Indian Institute of Technology Indore
भारतीय प्रौद्योगिकी संस्थान इंदौर

PDPM
Indian Institute of Information Technology,
Design and Manufacturing, Jabalpur

Object Oriented Programming with Java

Dr. Animesh Chaturvedi
Assistant Professor at IIIT Dharwad (Data Science and Artificial Intelligence)
Postdoc: King’s College London & The Alan Turing Institute
PhD: IIT Indore, MTech: IIITDM Jabalpur
Young Researcher: Heidelberg Laureate Forum and Pingala Interaction in Computing

KING’S College LONDON

The Alan Turing Institute

HEIDELBERG LAUREATE FORUM

IIIT Dharwad
ज्ञानेन विकासः
INDIAN INSTITUTE OF INFORMATION TECHNOLOGY

acm India Council
Pingala Interactions In Computing

Introduction

- Syntax of Java
- Java API
- How to build
  - stand-alone Java programs
- Example programs

Why Java?

- Highly popular and versatile programming language
- Fully Object-Oriented Programming (OOP) design
- Rich library of predefined classes and functions (Java Standard Library)
- Platform-independent: "Write Once, Run Anywhere" (WORA)
- Great for Web programming
- Strong security features
- Modern alternative to languages like C++

Java Program Types

- Applets (Deprecated)
  - Java programs designed to run in a web browser.
  - Applets operated in a restricted environment (sandbox).
- Servlets: A *servlet* is designed to be run by a web server
- Standalone Applications: Independent programs that run directly on the Java Virtual Machine (JVM).

Standalone JAVA Programs

- Write your Java code in file `foo.java` using an editor
- Compile the code with: `javac foo.java`
- This creates `foo.class`
- Run the compiled code using: `java foo`

Java Virtual Machine

- The .class files generated by the compiler are not executable binaries
  - so Java combines compilation and interpretation

- Contain "byte-codes" to be executed by the Java Virtual Machine

- Java source code is compiled into bytecode (.class files), which is interpreted and executed by the Java Virtual Machine.

- This provides platform independence and improved security.

HelloWorld (standalone)

- Note that String is built in
- println is a member function for the System.out class

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

Comments are almost like C++

- `/* This kind of comment can span multiple lines */`
- `// This kind is to the end of the line`

-
  ```
  /**
    * This kind of comment is a special
    * ‘javadoc’ style comment
    */
  ```

Primitive data types are like C

- Main data types are `int, double, boolean, char`
- Also have `byte, short, long, float`
- `boolean` has values `true` and `false`
- Declarations look like C, for example,
  - `double x, y;`
  - `int count = 0;`

Expressions are like C

- Assignment statements mostly look like those in C; you can use `=`, `+=`, `*=` etc.
- Arithmetic uses the familiar `+ - * / %`
- Java also has `++` and --
- Java has boolean operators `&& || !`
- Java has comparisons `< <= == != >= >`
- Java does *not* have pointers or pointer arithmetic

Control statements are like C

-
  ```
  if (x < y) smaller = x;
  ```

-
  ```
  if (x < y){ smaller=x;sum += x;}
  else { smaller = y; sum += y; }
  ```

-
  ```
  while (x < y) { y = y - x; }
  ```

-
  ```
  do { y = y - x; } while (x < y)
  ```

-
  ```
  for (int i = 0; i < max; i++)
     sum += i;
  ```
- BUT: conditions must be **boolean** !

Code snippets like C

```
int age = 25;
double salary = 55000.50;
boolean isEmployee = true;
char grade = 'A’;

if (age < 18) {
    System.out.println("Minor");
} else {
    System.out.println("Adult");
}

for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

Control statements II

- Java also introduces the `try` statement, about which more later

```
switch (n + 1) {
  case 0: m = n - 1; break;
  case 1: m = n + 1;
  case 3: m = m * n; break;
  default: m = -n; break;
}
```

Java isn't C!

- In C, almost everything is in functions
- In Java, almost everything is in classes
- There is often only one class per file
- There *must* be only one `public` class per file
- The file name *must* be the same as the name of that public class, but with a `.java` extension

Java program layout

- A typical Java file looks like:

```
import java.awt.*;
import java.util.*;

public class SomethingOrOther {
  // object definitions go here
  . . .
}
```

This must be in a file named SomethingOrOther.java !

What is a class?

- Early languages had only arrays
  - all elements had to be of the same type
- Then languages introduced structures (called **records**, or **structs**)
  - allowed different data types to be grouped
- Then Abstract Data Types (ADTs) became popular
  - grouped operations along with the data

So, what is a class?

- A class consists of
  - a collection of *fields*, or *variables*, very much like the named fields of a struct
  - all the operations (called *methods*) that can be performed on those fields
  - can be *instantiated*
- A class describes objects and operations defined on those objects

Name conventions

Conventions to follow:

- Java is case-sensitive; maxval, maxVal, and MaxVal are three different names
- Class names begin with a capital letter
- All other names begin with a lowercase letter
- Subsequent words are capitalized: theBigOne
- Underscores are not used in names
- These are *very strong* conventions!

Examples

- Class names start with an uppercase letter (Person, Animal).
- Variable and method names start with a lowercase letter (age, calculateTotal()).
- Subsequent words are capitalized (totalAmount, firstName).
- Avoid using underscores (_) in variable names.

The class hierarchy

- Classes are arranged in a hierarchy
- The root, or topmost, class is `Object`
- Every class but `Object` has at least one superclass
- A class may have subclasses
- Each class *inherits* all the fields and methods of its (possibly numerous) superclasses

An example of a class

```
class Person {
   String name;
   int age;

   void birthday ( ) {
      age++;
      System.out.println (name + ' is now ' + age);
   }
}
```

Another class Driver extending the class Person

```
class Driver extends Person {
   long driversLicenseNumber;
   Date expirationDate;
}
```

Creating and using an object

-
  ```
  Person john;
  john = new Person ( );
  john.name = "John Smith";
  john.age = 37;
  ```

-
  ```
  Person mary = new Person ( );
  mary.name = "Mary Brown";
  mary.age = 33;
  mary.birthday ( );
  ```

An array is an object

- `Person mary = new Person ( );`
- `int myArray[ ] = new int[5];`

  - or:

- `int myArray[ ] = {1, 4, 9, 16, 25};`
- `String languages [ ] = {"Prolog", "Java"};`

Classes and Objects in Java

- **Class Structure:** A class is a blueprint for creating objects.
- Class contains
  - fields (data) and
  - methods (operations).

```
public class Person {
    String name;
    int age;

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Method
    public void birthday() {
        this.age += 1;
    }
}

// Creating and using objects
Person john = new Person("John Smith", 30);
john.birthday();  // John's age is now 31
```

Example of Inheritance

```
class Animal {
    void eat() {
        System.out.println("This animal is eating.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("The dog is barking.");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Inherited from Animal
        dog.bark(); // Specific to Dog
    }
}
```

Best Practices for Java Programming

Key Guidelines:

- Follow naming conventions for readability.
- Ensure proper encapsulation by using access modifiers (private, public, protected).
- Handle exceptions using try-catch blocks.
- Close resources like files and streams after usage.

Streams and File I/O in Java

Goals and Objectives

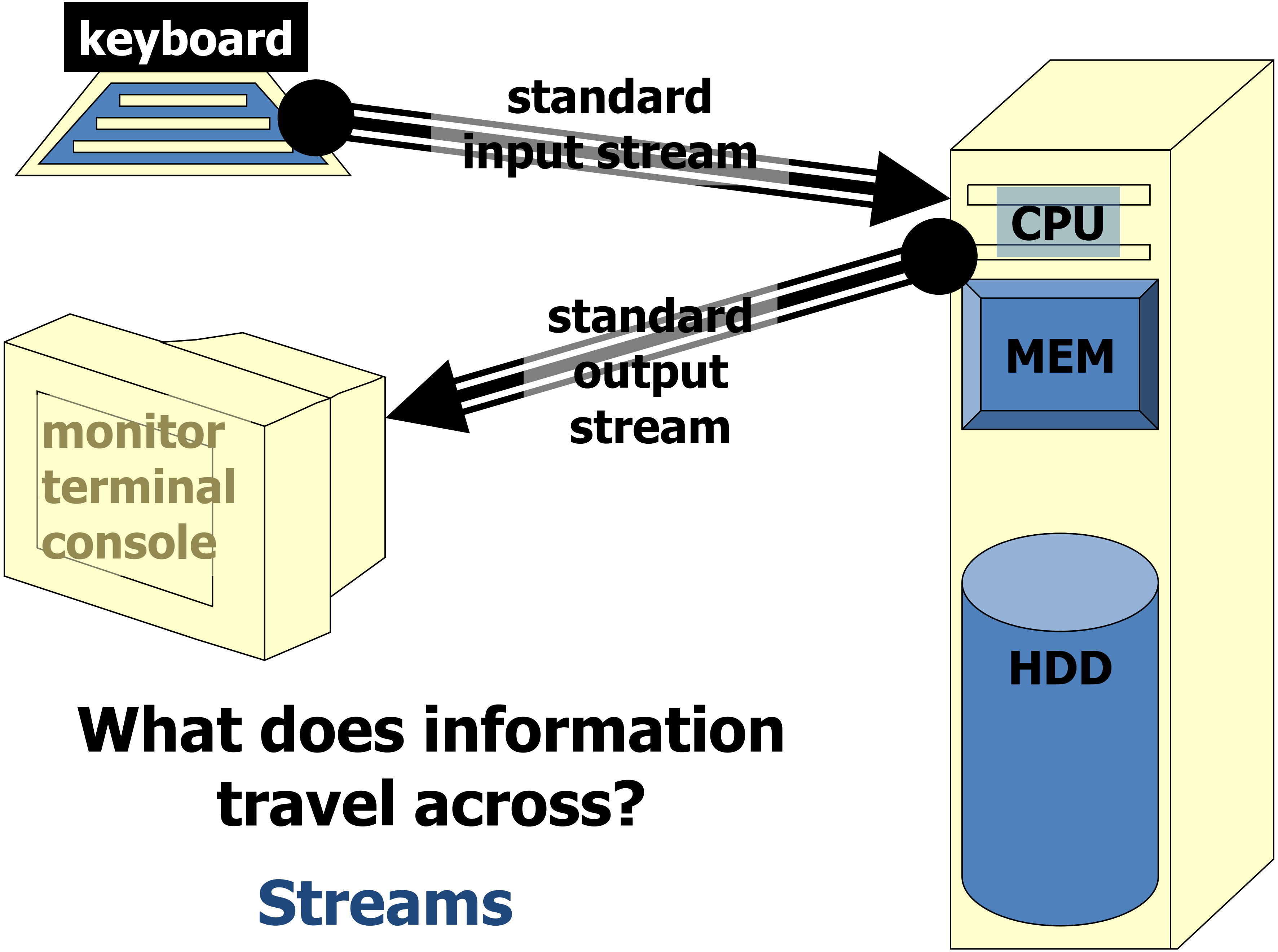
- Describe the concept of an I/O stream
  - The Concept of a Stream
  - Why Use Files for I/O?
  - Text Files and Binary Files
- Explain the difference between text and binary files
  - Save data, including objects, in a file
  - Read data, including objects, in a file
- Goals and Objectives
  - To be able to read and write text files
  - To become familiar with the concepts of text and binary formats
  - To understand when to use sequential and random file access
  - To be able to read and write objects using serialization
  - To learn about encryption

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)

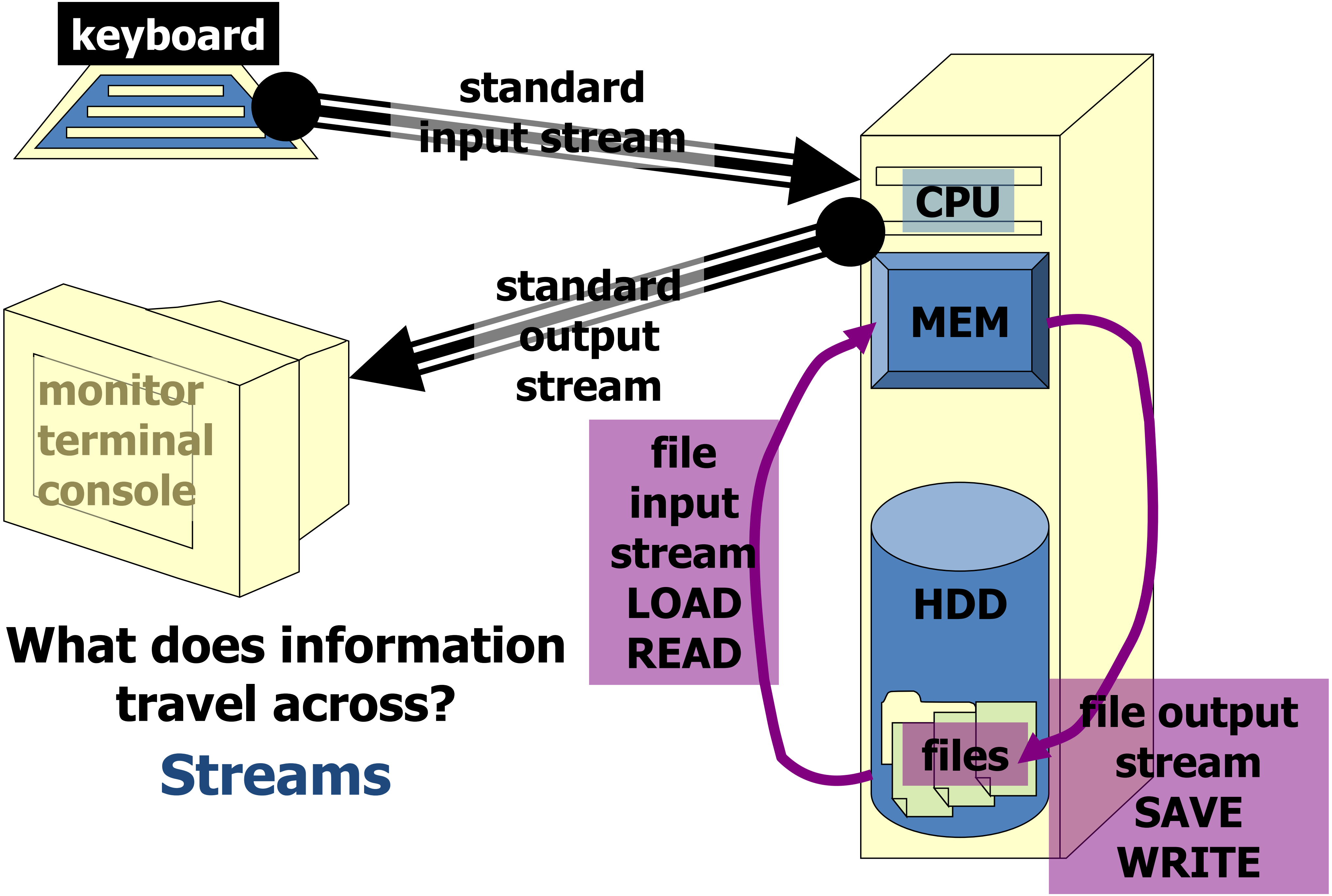
Why Use Files for I/O

- Keyboard input, screen output deal with temporary data
  - When program ends, data is gone
- Data in a file remains after program ends
  - Can be used next time program runs
  - Can be used by another program



What does information travel across?

Streams



What does information travel across?

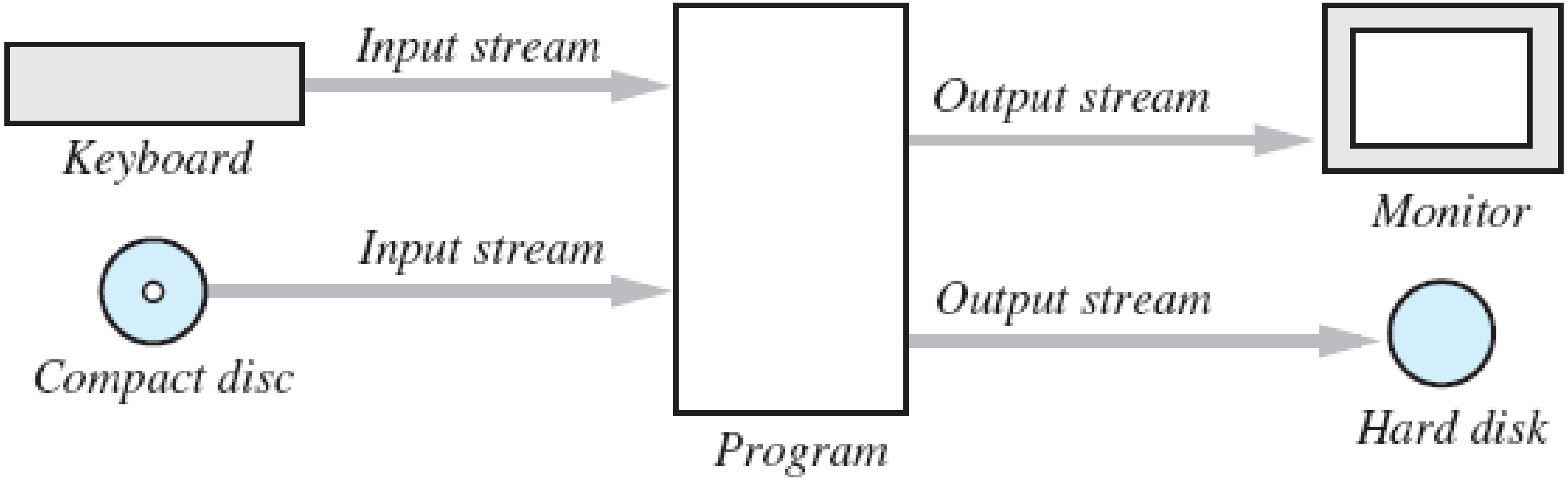
Streams

Stream

- A stream is a flow of data (either input or output) between a program and an external data source (file, keyboard, etc.).
- Input Stream: Reads data into the program (e.g., System.in, FileReader).
- Output Stream: Writes data from the program (e.g., System.out, PrintWriter).
- Use of files
  - Store Java classes, programs
  - Store pictures, music, videos
  - Can also use files to store program I/O
- A *stream* is a flow of input or output data: Characters, Numbers, Bytes

The Concept of a Stream

- Streams are implemented as objects of special stream classes
  - Class **`Scanner`**
  - Object **`System.out`**
- I/O Streams



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

Streams

- All modern I/O is stream-based
- A stream is a connection to a source of data or to a destination for data
- An input stream may be associated with the keyboard
- An input stream or an output stream may be associated with a file
- Different streams have different characteristics:
  - A file has a definite length, and therefore an end
  - Keyboard input has no specific end

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

- *Open* the stream
- *Use* the stream (read, write, or both)
- *Close* the stream

open
use
close

Opening a stream

- There is data external to your program that you want to get, or you want to put data somewhere outside your program
- When you open a stream, you are making a connection to that external place
- Once the connection is made, you forget about the external place and just use the stream

- A FileReader is a used to connect to a file that will be used for input:

```
FileReader fileReader =
    new FileReader(fileName);
```

- The fileName specifies where the (external) file is to be found
- You never use fileName again; instead, you use fileReader

open
use
close

Using a stream

- Some streams can be used only for input, others only for output, still others for both
- *Using* a stream means doing input from it or output to it
- But it's not usually that simple--you need to manipulate the data in some way as it comes in or goes out

```
int charAsInt;
charAsInt = fileReader.read( );
```

- The `fileReader.read()` method reads one character and returns it as an integer, or `-1` if there are no more characters to read
- The meaning of the integer depends on the file encoding (ASCII, Unicode, other)
- You can *cast* from `int` to `char`:

  ```
  char ch = (char)fileReader.read( );
  ```
- FileReaderExample1.java

open
use
close

Manipulating the input data

- Reading characters as integers isn't usually what you want to do
- A BufferedReader will convert integers to characters; it can also read whole lines
- The constructor for BufferedReader takes a FileReader parameter:

```
BufferedReader bufferedReader =
    new BufferedReader(fileReader);
```

Reading lines

```
String s;
s = bufferedReader.readLine( );
```

- A BufferedReader will return null if there is nothing more to read
- FileReaderExample2.java

open
use
close

Closing

- A stream is an expensive resource
- There is a limit on the number of streams that you can have open at one time
- You should not have more than one stream open on the same file
- You must close a stream before you can open it again
- *Always close your streams!*

- Java will normally close your streams for you when your program ends, but it isn't good style to depend on this

Text Files and Binary Files

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"

Binary Versus Text 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

<u>Java Text Files</u>

- ➢ Source files
- ➢ Occasionally input files
- ➢ Occasionally output files

<u>Java Binary Files</u>

- ➢ 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

```
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]

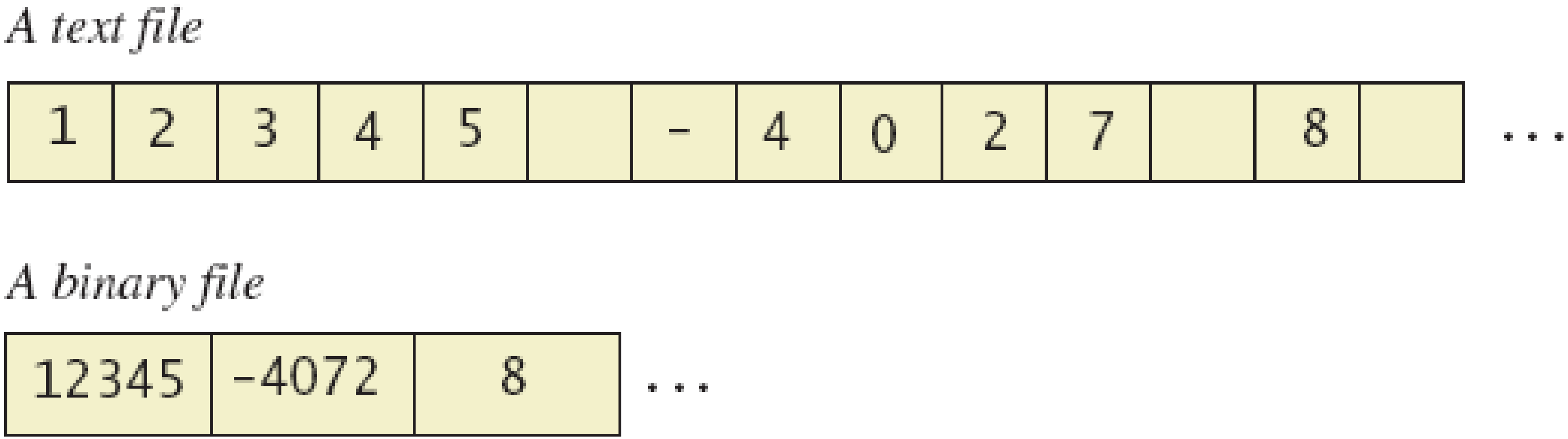
```
0000000 312 376 272 276 000 000 000 061
        312 376 272 276  \0  \0  \0   1
0000010 000 164 012 000 051 000 062 007
         \0   t  \n  \0   )  \0   2  \a
0000020 000 063 007 000 064 010 000 065
         \0   3  \a  \0   4  \b  \0   5
0000030 012 000 003 000 066 012 000 002
         \n  \0 003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

- All data in files stored as binary digits
  - Long series of zeros and ones
- Files treated as sequence of characters called *text files*
  - Java program source code
  - Can be viewed, edited with text editor
- All other files are called *binary files*
  - Movie, music files
  - Access requires specialized program

Text Files and Binary Files

- A text file and a binary file containing the same values



A text file

1	2	3	4	5		-	4	0	2	7		8	

. . .

A binary file

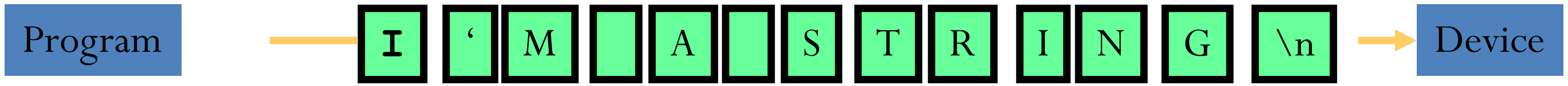
12345	-4072	8

. . .

Streams

- JAVA distinguishes between 2 types of streams:

- Text – streams, containing `characters`

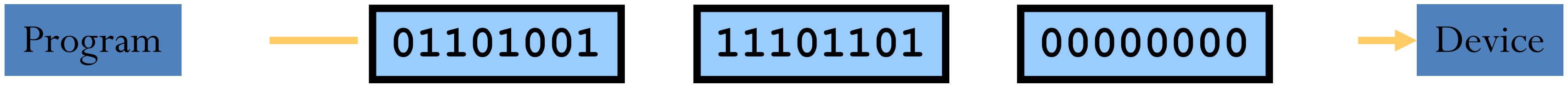


Program

I ‘ M   A   S T R I N G \n

Device

- Binary Streams, containing 8 – bit information



Program

01101001 11101101 00000000

Device

Binary vs. TextFiles

- When use Text / BinaryFiles ?
- Use TextFiles for efficient human text readability.
- Binary Files is used for non-final interchange between programs
- Binary Files are used for large amount of data (images, videos etc.),
  - with an exact definition of the meaning of the bytestream
  - Example: JPG, MP3, BMP

	pro	con
Binary	Efficient in terms of time and space	Preinformation about data needed to understand content
Text	Human readable, contains redundant information	Not efficient

Text Files I/O and Streams

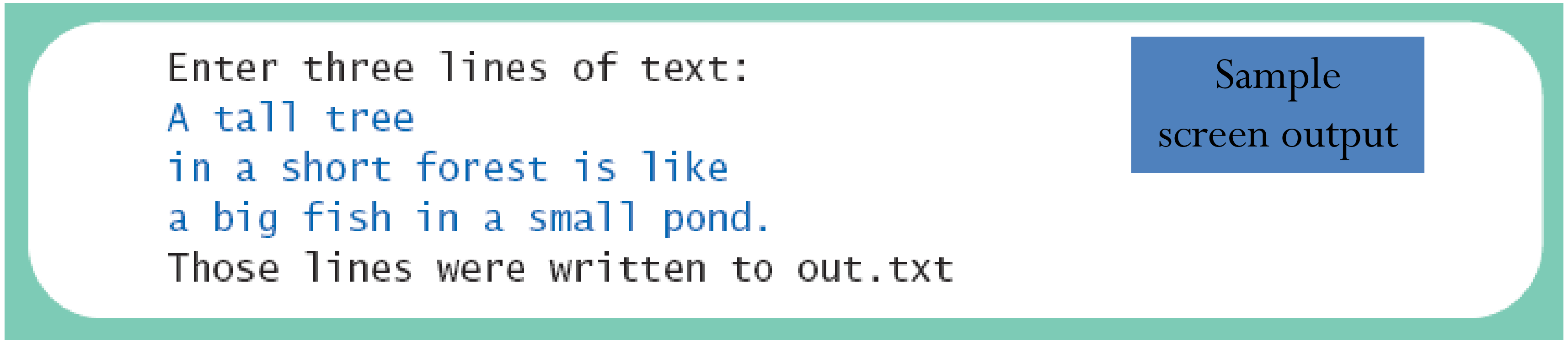
Creating a Text File

- File is empty initially
  - May now be written to with method `println`
- Data goes initially to memory buffer
  - When buffer full, goes to file
- Closing file empties buffer, disconnects from stream

Creating a Text File

- View sample program, “read three lines of a text file”

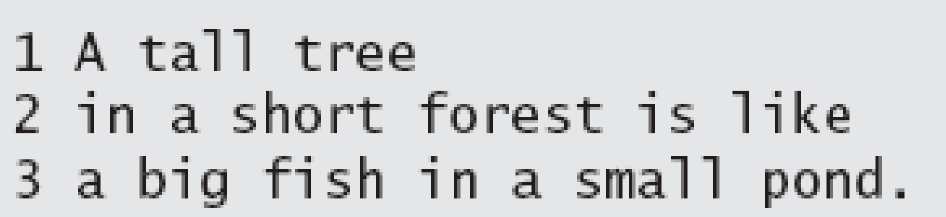
  class TextFileOutput



```
Enter three lines of text:
A tall tree
in a short forest is like
a big fish in a small pond.
Those lines were written to out.txt
```

Sample screen output

Resulting File



```
A tall tree
in a short forest is like
a big fish in a small pond.
```

You can use a text editor to read this file.

Creating a Text File

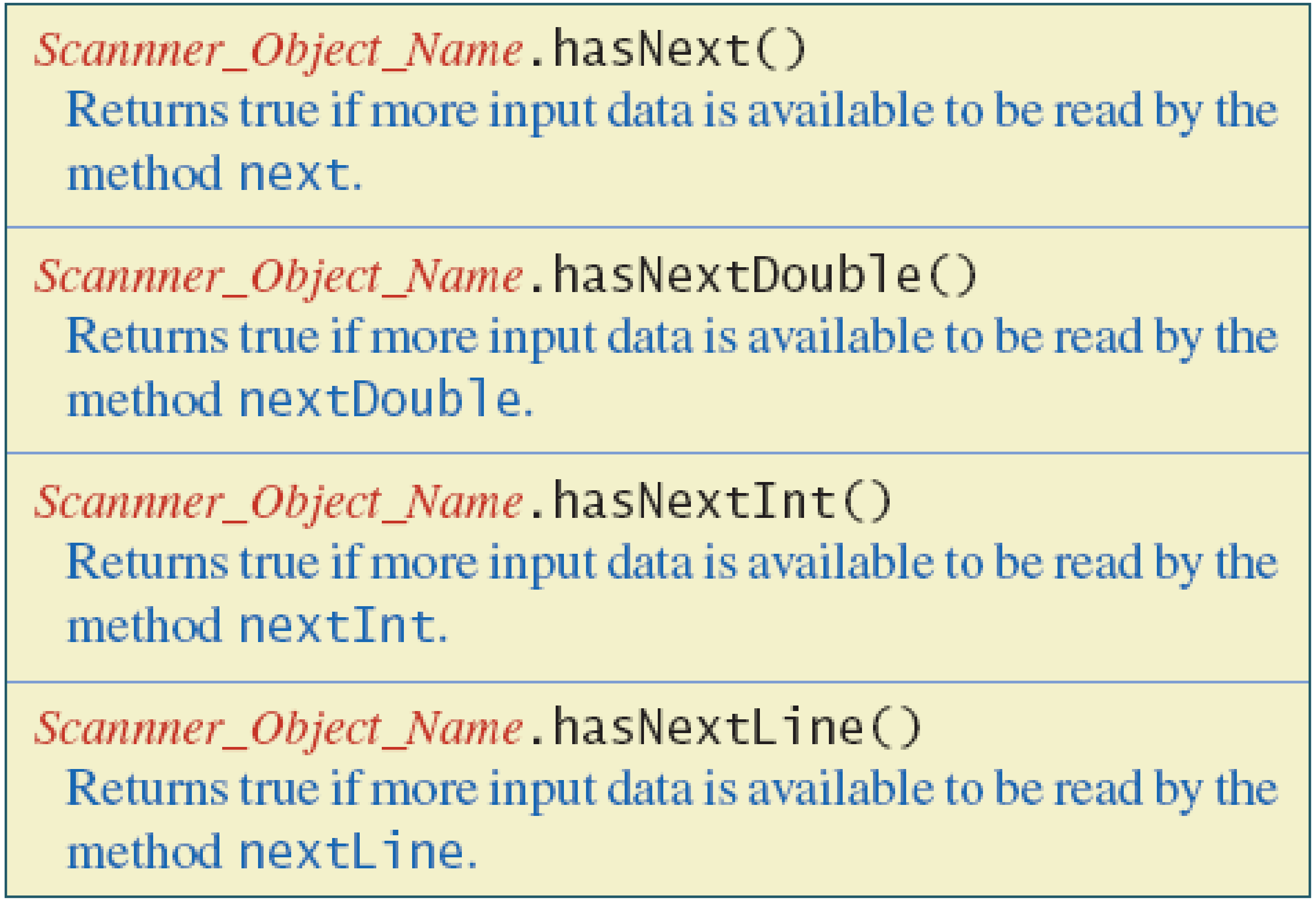
- When creating a file
  - Inform the user of ongoing I/O events, program should not be "silent"
- A file has two names in the program
  - File name used by the operating system
  - The stream name variable
- Opening, writing to file overwrites pre-existing file in directory

Reading from a Text File

- Reads text from file, displays on screen
- Note
  - Statement which opens the file
  - Use of `Scanner` object
  - Boolean statement which reads the file and terminates reading loop

Reading from a Text File

- Additional methods in class **Scanner**



Scannner_Object_Name.hasNext()

Returns true if more input data is available to be read by the method next.

Scannner_Object_Name.hasNextDouble()

Returns true if more input data is available to be read by the method nextDouble.

Scannner_Object_Name.hasNextInt()

Returns true if more input data is available to be read by the method nextInt.

Scannner_Object_Name.hasNextLine()

Returns true if more input data is available to be read by the method nextLine.

Techniques for Any File

- The Class **`File`**
- Programming Example: Reading a File Name from the Keyboard
- Using Path Names
- Methods of the Class **`File`**
- Defining a Method to Open a Stream

The Class **`File`**

- Class provides a way to represent file names in a general way
  - A **`File`** object represents the name of a file
- The object

  ```
  new File ("treasure.txt")
  ```

  is not simply a string
  - It is an object that *knows* it is supposed to name a file

Using Path Names

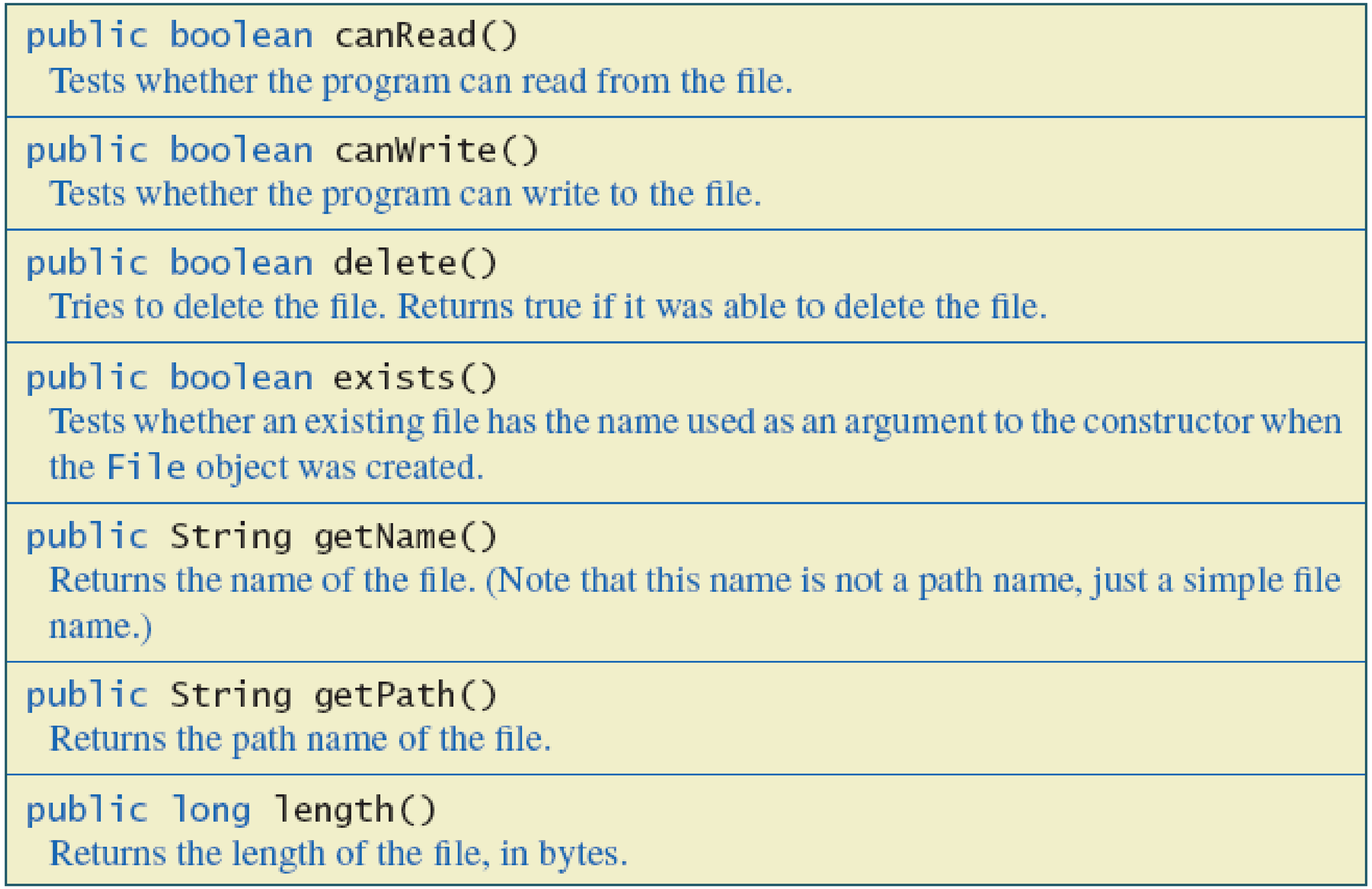
- Files opened assumes to be in same folder as where program run
- Possible to specify path names
  - Full path name
  - Relative path name
- Be aware of differences of pathname styles in different operating systems

Methods of the Class File

- Recall that a **`File`** object is a system-independent abstraction of file's path name
- Class **`File`** has methods to access information about a path and the files in it
  - Whether the file exists
  - Whether it is specified as readable or not
  - Etc.

Methods of the Class File

- Some methods in class **`File`**



`public boolean canRead()`

Tests whether the program can read from the file.

`public boolean canWrite()`

Tests whether the program can write to the file.

`public boolean delete()`

Tries to delete the file. Returns true if it was able to delete the file.

`public boolean exists()`

Tests whether an existing file has the name used as an argument to the constructor when the `File` object was created.

`public String getName()`

Returns the name of the file. (Note that this name is not a path name, just a simple file name.)

`public String getPath()`

Returns the path name of the file.

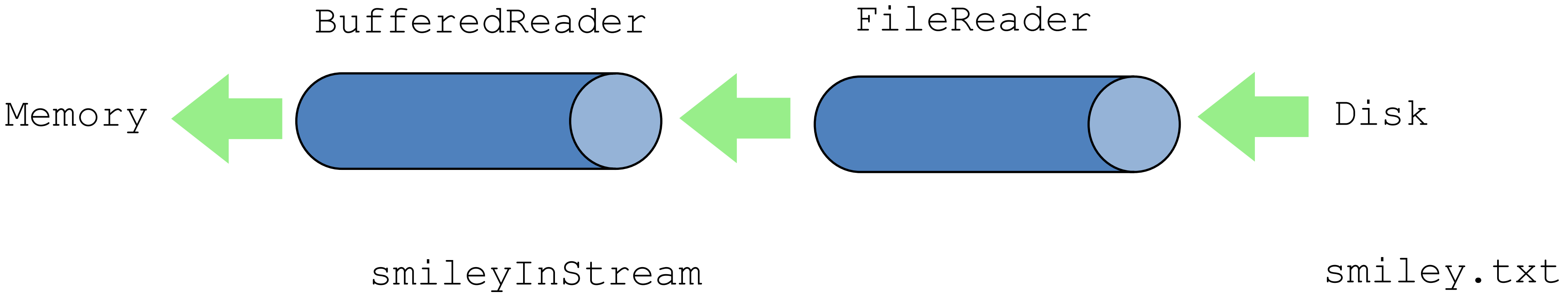
`public long length()`

Returns the length of the file, in bytes.

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

Input File Streams



BufferedReader

FileReader

Memory

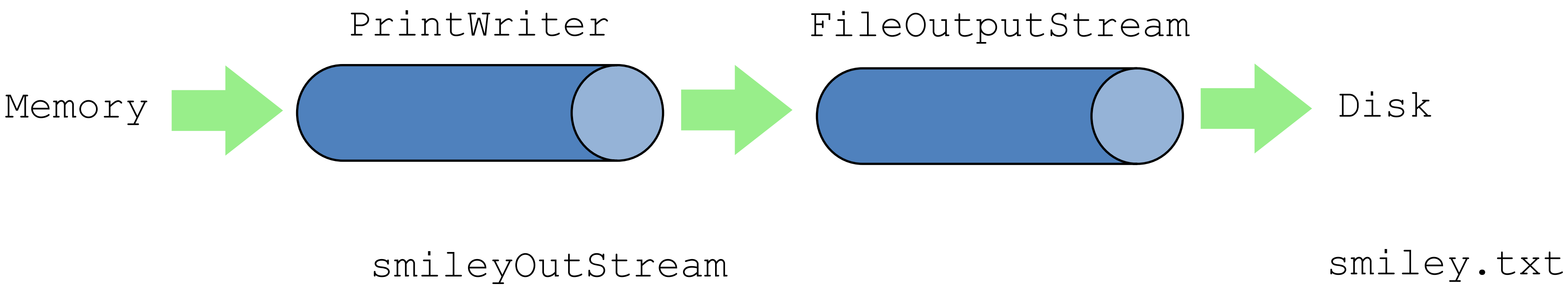
Disk

smileyInStream

smiley.txt

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

Output File Streams



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

Text File I/O

- Important classes for text file **input** (from the file):
  - **`BufferedReader`**
  - **`FileReader`**
- Important classes for text file **output** (to the file)
  - **`PrintWriter`**
  - **`FileOutputStream`**   **`[`**or **`FileWriter]`**
- **`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.*;
```

Reading and Writing Text Files

Reading and Writing Text Files

- Text files – files containing simple text
  - Created with editors such as notepad, html, etc.

- Simplest way to learn it so extend our use of **`Scanner`**
  - Associate with files instead of **`System.in`**

- All input classes, except Scanner, are in java.io
  - **`import java.io.*;`**

Scanner

- The constructor takes an object of type **`java.io.InputStream`** – stores information about the connection between an input device and the computer or program
  - Example: **`System.in`**
- Recall – only associate ***one*** instance of **`Scanner`** with **`System.in`** in your program
  - Otherwise, get bugs

Numerical Input

- 2 ways (we've learned one, seen the other)
  - Use **`int`** as example, similar for **`double`**

- First way:
  - Use **`nextInt()`**

```
int number = scanner.nextInt();
```

- Second way:
  - Use **`nextLine()`**, **`Integer.parseInt()`**

```
String input = scanner.nextLine();
int number = Integer.parseInt(input);
```

Numerical Input

- Exceptions
  - **`nextInt()`** throws **`InputMismatchException`**
  - **`parseInt()`** throws **`NumberFormatException`**

- Optimal use
  - **`nextInt()`** when there is multiple information on one line
  - **`nextLine() + parseInt()`** when one number per line

Reading Files

- The same applies for both console input and file input

- We can use a different version of a Scanner that takes a ***File*** instead of **`System.in`**

- Everything works the same!

Reading Files

- To read from a disk file, construct a **`FileReader`**

- Then, use the **`FileReader`** to construct a **`Scanner`** object

```
FileReader rdr = new FileReader("input.txt");
Scanner fin = new Scanner(rdr);
```

Reading Files

- You can use **`File`** instead of **`FileReader`**
  - Has an **`exists()`** method we can call to avoid **`FileNotFoundException`**

```
File file = new File ("input.txt");
Scanner fin;
if(file.exists()){
  fin = new Scanner(file);
} else {
  //ask for another file
}
```

Reading Files

- Once we have a Scanner, we can use methods we already know:
  - **`next, nextLine, nextInt`**, etc.

- Reads the information from the file instead of console

File Class

- **`java.io.File`**
  - associated with an actual file on hard drive
  - used to check file's status
- Constructors
  - **`File(<full path>)`**
  - **`File(<path>, <filename>)`**
- Methods
  - **`exists()`**
  - **`canRead()`**, **`canWrite()`**
  - **`isFile()`**, **`isDirectory()`**

File Class

- **`java.io.FileReader`**
  - Associated with `File` object
  - Translates data bytes from File object into a stream of characters (much like InputStream vs. InputStreamReader)

- Constructors
  - `FileReader( <File object> );`

- Methods
  - `read(),readLine()`
  - `close()`

Writing to a File

- We will use a **`PrintWriter`** object to write to a file
  - What if file already exists? → Empty file
  - Doesn’t exist? → Create empty file with that name

- How do we use a **`PrintWriter`** object?
  - Have we already seen one?

Writing to a File

- The out field of the System class is a **`PrintWriter`** object associated with the console
  - We will associate our **`PrintWriter`** with a file now

```
PrintWriter fout = new PrintWriter("output.txt");
fout.println(29.95);
fout.println(new Rectangle(5, 10, 15, 25));
fout.println("Hello, World!");
```

- This will print the exact same information as with **`System.out`** (except to a file "output.txt")!

Closing a File

- Only main difference is that we have to close the file stream when we are done writing

- If we do not, not all output will written

- At the end of output, call **`close()`**

```
fout.close();
```

Closing a File

- Why?
  - When you call **`print()`** and/or **`println()`**, the output is actually written to a buffer. When you close or flush the output, the buffer is written to the file
  - The slowest part of the computer is hard drive operations – much more efficient to write once instead of writing repeated times

File Locations

- When determining a file name, the default is to place in the same directory as your .class files

- If we want to define other place, use an absolute path (e.g. c:\My Documents)

```
in  = new FileReader(“c:\\homework\\input.dat”);
```

- Why `\\` ?

Java Input Review

CONSOLE:

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

FILE:

```
Scanner inFile = new Scanner( new FileReader(srcFileName ));
```

Java Output Review

- CONSOLE:

```
System.out.print("To the screen");
```

- FILE:

```
PrintWriter fout = new PrintWriter(new File("output.txt");
fout.print("To a file");
```

Defining a Method to Open a Stream

- Method will have a **`String`** parameter
  - The file name
- Method will return the stream object
- Will throw exceptions
  - If file not found
  - If some other I/O problem arises
- Should be invoked inside a **`try`** block and have appropriate **`catch`** block

Creating a Text File

- Class **`PrintWriter`** defines methods needed to create and write to a text file
  - Must import package **`java.io`**
- To open the file
  - Declare *stream variable* for referencing the stream
  - Invoke **`PrintWriter`** constructor, pass file name as argument
  - Requires **`try`** and **`catch`** blocks

Appending to a Text File

- Opening a file new begins with an empty file
  - If already exists, will be overwritten
- Some situations require appending data to existing file
- Command could be

```
outputStream =
    new PrintWriter(
    new FileOutputstream(fileName, true));
```

- Method `println` would append data at end

Sample Program

- Two things to notice:
  - Have to import from java.io
  - I/O requires us to catch checked exceptions
    - **`java.io.IOException`**

Reading from a File

```
import java.io.*;

public class FileReaderExample {
    public static void main(String[] args) {
        try {
            BufferedReader reader = new BufferedReader(new FileReader("example.txt"));
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

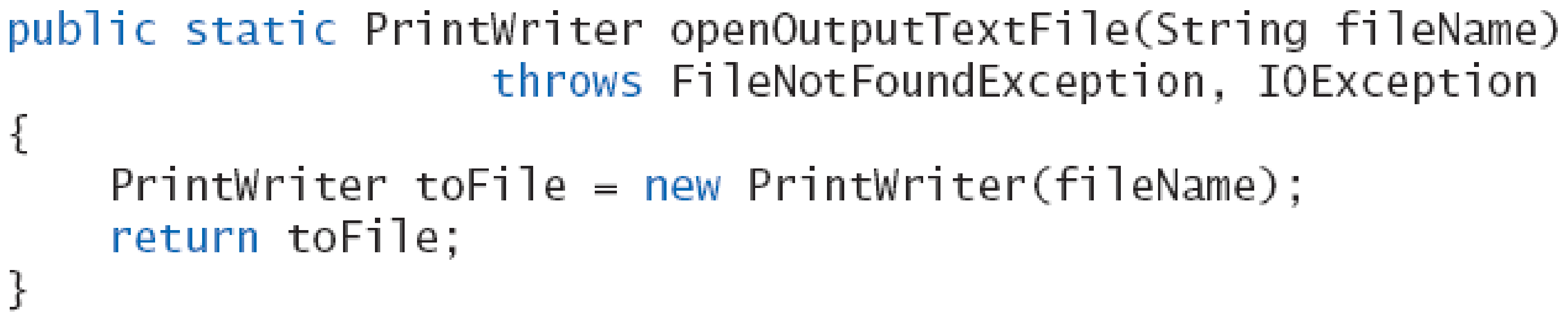
Writing to a File

```
import java.io.*;

public class FileWriterExample {
    public static void main(String[] args) {
        try {
            PrintWriter writer = new PrintWriter(new FileWriter("output.txt"));
            writer.println("This is a sample text.");
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```java
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;

public class LineNumberer{
  public static void main(String[] args){
      Scanner console = new Scanner(System.in);
      System.out.print("Input file: ");
      String inFile = console.next();

      System.out.print("Output file: ");
      String outFile = console.next();

      try{
            FileReader reader = new FileReader(inFile);
            Scanner in = new Scanner(reader);
      PrintWriter out = new PrintWriter(outputFileName);
      int lineNumber = 1;

      while (in.hasNextLine()){
        String line = in.nextLine();
        out.println("/* " + lineNumber + " */ " + line);
        lineNumber++;
      }

      out.close();
    } catch (IOException exception){
        System.out.println("Error processing file: " + exception);
    }
  }
}
```

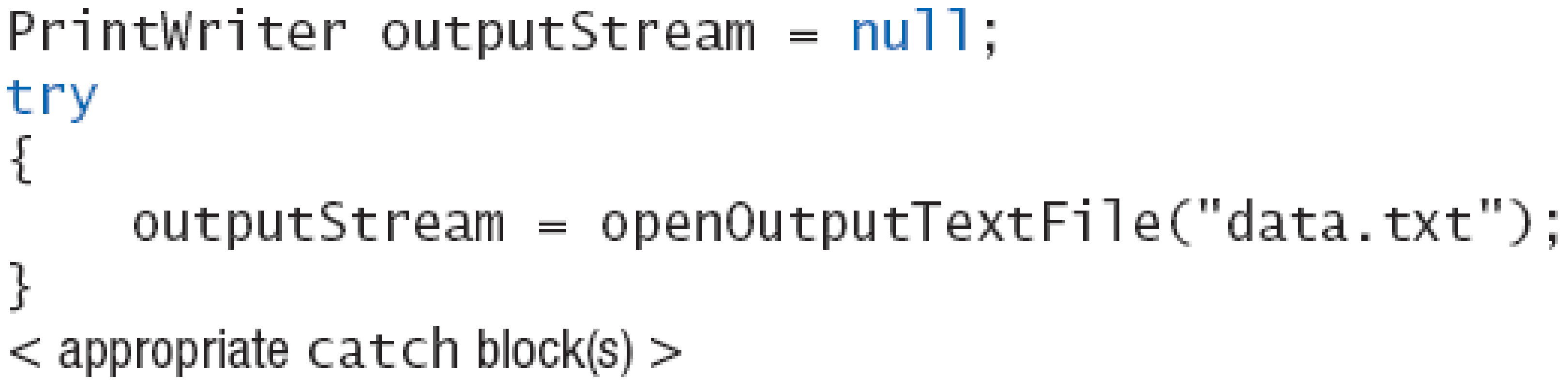
Defining a Method to Open a Stream

- Example code



```
public static PrintWriter openOutputTextFile(String fileName)
                                 throws FileNotFoundException, IOException
{
    PrintWriter toFile = new PrintWriter(fileName);
    return toFile;
}
```

- Example call



```
PrintWriter outputStream = null;
try
{
    outputStream = openOutputTextFile("data.txt");
}
< appropriate catch block(s) >
```

LineReader and LineWriter

Text files

- Text (`.txt`) files are the simplest kind of files
  - Text files can be used by many different programs
- Formatted text files (such as `.doc` files) also contain binary formatting information
- Only programs that "know the secret code" can make sense of formatted text files
- Compilers, in general, work only with text

My LineReader class

```
class LineReader {
  BufferedReader bufferedReader;

  LineReader(String fileName) {…}

  String readLine( ) {…}

  void close( ) {…}
}
```

Basics of the LineReader constructor

- Create a FileReader for the named file:

```
FileReader fileReader = new FileReader(fileName);
```

- Use it as input to a BufferedReader:

```
BufferedReader bufferedReader = new BufferedReader(fileReader);
```

- Use the BufferedReader; but first, we need to catch possible Exceptions

The full LineReader constructor

```
LineReader(String fileName) {
  FileReader fileReader = null;
  try { fileReader = new FileReader(fileName); }
  catch (FileNotFoundException e) {
    System.err.println
       ("LineReader can’t find input file: " + fileName);
    e.printStackTrace( );
  }
  bufferedReader = new BufferedReader(fileReader);
}
```

readLine

```
String readLine( ) {
    try {
        return bufferedReader.readLine( );
    }
    catch(IOException e) {
         e.printStackTrace( );
    }
    return null;
}
```

close

```
void close() {
    try {
        bufferedReader.close( );
    }
    catch(IOException e) { }
}
```

How did I figure that out?

- To read lines from a file
- There might be a suitable read*Something* method in the API
- A readLine method in several classes e.g. the BufferedReader class
- The constructor for BufferedReader takes a Reader as an argument
- Reader is an abstract class, but it has several implementations, including InputStreamReader
- FileReader is a subclass of InputStreamReader
- There is a constructor for FileReader that takes as its argument a (String) file name

The LineWriter class

```
class LineWriter {
    PrintWriter printWriter;

    LineWriter(String fileName) {...}

   void writeLine(String line) {...}

   void close( ) {...}
}
```

The constructor for LineWriter

```
LineWriter(String fileName) {
    try {
       printWriter =
          new PrintWriter(
              new FileOutputStream(fileName), true);
    }
    catch(Exception e) {
       System.err.println("LineWriter can’t " +
           "use output file: " + fileName);
    }
  }
```

Flushing the buffer

- Put information into a buffered output stream, it goes into a buffer

- The buffer may *or may not* be written out right away

- Program crashes then may not known how far it got before it crashed

- Flushing the buffer forces the information to be written out

The PrintWriter class

- Buffers are automatically flushed when the program ends normally
- Usually it is your responsibility to flush buffers if the program does not end normally
- PrintWriter can do the flushing for you

```
public PrintWriter(OutputStream out, boolean autoFlush)
```

writeLine

```
 void writeLine(String line) {
     printWriter.println(line);
  }
```

close

```
void close( ) {
    printWriter.flush( );
    try {
        printWriter.close( );
    }
    catch(Exception e) { }
}
```

Tokens

Tokenizing

- Often several text values are in a single line in a file to be compact

  `“25 38 36 34 29 60 59”`

- The line must be broken into parts (i.e. ***tokens***)

  `“25”`

  `“38”`

  `“36”`

- tokens then can be parsed as needed

  `“25”` can be turned into the integer `25`

Tokenizing

- Inputting each value on a new line makes the file very long

- May want a file of customer info – name, age, phone number all on one line

- File usually separate each piece of info with a **delimiter** – any special character designating a new piece of data (space in previous example)

Tokenizing in Java

- use a `StringTokenizer` object
  - default delimiters are: space, tab, newline, return
  - requires: `import java.util.*`

- Constructors
  - `StringTokenizer(String line)//default dlms`
  - `StringTokenizer(String ln, String dlms)`

- Methods
  - `hasMoreTokens()`
  - `nextToken()`
  - `countTokens()`

StringTokenizing in Java

```
Scanner stdin = new…
System.out.print( "Enter a line with comma seperated
  integers(no space): " );
String input = stdin.nextLine();

StringTokenizer st;
String delims = ",";
st = new StringTokenizer( input, delims );

while ( st.hasMoreTokens() )
{
  int n = Integer.parseInt(st.nextToken());
  System.out.println(n);
}
```

```
File gradeFile = new File(“scores.txt”);
if(gradeFile.exists()){
  Scanner inFile = new Scanner(gradeFile);

  String line = inFile.nextLine();

  while(line != null){
      StringTokenizer st = new
            StringTokenizer(line, ":");
      System.out.print(" Name: " + st.nextToken());

      int num = 0;
      double sum = 0;

      while ( st.hasMoreTokens() )
      {
            num++;
            sum += Integer.parseInt(st.nextToken());
      }
      System.our.println(" average = "+ sum/num);
      line = inFile.nextLine();
```

```
    }

    inFile.close();
}
```

If you call `nextToken()` and there are no more tokens, `NoSuchElementException` is thrown

Tokenizing

- Scanner tokenizes already…

```
Scanner in = new Scanner(…);
while(in.hasNext()) {
  String str = in.next();
  …
}
```

Examples

```java
// Java program for simple calculator
import java.io.*;
import java.lang.*;
import java.lang.Math;
import java.util.Scanner;

// Driver class
public class BasicCalculator {
    // main function
    public static void main(String[] args)
    {
        // Stores two numbers
        double num1, num2;

        // Take input from the user
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the numbers:");

        // Take the inputs
        num1 = sc.nextDouble();
        num2 = sc.nextDouble();

        System.out.println("Enter the operator (+,-,*,/):");

        char op = sc.next().charAt(0);
        double o = 0;

        switch (op) {
        // case to add two numbers
        case '+':
            o = num1 + num2;
            break;

        // case to subtract two numbers
        case '-':
            o = num1 - num2;
            break;

        // case to multiply two numbers
        case '*':
            o = num1 * num2;
            break;

        // case to divide two numbers
        case '/':
            o = num1 / num2;
            break;

        default:
            System.out.println("You enter wrong input");
        }

        System.out.println("The final result:");
        System.out.println();

        // print the final result
        System.out.println(num1 + " " + op + " " + num2
                                 + " = " + o);
    }
}
```

```
Enter the numbers:
2
2

Enter the operator (+,-,*,/)
+
The final result:
2.0 + 2.0 = 4.0
```

Java File Class

```
// importing the File class
import java.io.File;

class Main {
  public static void main(String[] args) {

    // create a file object for the current location
    File file = new File("newFile.txt");

    try {

      // trying to create a file based on the object
      boolean value = file.createNewFile();
      if (value) {
        System.out.println("The new file is created.");
      }
      else {
        System.out.println("The file already exists.");
      }
    }
    catch(Exception e) {
      e.getStackTrace();
    }
  }
}
```

```
// importing the FileReader class
import java.io.FileReader;

class Main {
  public static void main(String[] args) {

    char[] array = new char[100];
    try {
      // Creates a reader using the FileReader
      FileReader input = new FileReader("input.txt");

      // Reads characters
      input.read(array);
      System.out.println("Data in the file:");
      System.out.println(array);

      // Closes the reader
      input.close();
    }
    catch(Exception e) {
      e.getStackTrace();
    }
  }
}
```

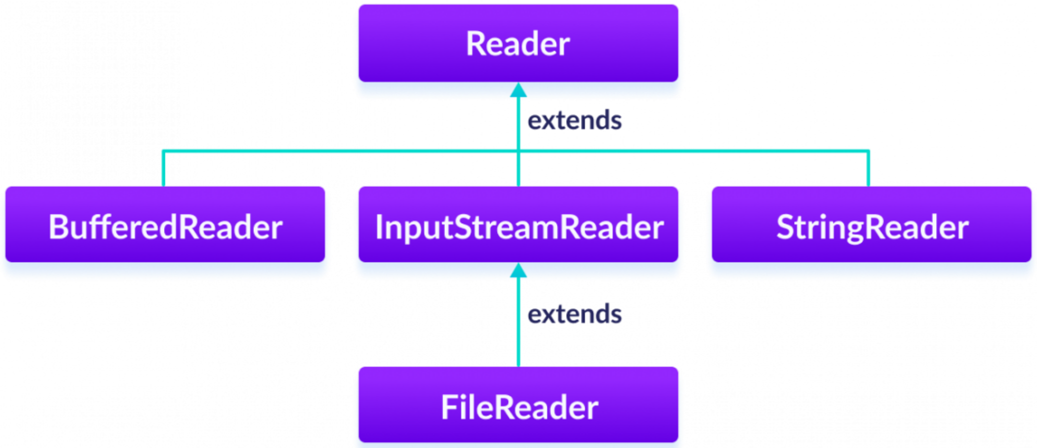
Java File Class

```
// importing the File class
import java.io.File;

class Main {
 public static void main(String[] args) {

  // create a file object for the current location
  File file = new File("newFile.txt");

  try {

   // trying to create a file based on the object
   boolean value = file.createNewFile();
   if (value) {
    System.out.println("The new file is created.");
   }
   else {
    System.out.println("The file already exists.");
   }
  }
  catch(Exception e) {
   e.getStackTrace();
  }
 }
}
```

```
import java.io.File;

class Main {
 public static void main(String[] args) {

  // creates a file object
  File file = new File("file.txt");

  // deletes the file
  boolean value = file.delete();
  if(value) {
   System.out.println("The File is deleted.");
  }
  else {
   System.out.println("The File is not deleted.");
  }
 }
}
```

Java Reader Class

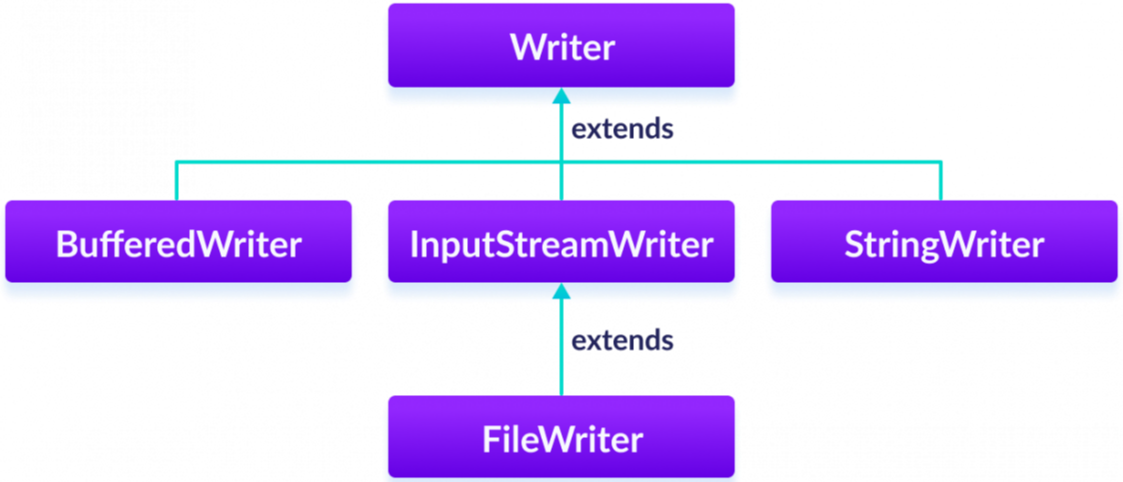
```
import java.io.Reader; import java.io.FileReader;

class Main {
  public static void main(String[] args) {

    // Creates an array of character
    char[] array = new char[100];

    try {
      // Creates a reader using the FileReader
      Reader input = new FileReader("input.txt");

      // Checks if reader is ready
      System.out.println("Is there data in the stream?  " + input.ready());

      // Reads characters
      input.read(array);
      System.out.println("Data in the stream:");
      System.out.println(array);

      // Closes the reader
      input.close();
    }

    catch(Exception e) {
      e.getStackTrace();
    }
  }
```



```
Is there data in the stream?  true
Data in the stream:
This is a line of text inside the file.
```

Java Writer Class

```
// importing the FileWriter class
import java.io.FileWriter;

 class Main {
  public static void main(String args[]) {

    String data = "This is the data in the output file";
    try {
      // Creates a Writer using FileWriter
      FileWriter output = new FileWriter("output.txt");

      // Writes string to the file
      output.write(data);
      System.out.println("Data is written to the file.");

      // Closes the writer
      output.close();
    }
    catch (Exception e) {
      e.getStackTrace();
    }
  }
}
```



Java InputStream Class

```java
import java.io.FileInputStream;
import java.io.InputStream;

class Main {
 public static void main(String args[]) {

  byte[] array = new byte[100];

  try {
   InputStream input = new FileInputStream("input.txt");

   System.out.println("Available bytes in the file: " + input.available());

   // Read byte from the input stream
   input.read(array);
   System.out.println("Data read from the file: ");

   // Convert byte array into string
   String data = new String(array);
   System.out.println(data);

   // Close the input stream
   input.close();
  } catch (Exception e) {
   e.getStackTrace();
  }
 }
}
```

```
Available bytes in the file: 39
Data read from the file:
This is a line of text inside the file
```

Java OutputStream Class

```
import java.io.FileOutputStream;
import java.io.OutputStream;

public class Main {

    public static void main(String args[]) {
        String data = "This is a line of text inside the file.";

        try {
            OutputStream out = new FileOutputStream("output.txt");

            // Converts the string into bytes
            byte[] dataBytes = data.getBytes();

            // Writes data to the output stream
            out.write(dataBytes);
            System.out.println("Data is written to the file.");

            // Closes the output stream
            out.close();
        }

        catch (Exception e) {
            e.getStackTrace();
        }
    }
}
```

Java FileInputStream Class

```
import java.io.FileInputStream;

public class Main {

  public static void main(String args[]) {

    try {
      FileInputStream input = new FileInputStream("input.txt");

      System.out.println("Data in the file: ");

      // Reads the first byte
      int i = input.read();

     while(i != -1) {
        System.out.print((char)i);

        // Reads next byte from the file
        i = input.read();
      }
      input.close();
    }

    catch(Exception e) {
      e.getStackTrace();
    }
  }
}
```

```
import java.io.FileInputStream;

public class Main {

  public static void main(String args[]) {

    try {
      // Suppose, the input.txt file contains the following text
      //This is a line of text inside the file.
      FileInputStream input = new FileInputStream("input.txt");

      // Returns the number of available bytes
      System.out.println("Available bytes at the beginning: " + input.available());

      // Reads 3 bytes from the file
      input.read();
      input.read();
      input.read();

      // Returns the number of available bytes
      System.out.println("Available bytes at the end: " + input.available());

      input.close();
    }

    catch (Exception e) {
      e.getStackTrace();
    }
```

Java FileInputStream Class

```
import java.io.FileInputStream;

public class Main {

  public static void main(String args[]) {

    try {
      // Suppose, the input.txt file contains the following text
      //This is a line of text inside the file.
      FileInputStream input = new FileInputStream("input.txt");

      // Skips the 5 bytes
      input.skip(5);
      System.out.println("Input stream after skipping 5 bytes:");

      // Reads the first byte
      int i = input.read();
      while (i != -1) {
        System.out.print((char) i);

        // Reads next byte from the file
        i = input.read();
      }

      // close() method
      input.close();
    }
    catch (Exception e) {
      e.getStackTrace();
    }
  }
}
```

Output

```
Input Stream after skipping 5 bytes:
is a line of text inside the file.
```

Java FileOutputStream Class

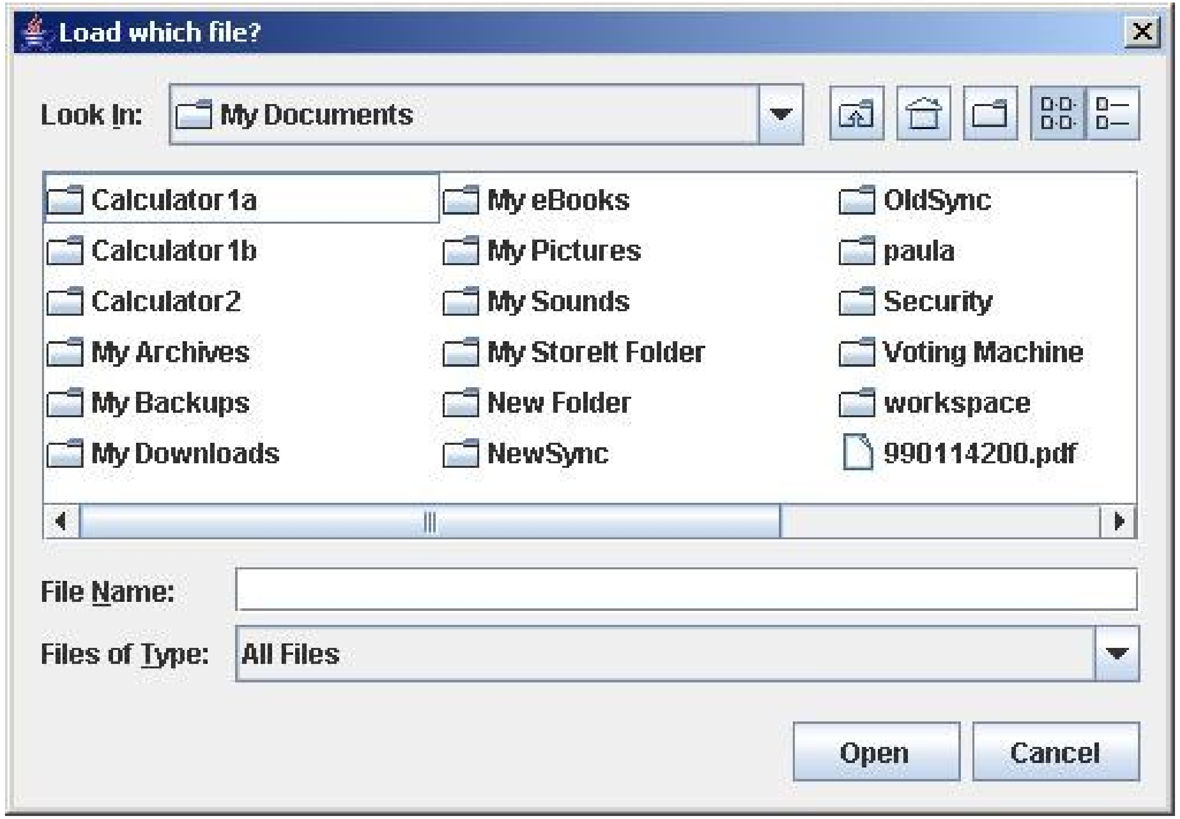
```
import java.io.FileOutputStream;

public class Main {
    public static void main(String[] args) {

        String data = "This is a line of text inside the file.";

        try {
            FileOutputStream output = new FileOutputStream("output.txt");

            byte[] array = data.getBytes();

            // Writes byte to the file
            output.write(array);

            output.close();
        }

        catch(Exception e) {
            e.getStackTrace();
        }
    }
}
```

```
import java.io.FileOutputStream;
import java.io.IOException;

public class Main {
    public static void main(String[] args) throws IOException {

        FileOutputStream out = null;
        String data = "This is demo of flush method";

        try {
            out = new FileOutputStream(" flush.txt");

            // Using write() method
            out.write(data.getBytes());

            // Using the flush() method
            out.flush();
            out.close();
        }
        catch(Exception e) {
            e.getStackTrace();
        }
    }
}
```

JFileChooser

About JFileChoosers

- The JFileChooser class displays a window from which the user can select a file
- The dialog window is modal--the application cannot continue until it is closed
- Applets cannot use a JFileChooser, because applets cannot access files



Load which file?

Look In: My Documents

Calculator1a
Calculator1b
Calculator2
My Archives
My Backups
My Downloads
My eBooks
My Pictures
My Sounds
My StoreIt Folder
New Folder
NewSync
OldSync
paula
Security
Voting Machine
workspace
990114200.pdf

File Name:

Files of Type: All Files

JFileChooser constructors

- JFileChooser()
  - Creates a JFileChooser starting from the user's directory

- JFileChooser(File *currentDirectory*)
  - Constructs a JFileChooser using the given File as the path

- JFileChooser(String *currentDirectoryPath*)
  - Constructs a JFileChooser using the given path

Useful JFileChooser methods I

- int showOpenDialog(Component ***enclosingJFrame***);
  - Asks for a file to read; returns a flag (see below)

- int showSaveDialog(Component ***enclosingJFrame***);
  - Asks where to save a file; returns a flag (see below)

- Returned flag value may be:
  - JFileChooser.APPROVE_OPTION
  - JFileChooser.CANCEL_OPTION
  - JFileChooser.ERROR_OPTION

Useful JFileChooser methods II

- File getSelectedFile()
  - showOpenDialog and showSaveDialog return a flag telling what happened, but don't return the selected file
  - After we return from one of these methods, we have to ask the JFileChooser what file was selected
  - If we are saving a file, the File may not actually exist yet—that's OK, we still have a File *object* we can use

Using a File

- Assuming that we have successfully selected a File:

```
File file = chooser.getSelectedFile();
if (file != null) {
    String fileName = file.getCanonicalPath();
    FileReader fileReader = new FileReader(fileName);
    BufferedReader reader = new BufferedReader(fileReader);
 }
```

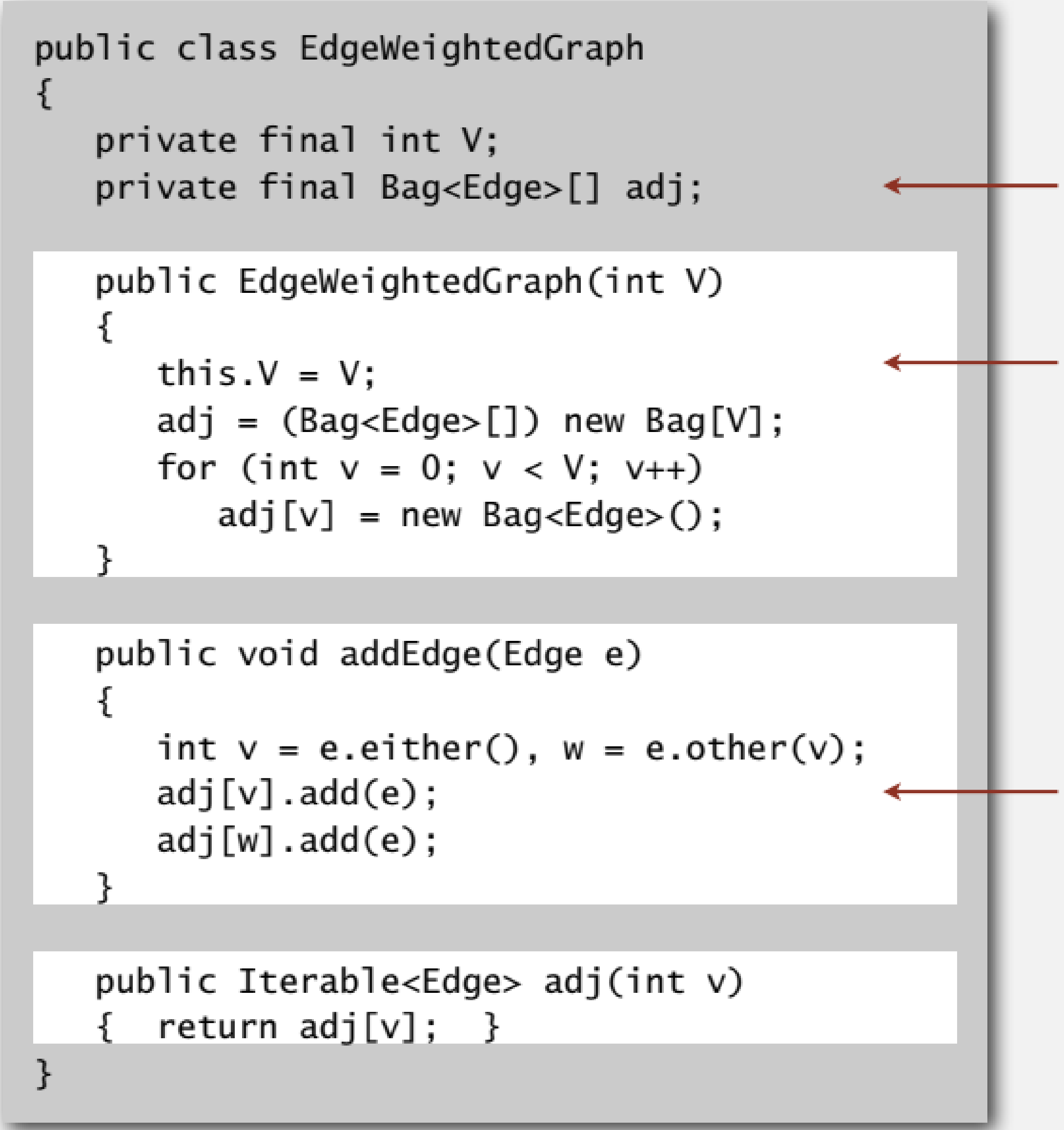
```
File file = chooser.getSelectedFile();
if (file != null) {
    String fileName = file.getCanonicalPath();
    FileOutputStream stream = new FileOutputStream(fileName);
    writer = new PrintWriter(stream, true);
}
```

Summary

- Files with characters are text files
  - Other files are binary files
- Programs can use `PrintWriter` and `Scanner` for I/O
- Always check for end of file
- File name can be literal string or variable of type `String`
- Class `File` gives additional capabilities to deal with file names

Java code Implementation of Kruskal’s algorithm and Prim’s algorithm

Java Implementation

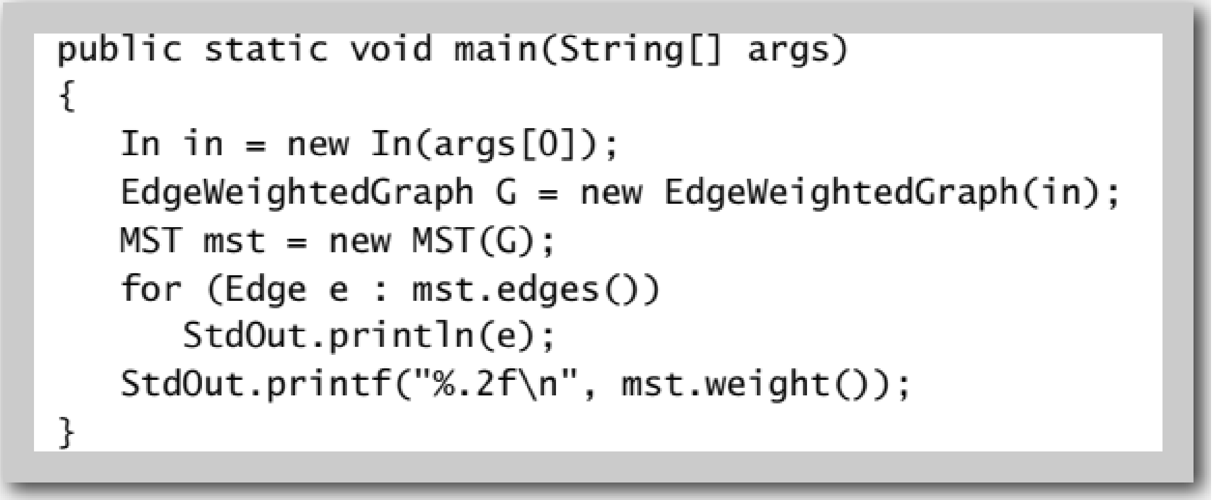


```
public class EdgeWeightedGraph
{
   private final int V;
   private final Bag<Edge>[] adj;

   public EdgeWeightedGraph(int V)
   {
      this.V = V;
      adj = (Bag<Edge>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Edge>();
   }

   public void addEdge(Edge e)
   {
      int v = e.either(), w = e.other(v);
      adj[v].add(e);
      adj[w].add(e);
   }

   public Iterable<Edge> adj(int v)
   {  return adj[v];  }
}
```

same as Graph, but adjacency lists of Edges instead of integers

constructor

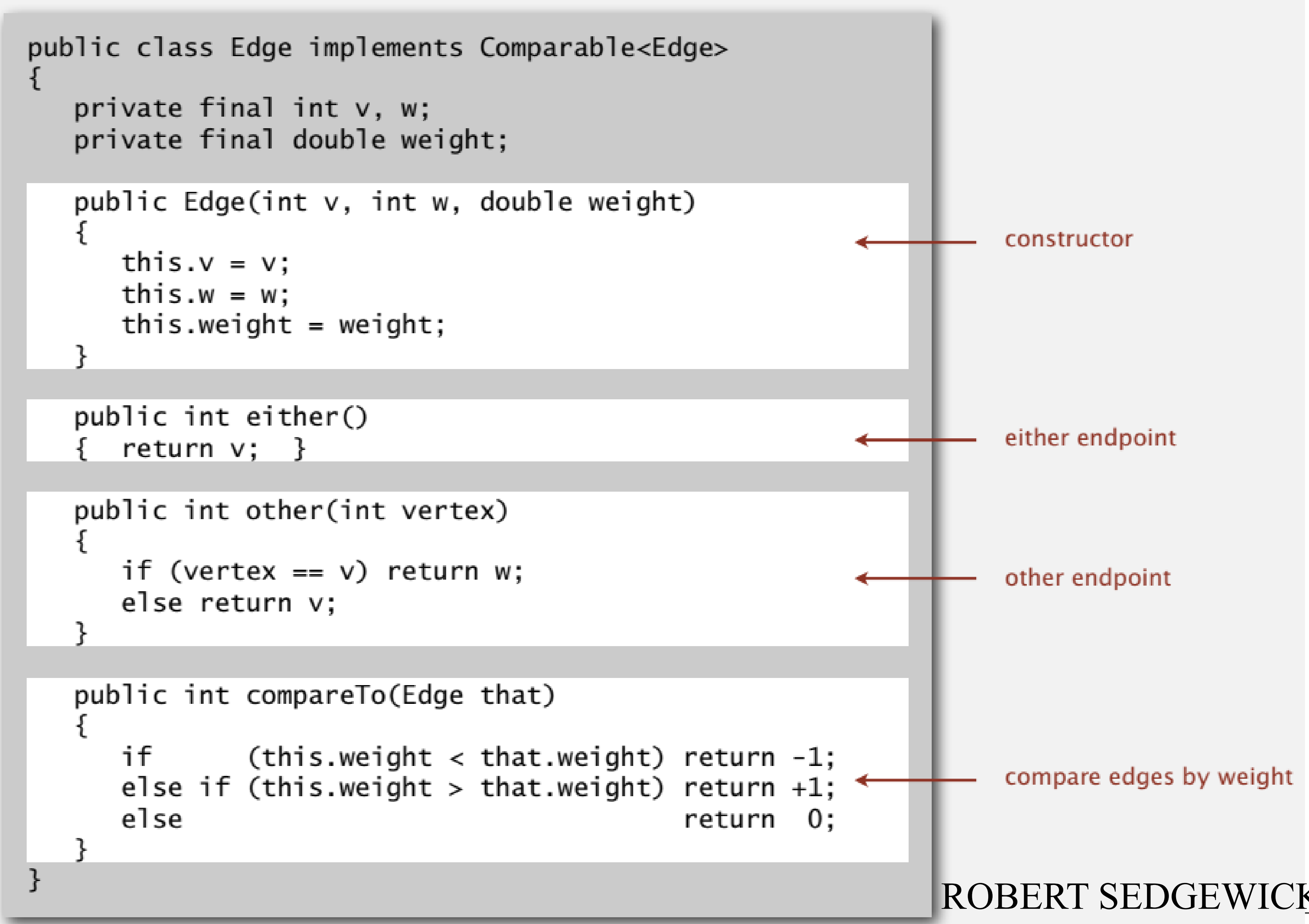
add edge to both adjacency lists



```
public static void main(String[] args)
{
   In in = new In(args[0]);
   EdgeWeightedGraph G = new EdgeWeightedGraph(in);
   MST mst = new MST(G);
   for (Edge e : mst.edges())
      StdOut.println(e);
   StdOut.printf("%.2f\n", mst.weight());
}
```

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition

Java Implementation



```
public class Edge implements Comparable<Edge>
{
   private final int v, w;
   private final double weight;

   public Edge(int v, int w, double weight)
   {
      this.v = v;
      this.w = w;
      this.weight = weight;
   }

   public int either()
   {  return v;  }

   public int other(int vertex)
   {
      if (vertex == v) return w;
      else return v;
   }

   public int compareTo(Edge that)
   {
      if      (this.weight < that.weight) return -1;
      else if (this.weight > that.weight) return +1;
      else                                return  0;
   }
}
```

constructor

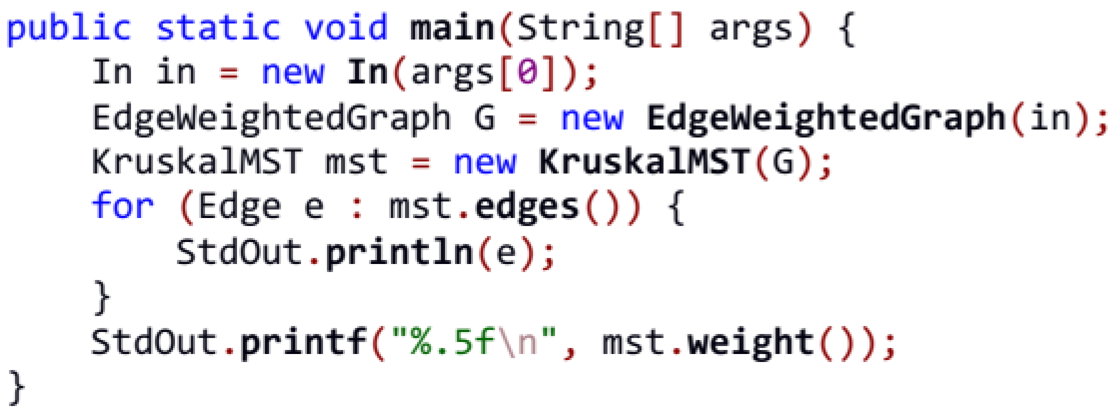
either endpoint

other endpoint

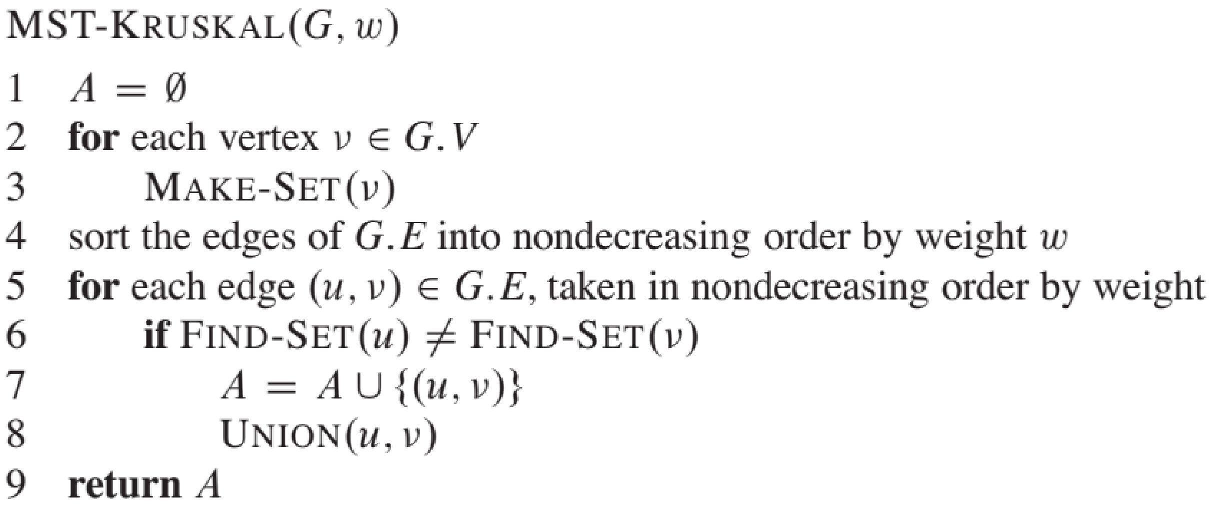
compare edges by weight

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition

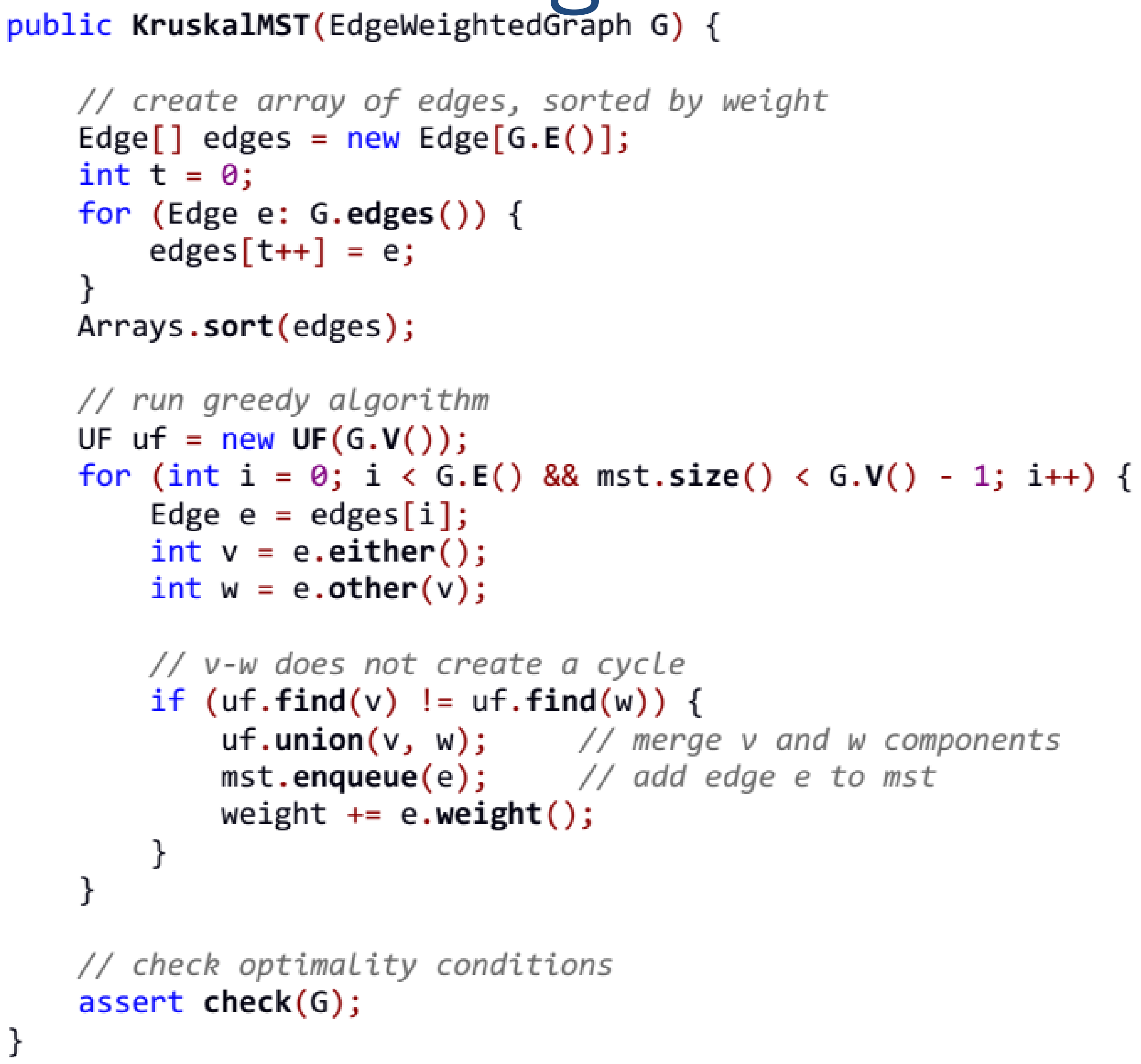
Java Implementation – Kruskal's Algorithm



```java
public static void main(String[] args) {
    In in = new In(args[0]);
    EdgeWeightedGraph G = new EdgeWeightedGraph(in);
    KruskalMST mst = new KruskalMST(G);
    for (Edge e : mst.edges()) {
        StdOut.println(e);
    }
    StdOut.printf("%.5f\n", mst.weight());
}
```



```
MST-KRUSKAL($G, w$)
  $A = \emptyset$
  for each vertex $v \in G.V$
      MAKE-SET($v$)
  sort the edges of $G.E$ into nondecreasing order by weight $w$
  for each edge $(u, v) \in G.E$, taken in nondecreasing order by weight
      if FIND-SET($u$) $\neq$ FIND-SET($v$)
          $A = A \cup \{(u, v)\}$
          UNION($u, v$)
  return $A$
```

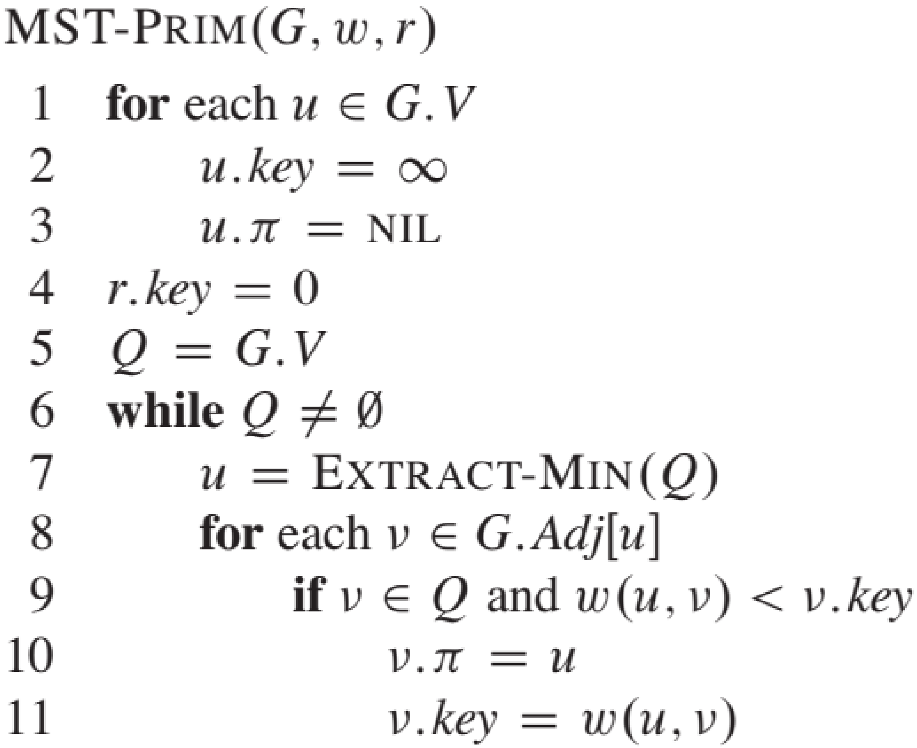


```java
public KruskalMST(EdgeWeightedGraph G) {

    // create array of edges, sorted by weight
    Edge[] edges = new Edge[G.E()];
    int t = 0;
    for (Edge e: G.edges()) {
        edges[t++] = e;
    }
    Arrays.sort(edges);

    // run greedy algorithm
    UF uf = new UF(G.V());
    for (int i = 0; i < G.E() && mst.size() < G.V() - 1; i++) {
        Edge e = edges[i];
        int v = e.either();
        int w = e.other(v);

        // v-w does not create a cycle
        if (uf.find(v) != uf.find(w)) {
            uf.union(v, w);     // merge v and w components
            mst.enqueue(e);     // add edge e to mst
            weight += e.weight();
        }
    }

    // check optimality conditions
    assert check(G);
}
```

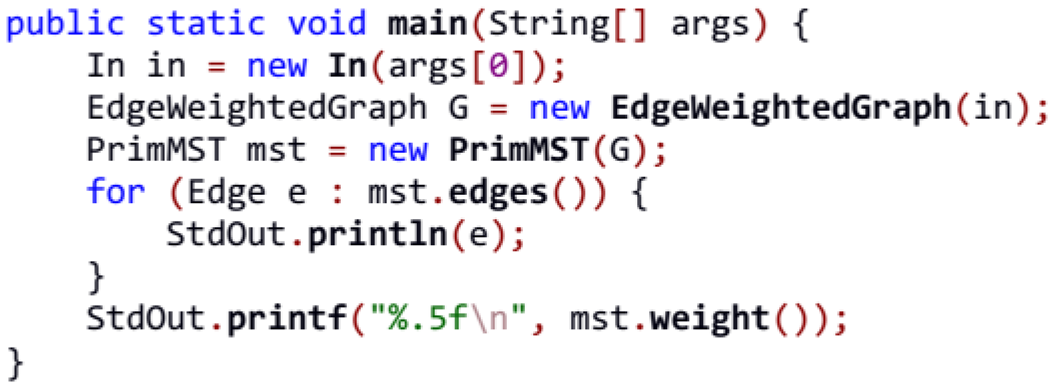
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition

Java Implementation – Prim’s Algorithm



```
MST-PRIM$(G, w, r)$
for each $u \in G.V$
    $u.key = \infty$
    $u.\pi = \text{NIL}$
$r.key = 0$
$Q = G.V$
while $Q \neq \emptyset$
    $u = \text{EXTRACT-MIN}(Q)$
    for each $v \in G.Adj[u]$
        if $v \in Q$ and $w(u, v) < v.key$
            $v.\pi = u$
            $v.key = w(u, v)$
```

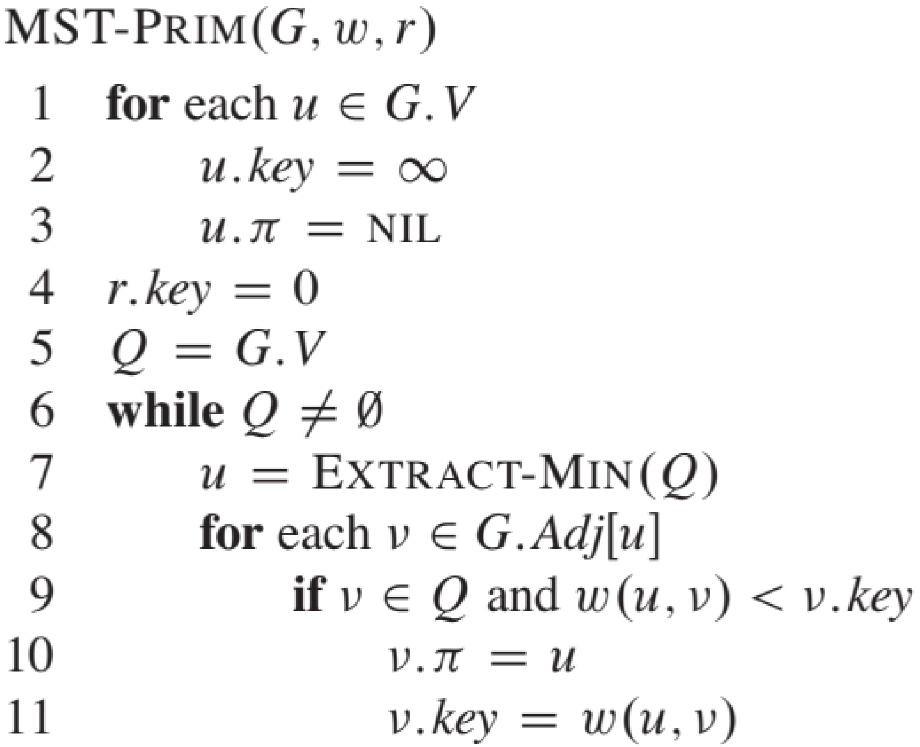


```java
public static void main(String[] args) {
    In in = new In(args[0]);
    EdgeWeightedGraph G = new EdgeWeightedGraph(in);
    PrimMST mst = new PrimMST(G);
    for (Edge e : mst.edges()) {
        StdOut.println(e);
    }
    StdOut.printf("%.5f\n", mst.weight());
}
```

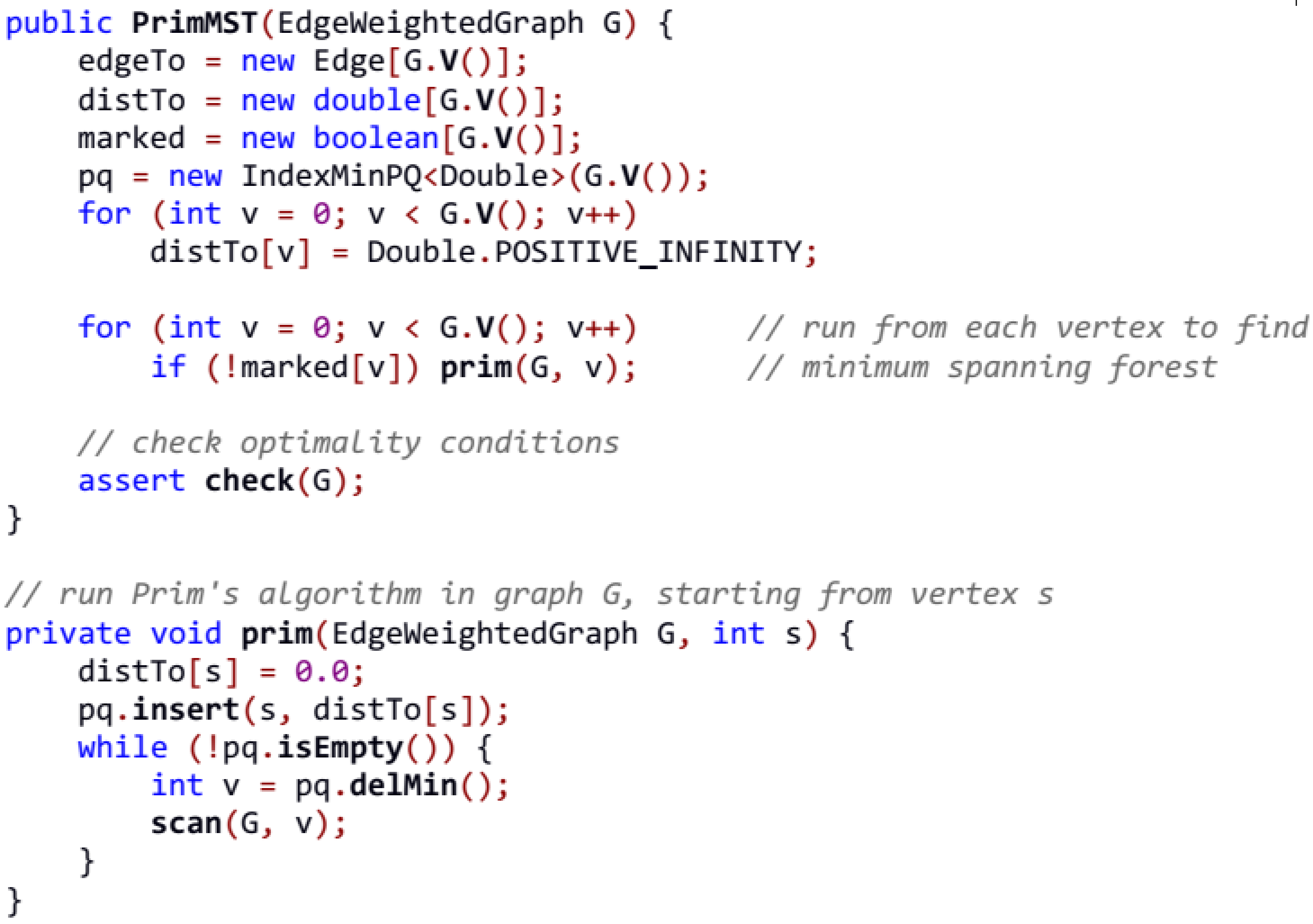
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition

Java Implementation – Prim’s Algorithm



```
MST-PRIM($G, w, r$)
for each $u \in G.V$
    $u.key = \infty$
    $u.\pi = \text{NIL}$
$r.key = 0$
$Q = G.V$
while $Q \neq \emptyset$
    $u = \text{EXTRACT-MIN}(Q)$
    for each $v \in G.Adj[u]$
        if $v \in Q$ and $w(u, v) < v.key$
            $v.\pi = u$
            $v.key = w(u, v)$
```

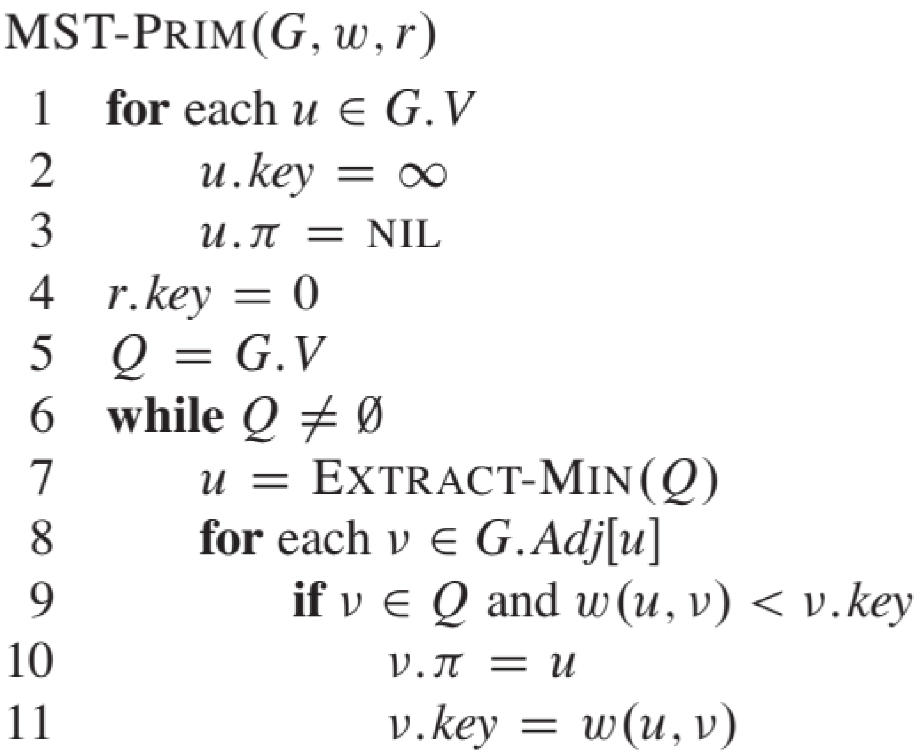


```
public PrimMST(EdgeWeightedGraph G) {
    edgeTo = new Edge[G.V()];
    distTo = new double[G.V()];
    marked = new boolean[G.V()];
    pq = new IndexMinPQ<Double>(G.V());
    for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;

    for (int v = 0; v < G.V(); v++)      // run from each vertex to find
        if (!marked[v]) prim(G, v);      // minimum spanning forest

    // check optimality conditions
    assert check(G);
}

// run Prim's algorithm in graph G, starting from vertex s
private void prim(EdgeWeightedGraph G, int s) {
    distTo[s] = 0.0;
    pq.insert(s, distTo[s]);
    while (!pq.isEmpty()) {
        int v = pq.delMin();
        scan(G, v);
    }
}
```

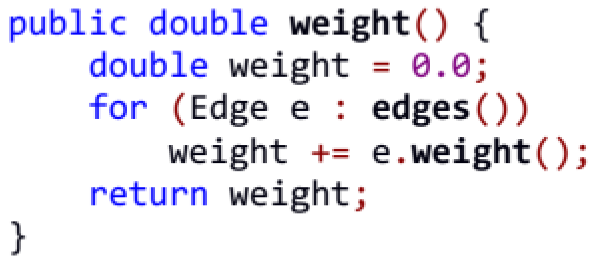
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition

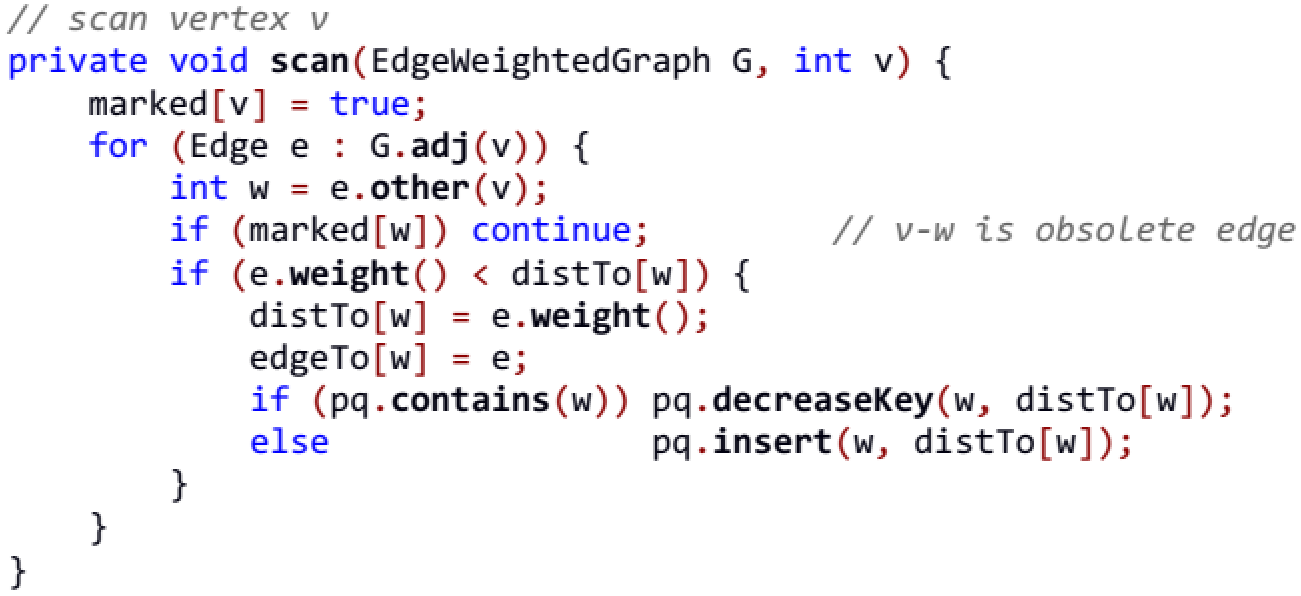
Java Implementation – Prim’s Algorithm



```
MST-PRIM(G, w, r)
for each $u \in G.V$
    $u.key = \infty$
    $u.\pi = \text{NIL}$
$r.key = 0$
$Q = G.V$
while $Q \neq \emptyset$
    $u = \text{EXTRACT-MIN}(Q)$
    for each $v \in G.Adj[u]$
        if $v \in Q$ and $w(u, v) < v.key$
            $v.\pi = u$
            $v.key = w(u, v)$
```



```java
public double weight() {
    double weight = 0.0;
    for (Edge e : edges())
        weight += e.weight();
    return weight;
}
```



```java
// scan vertex v
private void scan(EdgeWeightedGraph G, int v) {
    marked[v] = true;
    for (Edge e : G.adj(v)) {
        int w = e.other(v);
        if (marked[w]) continue;         // v-w is obsolete edge
        if (e.weight() < distTo[w]) {
            distTo[w] = e.weight();
            edgeTo[w] = e;
            if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
            else                pq.insert(w, distTo[w]);
        }
    }
}
```

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Vol. 3, pp. 624-642). Cambridge: MIT press.

ROBERT SEDGEWICK | KEVIN WAYNE. *Algorithms* 4th Edition

תודה רבה
Hebrew

Ευχαριστώ
Greek

Спасибо
Russian

Danke

German

धन्यवादः
Sanskrit

Merci

French

ধন্যবাদ
Bangla

நன்றி
Tamil

شكراً
Arabic

ಧನ್ಯವಾದಗಳು
Kannada

Thank You
English

നന്ദി
Malayalam

多謝
Traditional Chinese

Grazie

Italian

ధన్యవాదాలు
Telugu

આભાર
Gujarati

ਧੰਨਵਾਦ
Punjabi

धन्यवाद
Hindi & Marathi

Gracias

Spanish

多谢
Simplified Chinese

https://sites.google.com/site/animeshchaturvedi07

Obrigado
Portuguese

ありがとうございました
Japanese

ขอบคุณ
Thai

감사합니다

Korean