Java Exception Handling - Exceptions Types

An *exception* (extends `java.lang.Exception`) defines an **unusual or erroneous situation** that must either be handled by the programmer or declared in a *throws* clause. e.g. `IOException`.

A *runtime/unchecked exception* (extends `java.lang.RuntimeException`) is an exception that can readily occur (e.g. **due to programming errors**) and is thus impractical to check for and thus does not need to be caught or declared (but can be) e.g. `NullPointerException`

An *error* (extends `java.lang.Error`) defines a generally **fatal machine-related problem** that cannot be handled and thus cannot be caught (i.e. Causes program/VM termination) `e.g.OutOfMemoryError`.

The Exception mechanism allows normal execution and exception execution flows to be separated. This enhances program readability/traceability and allows programmers to **decide** where to most effectively handle an exception.

Java Exceptions (1)

An *exception object* is unique in that it can be returned from a method without the use of the `return` keyword. (An exception object is `thrown`, not returned.)

All objects that can be thrown are descendants of `java.lang.Throwable`. This class has two direct descendents, `java.lang.Error` and `java.lang.Exception` which do not add any extra methods (i.e. They exist for typing/classification purposes only)

Exceptions should be caught by the programmer – either:

- where the exception occurs, or
- elsewhere in the program (higher up in the method stack)
- in general handle as close as is practical
- if in doubt throw the exception rather than handle badly
- is common to throw exception to the UI layer and let user decide

Java Exceptions (2)

If an exception is not caught by the program, the virtual machine ultimately catches the exception, and terminates the program abnormally (it crashes.) When an exception is caught in this way, the message string, and the place where it occurred in the program, are output:

```
class Zero {
  public static void main(String[] args) {
      int a = 10;
      int b = 0;
      System.out.println( a/b );
   }
}
```

```
java.lang.ArithmeticException: / by zero
   at Zero.main(Zero.java:7)
```


The Exception Crash Message

```
java.lang.ArithmeticException: / by zero
   at Zero.main(Zero.java:7)
```

An exception crash message contains not only the message string (in this case, “divide by zero”), but also the method stack trace which specifies exactly where the error occurred, in the form method name (filename:line).

In general, a trace has more than one line.

Each line shows the method that was called to get to the method above, i.e. the trace represents the method calling hierarchy, and depicts the runtime stack.

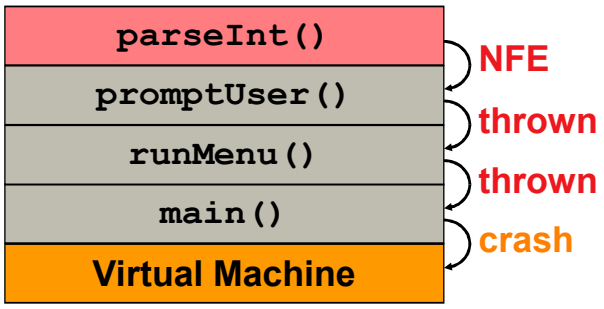
Exception Propagation

When an exception is thrown, it is passed to the calling method (i.e. next method in the stack).

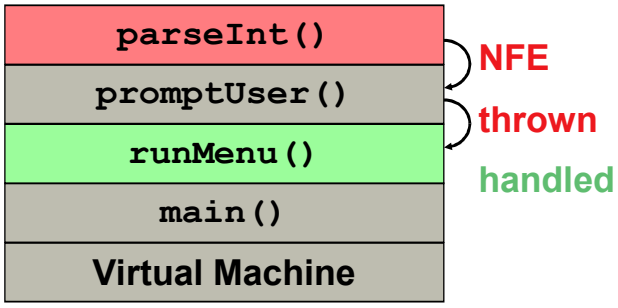
This process continues until it is either caught in a **`try..catch`** block, or it is thrown all the way to the Virtual Machine (which will cause the program to crash.)

The diagram on the right depicts how an exception can propagate through the method stack.

Without being caught:



Caught in `runMenu()`:



The throws Clause

The **`throws`** clause specifies that

(1) Exceptions are not going to be handled in the method where they are generated, and

(2) That they should be thrown to the calling method.

This has been used in programming 1:

```
public static void main(String[] args) throws IOException {
    // code is here
}
```

All programs so far that included data input from the keyboard were defined as above.

The '**`throws IOException`**' tells Java that this method can throw an **`IOException`** to the calling method (that can be thrown from the **`readLine()`** method calls.)

The try..catch Block (1)

If, instead of crashing the program, we want to catch the exception and handle it, the exception-prone code can be isolated inside of a `try..catch` block, such as:

```
try {
  int a = Integer.parseInt("dog");
}
catch (NumberFormatException e) {
  System.err.println("Not an integer!");
}
```

If the call to **`parseInt()`** has a problem running, the **`NumberFormatException`** that it throws will make the **`catch`** block code run, instead of the exception being thrown.

The try..catch Block (2)

A **`try..catch`** block can have multiple **`catch`** statements.

However, due to the polymorphic behaviour of exception types, care must be taken to order them such that the most specific exception types are the first **`catch`** statements, and the more generic types caught last.

```
try {
   // source code here with
   // many possible exceptions
}
catch (NumberFormatException e){
   // handle this exception
}
catch (FileNotFoundException e){
   // handle this exception
}
catch (Exception e){
   // handle this exception
}
```

The finally Block

In some cases, there might be code that needs to be executed regardless of **whether or not** an operation completed successfully (for example, a file output stream that needs to be closed.)

A **`finally`** clause can be appended to the end of a **`try..catch`** block to allow this functionality.

```
try {
    // code
}
catch (exception-type) {
    // error handling
}
finally {
    // this code will always run
}
```


Creating an Exception Type (1)

To create a custom exception type, we can simply **extend** the appropriate subclass of the **`Exception`** hierarchy, and **inherit the properties** that make it usable as an exception.

For example, if our custom exception was to model a problem related to the process of inputting data, it would make logical sense to extend the **`IOException`** class.

All exceptions and errors inherit **basic error properties**, such as a **message** string and the **stack trace** information.

If those properties are all that we need (which is often the case), we simply extend the appropriate `Exception` class, and fill in the constructors (since constructors are not inherited).

Creating an Exception Type (2)

```
class RangeException extends Exception {

  RangeException() {
    super();
  }

  RangeException(String message) {
    super(message);
  }
}
```


Exceptions versus boolean return value

A boolean can be accidentally ignored, a checked exception must be caught and handled (and an unchecked exception still thrown at runtime).

A well named exception provides additional semantics/meaning about the exception whereas the boolean requires commenting/documentation to describe its purpose.

The boolean cannot distinguish between different error types, whereas you can use polymorphic catch blocks to catch multiple custom exceptions.

The main caution with exceptions is not to use them for generic message passing (i.e. throwing an exception object containing state rather than passing an object as a parameter).

In contrast, in cases where there is a single clear mode of failure that can be reasonably ignored then use a boolean.

Exception Test 1: Local (1)

```
public class LocalExceptionTest {
  private static void numberTest (int num) {
    boolean valid;

    try {
      if (num < 0 || num > 9)
        throw new RangeException("out of range, ");
      else
        valid = true;
    }
    catch (RangeException e) {
      System.err.print(e.getMessage());
      valid = false;
    }
```


Exception Test 1: Local (2)

```
    finally {
      System.out.print("always do this");
    }

    if (valid)
      System.out.print(", valid data");
  }
}
```

Testing Input:

When `main()` executes:

```
1. numberTest(5);
2. numberTest(12);
3. numberTest(0);
4. numberTest(-1);
```

Testing Output:

1. always do this, valid data
2. out of range, always do this
3. always do this, valid data
4. out of range, always do this

Exception Test 2: Forward (1)

```
public class ForwardExceptionTest {
  private static void numberTest (int num)
                        throws RangeException {
    if (num < 0 || num > 9)
      throw new RangeException("Number out of Range");
    else
      System.out.println("A valid number was entered");
  }

  public static void main(String args[]) {
    try {
      numberTest(5);
      numberTest(12);
      numberTest(0);
      numberTest(-1);
    }
```


Exception Test 2: Forward (2)

```
    catch (RangeException e) {
      System.err.println(e.getMessage());
    }
    finally {
      System.out.println("Do this whatever happens");
    }
  }
}
```

Program Output:

A valid number was entered

Number out of range

Do this whatever happens

Exception Test 3: Uncaught

```
public class UncaughtExceptionTest {

  private static void numberTest (int num)
                        throws RangeException {

    if (num < 0 || num > 9)
      throw new RangeException("Number out of Range");
    else
      System.out.println("A valid number was entered");
  }

  public static void main(String args[]) throws RangeException{

    numberTest(5);
    numberTest(12);
    numberTest(0);
    numberTest(-1);
  }
}
```

Program Output:

A valid number was entered

<crash; exception stack trace>

Java Exceptions and Repetition (1)

A common task is to repeatedly prompt the user for input until it is correct.

```
public class Adder {

  public static void main (String[] args) {
    int n1 = UserReader.getInt("Enter a number: ");
    int n2 = UserReader.getInt("Enter another number: ");

    System.out.println ("The sum is " + (n1+n2));
  }
}
```

The `UserReader` class is defined on the next slide.

Java Exceptions and Repetition (2)

```
class UserReader {

  public static int getInt(String prompt) {

    BufferedReader stdin = new BufferedReader
                          (new InputStreamReader(System.in));
    int number = 0;
    boolean valid = false;

    while (! valid) {
      System.out.print (prompt);

      try {
        number = Integer.parseInt (stdin.readLine());
        valid = true;
      }
```


Java Exceptions and Repetition (3)

```
      catch (NumberFormatException exception) {

        System.out.println("Invalid input." +
                           "Try again.");
      }
      catch (IOException exception) {
        System.out.println ("Input problem." +
                            "Terminating.");
        System.exit(0);
      }

    }// end while loop

    return number;
  }
}
```