ADVANCED PROGRAMING PRACTICES

Exception handling

Exception handling: introduction

- Programs are meant to work correctly **within their specifications**.

- However, a program might be faced with unforeseen circumstances that are outside of its specifications.

- Unforeseen situations may come:

  - Externally from the environment of the program

    - When a user or software client tries to use the software outside of its specified usage characteristics.

    - When the program tries to use another piece of software and is faced with unforeseen behavior.

  - Internally from its own execution

    - When the program misbehaves due to an internal logical error and/or being in an inconsistent state.

- A robust program should be able to **handle** all kinds of circumstances, foreseen or unforeseen, whether they are coming from the exterior or are a result of its own faults.

Exception handling: error handling

- Exception handling is a mechanism that allows two separately developed program components to **communicate** when a program anomaly is encountered during the execution of the program.

- Such communication upon erroneous behavior has been long part of programming practice in the form of **error codes** and **error handling**.

- In error handling, functions set or return special **error codes** in case of malfunction and finish execution normally.

- It is then assumed that any function that might be affected will use the error code and react by **handling the error** i.e. to continue normal execution despite the error.

Exception handling: error handling

- Error handling code can create complexity in simple programs.

- There can be many different error codes, some error states can even be combinations of more than one error.

- The error code is a value to be returned. What if the functions also needs to return a value?

```
errorCodeType readFile (){
    errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

Exception handling: introduction

- However, error handling introduces **confusion** as it does not enable to separate normal behavior from error-handling behavior.

- To be more structured, functions should be first programmed according to the specifications of their **normal behavior**, and clearly separate code should be provided for **abnormal cases**, i.e. cases outside of the function's specifications of normal behavior.

```
returnType readFile (){
    part of code that cannot fail;
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
    other part of code that cannot fail;
}
```

Exception handling: introduction

- The first programming language to provide the early concept of exception handling was LISP in the early 1960. PL/1 later extended the concept in the early 1970s.

- An **exception** is a data structure that is generated when a special erroneous condition is met, that contains information about the nature and context of this erroneous condition.

- Exceptions are processed by an **exception handling mechanism** that only takes effect when an exception has been identified.

- The exception handling mechanism will then **take over** the normal execution mechanism until the exception is resolved.

- If the exception can be properly resolved, normal execution is resumed.

- If the exception cannot be resolved, the program execution is terminated.

Exception handling: try-throw-catch-finally

- Syntactically, handling exceptions in Java is made through of the `try-throw-catch` keyword trio, which was borrowed from C++.

- The `try` block contains a part of the code for the normal execution of the program.

- It is called a `try` block because it *tries* to execute the normal execution behavior, but where something is likely to be subject to exceptionally erroneous behavior, <u>and</u> we want to handle the erroneous case <u>locally</u>.

```
// Code for which thrown exceptions
// are not handled locally
try {
  // Code that may throw an exception
  // that may be handled locally
}
// Code for which thrown exceptions
// are not handled locally
```

Exception handling: try-throw-catch-finally

- Whenever a piece of code identifies an exceptionally wrong situation, it can create an exception and trigger the exception handling mechanism by using a `throw` statement:

```
throw new ExceptionClassName(/*PossiblySomeArguments*/);
```

- When an exception is thrown, the normal execution of the surrounding `try` block (if any) is stopped and the **exception handling mechanism takes over** the execution of the program.

- Normally, the flow of control is transferred by the exception handling mechanism to another portion of code known as a `catch` block.

- The value thrown is the argument to the `throw` operator, which must be an instance of the `Throwable` class (or in most cases of the `Exception` class) or a subclass of it.

Exception handling: try-throw-catch-finally

- A `throw` statement is similar to a method call, as it receives a "parameter" (the thrown object) and "branches" to a catch block:

```
throw new ClassName("Some Descriptive String");
```

- In the above example, an object of class `ClassName` (which must be a subclass of `Exception`) is created using a `String` as its argument (which is usual for exception classes) and used as an exception object.

- The throw statement has the effect of temporarily halting the normal execution of the program and handing it over to the **exception handling mechanism**.

Exception handling: try-throw-catch-finally

- When an exception is thrown and the exception handling mechanism takes over, it tries to find a corresponding `catch` block to handle the exception.
  - A `catch` block has one and only one parameter.
  - The exception object thrown is passed to the `catch` block using a similar mechanism as for a function call's parameter passing.
  - The type of the catch block's exception object parameter determines what kind of exception a `catch` block is meant to handle.

- The execution of the `catch` block is called *catching the exception*, or *handling the exception.*

```
catch(Exception e) {
    // ExceptionHandlingCode
}
```

Exception handling: try-throw-catch-finally

- Any `catch` block is attached to one specific `try` block.
- A `catch` block is an **exception handler** that is meant to handle some exception thrown in the `try` block it is attached to.
- The type of exception it is meant to handle is specified by its parameter type.
- A `catch` block can catch any exception of the type it specifies, or any of its subtypes.
- A single `try` block can be attached to as many `catch` blocks as there are different kinds of exceptions potentially thrown in the `try` block's code.
- The exception handling mechanism goes **sequentially** over the `catch` block types and branches upon the **first one** that matches the thrown exception type.

```
try {
    // Code that potentially throws some exception(s) to be
    // handled locally in one of the following catch blocks
}
catch (ExceptionType1 e){
    // Exception handling code for ExceptionType1
}
catch (ExceptionType2 e){
    // Exception handling code for ExceptionType1
}
```

Exception handling: try-throw-catch-finally

- Frequently, some code needs to be executed whether or not an exception was thrown in the `try` block, such as releasing a resource that was allocated in the `try` block before the exception is thrown. For this, an optional `finally` block can be used.

```java
try {
    // Code that potentially throws some exception(s)
    // to be handled locally in one of the following catch blocks
}
catch (ExceptionType1 e){
    // Exception handling code for ExceptionType1
}
finally {
    // Code that is executed whether or not
    // an exception was thrown in the try block
}
```

- Java SE 7.0 introduced the **try-with-resources block**, which allows to declare an object as a resource managed during the execution of a `try` block

```java
try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
    String s = br.readLine();
}
```

- where a class implementing the `java.lang.AutoCloseable` interface is opened as the `try` is declared, and will be automatically closed regardless of whether the `try` will complete normally or abruptly.

Exception handling: try-throw-catch-finally

- When a try block is executed, three things can happen:

1. **No exception is thrown in the try block**
   - The code in the try block is executed to the end of the block.
   - The catch blocks are skipped.
   - The finally block is executed.
   - The execution continues with the code placed after the finally block.

2. **An exception is thrown in the try block and caught in a local catch block**
   - The rest of the code in the try block is skipped.
   - Control is transferred to a following catch block.
   - The thrown object is passed to the catch block using parameter passing.
   - The code in the catch block is executed.
   - The finally block is executed.
   - Normal execution resumes using the code that follows the finally block.

3. **An exception is thrown in the try block and there is no corresponding local catch block to handle the exception**
   - The rest of the code in the try block is skipped.
   - As there I no suitable local catch block, the exception cannot be handled locally in the function.
   - The finally block is executed.
   - The function throws the exception to its calling function.
   - The calling function either catches the exception using a catch block, or itself throws the exception.
   - If all functions fail to catch the exception, it will eventually be thrown all the way to the main function.
   - If the main function cannot catch the exception, the program ends, itself throwing the exception.

Exception handling: stack unwinding

- Once the exception handling mechanism takes control as a `throw` statement is executed, control moves from the `throw` statement to the first `catch` statement that can handle the thrown type.

- Once the control goes to the `catch` handler, a process known as **stack unwinding** is used to **release all the local variables** that were used between the execution site where the exception was thrown and the execution site where the exception is caught.

- The physical deallocation of released objects is eventually done by the garbage collector.

- However, other kinds of allocated resources are not managed by the garbage collector.

- If, for example a file is opened, then the read/write operation fails by throwing an exception, the file resource release must appropriately be managed.

```
BufferedWriter file = new BufferedWriter(new FileWriter("DummyOutput.txt"));
file.write(pd.dummyName + ", " + pd.age);
// never closed if write() throws an exception!
// solution is to catch here, or else the file resource is leaked.
file.close();
```

Exception handling: stack unwinding

```java
public class StackUnwindingDemo {

  public static char whichToFail = ' ';

  public static void main(String[] args) throws IOException{
    System.out.print("Enter which function you want to fail (m, f, g, x) : ");
    whichToFail = (char) System.in.read();

    MyClass mc1 = new MyClass("mc1");
    try {
      MyClass mc6 = new MyClass("mc6");
      if (whichToFail == 'm') throw new Exception();
      f();
      MyClass mc7 = new MyClass("mc7");
    }
    catch (Exception e) {
      MyClass mc8 = new MyClass("mc8");
    }
    // resume here after handling the exception
    MyClass mc9 = new MyClass("mc9");
  }

  public static void f() throws Exception {
    System.out.println("Entering function f()");
    MyClass mc2 = new MyClass("mc2");
    if (whichToFail == 'f') throw new Exception();
    g();
    MyClass mc3 = new MyClass("mc3");
    System.out.println("Exiting function f()");
  }

  public static void g() throws Exception {
    System.out.println("Entering function g()");
    MyClass mc4 = new MyClass("mc4");
    if (whichToFail == 'g') throw new Exception();
    MyClass mc5 = new MyClass("mc5");
    System.out.println("Exiting function g()");
  }
}
```

```
Enter which function you want to fail (m, f, g, x) : m
in MyClass constructor: mc1
in MyClass constructor: mc6
in MyClass constructor: mc8
in MyClass constructor: mc9
```

```
Enter which function you want to fail (m, f, g, x) : f
in MyClass constructor: mc1
in MyClass constructor: mc6
Entering function f()
in MyClass constructor: mc2
in MyClass constructor: mc8
in MyClass constructor: mc9
```

```
Enter which function you want to fail (m, f, g, x) : g
in MyClass constructor: mc1
in MyClass constructor: mc6
Entering function f()
in MyClass constructor: mc2
Entering function g()
in MyClass constructor: mc4
in MyClass constructor: mc8
in MyClass constructor: mc9
```

```
Enter which function you want to fail (m, f, g, x) : x
in MyClass constructor: mc1
in MyClass constructor: mc6
Entering function f()
in MyClass constructor: mc2
Entering function g()
in MyClass constructor: mc4
in MyClass constructor: mc5
Exiting function g()
in MyClass constructor: mc3
Exiting function f()
in MyClass constructor: mc7
in MyClass constructor: mc9
```

Exception handling: stack unwinding

- This is even more problematic if exceptions are thrown in constructors.

```java
public class ConstructorExceptionsDriver {
  public static void main(String[] args) {
    char classToThrow = 'Z';
    A a = null;
    Scanner keyboard = new Scanner(System.in);
    while (classToThrow != 'X') {
      System.out.println("class to throw : ");
      classToThrow = keyboard.next().charAt(0);
      try {
        A.classToThrow = classToThrow;
        a = new A();
      } catch (Exception e) {
        System.out.println("catching Exception");
      }
    }
    keyboard.close();
  }
}
```

```java
public class A extends B {
  static char classToThrow;
  C mC = new C(classToThrow);
  D mD = new D(classToThrow);
  A() throws Exception {
    super(classToThrow);
    System.out.println("in ctor A(), throwing: " + classToThrow);
    if (classToThrow == 'A')
      throw new Exception();
  }
}
```

```java
public class B {
  B(char classToThrow) throws Exception {
    System.out.println("in ctor B(), throwing: " + classToThrow);
    if (classToThrow == 'B')
      throw new Exception();
  }
}
```

```java
public class C {
  C(char classToThrow) throws Exception {
    System.out.println("in ctor C(), throwing: " + classToThrow);
    if (classToThrow == 'C')
      throw new Exception();
  }
}
```

```java
public class D {
  D(char classToThrow) throws Exception{
    System.out.println("in ctor D(), throwing: " + classToThrow);
    if (classToThrow == 'D')
      throw new Exception();
  }
}
```

Exception handling: stack unwinding

- An object is only created if its constructor fully resolves. If it successfully calls other constructors before throwing, some resources allocated during the execution of the called constructors will be **leaked**.

```
Enter the class name to throw an exception (A,B,C,D or N for None or X for eXit):
B
in Constructor B() with class to throw: B
catching Exception

Enter the class name to throw an exception (A,B,C,D or N for None or X for eXit):
D
in Constructor B() with class to throw: D
in Constructor C() with class to throw: D
in Constructor D() with class to throw: D
catching Exception

Enter the class name to throw an exception (A,B,C,D or N for None or X for eXit):
A
in Constructor B() with class to throw: A
in Constructor C() with class to throw: A
in Constructor D() with class to throw: A
in Constructor A() with class to throw: A
catching Exception

Enter the class name to throw an exception (A,B,C,D or N for None or X for eXit):
N
in Constructor B() with class to throw: N
in Constructor C() with class to throw: N
in Constructor D() with class to throw: N
in Constructor A() with class to throw: N
```

Exceptions: handle or declare rule

- Exception specifications: `throws` clause

```
public void readFile(String file) throws FileNotFoundException
```

- This `throws` clause signifies that the function `readFile` can only throw exceptions of type `FileNotFoundException`.

- This informs any function calling `readFile` that it may throw this specific kind of exception, and that it will not handle it internally.

- It forces any calling function to either catch this exception, or itself declare that it may throw it using its own `throws` clause.

- A method can throw any subclass of the exception type mentioned in its `throws` clause.

- The `throws` clause is a very useful annotation for the programmer when programing with exceptions.

- Any function that contains code that may throw exceptions either needs to make the call to the throwing function in a `try` block and catch the exception, or itself have a `throws` clause that specifies that it may throw.

- This is referred to as the "*handle or declare rule*".

Checked and unchecked exceptions

- The handle or declare rule applies only to **checked exceptions**, which are of the `Exception` type that are not of the `RunTimeException` type.

- There is a debate about the relevance of the exception checking mechanism and the handle or declare rule.

- As a consequence, some programmers are using only unchecked exception types to avoid the handle or declare rule.

- An `Error` is a subclass of `Throwable` that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions.

- `RuntimeException` is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.
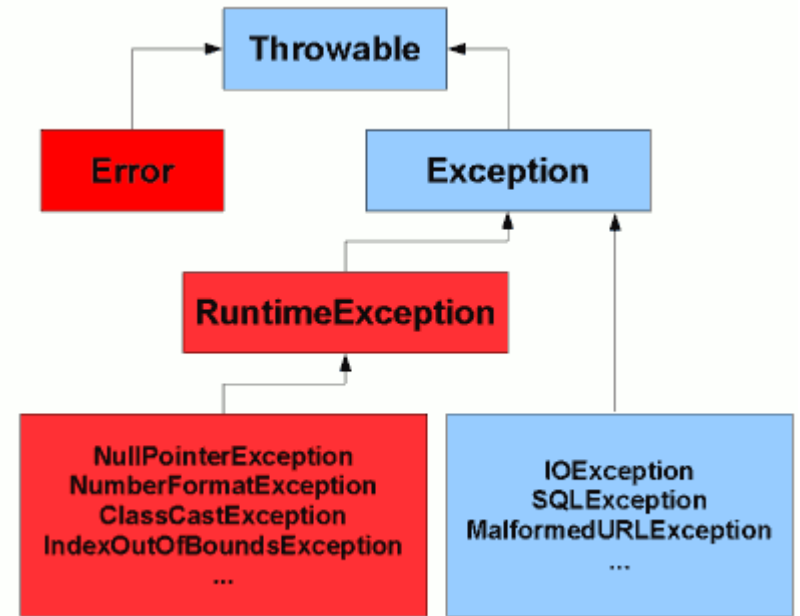
Custom exception classes

- Any instance of the Throwable class can be thrown as an exception.

- Custom exception classes can be derived from the standard Exception class.

- The only particularity of this class is that it offers a members function getMessage() to return a String that allows the programmer to store a descriptive message on the nature of the circumstances that led to the error.

- This string is passed to the constructor upon creation of the exception.

```java
public class HardwareException extends Exception {
  public
    HardwareException() {
      exc_time = getTime();
    }
    String getTime() {
      DateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
      return format.format(new Date());
    }
    String exc_time;
}
```

```java
public class ActuatorException extends HardwareException {
  public
    ActuatorException(HardwareState new_s) {
      super();
      hw_state = new_s;
    }
    String toString() {
      return this.exc_time+":"+this.getClass()+":"+this.hw_state.name();
    }
    HardwareState hw_state;
}
```

```java
public class StuckValveException extends ActuatorException {
  public
    StuckValveException(HardwareState new_s) {
      super(new_s);
    }
}
```

Custom exception classes

- An exception class can be made to store any other useful information, e.g. the time where the exception was thrown.

- The `getMessage()` function of the `Exception` class can also be overridden to provide information otherwise than assuming the programmer to provide a `String` upon construction.

- As for any other class, one can also override the `toString()` function to enable output to console, files, or other streams.

```java
public class SensorException extends HardwareException {
  public
    String toString() {
      return this.exc_time+":"+this.getClass()+":"+valueRead;
    }
    SensorException(int new_v) {
      super();
      valueRead = new_v;
    }
    int valueRead;
}
```
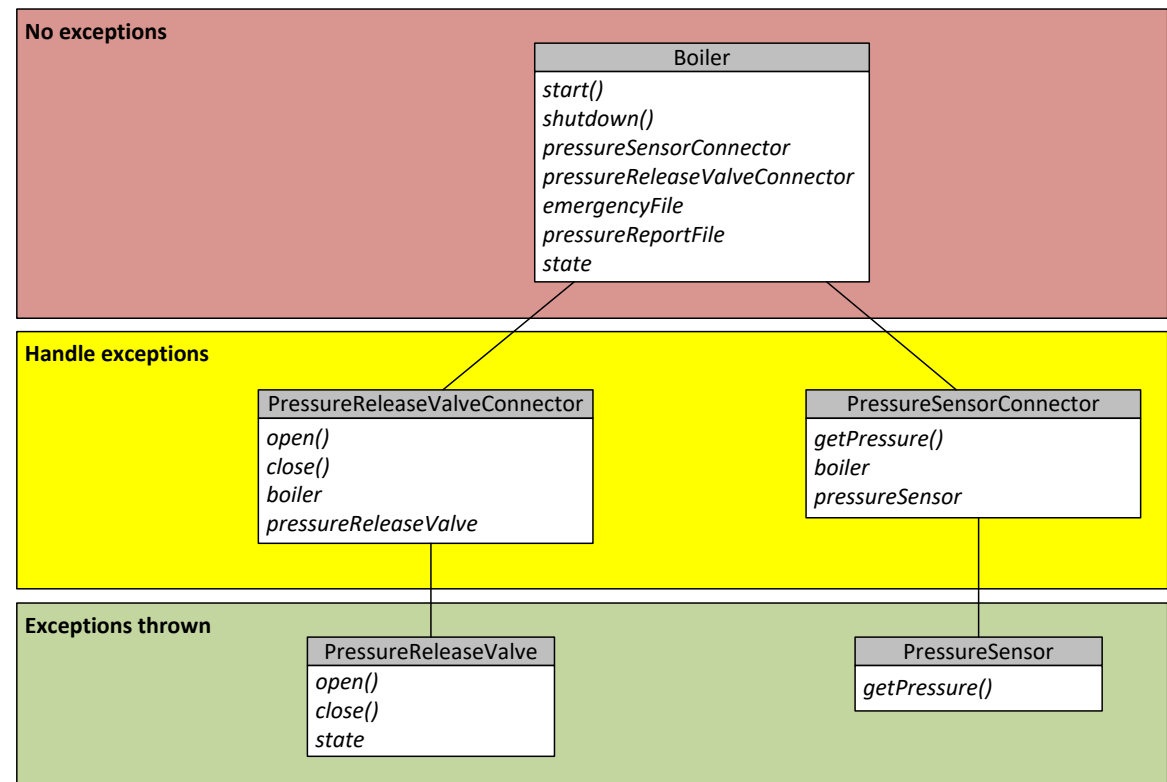
```java
public class HardwareException extends Exception {
  public
    HardwareException() {
      exc_time = getTime();
    }
    String getTime() {
      DateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
      return format.format(new Date());
    }
    String exc_time;
}
```

```java
public class OverloadPressureException extends SensorException {
  public
    OverloadPressureException(int new_v) {
      super(new_v);
    }
}
```

```java
public class NegativePressureException extends SensorException {
  public
    NegativePressureException(int new_v) {
      super(new_v);
    }
}
```

Exception handling: example

- Industrial boiler controlled by software.

- Connected to a pressure sensor and a pressure release valve.

- Keeps the pressure within an acceptable range.

- If the sensor is misbehaving, it shuts down the boiler by opening the valve.

- If the valve is stuck, it calls an emergency.

- Keeps a log of the pressure readings, as well as another log for operational events.

- Hardware drivers can throw exceptions.

- **Requirement:** For security, the boiler controller should be shielded from those exceptions.

- Thus, an exception handling layer is added.

No exceptions

Boiler
start()
shutdown()
pressureSensorConnector
pressureReleaseValveConnector
emergencyFile
pressureReportFile
state

Handle exceptions

PressureReleaseValveConnector
open()
close()
boiler
pressureReleaseValve

PressureSensorConnector
getPressure()
boiler
pressureSensor

Exceptions thrown

PressureReleaseValve
open()
close()
state

PressureSensor
getPressure()

Exception handling: example

```java
public class Boiler {
  public
    Boiler() {
      emergencyFile = new FileWriterWrapper("EmergencyFile.txt");
      emergencyFile.write("STARTING BOILER CONTROLLER\r\n");
      pressureReportFile = new FileWriterWrapper("pressureReportFile.txt");
      pressureReportFile.write("STARTING BOILER CONTROLLER\r\n");
      ps = new PressureSensorConnector(new PressureSensor(), this);
      prv = new PressureReleaseValveConnector(new PressureReleaseValve(
          HardwareState.stuck), this);
      boilerState = BoilerState.safe;
    }
    void shutdown() {
      emergencyFile.write("Engaging shutdown procedure.\r\n");
      prv.open();
      emergencyFile.write("BOILER CONTROLLER SHUT DOWN\r\n");
      emergencyFile.close();
      pressureReportFile.write("BOILER CONTROLLER SHUT DOWN\r\n");
      pressureReportFile.close();
    }
    void start() {
      while (boilerState == BoilerState.safe) {
        try {
          Thread.sleep(1000);
        }
        catch (InterruptedException e) {
        }
        System.out.println(ps.getPressure());
      }
    }
    PressureSensorConnector ps;
    PressureReleaseValveConnector prv;
    BoilerState boilerState;
    FileWriterWrapper emergencyFile;
    FileWriterWrapper pressureReportFile
}
```

```java
public enum ActuatorState {
  opened, closed }

public enum BoilerState {
  safe, unsafe, critical }

public enum HardwareState {
  operational, stuck }
```

```java
public class BoilerDriver {
  public
    static void main(String[] args) {
      Boiler b = new Boiler();
      b.start();
    }
}
```

- **Boiler**: connected to a temperature sensor and pressure release valve.
- Reports pressure readings in **pressureReportFile**.
- Reports erroneous behaviors in **EmergencyFile**.
- Repeatedly reads the pressure.
- No exception handling here.
- Could there be?

Exception handling: example

```java
public class FileWriterWrapper {
  public
    FileWriterWrapper(String new_file) {
      try {
        file = new BufferedWriter(new FileWriter(new_file));
      } catch (IOException e) {
        System.out.println("WARNING: file cannot be opened");
      }
    }

    void close() {
      try {
        file.close();
      }
      catch (IOException e) {
        System.out.println("WARNING: file cannot be closed");
      }
    }

    void write(String new_string) {
      try {
        file.write(new_string);
      }
      catch (IOException e) {
        System.out.println("WARNING: file cannot be written");
      }
    }

  private BufferedWriter file;
}
```

- The `FileWriterWrapper` class is created to isolate the file handling exceptions.

- Any exception thrown while opening, writing or closing a file are kept local to this class and thus do not reach the `Boiler` class

- This simplifies the code of the `Boiler` class.

Exception handling: example

```java
public class PressureSensorConnector {
  private
    PressureSensor ps;
    Boiler b;
  public
    PressureSensorConnector(PressureSensor new_ps, Boiler new_b) {
      ps = new_ps;
      b = new_b;
    }
    int getPressure() {
      int pressure = 999;
      try {
        pressure = ps.getPressure();
        b.pressureReportFile.write(pressure+"@"+getTime()+"\r\n");
      }
      catch (SensorException e) {
        b.emergencyFile.write(e.toString() + "\r\n");
        b.boilerState = BoilerState.unsafe;
        b.shutdown();
      }
      return pressure;
    }
    String getTime() {
      DateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
      return format.format(new Date());
    }
}
```

- **PressureSensor**: hardware component that reads temperature. If out of range, throw exception.
- Connected to the boiler using a **PressureSensorConnector** that reports pressure readings and catches and reports the exceptions thrown by the sensor.
- Exceptions thrown by the sensors do not reach the **Boiler** class.

```java
public class PressureSensor {
  public
    PressureSensor() {
      randomGenerator = new Random();
    }
    int getPressure() throws NegativePressureException,
      OverloadPressureException {
      int pressure = randomGenerator.nextInt(105) - 1;
      if (pressure < 0) {
        throw new NegativePressureException(pressure);
      }
      if (pressure > 100) {
        throw new OverloadPressureException(pressure);
      }
      return pressure;
    }
    Random randomGenerator;
}
```

Exception handling: example

```java
public class PressureReleaseValveConnector {
  private
    PressureReleaseValve prv;
    Boiler b;
  public
    PressureReleaseValveConnector(PressureReleaseValve new_prv, Boiler new_b) {
      prv = new_prv;
      b = new_b;
    }
    void close() {
      try {
        prv.close();
      }
      catch (StuckValveException e) {
        b.emergencyFile.write(e.toString());
      }
    }
    void open() {
      try {
        prv.open();
      }
      catch (StuckValveException e) {
        b.boilerState = BoilerState.critical;
        b.emergencyFile.write(e.toString() + "\r\n");
        b.emergencyFile.write("Evacuation!!\r\n");
      }
    }
}
```

```java
public class PressureReleaseValve {
  public
    PressureReleaseValve(HardwareState new_s) {
      hw_state = new_s;
      act_state = ActuatorState.closed;
      randomGenerator = new Random();
    }
    void close() throws StuckValveException {
      if (randomGenerator.nextInt(100) <= 2) {
        hw_state = HardwareState.stuck;
      }
      if (hw_state == HardwareState.stuck && act_state == ActuatorState.opened) {
        throw new StuckValveException(hw_state);
      }
    }
    void open() throws StuckValveException {
      if (randomGenerator.nextInt(100) <= 2) {
        hw_state = HardwareState.stuck;
      }
      if (hw_state == HardwareState.stuck && act_state == ActuatorState.closed) {
        throw new StuckValveException(hw_state);
      }
    }
  private
    ActuatorState act_state;
    HardwareState hw_state;
    Random randomGenerator;
}
```

- **PressureReleaseValve**: hardware component that opens/closes the boiler's container. If the valve is stuck throw an exception. If the valve is stuck closed, put the boiler in critical state.

- **PressureRelease ValveConnector** reports pressure readings to the Boiler while catching and reporting the exceptions thrown by the sensor.

Exception handling: significance

- Does the exception handling mechanism solve our error handling problems?
  - No, it is only a mechanism.

- Does the exception handling mechanism provide a radically new way of dealing with errors?
  - No, it simply provides a formal and explicit way of applying the standard techniques.

- The exception handling mechanism
  1. Makes it easier to adhere to good programming practices.
  2. Gives error handling a more regular style.
  3. Makes error handling code more readable.
  4. Makes error handling code more amenable to tools.

[Bjarne Stroustrup]

Exceptions: overhead

- The exception mechanism has a very minimal performance cost if no exception is thrown.

- If an exception is thrown, the cost of the stack traversal and unwinding is roughly comparable to the cost of a function call.

- Additional data structures are required to track the call stack after a try block is entered, and additional instructions are required to unwind the stack if an exception is thrown.

- However, in most scenarios, the cost in performance and memory footprint is not significant.

- The adverse effect of exceptions on performance is likely to be significant only on very memory-constrained systems, or in performance-critical loops where an error is likely to occur regularly and the code to handle it is tightly coupled to the code that reports it.

Exceptions: overhead

- The real cost of exception handling is in the difficulty of designing exception-safe code.

  - Constructors that throw exceptions are problematic: if a constructor fails, the object is not created, which makes it hard to recover from. This is even more problematic with class hierarchies, which require a sequence of constructors to succeed in order for an object to be fully constructed. In Java, an object is either successfully fully constructed or it does not exist.

  - Garbage collection definitely helps making things more easy for stack unwinding.

  - Additional design considerations may be needed in order to contain exceptions inside a certain scope.

References

- Oracle Corporation. [The Java Tutorials: Exceptions.](#)

- Oracle Corporation. [The Java Tutorials: The try-with-resources Statement](#).

- Oracle Corporation. [The Java Tutorials: Advantages of Exceptions.](#)

- Java.net. Mala Gupta. [Using Throws and Throw Statements in Java.](#)