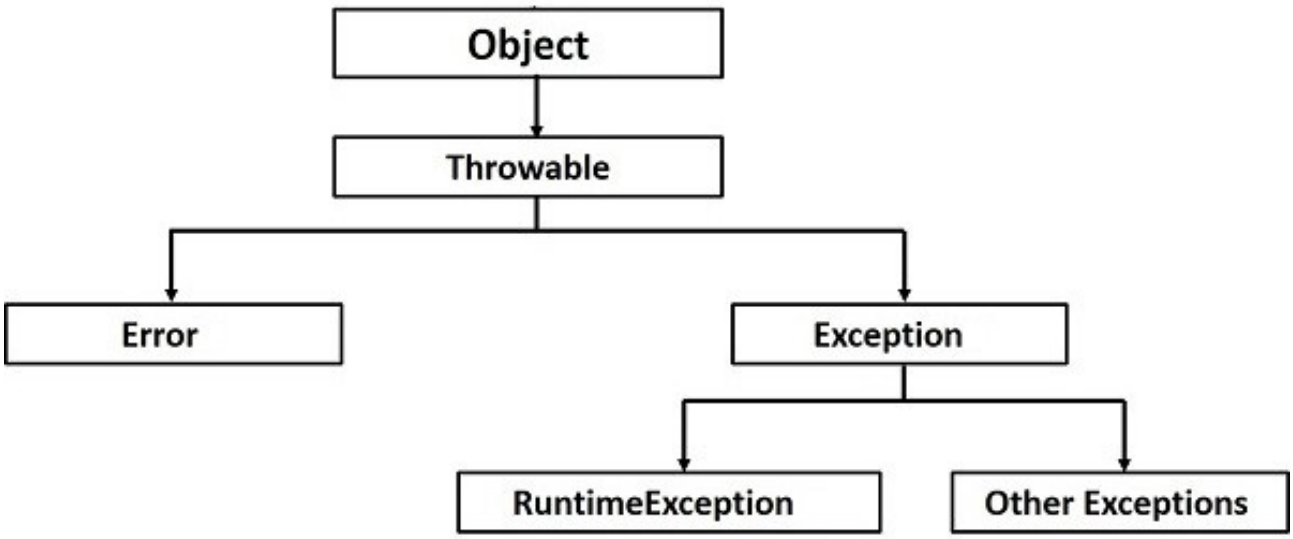
Wyjątki

Wyjątki są sytuacjami nietypowymi w programie. Możemy je rozumieć jako “błędy”, na które programista musi jakoś zareagować. Najprostszym przykładem jest dzielenie przez zero, które jak wiemy jest w matematyce zabronione. W językach niższego rzędu (np. C) należało zabezpieczać kod przed taką możliwością wprost w kodzie używając instrukcji **if** wszędzie tam gdzie istniała taka możliwość. Było to żmudne i nie do końca odpowiadało zadaniu jakie programista powinien zrobić - powinien on skupić się na oprogramowaniu danej funkcjonalności a nie szukać wszelkich możliwych matematycznych błędów jakie mogą się zdarzyć. W Javie jest inaczej, tu dostajemy gotowy system obsługi wyjątków. To sama JAVA zasygnalizuje, że coś jest nie tak poprzez system wyjątków. Programista jedynie będzie musiał zareagować odpowiednio - tj. obsłużyć wyjątek.

Zanim zaczniemy programować - przyjrzyjmy się strukturze wyjątków w Javie:



Z przedstawionego wykresu wynika, że mamy 3 rodzaje wyjątków:

- wyjątki dziedziczące po **Error**, które nie wynikają bezpośrednio z błędu programisty a z ograniczeń maszyny. Przykładowym błędem jest sytuacja kiedy zabraknie pamięci RAM w programie lub ilość wywołań rekurencyjnych przekroczy limit. Takich błędów raczej nie obsługujemy (choć możemy). Gdy takowy błąd wystąpi - próbujemy zaradzić zmieniając parametry maszyny na której program jest uruchamiany.

- wyjątki dziedziczące po **RuntimeException**, których nie trzeba (choć można) obsługiwać. Najczęstszym wyjątkiem tego typu jest znany nam dobrze **NullPointerException**, który jest rzucany gdy odwołujemy się do metod obiektu, który nie istnieje.

```
String text = null;
// ...
text.substring(1,4); // throw: NullPointerException
```

- wyjątki dziedziczące po **Exception** (ale nie po RuntimeException) - wyjątki, które trzeba łapać. Kompilator nie pozwoli nam na ich pominięcie.

W przypadku wyjątków mechanizm wygląda tak, że program zakończy wykonywanie kodu w miejscu gdzie wyjątek zostanie rzucony. Wyjątek ten zacznie wydostawać się z metody w której powstał do metody, która tą metodę uruchomiła - tak długo aż dojdzie to metody **main**. Gdy opuści main - program zakończy swoje działanie. Zasada jest taka, że każdy wyjątek który opuści metodę main - powoduje zatrzymanie programu. To od programisty zależy czy będzie temu przeciwdziałał próbując wyjątki obsłużyć. O ile obsługę wyjątków dziedziczących po **Exception** wymusza sam kompilator - to pozostałych już nie wymaga, więc programista często nie jest świadom, że mogą zostać w tym miejscu rzucone.

Jak działa przymus obsługi wyjątków? Wrzućmy do kompilatora następujący kod:

```
import java.io.FileReader;

public class ExceptionExample {

   public static void main(String args[]) {
      new ExceptionExample().readFile();
   }

   public void readFile(){
       FileReader fr = new FileReader("E://file.txt");
       char [] a = new char[50];
       fr.read(a);   // wczytaj 50 znaków
       for(char c : a){
         System.out.print(c);   // drukuje znak po znaku
       }
   }
}
```

Powyższy kod nie będzie się kompilował. Dlaczego? Ponieważ **FileReader** rzuca wyjątkiem w przypadku braku pliku podanego w konstruktorze. Widać to, gdy wejdziemy w definicję konstruktora gdzie deklaracja wygląda następująco:

```
public FileReader(String fileName) throws FileNotFoundException { ... }
```

mało tego: metoda **fr.read(a);** rzuca wyjątkiem **IOException**:

```
public int read(char[] cbuf) throws IOException { ... }
```

Co więc powinien zrobić programista w takiej sytuacji? Ma dwa wybory:

- Zadeklarować w metodzie **readFile**, że rzuca wyjątkiem **FileNotFoundException** oraz **IOException** i zrzucić obowiązek ich obsługi w metodzie **main**. Możemy nawet skorzystać z abstrakcji i powiedzieć, że metoda **readFile** rzuca tylko **IOException** gdyż **FileNotFoundException** dziedziczy po **IOException** (zobacz https://docs.oracle.com/javase/7/docs/api/java/io/FileNotFoundException.html).

  ```
  ...
  public void readFile() throws FileNotFoundException, IOException {
  // lub samo: IOException
  ```

```
        FileReader fr = new FileReader("E://file.txt");
        char [] a = new char[50];
        fr.read(a);   // wczytaj 50 znaków
        for(char c : a){
          System.out.print(c);   // drukuje znak po znaku
        }
    }
...
```

Od tej chwili kompilator nie będzie zgłaszał błędu w metodzie **readFile** a w metodzie **main** gdzie teraz tam faktycznie nie będzie obsługi owych wyjątków. Oczywiście możemy zadeklarować również, że sama metoda **main** rzuca wyjątkiem:

```
public static void main(String args[]) throws
FileNotFoundException, IOException {
      new ExceptionExample().readFile();
}
```

wtedy kompilator się od nas odczepi - ale czy to jest wyjście z sytuacji? Wątpię.

- Sensowniejszym wyjściem z sytuacji częściej jest obsługa wyjątku i nie dopuszczenie do przerwania programu gdy jakiś błąd wystąpi.

Do obsługi wyjątków służy blok **try - catch**, który przerywa działanie kodu i przechodzi do odpowiedniej procedury obsługi wątku. To my decydujemy co stanie się dalej z kodem po wykryciu wyjątku. Oczywiście możemy przerwać działanie kodu lub nie robić nic. Wygląda to następująco:

```
//...
public static void main(String args[]) {
    try {
      new ExceptionExample().readFile();
      System.out.println("TO JEST TEXT");
    } catch (FileNotFoundException e) {
        System.out.println("Brak pliku. Powód: " + e.getMessage());
    } catch (IOException e) {
        System.out.println("Problem ogólny WE/WY. Powód: " +
e.getMessage());
    }
    //... dalsza część kodu
}
// ...
```

Jak widzimy: metoda **main** nie ma już klauzuli **throws FileNotFoundException**, **IOException**. Jest za to obsługa każdego z błędów w postaci wydrukowania treści błędu. Oczywiście taka obsługa może być bardziej wyrafinowana - włącznie ze zmianą logiki działania całego programu.

Kilka uwag do kodu powyżej:

- Jeśli wystąpi wyjątek w **readFile()** to na ekran nie zostanie wypisany napis **“TO JEST TEXT”**
- Jeśli chcemy - możemy obsłużyć tylko część błędów. Resztę można zadeklarować w klauzuli **throws** metody.

- Catch w wyjątkach piszemy w kolejności - bardziej szczegółowe a potem bardziej ogólne. Taki zapis by się nie skompilował:

```
    try {
      new ExceptionExample().readFile();
      System.out.println("TO JEST TEXT");
    } catch (IOException e) {
        System.out.println("Problem ogólny WE/WY. Powód: " +
e.getMessage());
    } catch (FileNotFoundException e) {
        System.out.println("Brak pliku. Powód: " + e.getMessage());
    }
```

  Dlaczego? Bo **FileNotFoundException** dziedziczy akurat po **IOException**. Pisanie więc najpierw **IOException** powoduje, że bez sensu jest pisać **FileNotFoundException** gdyż nie ma szans aby kiedykolwiek ten catch się wykonał.

- Jeśli kilka wyjątków dziedziczy po jednym (wspólny rodzic) to możemy użyć abstrakcji i złapać kilka wyjątków za jednym razem:

```
    try {
      new ExceptionExample().readFile();
      System.out.println("TO JEST TEXT");
    } catch (Exception e) {
        System.out.println("To złapie każdy wyjątek");
    }
```

  Jako, że każdy wyjątek dziedziczy po **Exception** - kod powyżej złapie każdy wyjątek.

- Możemy łączyć obsługę kilku różnych wyjątków za pomocą konstrukcji:

```
    try {
      new ExceptionExample().readFile();
      System.out.println("TO JEST TEXT");
    } catch (FileNotFoundException | ArrayOutOfBoundException e) {
        System.out.println("Obsługa kilku wyjątków naraz");
    }
```

  Połączone zostały tutaj wyjątki spójnikiem **|**. Proszę zauważyć, że bezsensowne byłoby połączenie wyjątków: **catch(FileNotFoundException | IOException e) {...}** bo wystarczy tylko ten drugi. Java zasygnalizuje nam to błędem.

Istnieje również blok: **try - catch - finally** różni się on tym, że istnieje jeszcze część kodu w klauzuli **finally**, która wykona się zawsze - nawet wtedy gdy wystąpi wyjątek. Zamiast opisywać - napiszę kod, który wyraża 1000 słów:

```
class TestClass {
   static void myMethod(int number) throws Exception {
      System.out.println ("start - myMethod");
      if (testnum == 2) {
         throw new Exception();
      }
      System.out.println("end - myMethod");
```

```
   }

   public static void main(String args[]) {
      int number = 2;
      try {
         System.out.println("try - first statement");
         myMethod(number);
         System.out.println("try - last statement");
      } catch ( Exception ex) {
         System.out.println("Exception thrown");
      } finally {
         System. out. println( "finally") ;
      }
      System.out.println("Out of try/catch/finally - statement");
   }
}
```

Widzimy tutaj przy okazji jak możemy rzucić wyjątkiem. Robi się to za pomocą komendy **throw** powołując nowy wyjątek do życia. Zamiast tłumaczyć co się tu wydarzy… przeklej kod do swojego IDE i sprawdź sam które linijki kodu się wykonają. Sam wtedy wyciągnij wnioski jak działa ta konstrukcja. W końcu to laboratorium a nie wykład ;)

Własny wyjątek

W kodzie wyżej widzieliśmy, jak rzuciłem wyjątkiem **Exception**. Można stworzyć własny wyjątek. Wystarczy zdziedziczyć po odpowiednim wyjątku (Exception, RuntimeException bądź którymś z ich “dzieci”) i gotowe. Możemy rzucać wtedy już własnymi wyjątkami (**throw new MyException(“Coś jest nie tak”)**