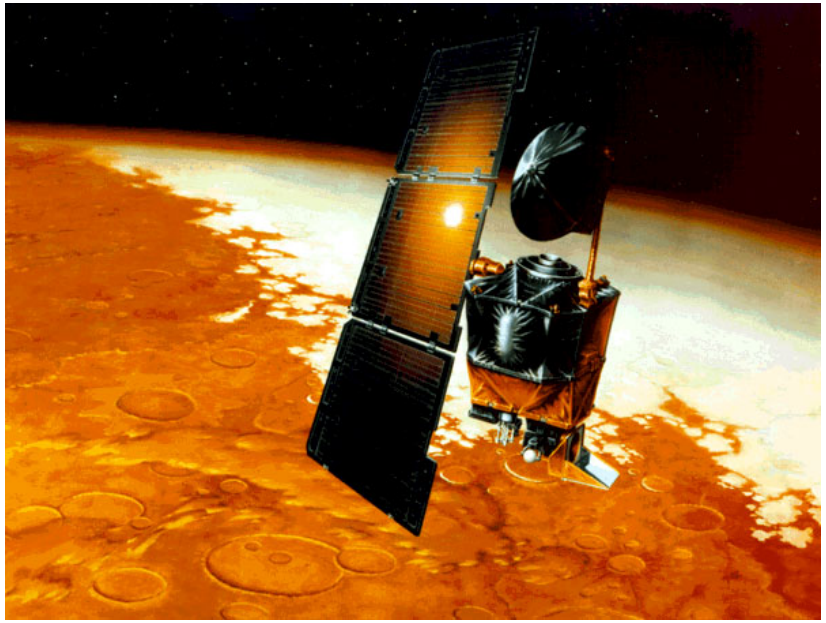
# Scientific Programming Using Object-Oriented Languages
## Module 3: Exceptions

**Aims of Module 3:**

- Understand how to structure programs to handle errors.
- Be able to:
  - implement `try`, `catch` and `finally` structures.
  - throw Exceptions in your code.
  - declare Exceptions in your method/class definitions.

# Famous Software Failures

## Mars Climate Orbiter, 1999



- The software to control the thrusters operated in Imperial units… the rest of the software operated in metric units.
- Caused the loss of the $327.6 million project as the thrusters were applying 4.5 times too much thrust.
- https://blog.bugsnag.com/bug-day-mars-climate-orbiter/

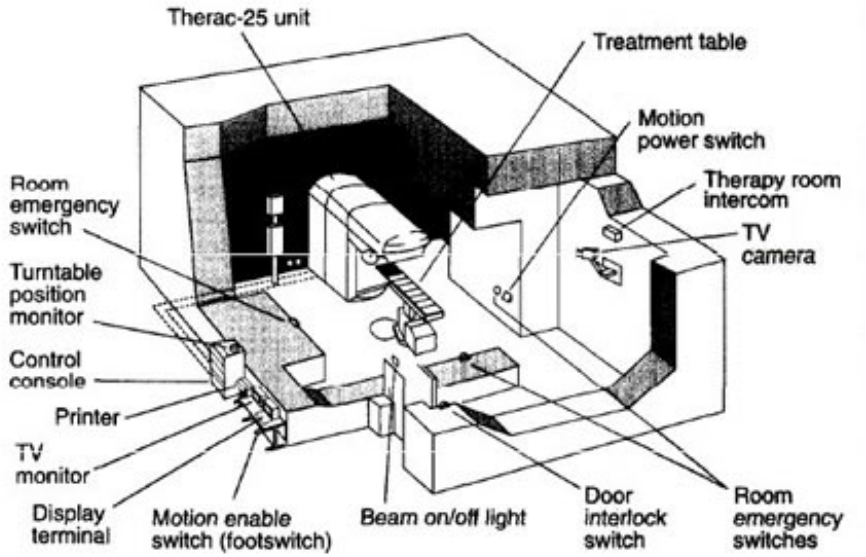
# Famous Software Failures

## Ariane-5 Explosion, 1996

- Error in navigation code: 64-bit floating point number larger than 32,767 “downcast” into a 16-bit integer.
- The guidance package was taken from Ariane-4, but the larger variable range required by Ariane-5 exposed latent bug.
- https://blog.bugsnag.com/bug-day-ariane-5-disaster/

# Famous Software Failures

## Therac-25 Radiation Therapy, 1980's




- Hardware interlocks replaced with software interlocks.
- Subtle bug allowed for high power electron beam to be activated without target/shielding in place when operator changed modes quickly.
- Several people killed by radiation poisoning.
- https://blog.bugsnag.com/bug-day-race-condition-therac-25/

# Famous Software Failures

## Boeing 787 Dreamliner, 2015


- Software bug in generator control units would cause them to fall into failsafe mode if kept continuously powered on for 248 days.
- Caused by 32-bit unsigned integer overflow bug: 248.55 days = 2^31 100ths of a second.
- If the four main generator control units (associated with the engine-mounted generators) were powered up at the same time, after 248 days of continuous power, all four GCUs will go into failsafe mode at the same time, resulting in a loss of all AC electrical power regardless of flight phase…
- Boeing's solution?  Turn every aircraft off and back on again…
- http://www.theguardian.com/business/2015/may/01/us-aviation-authority-boeing-787-dreamliner-bug-could-cause-loss-of-control

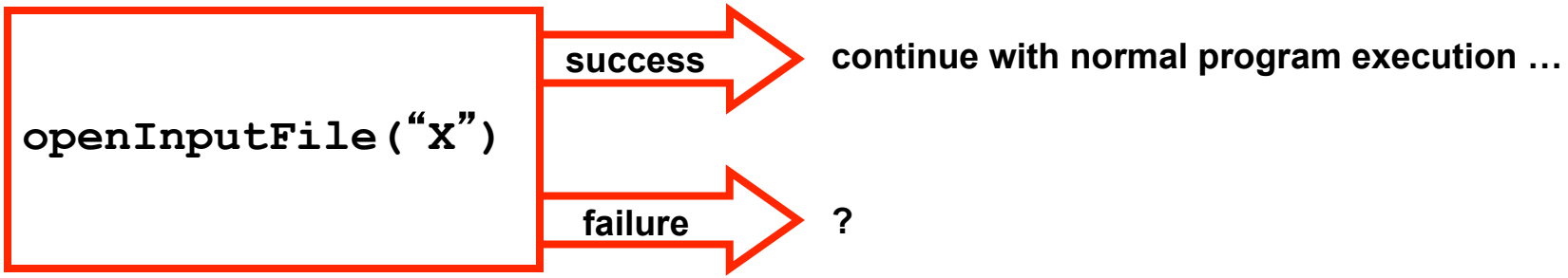
# Bugs

- All complex code contains bugs.
- There are many engineering strategies for reducing the level of bugs : failure analysis, component testing etc.

- Implementing proper exception handling will *not* eliminate bugs from your code, especially these kinds of subtle bugs.
- But it should make you think more carefully about the various unexpected circumstances that can arise during the execution of your programs, and how to deal with them.
- This helps to make our programs more *robust*.

# Exceptions

- Java's built-in exception handling framework provides a way of dealing with unexpected circumstances in a controlled way.
- A simple example of a common failure mode :




- The response to failure depends on the application. Actions might include :
  - Looking for an alternative input file or source of data.
  - Assuming certain default values for the data that were to be read from the file and proceeding as normal.
  - Terminating program execution with an informative error message.
  - … etc.

# Throwing An Exception

```
public static int process(int j) throws Exception {

    if (j==13) {
        throw new Exception("Unlucky number: "+j);
    }

    return 2*j;

}
```

- Method needs to declare that it can throw an exception with the `throws Exception` declaration.
- Within that method, the point at which the exception needs to be thrown is specified with `throw new Exception`.
- The `Exception` object is instantiated with text containing the relevant error message: `Exception("Unlucky number: "+j)`. **Make sure it's descriptive enough!** `"Error"` doesn't count…

# Catching the Exception

```
for (int k=10; k<20; k++) {
    try {
        // Code that can throw an exception
        int n = process(k);
        System.out.println("in: "+k+"  out: "+n);
    }
    catch (Exception e) {
        // Code that handles the exception
        System.out.println(e)
    }
    finally (Exception e) {
        // Code that executes regardless of whether an
        // exception is thrown or not
        System.out.println("done "+k);
    }
}
```

- For any methods that throw an exception, they need to be enclosed in a **`try-catch`** block.
- If there is code that needs to execute regardless of whether an exception is thrown, include it in the optional **`finally`** block.

# Multiple Exceptions (1)

```
try {
    // Read data from text files
    String urlData = getDataFromURL("http://webaddress.net");
    String fileData = getDataFromFile("textFile.txt");
}
catch (Exception e) {
    // Code that handles the exception
    System.out.println("Could not read some data");
}
```

- Here, we have two methods — `getDataFromURL()` and `getDataFromFile()` — both of which throw exceptions.
- With the **`catch`** block that we've written, how do we know which method has thrown the exception?

# Multiple Exceptions (2)

```
try {
    // Read data from URL
    String urlData = getDataFromURL("http://webaddress.net");
}
catch (Exception e) {
    // Handle the URL exception
    System.out.println("Could not read data from URL");
}
try {
    // Read data from text file
    String fileData = getDataFromFile("textFile.txt");
}
catch (Exception e) {
    // Handle the File exception
    System.out.println("Could not read data from File");
}
```

- Better to have two discrete **`try-catch`** blocks, since the code is not interdependent.
- Also demonstrates the importance of having **clear error messages**: that way, you are at less risk of giving ambiguous information.

# Interdependent Code (1)

```
String urlData;
try {
    // Read data from URL
    urlData = getDataFromURL("http://webaddress.net");
}
catch (Exception e) {
    // Handle the URL exception
    System.out.println("Could not read data from URL");
}
try {
    // Process data
    int numOfInstances = processData(urlData);
}
catch (Exception e) {
    // Handle the data processing exception
    System.out.println("Could not process data");
}
```

- What happens if `getDataFromURL()` throws an exception? There is no point trying to process the data!
- In this case we would be better including both methods in a single **`try`** block, since they are interdependent…

# Interdependent Code (2)

```
String urlData;
try {
    // Read data from URL
    urlData = getDataFromURL("http://webaddress.net");
    // Process data
    int numOfInstances = processData(urlData);
}
catch (Exception e) {
    // Handle the exceptions
    System.out.println("Could not read data from URL");
    System.out.println(e);
}
```

- In this case, we don't want `processData()` to run if `getDataFromURL()` couldn't read anything from the URL.
- Including both methods in a single **`try`** block means that the code won't continue to execute, but…
- Now you **have** to make sure your exception error message is clear enough to tell what threw the exception.
- **Nesting `try-catch` blocks is not a solution!**

# Exceptions

- Exception handling forces you to consider:

**HOW** to handle the error

**WHERE** to handle the error

- See the Module-3 notes and other documentation for technical details.
- Start working on Module-3 exercises if you haven't already done so.