Programming Foundations
FIT9131

Error handling and File I/O

Week 10

Lecture outline

- Causes of errors
- Defensive programming
  - anticipating that things could go wrong
- Error reporting
- Exception "throwing" and handling
  - "checked" and "unchecked" exceptions
  - error recovery
- File I/O processing
  - writing to a text file
  - reading from a text file

Some common causes of errors

- Incorrect implementation of solution.
  - eg. solution does not meet the specification.
- Inappropriate object request.
  - eg. using invalid index for an array
  - eg. attempting to access an object in an array that has not been initialised – ie. the requested object does not exist yet
- Unanticipated use of an object which leaves an object in an inconsistent or inappropriate state.
  - eg. not initialising attributes of an object in its default constructor, or initialising them with illegal values

Other common causes of errors

- **Errors often arise from the operating environment:**
  - **eg. Invalid URL for web pages**
  - **eg. network interruptions**

- **File processing is particularly error-prone:**
  - **eg. missing files**
  - **eg. lack of appropriate access permissions**

Errors *do* happen and programs need to deal with them.
They should *not* be simply ignored.

Example : Exploring errors

We will explore error situations through the *address-book* (Chapter 14) projects.

We will consider two aspects:

- Error reporting
- Error handling

The *Address Book* class represents a typical *server* object, with all activities driven by *client* requests. We will examine the "*address-book-v1t*" project – which is the simplest version, and contains some potential errors.

Server classes Vs Client Classes

- **We often write classes to provide functionality to be used (or "*consumed*") by other classes. These are typically called "*Server Classes*", while the classes consuming the functionality provided are called "Client classes". In other words, *objects of the client class will call/use methods from objects of the server class, to perform some tasks*.**

Error-Handling responsibilities

So which class should be responsible for handling/preventing possible errors – the server class, or the client class?

Defensive programming

- *During Client-Server* **interactions :**
  1) **Should a server assume that clients are well-behaved?**
  2) *Or* **should it assume that clients are potentially hostile?**

- **In other words, who should be responsible for trapping/handling the errors?**

- **There are significant differences in the implementations required for these two views.**

Issues to be addressed

Need to consider issues such as:

Server

- How much checking should a server perform on client requests?
- How should a server report errors to clients?
- How should a server deal with failure?

Client

- How can a client anticipate failure of a request?
- How can a client detect an error condition coming from the server?
- How should a client deal with failure?

An example (address-book-v1t)

Let's start by creating an `AddressBook` object, and 2 `ContactDetails` objects. Then we use the `addDetails()` method in the `AddressBook` object to add the 2 `ContactDetails` objects to the `AddressBook`.

If we now access the `AddressBook` object from another object, we are treating it as a *server* object, since it is providing some service (eg. `addDetails()`, `removeDetails()`, `search()`, etc) via its public methods.

An example (address-book-v1t)

Now we can try out the various methods in the `AddressBook` object. These methods work if we use them in the "expected" way, but will fail if used in the wrong way – eg. trying to delete a contact detail which does not exist in the address book will cause the program to crash!

Side note : sometimes a object can act both as a server and a client in the same program – for instance, consider the relationship between an `AddressBook` object and its `ContactDetail` attributes.

AddressBook attribute

```
public class AddressBook
{
  private TreeMap<String, ContactDetails> book;
  private int numberOfEntries;
  …
}
```

Note : the `AddressBook` actually uses a `TreeMap` as its main attribute. A `TreeMap` works in a similar way to an `ArrayList` – the main difference being that each element in a `TreeMap` is a "pair" (`Key`+`Value`). To access an element, we provide a "key". In this example, the key can be either the name or the tel#.

Adding an entry to AddressBook

Create an `AddressBook` object.
Then try to add an entry.

parameter is a single ContactDetails object

```
public void addDetails(ContactDetails details)
{
  book.put(details.getName(), details);
  book.put(details.getPhone(), details);
  numberOfEntries++;
}
```

both the name and phone are added to the map, to allow the key to be either name or phone

one single entry added

Each time a single entry (a `ContactDetails` object) is added, 2 key-pairs are inserted into the map – so that the client can search the addressbook by either name/phone#.

Adding an entry to AddressBook

Create an `AddressBook` object.
Then try to remove an entry.

potential error
key may not be valid
null value may be returned

```
public void removeDetails(String key)
{
    ContactDetails details = book.get(key);
    book.remove(details.getName());
    book.remove(details.getPhone());
    numberOfEntries--;
}
```

BlueJ will report a runtime error if the key was invalid

If the key passed to `removeDetails()` is not valid, a runtime error results. Whose 'fault' is this?

Anticipation and prevention are preferable to apportioning blame.

Checking actual argument values

Receiving arguments through parameters represents a major area of vulnerability for a server object. Some typical situations:

- arguments passed to Constructors and Methods should be checked
- passing in incorrect/illegal arguments is a very common source of errors

Argument-checking is one *defensive* measure.

(eg. user input via `Scanner` objects – what if the user enters data of the wrong type?)

Example : Checking the key

```
// improved version
public void removeDetails(String key)
{
     if (keyInUse(key))
     {
          ContactDetails details = book.get(key);
          book.remove(details.getName());
          book.remove(details.getPhone());
          numberOfEntries--;
     }
}

public boolean keyInUse(String key)
{
     return book.containsKey(key);
}
```

first check that the key exists

only try the remove operations *IF* the key exists

new method to check if a key exists

Server : error reporting

How to *report* illegal arguments? Some common ways :

1. **Notify the user via a printed or displayed error message. However, need to consider these issues :**
   - **is there a human user to see the message?**
   - **can they solve the problem?**
   - ***this is not a good general solution***

2. **Notify the client object. Two main ways:**
   - **return a diagnostic value to indicate the error**
   - ***"throw an exception"***

Example : Returning a special diagnostic value (eg. using a boolean return value)

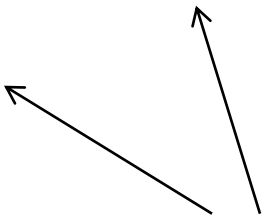
now returns a boolean (instead of void)

```
public boolean removeDetails(String key)
{
    if (keyInUse(key))
    {
        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
        return true;
    }
    else
        return false;
}
```

the client code (which called this method) can now check the ***return value*** from this method to determine if an error has occurred, and then proceed accordingly

Another Example : Returning a special diagnostic value (eg. using *null*)

```
public ContactDetails getDetails(String key)
{
    if (keyInUse(key))
        return book.get(key);
    else
        return null;

}
```



the client code can now check to see if this method failed (if it returns `null`) , or succeeded (if it returns a `ContactDetails` object properly), and then proceed accordingly

Client : responding to server errors

The client may then (via code) :

1. ***Test*** **the return value (good)**
   - **attempt recovery on error**
   - **avoid program failure**
2. ***Ignore*** **the return value (bad!)**
   - **cannot be prevented from doing this!**
   - **likely to lead to program failure later on**

Both techniques are not ideal. Another technique known as *Exception handling* is preferable.

Exception-handling principles

- *Exception handling* **is a special** *Object-Oriented* **language feature.**
- **No 'special' return value is needed.**
- **Errors *cannot* be ignored in the client.**
  - ➢**the normal flow-of-control is interrupted.**
- **Specific recovery actions are enforced/encouraged.**

Barnes & Kolling : "An *exception* is an *object* representing details of a program failure. An exception is *thrown* to indicate that a failure has occurred, when one of the methods causing the failure was called."

Exception-handling principles

- Error *discovery* is performed by the server object's method
  - this is typically known as *"throwing an exception"*
- Error *recovery* is performed by the client object's method
  - this is typically known as *"catching an exception"*

"Throwing" an exception

Two stages to throwing an exception:

1. An exception object is constructed:
   ```
   new ExceptionType("diagnostic message");
   ```
2. The exception object is then thrown:
   ```
   throw ...
   ```

A diagnostic string may be passed as a parameter, to indicate the nature of the error.

Javadoc documentation can include information about exceptions:

```
@throws ExceptionType reason
```

Example : Throwing an exception

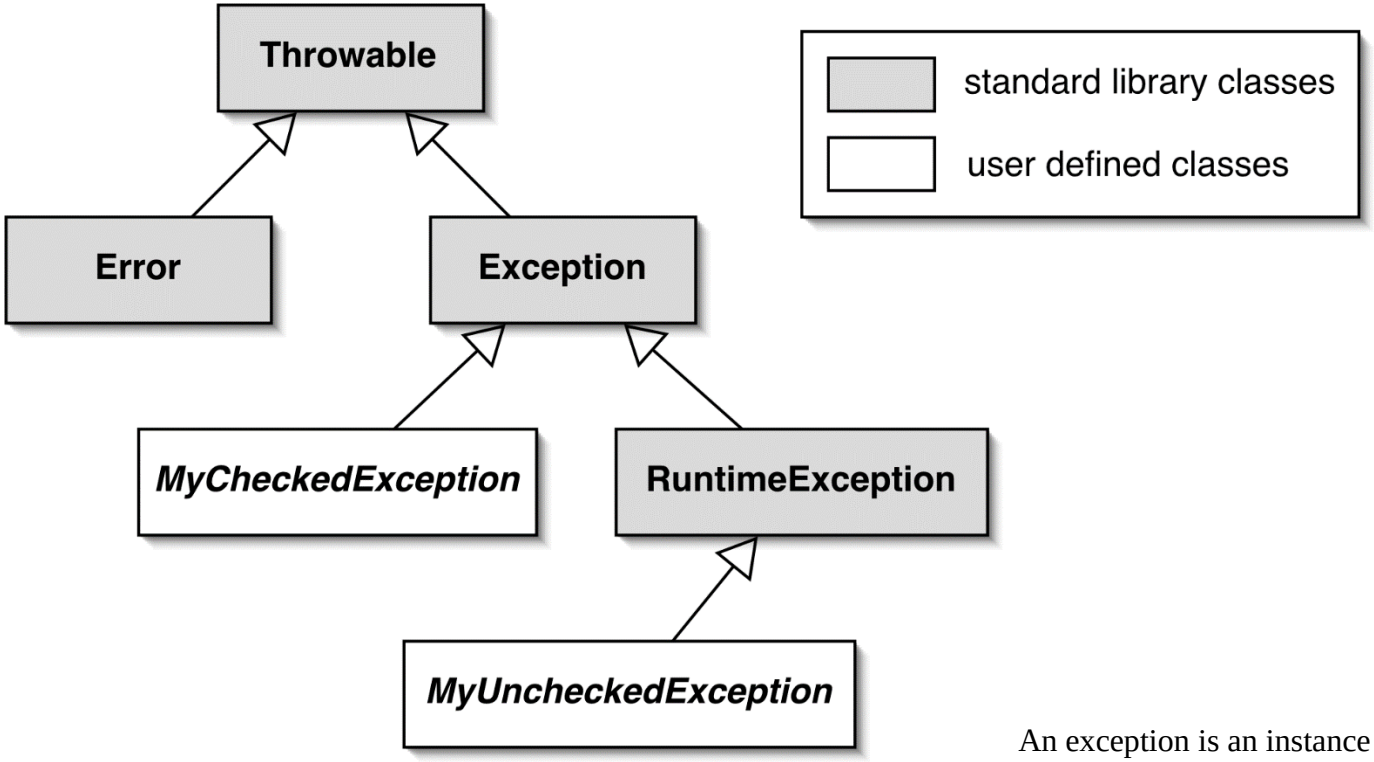
```
/**
 * Look up a name or phone number and return the
 * corresponding contact details.
 * @param key The name or number to be looked up.
 * @return The details corresponding to the key,
 *         or null if there are none matching.
 * @throws NullPointerException if the key is null.
 */
public ContactDetails getDetails(String key)
{
    if (key == null)
        throw new NullPointerException("null key in getDetails");

    return book.get(key);
}
```

Javadoc tag for throwing exception in a method

if no exception was thrown, return requested data as normal

an error has occurred – throw an exception – no data is returned

The Java exception class hierarchy



An exception is an instance of a class from a special inheritance hierarchy.

We can create new exception types by creating subclasses (more about creating subclass in a later lecture) in this hierarchy.

Java Exception categories

Checked exceptions:

- these exceptions must be checked when the program is compiled, ie. methods which throw such an exception must either handle it, or specify that the exception must be handled somewhere else – otherwise a compilation error will occur
- used for failure situations beyond the immediate control of the program, e.g. disk full, network failures, etc

Unchecked exceptions:

- these exceptions are not checked at compile-time, ie. methods which throw such an exception are not obliged to handle it
- used for situations of suspected logic errors within the program, eg. passing null parameters to a method, etc

The principles of exception throwing apply equally to both unchecked and checked exceptions. However, **Exception handling is only a requirement with checked exceptions**.

The effect of an exception

When an exception is thrown:

- the throwing (server's) method finishes prematurely
- no return value is returned
- control does not return to the client's point of call - so the client cannot simply just carry on

For example, consider the following call to `getDetails()` :

```
AddressDetails details = addresses.getDetails(null);
// any other statements following this statement will not be
// executed if an exception was thrown by getDetails
```

The program will terminate unless the exception is 'caught'.

Using Unchecked exceptions

Unchecked exceptions – compiler does not apply any checks on:

- the method in which the exception is thrown.
- the place from where the method is called.

Unchecked exceptions cause program termination if not caught.

`IllegalArgumentException` and `NullPointerException` are typical examples of unchecked exceptions.

Unchecked exceptions example : argument-checking

```
public ContactDetails getDetails(String key)
{
    if (key == null)
        throw new NullPointerException(
                              "null key in getDetails");

    if (key.trim().length() == 0)
        throw new IllegalArgumentException(
                          "Empty key passed to getDetails");

    return book.get(key);
}
```

the client code can then catch these 2 exceptions

if no exception was thrown, return requested data as normal

Unchecked exceptions example : preventing object creation

```
public ContactDetails(String name, String phone, String address)
{
    if (name == null)
        name = "";
    if (phone == null)
        phone = "";
    if (address == null)
        address = "";

    this.name = name.trim();
    this.phone = phone.trim();
    this.address = address.trim();

    if (this.name.length() == 0 && this.phone.length() == 0)
        throw new IllegalStateException(
                "Either the name or phone must not be blank.");
}
```

A call to this ***constructor*** with null/blank strings will now fail completely - ***the object will not be created at all*** – instead of an object containing some (invalid) null values in its attributes.

Exception handling

An *exception handler* is program code that protects statements in which an exception might occur – provides reporting and/or recovery code.

Exception handling may be used for unchecked and checked exceptions – however, it is a compilation requirement for checked exceptions.

Checked exceptions are meant to be caught.

The compiler ensures that their use is tightly controlled in both server and client – ie. in both the method that throws a checked exception and the caller of that method.

When used properly, failures may be recoverable.

The `throws` clause

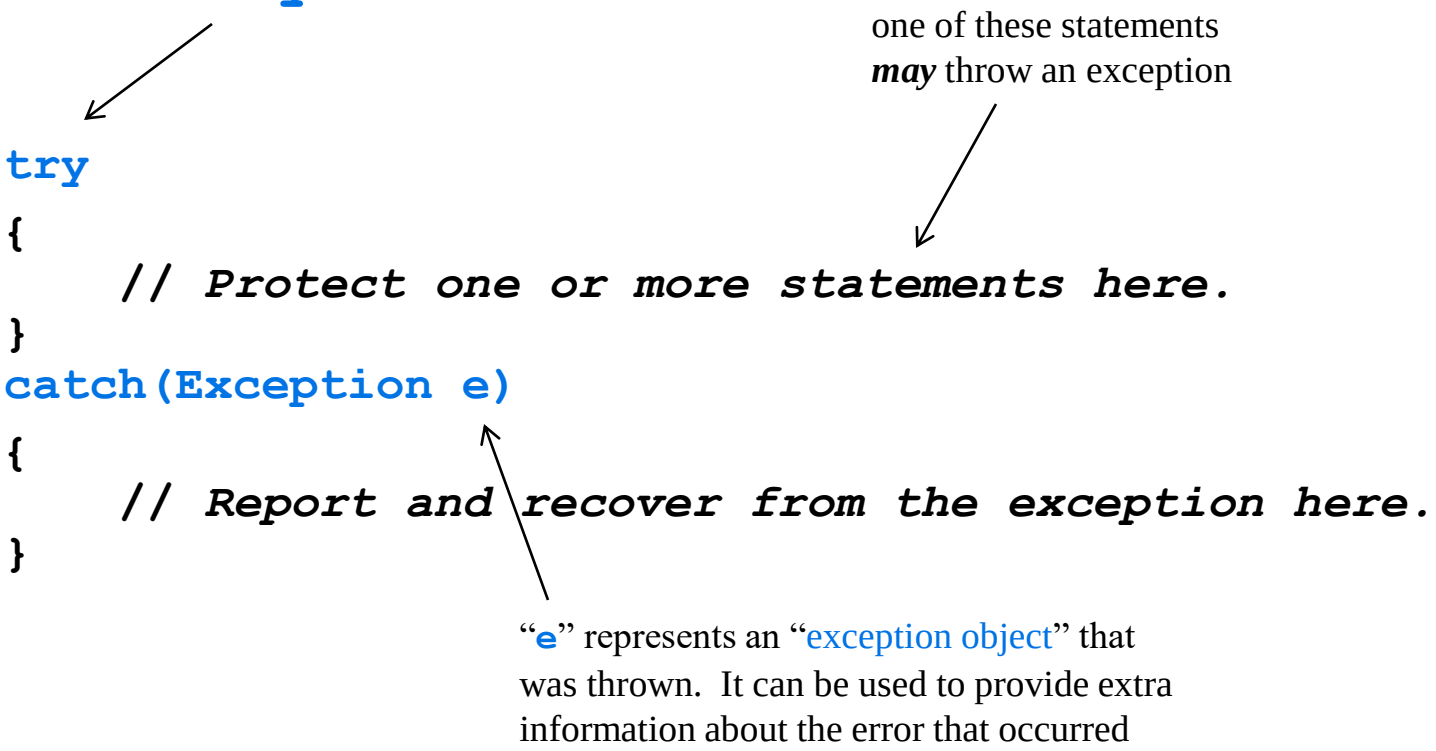
Methods throwing a *checked exception* must include a `throws` clause, added to the method header, e.g.

```
public void saveToFile(String destinationFile)
    throws IOException
{
      ......   // method body not shown
               // (an exception is thrown somewhere in here)
}
```

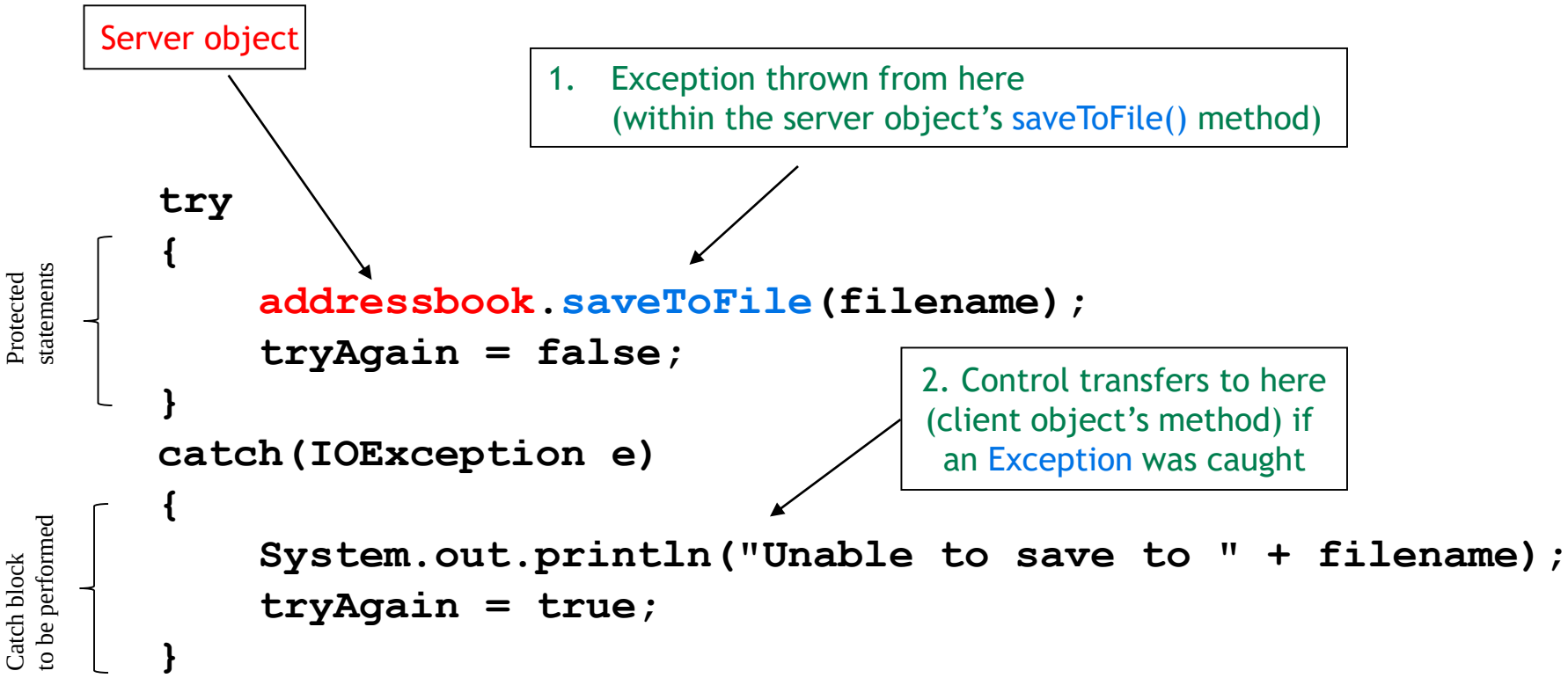
this clause indicates that this method can potentially throw an IOException error

The `try` statement

Clients catching an exception must protect the call with a `try` statement:



Example : Exception Handler - the `try` statement



Catching multiple exceptions

```
try
{
     ...
     someObject.someMethod();    // possible exception
     ...
}
catch(EOFException e)
{
     // Take action on an end-of-file exception.
     ...
}
catch(FileNotFoundException e)
{
     // Take action on a file-not-found exception.
     ...
}
```

Multi-catch – a Java 7 feature

```
try
{
     ...
     someObject.someMethod();
     ...
}
catch(EOFException | FileNotFoundException e)
{
     // Take common action appropriate to both types
     // of exceptions.
     ...
}
```

The `finally` clause

```
try
{
     Protect one or more statements here.
}
catch(Exception e)
{
     Report and recover from the exception here.
}
finally
{
     Perform any actions here regardless of whether
     or not an exception is thrown/caught above.
}
```

The `finally` *clause*

The `finally` clause is optional – except if there is no `catch` clause.

The `finally` clause contains statements that should be executed regardless of whether or not an exception is thrown by the protected statements.

A `finally` clause is executed even if a return statement is executed in the try or catch clauses.

A `finally` clause is still executed if an exception is thrown but not caught.

Error recovery summary

For successful error recovery, clients should take note of error notifications

- check return values
- should not ignore exceptions

This will usually include writing code to attempt recovery

- often requires using a loop to try again

Example : Attempting recovery

```
boolean successful = false;
int attempts = 0;

do
{
    try      // try to save the address book
    {
        addressbook.saveToFile(filename);
        successful = true;
    }
    catch(IOException e)   // save operation failed, exception caught
    {
        System.out.println("Unable to save to " + filename);
        attempts++;
        if (attempts < MAX_ATTEMPTS)
            ....  // code to ask for an alternative file name;
    }
} while (!successful && attempts < MAX_ATTEMPTS);

if (!successful)
    ....   // Too many errors, report the problem and give up;
```

Java Text input-output

Input-Output (I/O) is particularly error-prone.

- it involves interaction with the external environment. User inputs could be unpredictable.

The `java.io` package supports input-output.

`java.io.IOException` is a *checked* exception.

- so it must be caught

Readers, writers, streams

Readers and *Writers* deal with textual input.

- based around the `char` type.

Streams deal with binary data.

- based around the `byte` type.

The *address-book-io* project illustrates textual I/O.

Review : Output to the console

To print some text to the console you use:

```
System.out.println("hello there");
```

Components of this expression:

- `System` (a class)
- `out` (a *static* variable in the class `System`)
- `println(...)` a message sent to `System.out`

Writing text output to a file

Three steps involved in writing data to a file:

- Open a file
- Write to the file
- Close the file

`IOException` is a checked exception, and so **must** be caught

Failure at any point results in an `IOException`.

To write to a text file we need to represent the file as an object. The `io` package has a `PrintWriter` object which can process output to a file. `PrintWriter` has `print()` and `println()` methods – similar to `System.out.print()` and `System.out.println()`.

Writing text output to a file

To write to a text file in Java, you need to:

1. Include this statement at the top of the class:

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

2. Open the output text file by:
   a) declaring a `PrintWriter` object;
   b) instantiating this object with the name of the file.
3. Write to the file (using the `PrintWriter` object)
4. Close the file when you finish

Writing text to a file: example

```
import java.io.*;
…

public class FileOutputTest
{
    private String message;
     …
    public void writeLinesToFile(String filename)
    {
        PrintWriter outputFile = new PrintWriter(filename);
        outputFile.println("Writing a message to the file :");
        outputFile.println(message);
        outputFile.close();
    }
}
```

(1)

(2a)

(2b)

(3)

(4)

The above code will NOT compile!!! Java will insist that you catch this *checked* exception, or declare it to be thrown (hence to be handled elsewhere).

Writing text to a file: example

this declaration specifies that **writeLinesToFile** *may* ***throw*** an **IOException** - the exception can then be caught elsewhere

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

public class FileOutputTest
{
    private String message;
     ...
    public void writeLinesToFile(String filename)
                        throws IOException
    {
        PrintWriter outputFile = new PrintWriter(filename);
        outputFile.println("Writing a message to the file :");
        outputFile.println(message);
        outputFile.close();
    }
}
```

Writing text to a file: explicit exception handling

```
public void writeLinesToFile(String filename)
{
    try       // try to save to file
    {
        PrintWriter outputFile = new PrintWriter(filename);
        while (there is more text to write…)
        {
            ...
            outputFile.println(next piece of text);
            ...
        }
        outputFile.close();
    }
    catch(IOException e)      // save operation failed
    {
        // something went wrong with accessing the file
    }
}
```

Reading Text input from a file

Three steps involved in reading data from a file:

- Open a file
- Read from the file
- Close the file

`IOException` is a checked exception, and so **must** be caught

Failure at any point results in an `IOException`.

To read from a text file we need to represent the file as an object. We will use the `Scanner` class – but now we make it read from a file instead of from the keyboard.

Reading from a text file

To read a text file in Java, you need to:

1. Include these statements at the top of your class :

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

2. Open the input text file:
   a) declare a `FileReader` object
   b) instantiate this object with the name of the file
3. Instantiate a `Scanner` object to process input.
   HOWEVER: input is now from file NOT keyboard
4. Read the file (using the `FileReader` object)
5. Close the file when you finish

Example : Reading a file, line-by-line

Let's suppose that the file `Students.txt` contains the following text :

```
Peter Pan
John Tan
Kim Smith
Al Azad
```

"Students.txt"

Reading a text file: example

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

String filename = "Students.txt";
FileReader inputFile = new FileReader(filename);
Scanner parser = new Scanner(inputFile);
String name = parser.nextLine();
System.out.println(name);
inputFile.close();
```

(1) (2a) (2b) (3) (4) read one line from the file (5)

Same as

```
Scanner parser = new Scanner(new FileReader("Students.txt"));
```

The above code will **NOT** compile!!!

- What if a file is not found? Java will insist that you catch this error
- For `inputFile.close()` to succeed, `IOException` error must be caught

Reading from a text file: exception handling

```
import java.io.*;
import java.util.Scanner;
…
String filename = ("Students.txt");
try
{
    FileReader inputFile = new FileReader(filename);
    Scanner parser = new Scanner(inputFile);
    String name = parser.nextLine();
    System.out.println(name);
    inputFile.close();
}
catch(FileNotFoundException exception)
{
     System.out.println(filename + " not found");
}
 catch(IOException exception)
{
     System.out.println("Unexpected I/O error occured");
}
```

What if an exception occurs before we close the file?

Better exception handling

```
String filename = ("Students.txt");
try
{
    FileReader inputFile = new FileReader(filename);
    try
    {
        Scanner parser = new Scanner(inputFile);
        String name = parser.nextLine();
        System.out.println(name);
    }
    finally
    {
        System.out.println("Finally... closing file");
        inputFile.close();
    }
}
catch(FileNotFoundException exception)
{
   System.out.println(filename + " not found");
}
catch(IOException exception)
{
   System.out.println("Unexpected I/O exception occurs");
}
```

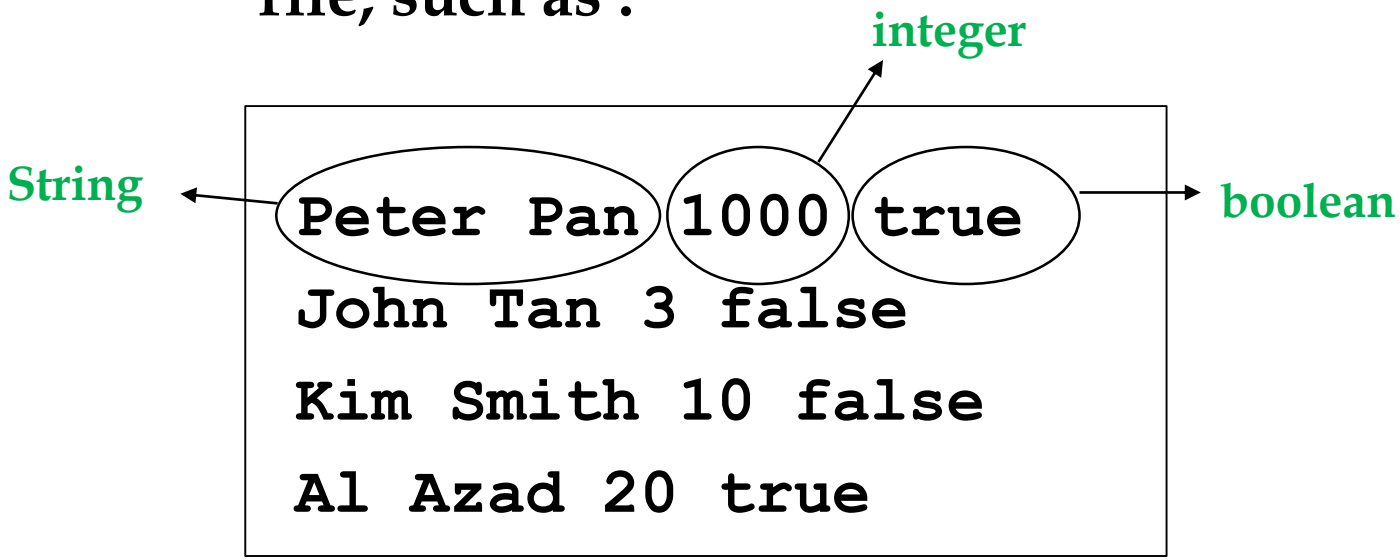
Example : Reading every line in a file using a loop

- the previous code example demonstrates how to read one single line from a file. To read lines repeatedly, we need to use a loop, such as :

```
...
while (parser.hasNextLine())
{
     name = parser.nextLine();
     System.out.println(name);
}
...
```

Reading a line with different data types

How to read more complicated data from a text file, such as :



"Students.txt"

Reading line with different data types

parser sees : "Peter Pan 1000 true"

(for line #1)

```
import java.io.*;
import java.util.Scanner;
...
String filename = "Students.txt";
FileReader inputFile = new FileReader(filename);
Scanner parser = new Scanner(inputFile);

String givenName = parser.next();          // "Peter"
String surname = parser.next();            // "Pan"
int age = parser.nextInt();                // 1000
boolean isStudent = parser.nextBoolean(); // true

System.out.println(givenName + " " + surname
            + " " + age + " " + isStudent);
```

Better I/O exception handling

Sometimes we need to consider situations where I/O exceptions occur before we can properly close a file :

- for input file :
  - not an issue, as the file's contents are not changed
- for output file :
  - need to use the `finally` clause to make sure that the file is closed properly; otherwise data (which has already been written) may be lost

Summary : Reading text file

Use for exception handling

- `try{}` → when opening/reading
- `catch{}` → catch the errors
- `Finally{}` → tidying up: close file

Reading from a text file: still use `Scanner` class, but read from a `FileReader` object instead of `System.in` (standard input/keyboard).

- to read a line containing data of different types : use the `Scanner` class's various input methods to "separate" the different data

Review

- Runtime errors arise for many reasons, eg :
  - an inappropriate client call to a server object.
  - a server unable to fulfill a request.
  - programming error in client and/or server.

- Runtime errors often lead to program failure.
- Defensive programming anticipates errors – in both client and server.
- Exceptions provide a reporting *and* recovery mechanism.