Dealing with Errors

Type	📒 Lecture
Date	@February 9, 2022
Lecture #	1
Lecture URL	https://youtu.be/5t0UPIdDYo4
Notion URL	https://21f1003586.notion.site/Dealing-with-Errors-8fc0f8b9d3a34990b6bce6edae5a4495
Week #	7

When things go wrong

- Our code could encounter many types of errors
  - User input
    - enter invalid filenames or URLs
  - Device errors
    - Printer jam
    - Network connection drops

    - Resource limitations
        - Disk full
    - Code errors
        - Invalid array index
        - Key not present in the hash table
        - Refer to a variable that is `null`
        - Divide by zero
        - `...`
- Signalling errors
    - Return an invalid value
        - `-1` at the end of the file
        - `null`
    - What if there is no obvious invalid value?

Exception Handling

- Code that generates error raises or throws an exception
- Notify the type of error
    - Information about the nature of the exception
    - Natural to structure an exception as an object
- Caller catches the exception and takes corrective action
    - Extract the information about the error from the exception object
    - Graceful interruption rather than a program crash
- Or pass the exception back up the calling chain
- Declare if a method can throw an exception
    - Compiler can check whether calling code has made a provision to handle the exception

Java's classification of errors

- All exceptions descend from class `Throwable`

    - Two branches, `Error` and `Exception`
- `Error` — relatively rare, “not the programmer’s fault”
    - Internal errors, resource limitations within Java runtime
    - No realistic corrective action possible, notify the caller and terminate gracefully
- `Exception` — two sub branches
    - `RunTimeException`, checked exceptions
- `RunTimeException` — programming errors that should have been caught by code
    - Array index out of bounds, invalid hash key, ...
- Checked exceptions
    - Typically user-defined, code assumptions violated
        - In a list of orders, quantities should be positive integers

Summary

- Exception handling — gracefully recover from errors that occur when running code
- Throw an exception — generate an object encapsulating information about the error
- Catch an exception — decode the nature of the error and take corrective action
- Java organizes exceptions in a hierarchy, by type
    - `Error` — internal errors within JVM, “not the programmer’s fault”
    - `RunTimeException` — coding errors, could have been avoided by runtime checks in code
    - Checked exceptions — user-defined, violations of assumptions mode by code
        - To contrast, `Error` and `RunTimeException` are called unchecked exceptions

Exceptions in Java

Type	📒 Lecture
Date	@February 9, 2022
Lecture #	2
Lecture URL	https://youtu.be/hsVEjkYKjpg
Notion URL	https://21f1003586.notion.site/Exceptions-in-Java-ef362d53fecf4772b79aa3aa15da872f
Week #	7

Catching and Handling Exceptions

- `try-catch`
  - Enclose the code that may generate exception in a `try` block
  - Exception handler in a `catch` block
  - It is a similar to python

```
try {
  ...
```

```
  call a function that may throw an exception
  ...
} catch(ExceptionType e) {
  ...
  examine e and handle it
  ...
}
```

- If `try` encounters an exception, rest of the code in the block is skipped
- If exception matches the type in `catch`, handler code executes
- Otherwise, uncaught exception is passed back to the code that called this code
- Top level uncaught exception — program crash

- Can catch more than one type of exception
  - Multiple `catch` blocks
- Exceptions are classes in the Java class hierarchy
  - `catch (ExceptionType e)` matches any subtype of `ExceptionType`
- Catch blocks are tried in sequence
  - Match exception type against each one in turn
- Order `catch` blocks by argument type, more specific to less specific
  - `IOException` would intercept `FileNotFoundException`

```
try {
  ...
  code that might throw exceptions
  ...
} catch(FileNotFoundException e) {
  ...
  handle the missing files
  ...
} catch(UnknownHostException e) {
  ...
  handle unknown hosts
  ...
} catch(IOException e) {
  ...
  handle all other I/O issues
  ...
}
```

Generating Exception

- When does a function generate an exception?
- `Error` — JVM runtime issue
- `RunTimeException`
  - Array index out of bounds
  - invalid hash key
  - ...
- Code calls another function that generates an exception
- Your code detects an error and generates an exception
  - `throw` a checked exception

Notify checked exceptions

- Example: You write a method `readData()`
  - Header line provides length of data
    - `Content-Length: 2048`
  - Actual data read is less than promised length
- Search Java documentation for suitable pre-defined exception
  - `EOFException`, subtype of `IOException`
  - “Signals that EOF has been reached unexpectedly during input”
- Created an object of exception type and `throw` it

  `throw new EOFException();`
- Can also pass a diagnostic message when constructing exception object

  `String errormsg = “Content-Length:” + contentlen + “, Received: ” + rcvdlen;`

  `throw new EOFException(errormsg);`

Throwing Exceptions

- How does the caller know that `readDate()` generates `EOFException`?
- Declare exceptions thrown in header

```
String readData(Scanner in) throws EOFException {
  ...
  while(...) {
    if(!in.hasNext()) {
```

```
      // EOF encountered
      if(n < len) {
        String errmsg = ...
        throw new EOFException(errmsg);
      }
      ...
    }
  }
  return s;
}
```

- Can throw multiple types of exceptions

```
String readFile(String filename) throw FileNotFoundException, EOFException {
  ...
}
```

- Can throw any subtype of declared exception type

  `String readFile(String filename) throw IOException { ... }`

  - Can throw `FileNotFoundException`, `EOFException`, both subclasses of `IOException`

- Method declares the excpetions it throws
- If you call such a method, you must handle it
- Or pass it on; your method should advertise that it throws the same exception
- Need not advertise unchecked exceptions
  - `Error, RunTimeException`
- Should not normally generate `RunTimeException`
  - Fix the error or report suitable checked exception

Customized Exceptions

- Don't want negative numbers in a `LinearList`
- Define a new class extending `Exception`

```
public class NegativeException extends Exception {
  private int error_value;
  // Negative value that generated exception

  public NegativeException(String message, int i) {
    super(message);  // Appeal to the super class
```

```
    error_value = i; // Constructor to set message
  }

  public int report_error_value() {
    return error_value;
  }
}
```

- Throw this from `LinearList`
  - Note that `add` advertises the fact that it throws a `NegativeException`

```
public class NegativeException extends Exception {
  ...
}

public class LinearList {
  ...
  public void add(int i) throws NegativeException {
    ...
    if(i < 0) {
      throw new NegativeException("Negative input", i);
    }
    ...
  }
}
```

More on catching exceptions

- We can extract information about the exception

```
try {
  ...
  call a function that may throw an exception
  ...
} catch(ExceptionType e) {
  ...
  String errormsg = e.getMessage();
  ...
}
```

- Chaining Exceptions
  - Process and throw a new exception from `catch`

```
try {
  ...
  access database
  ...
```

```
} catch(SQLException e) {
  String errormsg = "database error" + r.getMessage();
  throw new ServletException(errormsg);
  ...
}
```

- `Throwable` has additional methods to track chain of exceptions
  - `getCause()`
  - `initCause()`

```
try {
  ...
  access database
  ...
} catch(SQLException e) {
  String errormsg = "database error" + r.getMessage();
  ServletException newe = new ServletException(errormsg);
  newe.initCause(e);
  throw newe;
  ...
}
```

- Add information when you chain exceptions
- Retrieve information when you catch exception

```
try {
  ...
} catch(ServletException e) {
  ...
  Throwable original = e.getCause();
  ...
}
```

Cleaning up resources

- When exception occurs, rest of the `try` block is skipped
- May need to do some clean up (close files, deallocate resources, ...)
- Add a block labelled `finally`

```
try {
  ...
} catch(ExceptionType1 e) {
  ....
```

```
} catch(ExceptionType2 e) {
  ...
} finally {
  ...
  // Always executed, whether try terminates normally or exceptionally
  // Use it for cleanup
  ...
}
```

- Different scenarios
  - No error — 1,2,5,6
  - `IOException` in `try`, no exception in `catch` — 1,3,4,5,6
  - `IOException` in `try`, chained exception in `catch` — 1,3,5

```
FileInputStream in = new FileInputStream(...);
try {
  // 1
  code that might throw exceptions
  // 2
} catch(IOException e) {
  // 3
  show error message
  // 4
} finally {
  // 5
  in.close();
}
// 6
```

Summary

- Use try-catch to safely call functions that may generate errors
- Can throw an exception — usually checked exception
- Must advertise checked exceptions that are thrown in function header
  - Java compiler enforces that code that calls such a function handles the exception or passes it on
- Can inspect exceptions and chain them with information about original source
- Use `finally` to clean up resources that may be left open when code is interrupted by an exception

Packages

Type	📒 Lecture
Date	@February 10, 2022
Lecture #	3
Lecture URL	https://youtu.be/U_rxCLyJHgw
Notion URL	https://21f1003586.notion.site/Packages-f5284e2ccd894903b8d3b0cf07dae6c0
Week #	7

Packages

- Java has an organizational unit called `package`
- Can use `import` to use packages directly

  `import java.math.BigDecimal`
- All classes in `.../java/math`

  `import java.math.*`
- Note that `*` is not recursive, we cannot write ...

  `import java.*`

Creating and naming packages

- We can create our own hierarchy of packages
- Naming convention is similar to Internet domain name, but in reverse
  - Internet domain: `onlinedegree.iitm.ac.in`
  - Package name: `in.ac.iitm.onlinedegree`
- Add a package header to include a class in a package

  ```
  package in.ac.iitm.onlinedegree;
  public class Employee { ... }
  ```

- By default, all classes in a directory belong to the same anonymous package

More about visibility

- We have seen modifiers `public` and `private`
- If we omit these, the default visibility is public within the package
  - This applies to both methods and variables
- Can also restrict visibility w.r.t. inheritance hierarchy
  - `protected` means visible within the sub-tree, so all sub-classes
  - Normally, a sub-class cannot expand visibility of a function
  - However, `protected` can be made `public`

Assertions

Type	📒 Lecture
Date	@February 10, 2022
Lecture #	4
Lecture URL	https://youtu.be/hKl3tCQdfTI
Notion URL	https://21f1003586.notion.site/Assertions-ef6b535c2bba4aaf8ff671f548df705a
Week #	7

Documenting and checking assumptions

- Functions may have constraints on the parameters

```
public static double myfn(double x) {
  // Assume x >= 0
  ...
}
```

- We could check the condition and throw an exception

```
public static double myfn(double x) throws IllegalArgumentException {
  // Assume x >= 0
  if(x < 0) {
    throw new IllegalArgumentException("x < 0");
  }
}
```

- What if `myfn` is only used internally by our own code
  - Flag errors during development, debugging
  - But diagnostic code should not trigger at runtime
  - Performance, and other considerations
- Instead, “assert” the property you assume to hold

```
public static double myfn(double x) {
  asset x >= 0;
}
```

Assertions

- If assertion fails, the code throws `AssertionError`
- This should not be caught
  - Abort and print the diagnostic information (stack trace)
- We can also provide additional information to be printed with the diagnostic message

```
public static double myfn(double x) {
  asset x >= 0 : x;
}
```

Enabling and Disabling assertions

- Assertions are enabled or disabled at runtime
  - Does not require re-compilation
- Use the following flag to run with assertions enabled

  `java -enableassertions MyCode`
- We can use `-ea` as abbreviation for `-enableassertions`

- We can selectively turn on assertions for a class

  ```
  java -ea:Myclass MyCode
  ```

- or a package

  ```
  java -ea:in.ac.iitm.onlinedegree MyCode
  ```

- Similarly, disable assertions globally or selectively

  ```
  java -disableassertions MyCode
  ```

  ```
  java -da:MyClass MyCode
  ```

- We can combine the two

  ```
  java -ea in.ac.iitm.onlinedegree -da:MyClass MyCode
  ```

- Separate switch to enable assertions for system class

  ```
  java -enablesystemassertions MyCode
  ```

  ```
  java -esa MyCode
  ```

Summary

- Assertion checks are supposed to flag fatal unrecoverable errors
  - We don't `catch` them
- If we need to flag the error and take corrective actions, we instead use exceptions
- It is turned on only during development and testing
  - It is not checked at runtime after deployment

Logging

Type	📒 Lecture
Date	@February 10, 2022
Lecture #	5
Lecture URL	https://youtu.be/fkGfOzVC8zI
Notion URL	https://21f1003586.notion.site/Logging-dd33ff3f3c5f4a27af0ca313258e1cd2
Week #	7

Diagnostic messages

- It is rather typical to generate messages within code for diagnosis
- Naive approach is to use the print statements
  - The need to add/subtract as we go along
  - Enable and Disable explicitly
- Instead log diagnostic messages separately
  - Logs are arranged hierarchically — choose the level of logging needed

    - Can be displayed in different formats
    - Logs can be processed by other code — handlers
        - Can filter out uninteresting entries
    - Logging controlled by a config file

Logging

- Simplest: call `info()` method of global logger:

  ```
  Logger.getGlobal().info("Edit->Copy menu item selected");
  ```

- This prints the following

  ```
  January 10, 2022 10:12:15 PM LoggingImageViewer myFunction
  INFO: Edit->Copy menu item selected
  ```

- Suppress logging by executing the following code

  ```
  Logging.getGlobal().setLevel(Level.OFF);
  ```

- Create a custom logger

  ```
  private static final Logger myLogger = Logger.getLogger("in.ac.iitm.onlinedegree");
  ```

    - Logger names are hierarchical, like package names
    - Setting a property for `in.ac.iitm` automatically sets it for `in.ac.iitm.onlinedegree`

Logging Levels

- Seven logging levels
    - SEVERE
    - WARNING
    - INFO
    - CONFIG
    - FINE
    - FINER
    - FINEST
- By default, first three levels are logged
- Can set a different level

  ```
  logger.setLevel(Level.FINE);
  ```

- Turn on all the levels, or turn off all logging

  ```
  logger.setLevel(Level.ALL);
  ```

  ```
  logger.setLevel(Level.OFF);
  ```

- Can also change logging properties through a config file

Summary

- Logging gives us more flexibility and control over tracking diagnostic messages than simple print statments
- We can define a hierarchy of loggers
- Seven level of messages — control which levels are printed
- Control logging from within code or through enxternal config file