Exceptions

Exception basics, exception class hierarchy, checked vs. unchecked exceptions, creating exception classes, throwing exceptions, catching multiple exception types, finally

Exceptions basics

An **exception** is signal that something out of the ordinary has happened (usually something gone wrong).

- When the JVM detects an exception, it prints a stack trace that shows the name of the exception and the runtime stack showing where the exception occurred.
- Java features for handling exceptions:
  - **Exception:** A concrete class to represent the exception. This can be extended.
  - **try**: keyword before a block of code where a possible exception is anticipated.
  - **catch**: keyword before the block of code that handles the exception.
  - **throw**: keyword to indicate new exceptions.
  - **throws**: keyword to indicate a method may throw an exception.
  - **finally**: keyword to preface wrap-up code.

try/catch syntax

- A try block has the form:

```
try {
        statement t1;
        statement t2;
        ... statement tn;
}
```

- A catch block has the form:

```
catch (ExceptionType name) {
        statement c1;
        statement c2;
        ... statement ck;
}
```

- *ExceptionType* is a class in the Exception hierarchy.
- *name* is the name of the exception. The name is required, but you don't need to refer to it.

try/catch rules and execution

- Every try block must be followed immediately by a catch block (or a finally block).
- Every catch block must be preceded immediately by a try block or another catch block. The first catch block in the sequence must be preceded by a try block.
- Execution process:
  1. The statements in the try block are executed in the usual order.
  2. If executing a statement generates an exception:
     1. The rest of the try block is skipped.
     2. Control switches to the first catch block whose exception type matches the type of exception that was thrown.
     3. The statements in the catch block are executed.
     4. The remaining catch blocks are skipped.
  3. If no statement in the try block throws an exception, the catch blocks are skipped.

Example 1 - Simple exception

```
public class SillyExample {
  public static void main(String[] args) {
    int num = 1;
    try {
      int div = 14;
      if (num > 0)
        div = 0;
      num = num / div;   // EXCEPTION
      System.out.println("You won't see this message...");
    }
    catch (Exception e) {
      System.out.println("You tried to divide by 0.");
    }
    System.out.println("This message is at the end.");
  }
}
```

Example 1 discussion

Output from the code:

```
You tried to divide by 0.
This message is at the end.
```

- `num = num / div` generated an **ArithmeticException**.
- When exception occurs, the rest of try block is ignored, and the catch block executes.
- Code following the catch block is also executed.

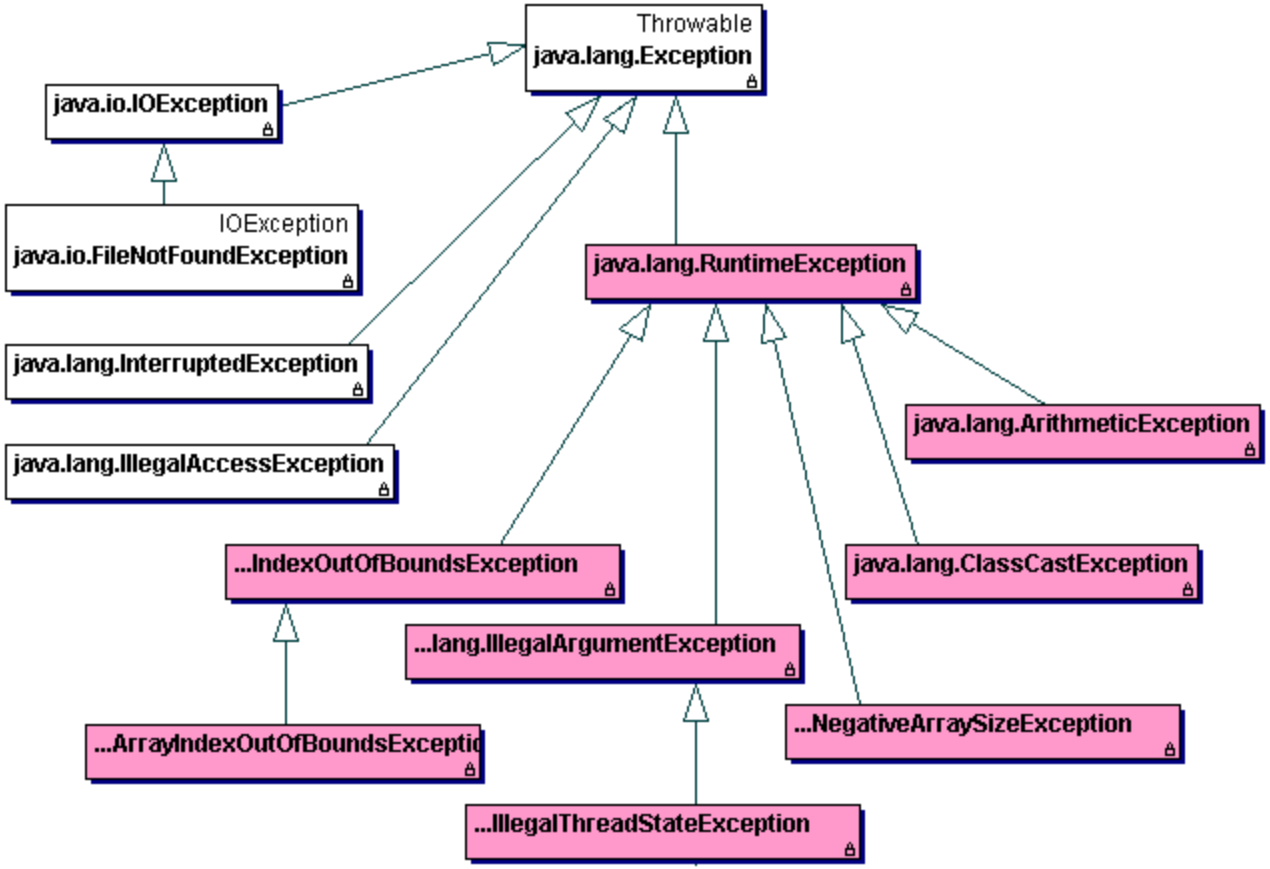
Example 2 – try/catch in a loop

```
public static void method() {
  Scanner s = new Scanner(System.in);
  System.out.print("Enter some numbers. 0 quits.");
  int count = 0;
  int sum = 0;
  int x = 1;
  do {
    try {
      count++;
      x = s.nextInt();
      sum += x;
    } catch (InputMismatchException e) {
      System.out.println("Line " + count + " is improper.");
      s.nextLine();
    }
  } while (x != 0);
  System.out.println("Sum of numbers entered: " + sum);
}
```

The exception does not break the loop, but it discards improper input.

Exception class hierarchy

The hierarchy is huge. This is part of it in java.lang.



Checked vs unchecked exceptions

- RuntimeExceptions are called **unchecked exceptions**. They include:
  - ArrayIndexOutOfBoundException
  - ArithmeticException
  - ClassCastException
  - NullPointerException
- Your code is not required to handle runtime/unchecked exceptions. When unchecked exceptions aren't handled:
  - Not a GUI-driven program: JRE prints out a stack trace and the program stops execution.
  - GUI-driven program: JRE prints out a stack trace but the program doesn't typically quit execution.
- All other exceptions are **checked exceptions**. Your code is required to handle them via try/catch.

Note: There is also an Error class for errors that are so serious that the program should not expect to catch and recover from. Error and Exception extend Throwable.

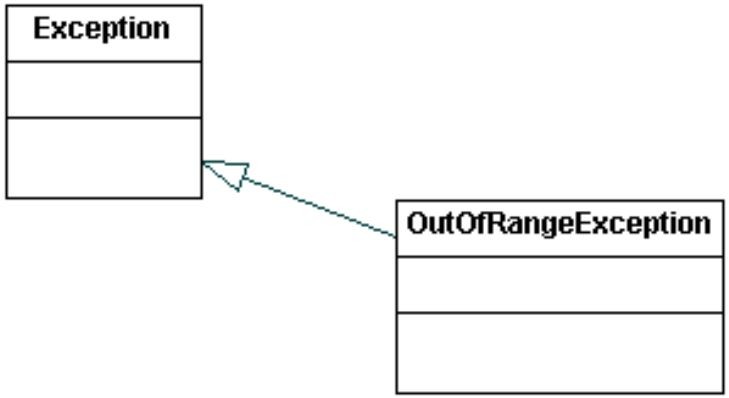
Custom exception classes

You can create your own exception classes. Eclipse can generate the constructors automatically.

```
public class OutOfRangeException extends Exception {

  public OutOfRangeException() {
  // TODO Auto-generated constructor stub
  }
  public OutOfRangeException(String message) {
     super(message);
  // TODO Auto-generated constructor stub
  }
  public OutOfRangeException(Throwable cause) {
     super(cause);
  // TODO Auto-generated constructor stub
  }
  public OutOfRangeException(String message, Throwable cause) {
     super(message, cause);
  // TODO Auto-generated constructor stub
  }
}
```

Creating exception classes (cont)



Exception class code is very simple. Typically just calling an Exception constructor is the only thing needed.

```
public class OutOfRangeException extends Exception {

  public OutOfRangeException() {
    super("Exception: Value is out of range")
  }
  public OutOfRangeException(String message) {
     super(message);}
}
```

Throwing exceptions

A method ***throws*** exceptions. The `throws` keyword on a method's header states that the method might generate an exception. The caller is responsible for handling the consequences.

```
public type name(params) throws KindOfException
```

If a method throws an exception, the method's header :

- *Must* list the checked exceptions for compilation.
- Can *optionally* list unchecked exceptions.
- Can list multiple exception types, separated by commas. Example:

```
public int calc(int x) throws NumberFormatException, MyException {
```

Exception throwing example

```
public char letter(int grade) throws OutOfRangeException {
 if (grade < 0 || grade > 100)
   throw new OutOfRangeException("Grade must be in [0,100]");
 if (grade > 89)
   return 'A';
 if (grade > 79)
   return 'B';
 if (grade > 69)
   return 'C';
 if (grade > 59)
   return 'D';
 return 'F';
}
```

The code has a **`throw`** statement to generate an exception. The **`throw`** statement almost always simply calls a constructor.

Catching multiple exceptions

You can catch multiple exceptions:

```
try {
   // statements
} catch (ExceptionType1 e1) {
   // code to handle e
} catch (ExceptionType2 e2) {
   // code to handle e2
}
```

For multiple exception types:

- The exception is caught by the first type that matches/is consistent with the type of exception that was thrown.
- ***`catch (Exception e)`*** always catches any exception. If you want to have it, put it last.

Multi-catch

If multiple exception types are to be handled the same way, you can use **multi-catch**, like this:

```
try {
    // statements
 } catch (ExceptionType1 | ExceptionType2 e) {
    // common exception handling code to handle e
 }
```

Catching the exception – or not

- doGrade() throws the exception down to the caller.
- doCatchGrade() handles the exception.

```
private char myGrade;

public void doGrade(int x) throws OutOfRangeException{
   myGrade = this.letter(x);
}

public void doCatchGrade(int x) {
   try {
      myGrade = letter(x);
   }
   catch (Exception e) {
      myGrade = 'F';
   }
}
```

Multiple exceptions example

```
public int translateTime(String time) {
    int totalSeconds = 0;
    Scanner s = new Scanner(time);
    s.useDelimiter(":");
    try {
        totalSeconds += 3600 * s.nextInt();
        totalSeconds += 60 * s.nextInt();
        totalSeconds += s.nextInt();
        return totalSeconds;
    }
    catch (NoSuchElementException e) {
        return totalSeconds;
    }
    catch (NumberFormatException e) {
        return 0;
    }
}
```

Finally blocks

- A try block can have a corresponding finally block.
- If any code in the try block is executed, the finally block is guaranteed to execute.
- Finally blocks are typically used to clean up resources and avoid code duplication (close file streams, close database streams, etc.).

```
public int tryAndFinally(String x) {
    try {
      return Integer.parseInt(x);
    }
    catch (NumberFormatException e) {
      return 25;
    }
    finally {
      System.out.println("Finally!!");
    }
}
```

Finally blocks

Executing ***tryAndFinally()***.

```
// Calling code…
System.out.println(tryAndFinally("92"));
System.out.println(tryAndFinally("abc"));
```

The output is:

```
Finally!!
92
Finally!!
25
```