## Exception handling

- An exception is generated (raised) with the *throw* statement:

```
throw object;
```

where *object* is an instance of a subclass of `Exception` or `Throwable`

- The *try-catch* statement:

```
try {
  statements;
}
catch (ExceptionSubclass1 e) {
  statements1;
}
catch (ExceptionSubclass2 e) {
  statements2;
}
.
.
.
```

McGill

## Exception handling

- A try-catch statement executes its default statements in sequence, and
  - If no exception is raised, then computation continues after the catch clauses
  - Otherwise, if an exception is raised, the sequence of statements is interrupted, and execution continues in the catch clause that matches the type of the exception

- After a catch clause finishes, computation continues after the try-catch. This is, the flow of control does not return to the point where the exception occurred. *Note: It never returns to the method that raised the exception, in contrast with a method call.*

- An exception which is not caught by a try-catch, is "propagated", i.e. it is raised again

## Exception handling

```
class MyException extends Exception {
  String message;
  MyException(String m) { message = m; }
  public String toString()
  {
    return ‘‘MyException occurred: ’’+message;
  }
}

class MyOtherException extends Exception {
  int code;
  MyOtherException(int c) { code = c; }
  public String toString() { return ‘‘’’+code; }
}
```

## Exception handling

```
static int q(float f) throws MyOtherException
{
  if (f < -5)
    throw new MyOtherException(7);
  return f * 3 + 1;
}
static float r(float f) throws MyException
{
  if (f > 15)
    throw new MyException(“r: ”+f);
  return f - 2;
}
```

## Exception handling

```
static void p()
{
  float n = Keyboard.readFloat();
  try {
    int m = q(r(n));
    System.out.println(m);
  }
  catch (MyException e) {
    System.out.println(e);
  }
  catch (MyOtherException e) {
    String s = e;
  }
}
```

## Exception handling

- If MyOtherException is not caught, it repropagates

```
static void p() throws MyOtherException
{
  float n = Keyboard.readFloat();
  try {
    int m = q(r(n));
    System.out.println(m);
  }
  catch (MyException e) {
    System.out.println(e);
  }
}
```

- Note: p does not throw an exception explicitely

## Exception handling

```
class Food {
  boolean fresh, smelly;
}
class FoulSmell extends Exception {
  public String toString() {
    return “Yuck”;
  }
}
class FoodPoison extends Exception {
  public String toString() {
    return “Ouch”;
  }
}
```

## Exception handling

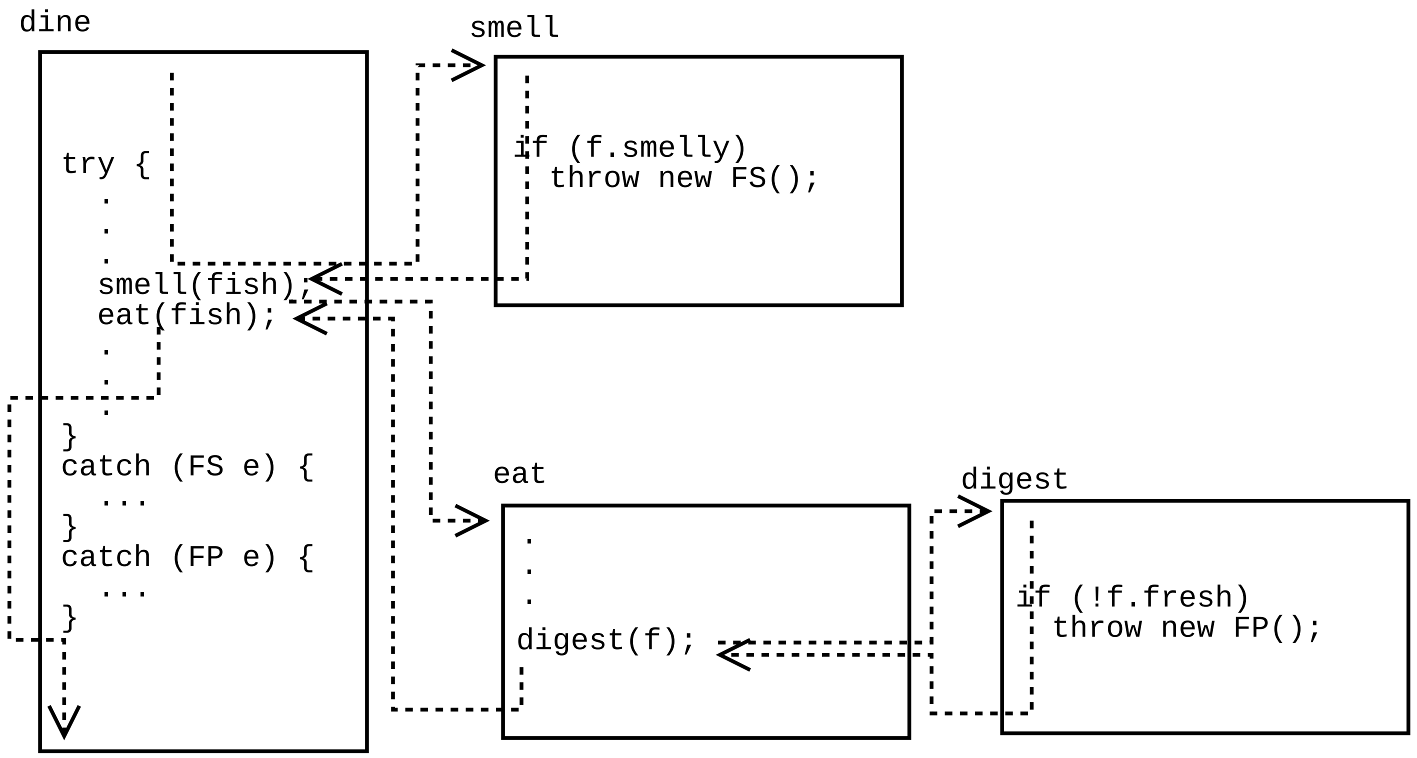
```
static void smell(Food f) throws FoulSmell
{
  if (f.smelly)
    throw new FoulSmell();
  System.out.println("Smells OK");;
}
static void eat(Food f) throws FoodPoison
{
  System.out.println("Hmmm...");
  digest(f);
}
static void digest(Food f) throws FoodPoison
{
  if (!f.fresh)
    throw new FoodPoison();
}
```

## Exception handling

```
static void dine()
{
  try {
    Food fish = new Food();
    fish.smelly = false;
    fish.fresh = false;
    smell(fish);
    eat(fish);
  }
  catch (FoulSmell e) {
    System.out.println(e);
  }
  catch (FoodPoison e) {
    System.out.println(e);
  }
}
```

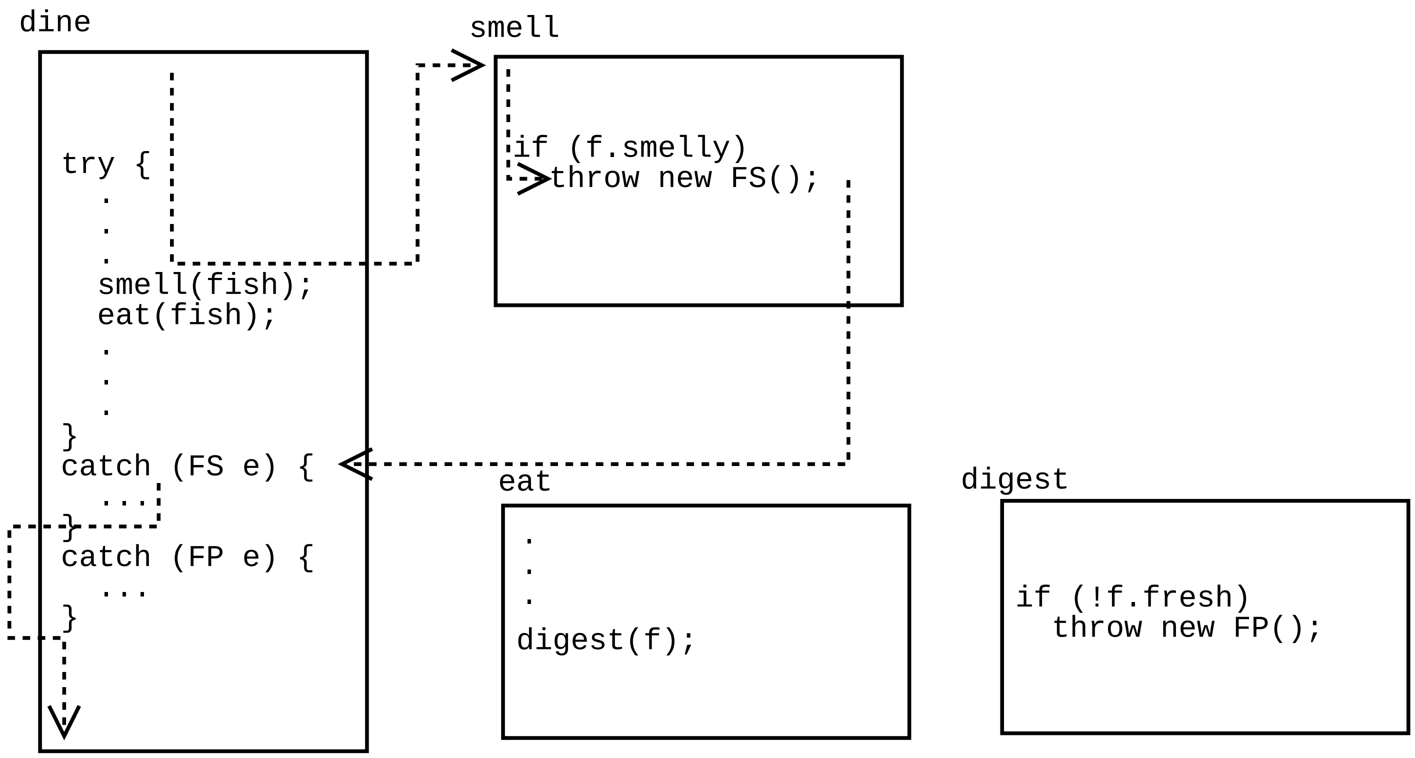
## Exception handling

```
// fish.smelly = false; fish.fresh = true;
```



# Exception handling

```
// fish.smelly = true;
```



```
dine
try {
  .
  .
  .
  smell(fish);
  eat(fish);
  .
  .
  .
}
catch (FS e) {
  ...
}
catch (FP e) {
  ...
}
```

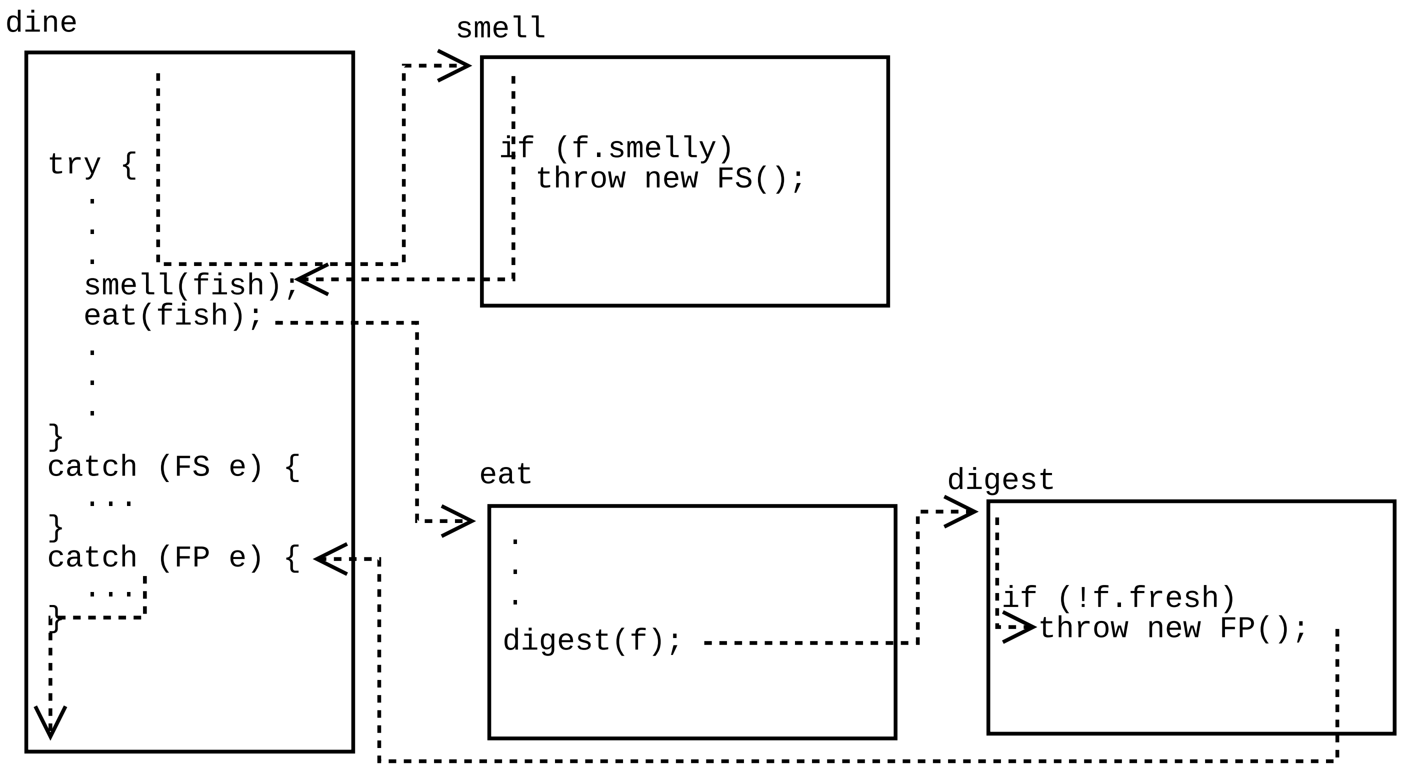
```
smell
if (f.smelly)
  throw new FS();
```

```
eat
.
.
.
digest(f);
```

```
digest
if (!f.fresh)
  throw new FP();
```

## Exception handling

```
// fish.smelly = false; fish.fresh = false;
```



## Exception handling

- A method can throw more than one class of exceptions:

```
void m() throws A, B, ...
{
  ... throw new A() ...
  ... throw new B(...) ...
}
```

- ... but the exception needs not be raised explicitly in the method itself: it can be raised by another method called by `m`.

## Exception handling

- Exceptions can be used not only for errors, but for control-flow too:

```
class Sheep {
  private int id;
  public Sheep(int i) { n = i; }
  public void jump()
  {
    System.out.println(‘‘Sheep #’’+id+’’ jumped’’);
    if (id == 6)
      throw new LoudSound(i);
  }
}
```

## Exception handling

```
class LoudSound extends Throwable {
  private int n;
  public LoudSound(int i) { n = i; }
  public toString()
  {
    return ‘‘I was in sheep #’’+n;
  }
}
```

## Exception handling

```
class GoToSleep {
  public static void main(String[] args)
  {
    try {
      for (int i = 1; i < 100; i++) {
        Sheep s = new Sheep(i);
        s.jump();
      }
      System.out.println(“zzzz...”);
    }
    catch (LoudSound s) {
      System.out.println(s);
    }
  }
}
```

## Exception handling

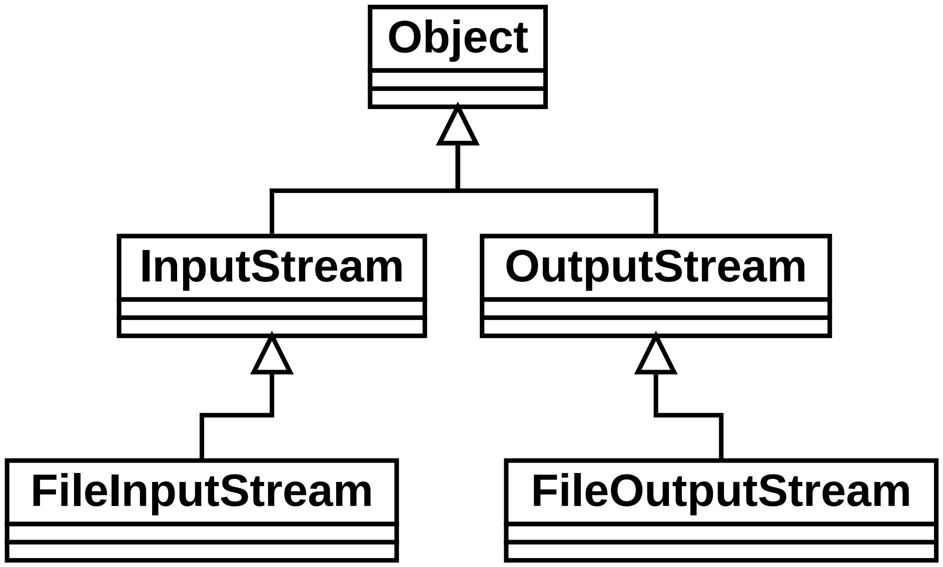
- Some exceptions arise without an explicit throw.
- Some standard exceptions

```
Exception
  RunTimeException
    IndexOutOfBounds
    StringIndexOutOfBounds
    ArithmeticException (e.g. division by 0)
    NullPointerException
  NoSuchMethodException
  ClassNotFoundException
```

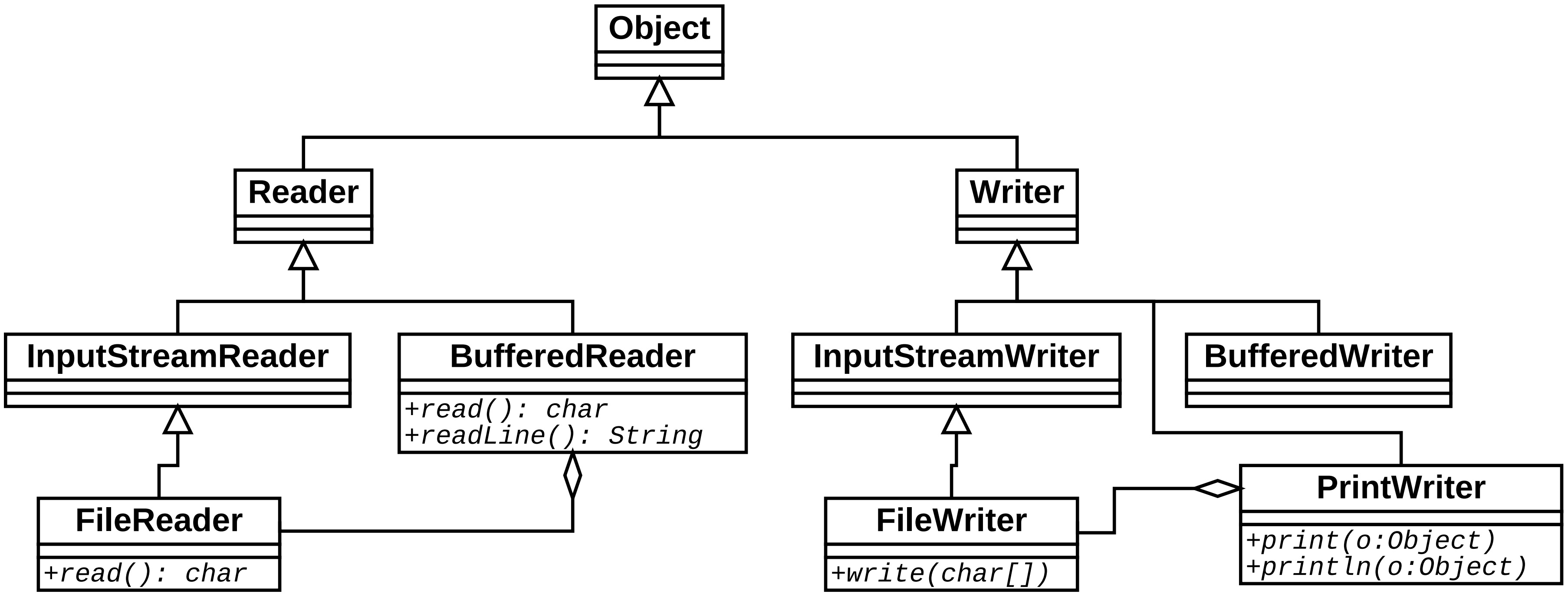
## I/O

- When an object is created, it is destroyed whenever there are no more references to it.
- Sometimes we want to save the information of an object beyond the lifetime of the program.
- Operations
  - Save information into a file
  - Load information from a file into some sata-structure
- `java.io` package
- Streams
  - A *stream* is a sequence of elements. Possibly with no fixed size.
  - I/O Streams (char vs. byte streams.)
  - Stream operations: read from the stream, and write to the stream.
  - Associate an I/O stream with a file.

## I/O


Object
InputStream
OutputStream
FileInputStream
FileOutputStream

## I/O


Object
Reader
Writer
InputStreamReader
BufferedReader
+read(): char
+readLine(): String
InputStreamWriter
BufferedWriter
FileReader
+read(): char
FileWriter
+write(char[])
PrintWriter
+print(o:Object)
+println(o:Object)

## I/O

- When writing files, decide a file format.
- Then any application reading the file should be aware of this format.
- Saving a directory of names and telephones; format:

```
name1  tel1
name2  tel2
name3  tel3
.
.
.
```

- Each line is a record
- Each line is divided in fields, separated by spaces
- Separators could be different but we must be consistent.

## I/O

```
import java.io.*;
import cs1.Keyboard;

public class SavingTest {
  public static void main(String[] args)
  {
    String[] names = new String[200];
    String[] tels = new String[200];

    enter_data(names, tels);
    String file_name = “agenda.dat”;

    save_file(file_name, names, tels);
  }
  // Continues below...
```

## I/O

```
static void enter_data(String[] names,
                       String[] tels)
{
  for (int i = 0; i < names.length; i++) {
    System.out.print(“Enter a name: ”);
    names[i] = Keyboard.readString();
    System.out.print(“Enter a telephone: ”);
    tels[i] = Keyboard.readString();
  }
}
```

## I/O

```
static void save_file(String file_name,
                        String[] names,
                        String[] tels)
  {
    FileWriter fw = new FileWriter(file_name);
    BufferedWriter bw = new BufferedWriter(fw);
    PrintWriter file = new PrintWriter(bw);
    int line = 0;
    while (line < names.length) {
      String record = names[line] + “   ”
                    + tels[line];
      file.println(record);
      line++;
    }
    file.close();
  }
} // End of SavingTest
```

## I/O

```
import java.io.*;
public class LoadingTest {
  public static void main(String[] args)
  {
    String[] names = new String[200];
    String[] tels = new String[200];

    String file_name = “agenda.dat”;

    load_file(file_name, names, tels);

    print_data(names, tels);
  }
}
```

## I/O

```
static void print_data(String[] names,
                       String[] tels)
{
  for (int i = 0; i < names.length; i++) {
    System.out.println(“Record #”+i+“: ”+
                       names[i] + “,” +
                       tels[i]);
  }
}
```

## I/O

```
static void load_file(String file_name,
                      String[] names,
                      String[] tels)
{
  String line;
  int line_num = 0;
  StringTokenizer tokenizer;
  try {
    FileReader fr = new FileReader(file_name);
    BufferedReader file = new BufferedReader(fr);
    line = file.readLine();
    while (line != null) {
      tokenizer = new StringTokenizer(line);
      names[line_num] = tokenizer.nextToken();
      tels[line_num] = tokenizer.nextToken();
      line = file.readLine();
      line++;
    }
    file.close();
  }
  // Continues below ...
```

```
      catch (FileNotFoundException e) {
        System.out.println(“The file was not found: 
      }
      catch (IOException e) {
        System.out.println(“An I/O error occurred: ”
      }
    } // End of load_file
  } // End of SavingTest
```

## I/O

- To write to a file we must:
  - Decide a format for the data
  - Create a FileWriter and associate it with the actual file
  - Possibly associate a BufferedWriter and PrintWriter to the FileWriter
  - Use methods print, println, or write, on each record to be saved
  - Close the file when finnished
- To read from a file
  - Create a FileReader and associate it with the actual file
  - Possibly associate a BufferedReader to the FileReader
  - Use methods read and readLine
  - Create a StringTokenizer associated to each line
  - Use the method nextToken to get each field in a record
  - Close the file when finnished