# Javaプログラミング実習

# 29. 例外

株式会社ジードライブ

# 今回学ぶこと

- 例外の概念と処理方法

# エラーの種類

- 文法的なエラー（Syntax Error）
  - 変数名の間違いやセミコロン忘れなど

  ⇒Eclipseが ⊗ で教えてくれる
- 論理的なエラー（Logical Error）
  - 例：100円と200円を購入したら、合計金額が100200円と表示された

  ⇒デバッガを使って調査する
- 実行時のエラー（Runtime Error）
  - プログラムには間違いがないが、想定外に(**例外的に**)発生してしまうエラー(異常事態)

  ⇒メモリ不足、読み込み予定のファイルがない…等

# 例外とは

- プログラム実行中に発生する異常
  - Exception (エクセプション) ともいう
- 例外が発生する場面の例
  - 配列で存在しない要素にアクセスした
  - ファイルを読み込もうとしたが指定したファイルが存在しなかった
  - nullが代入されているオブジェクトのメソッドを呼び出した

# 例外の例

- 配列の範囲外の要素を参照する例

```
String[] fruits = {"りんご", "バナナ", "ぶどう"};
System.out.println(fruits[1]);
System.out.println(fruits[3]); // fruits[3] は存在しない
```

- 実行結果

```
バナナ
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 3
        at exception.Example.main(Example.java:9)
```

# 例外の例

- ArrayIndexOutOfBoundsException
  - 不正なインデックスで配列へのアクセスした場合に発生する例外
- IOException
  - 入出力における例外
- FileNotFoundException
  - 指定されたファイルが見つからない場合の例外
- NullPointerException
  - nullオブジェクトのメソッドを呼び出そうとした場合などに発生する例外

etc...

# 例外処理の書式

- Javaでは、例外が発生した際に特別な処理を行う仕組み(try-catch構文)が存在する
  - tryブロックで例外が発生すると、catchブロックに移動する

例外が投げられる(throwされる)

書式

```
try {
    // 例外が発生する可能性のあるステートメント
    …
} catch（例外クラス型 変数）{
    // 例外が発生した際に実行するステートメント
    …
}
```

例外クラスのオブジェクトが入る
⇒ catchブロックで使用可能

# 例外処理の流れ

**try**ブロック内で例外が発生すると…

- 発生した例外の種類が**catch**ブロックで指定した例外と一致する場合は**catch**ブロック内の処理が実行される
  - catchブロック内の処理が実行された後は、その後に続く処理が実行される
- catchブロックで指定していない例外(キャッチされない例外)が発生すると、プログラムの処理は中断する

# 例外処理の例

```
String[] fruits = {"りんご", "バナナ", "ぶどう"};
try {
    System.out.println(fruits[1]);
    System.out.println(fruits[3]); // 例外発生⇒catchブロックへ
    System.out.println(fruits[2]); // 実行されない
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("例外が発生しました");
}
```

実行

```
バナナ
例外が発生しました
```

# 練習

- 練習29-1

# 複数の例外処理

- catchブロックは連結することができる

```
try {
  …
} catch（例外クラス型1 変数）{
  // 例外1が発生した際に実行するステートメント
  …
} catch（例外クラス型2 変数）{
  // 例外2が発生した際に実行するステートメント
  …
} catch（例外クラス型3 変数）{
  // 例外3が発生した際に実行するステートメント
  …
}
```

例外には階層関係(継承関係)がある ⇒ 階層の深い例外を先に記述する

# マルチキャッチ

- 複数の例外を一括でキャッチして処理する方法

```
try {
  // 例外が発生する可能性のあるステートメント
  …
} catch（例外クラス型1 | 例外クラス型2 | 例外クラス型3 変数）{
  // 例外が発生した際に実行するステートメント
  …
}
```

# finally

- 例外発生の有無に関わらず何らかの処理を行いたい場合に使用するブロック
  - キャッチされない例外が発生した場合も実行される

書式

```
try {
  …
} catch（例外クラス型 変数）{
  …
} finally {
  // 例外が発生してもしなくても実行するステートメント
  …
}
```

# finallyの利用

- 以下のような場面で利用する
  - ファイルのクローズ処理
  - データベースのクローズ処理
  - 通信のクローズ処理 など
- `finally`ブロックの中では、`break`, `continue`, `return` など、`finally`ブロックの外に制御が移る処理を行うべきではない
  - `catch`されなかった例外が無視されてしまうため

⇒例外が発生した場合は正しく処理をする、または、処理を中断すべき

# 練習

- 練習29-2

# try-with-resources

- この構文を利用すると、AutoCloseableインターフェースを実装しているクラスのclose処理が省略できる
  - finallyブロックでのclose処理が不要になる

```
try (AutoCloseableインターフェースを実装するクラス) {
  …
} catch (例外クラス型 変数) {
  …
}
```

# try-with-resources

例

```
InputStream is = new FileInputStream("data.txt");
InputStreamReader isr = new InputStreamReader(is);
try (BufferedReader br = new BufferedReader(isr)) {
   String text = br.readLine();
   System.out.println("テキスト：" + text);
} catch (IOException e) {
   System.out.println("IOException発生");
}
```

以下の記述が不要になる

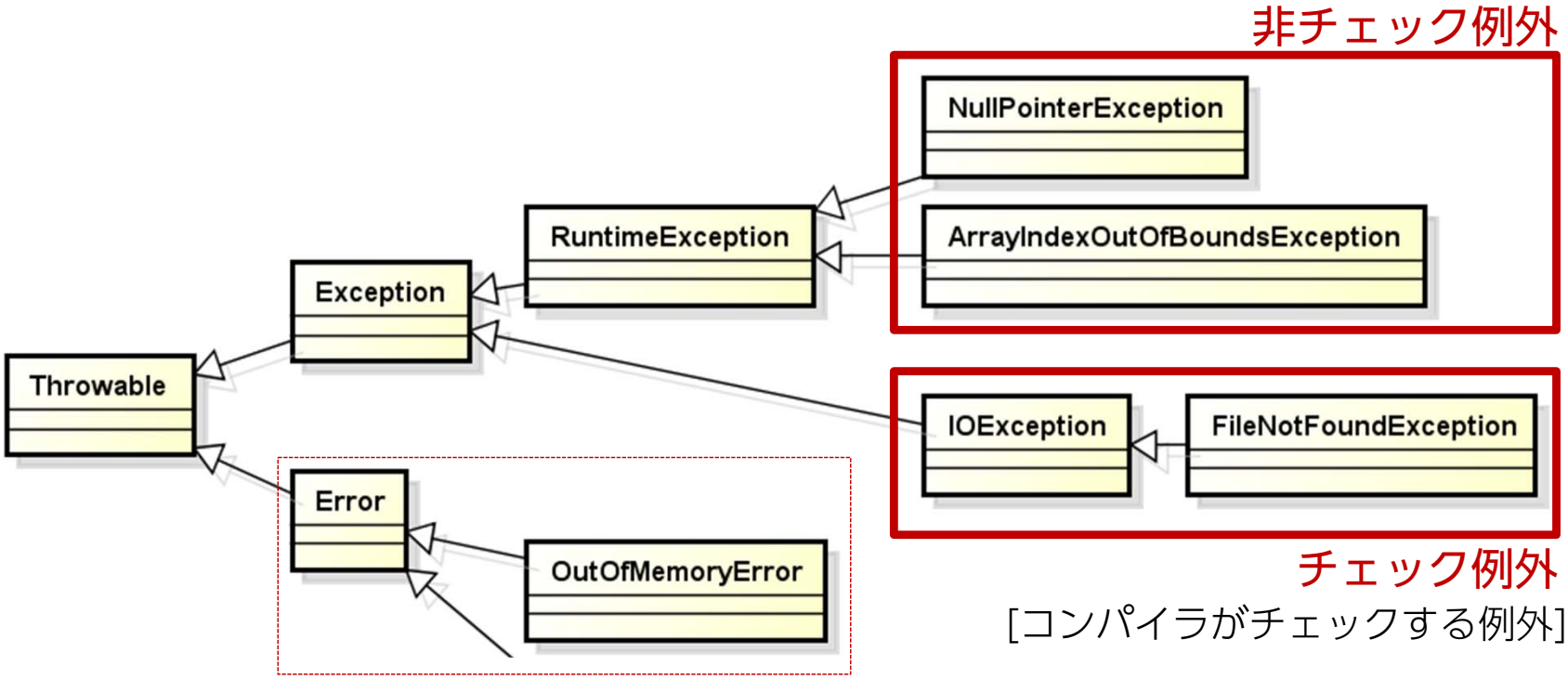
```
finally {
  br.close();
}
```

# 練習

- 練習29-3
- 練習29-4

# 例外クラスの継承関係

- RuntimeExceptionから派生：非チェック例外
- Exceptionから直接派生：チェック例外




[コンパイラがチェックする例外]

メモリ不足やクラスファイルの破損等
⇒ 回復の見込みがなく、キャッチしても対処のしようがない(キャッチしない)

# 継承関係にある例外の処理

- スーパークラスの例外をキャッチする記述でサブクラスの例外もキャッチされる

例：ファイル入出力の処理
ここで発生し得る FileNotFoundException は IOExceptionのサブクラス

```
try {
  …ファイル入出力などの処理…
} catch (IOException e) {
  System.out.println("入出力エラーが発生しました");
}
```

IOException も FileNotFoundException もまとめてキャッチされる

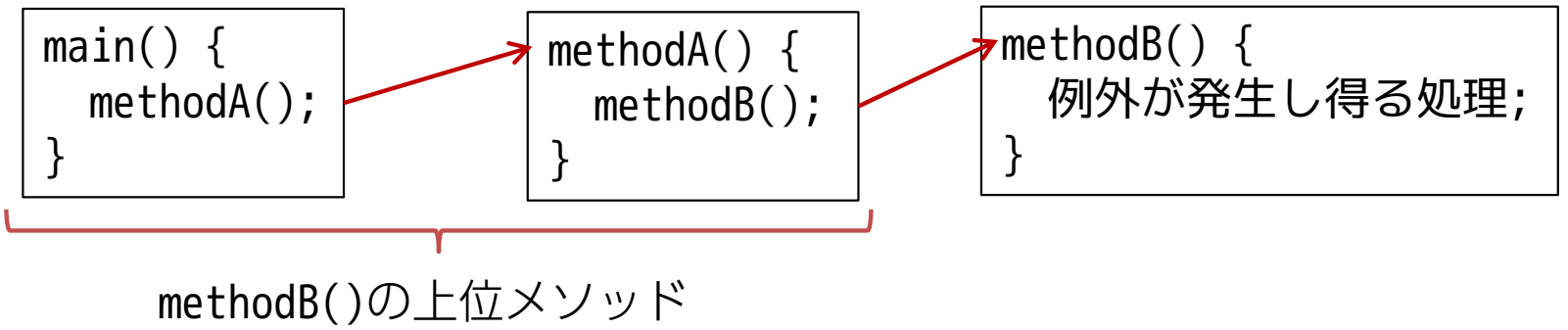
# 非チェック例外への対応

- 非チェック例外は、想定しきれない(一つ一つ対応していたらきりがないような)ものが多い
  - nullオブジェクトの利用(NullPointerException)
  - 不正な配列インデックスへのアクセス(ArrayIndexOutOfBoundsException)
- 非チェック例外は、キャッチしなくてもコンパイルエラーにはならない
- 非チェック例外が発生した場合は自動的に上位メソッドへスローされる

# チェック例外への対応

- チェック例外は、発生を予期し、対処を考える必要のある事態
  - ファイルの読み書きができない(IOException)
  - ネットワーク接続ができない(ConnectException)
- try-catchでキャッチするか、上位メソッドに例外をスローする必要がある
  - 記述がないとコンパイルエラーが発生する

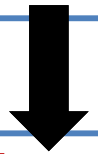
  ⇒Eclipseが対応が必要であることを教えてくれる



# 上位メソッドへの例外の伝搬

- throwsの宣言を記述することで、例外処理を上位メソッドに委ねることができる
  - try-catch構文の記述が不要になる

```
public void methodB() {
    try {
        …IOException が発生し得る処理…
    } catch (IOException e) {
        …例外処理…
    }
}
```



複数の例外をスローする場合は、カンマ区切りで列挙

```
public void methodB() throws IOException {
    …IOException が発生し得る処理…
}
```

# 上位メソッドへの例外の伝搬

- 例外のスローが行われた場合、上位メソッドのどこかでtry-catch構文の記述が必要になる

```
public void methodB() throws IOException {
  …IOException が発生し得る処理…
}
```

```
public void methodA() throws IOException {
  methodB();
}
```

```
public void main(String[] args) {
  try {methodA();}
  catch (IOException e) {
    …例外処理…
  }
}
```

```
public void methodA() {
  try {methodB();}
  catch (IOException e) {
    …例外処理…
  }
}
```

```
public void main(String[] args) {
  methodA();
}
```

# 意図的な例外のスロー

- メソッド内に throw new Exception() を記述することで、意図的に例外をスローすることができる
  - IOExceptionなど、他の例外も意図的にスロー可能
  - catchブロックに移動するか、上位メソッドに例外が伝搬する

```
if (age < 0 || age > 150) {
  throw new Exception("年齢が範囲外");
}
else if (age >= 20) {
  System.out.println("飲酒可能です");
}
else {
  System.out.println("飲酒できません");
}
```

# 例外クラスのメソッド

- getMessage() や printStackTrace() といったメソッドを使用することができる

```
try {
   ...
}
catch (Exception e) {
   System.out.println(e.getMessage());
   e.printStackTrace();
}
```

# 練習

- 練習29-5