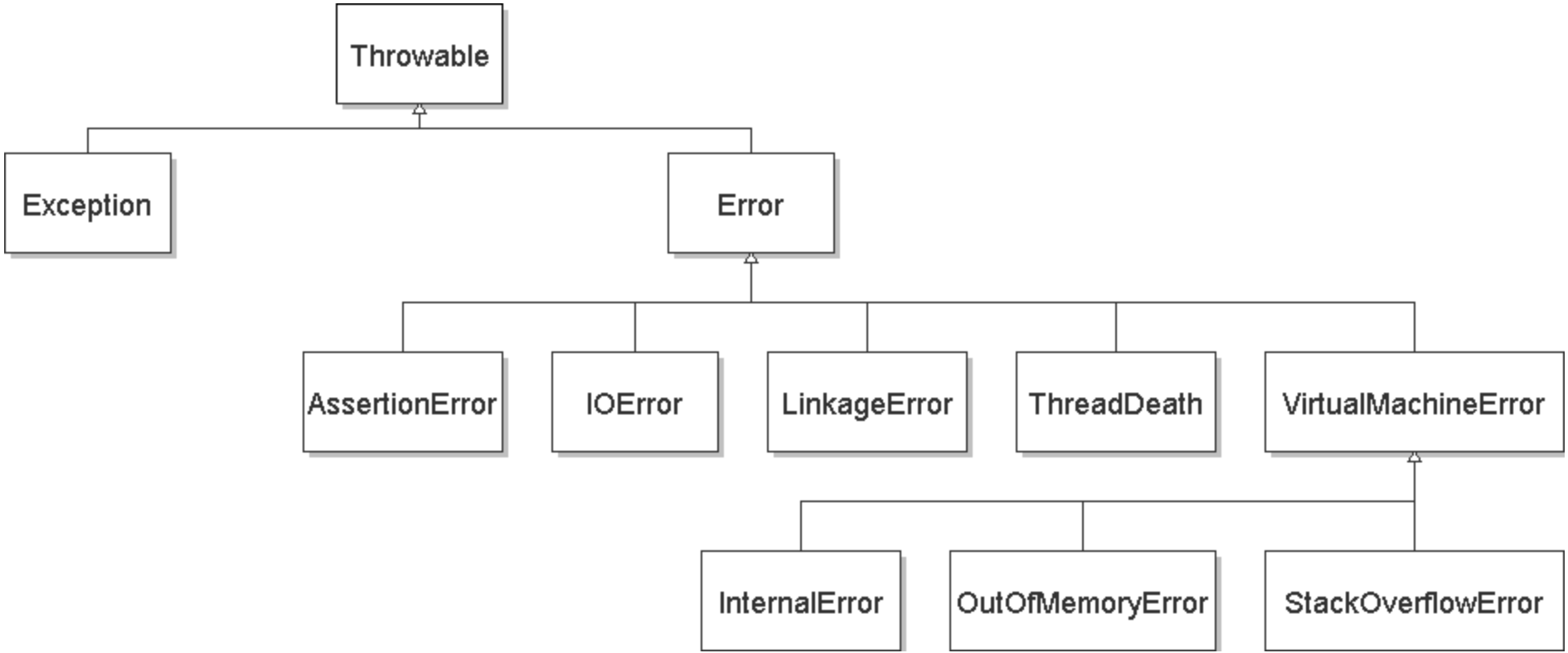
- Similarly, you can state that a method throws any exception:

```
public static void foo() throws Exception { ...
```

  - Are there any disadvantages of doing so?

Exceptions and errors

- There are also `Error`s, which represent serious Java problems.
  - `Error` and `Exception` have common superclass `Throwable`.
  - You can catch an `Error` (but you probably shouldn't)



Reading from the web

- class `java.net.URL` represents a web page's URL
- we can connect to a URL and read data from that web page

Method/Constructor	Description
`public URL(String address) throws MalformedURLException`	creates a URL object representing the given address
`public String getFile(), getHost(), getPath(), getProtocol()` `public int getPort()`	returns various parts of the URL as strings/integers
`public InputStream openStream() throws IOException`	opens a stream for reading data from the document at this URL

Exercise 1

- Write a class `Downloader` with the following behavior:
  - `public Downloader(String url)`
    - Initializes the downloader to examine the given URL.
  - `public void download(String targetFileName)`
    - Downloads the file from the URL to the given file name on disk.

- Write client program `DownloadMain` to use `Downloader`:

```
URL to download? foo bar
Bad URL! Try again: http://hanu.vn/
Target file name: hanu.html
```

Then you can use notepad to see the Contents of `hanu.html`

Exercise 1: solution

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

public class Downloader {
    private URL url;

    // Constructs downloader to read from the given URL.
    public Downloader(String urlString) throws MalformedURLException {
        url = new URL(urlString);
    }

    // Reads downloader's URL and writes contents to the given file.
    public void download(String targetFileName) throws IOException {
        InputStream in = url.openStream();
        FileOutputStream out = new FileOutputStream(targetFileName);
        while (true) {
            int n = in.read();
            if (n == -1) {      // -1 means end-of-file
                break;
            }
            out.write(n);
        }
        in.close();
        out.close();
    }
}
```

Exercise 1: solution (cont.)

```
import java.io.*;
import java.net.*;
import java.util.*;

public class DownloadMain {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        System.out.print("URL to download? ");
        String urlString = console.nextLine();

        Downloader down = null;    // create a downloader;
        while (down == null) {     // re-prompt the user if this fails
            try {
                down = new Downloader(urlString);
            } catch (MalformedURLException e) {
                System.out.print("Bad URL! Try again: ");
                urlString = console.nextLine();
            }
        }

        System.out.print("Target file name: ");
        String targetFileName = console.nextLine();

        try {    // download bytes to file (print error if it fails)
            down.download(targetFileName);
        } catch (IOException e) {
            System.out.println("I/O error: " + e.getMessage());
        }
    }
}
```

Exercise 2

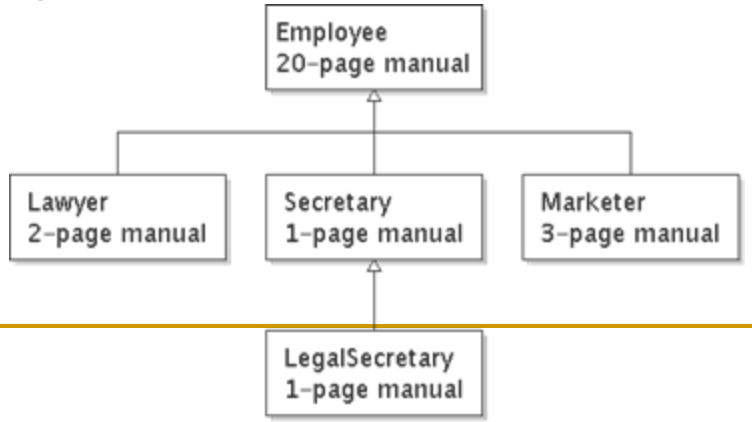
- Write class `TallyDownloader` to add behavior to `Downloader`:
  - `public TallyDownloader(String url)`
  - `public void download(String targetFileName)`
    - Downloads the file, and also prints the file to the console, and prints the number of occurrences of each kind of character in the file.

```
URL to download? http://fit.hanu.vn/
<!DOCTYPE html>
<html  dir="ltr" lang="vi" xml:lang="vi">
<head>
    <title>FIT HANU</title>
    <link rel="shortcut icon"
href="http://fit.hanu.vn/theme/image.php/clean/theme/1574932483/favicon" />
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<meta name="keywords" content="moodle, FIT HANU" />
```

- .....

Inheritance

- **inheritance**: Forming new classes based on existing ones.
  - a way to share/**reuse code** between two or more classes
  - **superclass**: Parent class being extended.
  - **subclass**: Child class that inherits behavior from superclass.
    - gets a copy of every field and method from superclass
  - **is-a relationship**: Each object of the subclass also "is a(n)" object of the superclass and can be treated as one.



Inheritance syntax

```
public class name extends superclass {

public class Lawyer extends Employee {
    ...
}
```

- **override**: To replace a superclass's method by writing a new version of that method in a subclass.

```
public class Lawyer extends Employee {
    // overrides getSalary method in Employee class;
    // give Lawyers a $5K raise
    public double getSalary() {
        return 55000.00;
    }
}
```

super keyword

- Subclasses can call inherited methods/constructors with `super`

```
super.method(parameters)
super(parameters);
```

```
public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years);  // calls Employee constructor
    }

    // give Lawyers a $5K raise
    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 5000.00;
    }
}
```

  - Lawyers now always make $5K more than Employees.

Exercise 2: solution

```
public class TallyDownloader extends Downloader {
    public TallyDownloader(String url) throws MalformedURLException {
        super(url);   // call Downloader constructor
    }

    // Reads from URL and prints file contents and tally of each char.
    public void download(String targetFileName) throws IOException {
        super.download(targetFileName);

        Map<Character, Integer> counts = new TreeMap<Character, Integer>();
        FileInputStream in = new FileInputStream(targetFileName);
        while (true) {
            int n = in.read();
            if (n == -1) {
                break;
            }
            char ch = (char) n;
            if (counts.containsKey(ch)) {
                counts.put(ch, counts.get(ch) + 1);
            } else {
                counts.put(ch, 1);
            }
            System.out.print(ch);
        }
        in.close();
        System.out.println(counts);   // print map of char -> int
    }
}
```

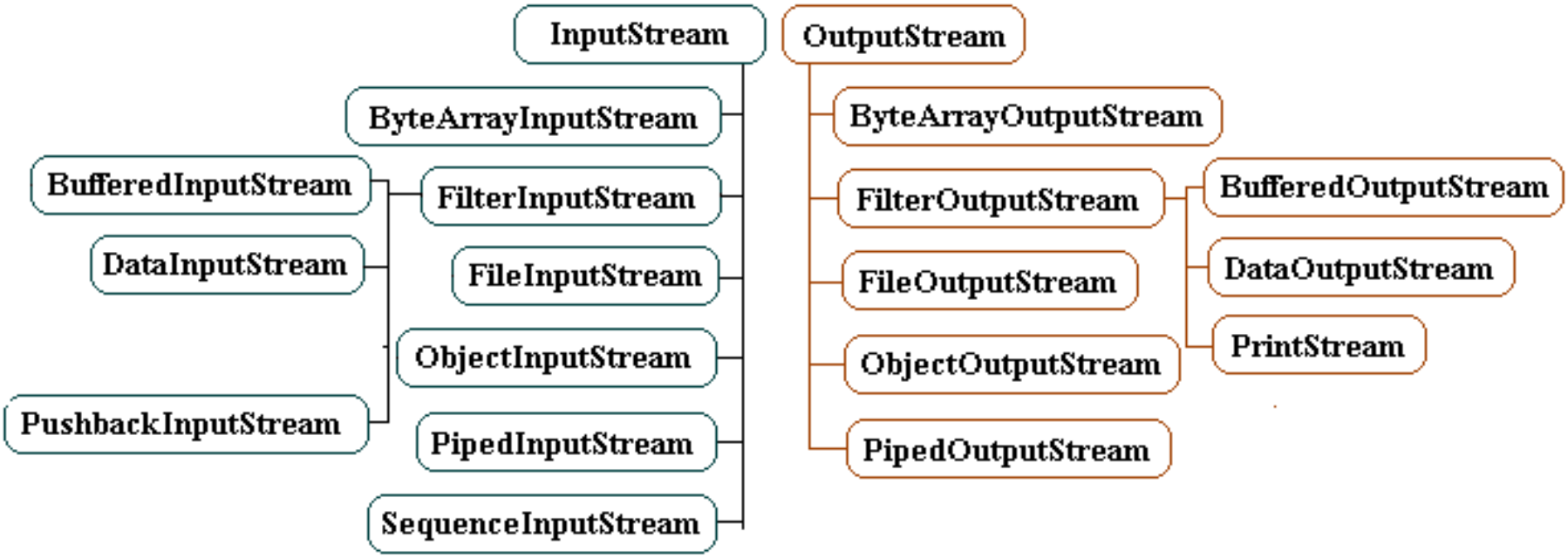
Exercise 2 : solution (cont.)

```
import java.io.*;
import java.net.*;
import java.util.*;

public class DownloadMain {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        System.out.print("URL to download? ");
        String urlString = console.nextLine();

        Downloader down = null;    // create a tallying downloader;
        while (down == null) {     // re-prompt the user if this fails
            try {
                down = new TallyDownloader(urlString);
            } catch (MalformedURLException e) {
                System.out.print("Bad URL! Try again: ");
                urlString = console.nextLine();
            }
        }

        ...
    }
}
```

Difference of `InputStream`, `InputStreamReader`, `BufferedReader`

- `InputStream.read`(): Reads the next byte of data from the input stream
- `InputStreamReader.read`(): Reads a single character from specified stream and remaining characters still remain in the stream.
- `BufferedReader` has methods to read a couple/string/line of characters from the specified stream and store it in a buffer. This makes input faster.

IO streams


InputStream
OutputStream
ByteArrayInputStream
ByteArrayOutputStream
BufferedInputStream
FilterInputStream
FilterOutputStream
BufferedOutputStream
DataInputStream
FileInputStream
FileOutputStream
DataOutputStream
ObjectInputStream
ObjectOutputStream
PrintStream
PushbackInputStream
PipedInputStream
PipedOutputStream
SequenceInputStream