

# Javaプログラミング(7)

## 例外処理, ファイル入出力

成蹊大学理工学部  
情報科学科

# 本日の内容(例外処理, ファイル入出力)

- 例外処理
- ファイルの読み込み
- ファイルの書き込み
- Javaでの文字コードの扱い
- splitメソッドについて

# 例外処理

- 例外処理(Exception)
  - プログラム動作中に起こった問題を通知し、エラー処理をする機構
- 変数の宣言や構文誤りといった実行する前にわかるプログラムの誤りは、コンパイラがチェックをしてくれるが、コンパイラは通っても、実行時に問題が起こることはよくある。
  - (例) 配列の長さ以上の要素を参照しようとする、ファイルを開こうとしたらファイルがない、メソッドに入力された引数が不適切等

# 例外が起こるプログラム例

```
1: public class ExceptionTestError {
2:     public static void main(String[] args) {
3:         int[] myarray = new int[3]; // 大きさ3の配列を確保
4:         System.out.println("代入");
5:         myAssign(myarray, 100, 0); ←
6:         System.out.println("代入完了");
7:         System.out.println("プログラム終了");
8:     }
9:
10:    static void myAssign(int[] arr, int index, int value) {
11:        System.out.println("myAssign開始");
12:        arr[index]=value;
13:        System.out.println("myAssin完了");
14:    }
15: }
```

大きさ3の配列の  
100番目に代入し  
ようとしている！

# 例外が起こるプログラム例(Cont.)

代入

myAssign開始

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:  
100

at ExceptionTestError.myAssign(ExceptionTestError.java:12) ←12行目  
で例外

at ExceptionTestError.main(ExceptionTestError.java:5)←5行目で例外

- 例外処理がないと, プログラムが途中で止まって, 異常終了する.

# 例外をキャッチする

## 例外をキャッチする方法

```
try {  
} catch {  
}
```

```
1: public class ExceptionTest {  
2:     public static void main(String[] args) {  
3:         int[] myarray = new int[3]; // 大きさ3の配列を確保  
4:         try {  
5:             System.out.println("代入");  
6:             myAssign(myarray, 100, 0);  
7:             System.out.println("代入完了");  
8:         } catch (ArrayIndexOutOfBoundsException e) {  
9:             System.out.println("代入失敗");  
10:            System.out.println("例外 : "+e);  
11:            e.printStackTrace(); // スタックトレース  
12:        } finally {  
13:            System.out.println("プログラム終了");  
14:        }  
15:    }  
}
```

# 例外をキャッチした場合の実行例

myAssign開始

代入失敗

例外：java.lang.ArrayIndexOutOfBoundsException: 100

← 例外の内容

<スタックトレースで表示されるメッセージ>

java.lang.ArrayIndexOutOfBoundsException: 100

at **ExceptionTest.myAssign**(ExceptionTest.java:19) ←19行目で例外

at **ExceptionTest.main**(ExceptionTest.java:6) ←6行目で例外

<ここまで>

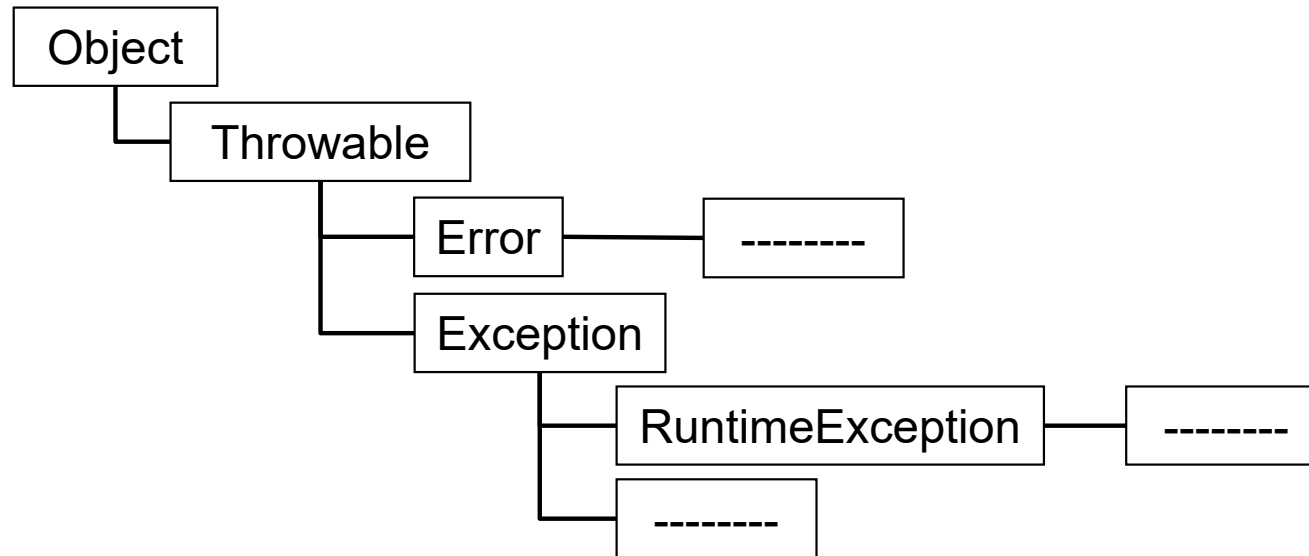
プログラム終了 ←先の処理に進んでいる

# 例外処理の内容

- プログラムが途中で終了することはない。例外を受けて、先の処理に進むことができる。（システムの異常終了を避けることができる）
- デバッグプリントメッセージに”e”を入れると、その例外の内容が文字列に変換されるので起こった例外が何であるのかを調べるのに便利。
- メソッド呼び出しの積み重ねのことを「コールスタック (Call Stack)」と呼ぶ。
- `printStackTrace()`: コールスタックのトレースを表示
- スタックトレース中に示される行番号で、プログラムの何行目で例外が起こったかがわかる。



# 例外の階層



- ErrorとExceptionはThrowable(投げられる)クラスの拡張である.
- 例外処理で投げられるものは, これらのクラスのインスタンスである.
- クラスライブラリのマニュアルで, コンストラクタやメソッドの説明に投げられる例外の種類が記述されている.
- 親クラスのExceptionクラスを使ってキャッチすることもできる. 例外を種類に応じて個別に処理しない(する必要がない)場合は, Exceptionクラスでキャッチしてしまっても良い.

# Exceptionクラスの2種類のサブクラス

Exception(例外)クラスのサブクラスは, コンパイラによって例外処理を強制されるものとそうでないものがある.

A) RuntimeException: 例外処理が不要ない.

(例) 配列の範囲を超えたアクセス. 整数を0で割る. 不当な型変換.

B)(RuntimeException以外の)サブクラス: **例外処理を必ずしなければならない.**

(例) ファイルが存在しない. 通信でエラーが発生した.

# 例外処理記述方法

- (方法1) メソッド内のcatch節で例外をキャッチする(先述)
- (方法2) メソッド宣言の中で, throws節を宣言する(例外を投げる可能性があることを宣言)

# throwsの例

```
import java.io.*;
public class ThrowsSample {
    static int getInt(String s) throws IOException {
        //文字入力ストリームを作成
        BufferedReader r =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print(s);    // メッセージ出力
        String t = r.readLine(); // 文字列の入力
        return Integer.parseInt(t); // 整数に変換して返す
    }
    public static void main(String[] args) {
        try{
            int a = getInt("たて : ");
            int b = getInt("よこ : ");
            System.out.println("面積 : "+(a*b));
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

# finally

- catchは、例外が起こったときに、単に処理を中断するのみ.
- finallyブロックは例外が投げられても投げられなくても実行されるブロック.
- tryブロック中での処理の後始末を書いておくことにより、例外が起こった場合でもプログラムの終了処理が確実に行われる.

```
finally {  
    System.out.println("プログラム終了");  
}
```

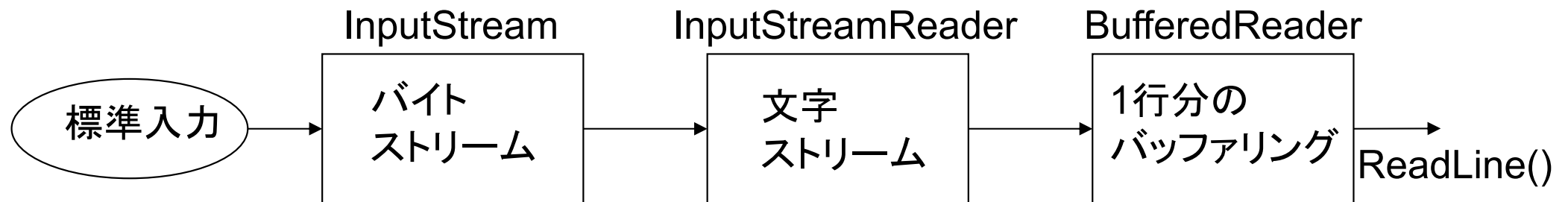
# 例外処理の意義

- プログラムが異常終了してしまうことを避けることができる.
- プログラムの誤りを効率よく検出するための手助けにもなる.

# 入出力とストリーム

ファイルの入出力や標準入力では「ストリーム」が用いられる。

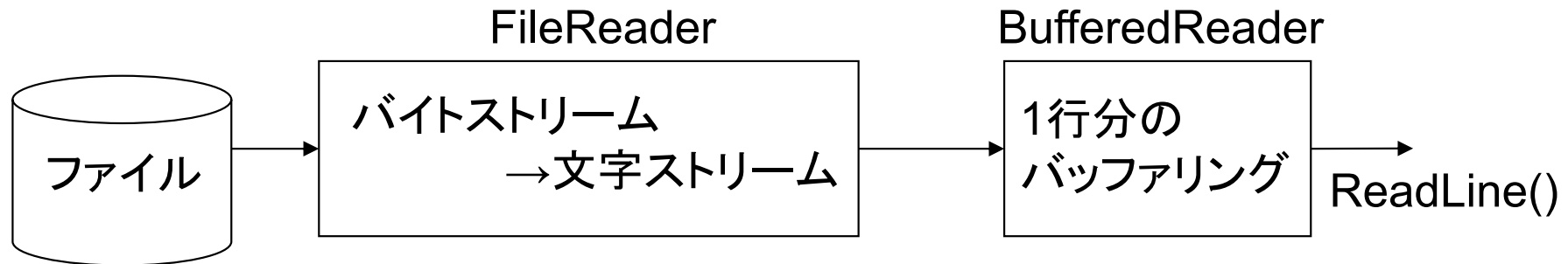
## <標準入力からの読み取り>



```
BufferedReader reader = new BufferedReader (new InputStreamReader(System.in));
```

InputStreamReaderは、文字ストリーム(16ビットの文字単位)で読み取りを行うクラス

# ファイルの読み込み



```
BufferedReader reader = new BufferedReader(new FileReader(filename));
```

- バッファリング

- ハードディスク内のデータをメモリに転送することにより、データの入出力を高速，効率的に行う方法.
- FileReaderはディスクにアクセスしに行ってしまうが，BufferedReaderをかぶせておくことにより，バッファリングが行われ，ディスクへのアクセスを減らすことができる。



# ファイル読み込みの方法

ファイル読み込みためのオブジェクトとメソッド

作成するオブジェクト

```
BufferedReader reader = new BufferedReader(new FileReader(filename));
```

ファイルから1行ずつ読み込むメソッド

```
reader.readLine();
```


- ・入れ子構造になっているところは、分割しても同じこと

```
FileReader freader = new FileReader(filename);
```

```
BufferedReader reader = new BufferedReader(freader);
```

# ファイル読み込みのプログラム例

```
import java.io.*;
public class ShowFile1 {
    public static void main(String[] args) {
        if (args.length!=1) {
            System.out.println("使用法: java ShowFile1 ファイル名");
            //ファイルが指定されていなければシステム終了
            System.exit(0);
        }
        String filename = args[0];
```



java.ioパッケージをインポートしておく

# ファイル読み込みのプログラム例 (Cont.)

... つづき

**try** {

BufferedReader reader =

**new** BufferedReader(**new** FileReader(filename));

String line;

**while** ((line=reader.readLine())!=**null**) {

System.out.println(line);

}

reader.close();

} **catch** (FileNotFoundException e) {

System.out.println("FileReaderから投げられた例外"+e);

} **catch** (IOException e) {

System.out.println("readLineから投げられた例外"+e);

}

}

}

FileReaderのインスタンスを使って、  
BufferedReaderのインスタンスを作る

readLineメソッドで1行ずつ読み込む

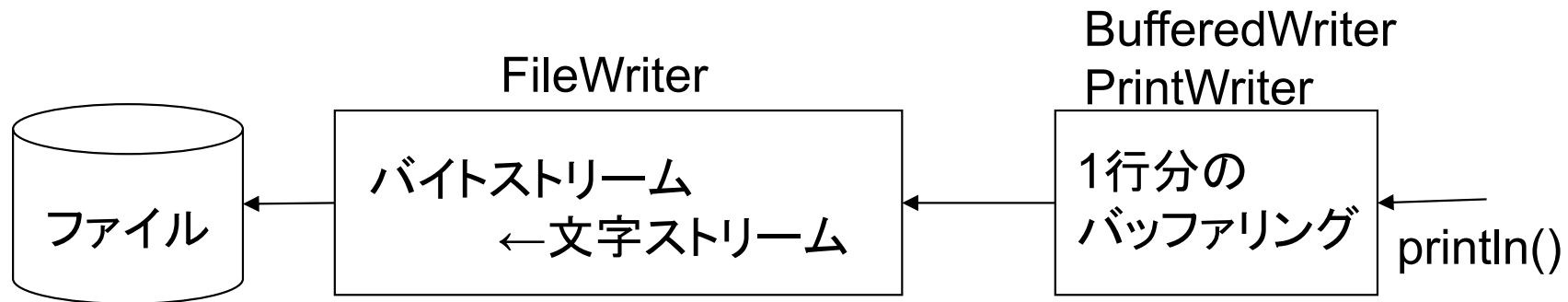
closeメソッドで  
ファイルを閉じる

ファイルの最後まできたらreadLineメソッド  
はnullを返すので、ここでwhile文が止まる

# ファイル読み込みのプログラム例 (Cont.)

- ファイル読み込みのためのFileReaderクラスがFileNotFoundExceptionという例外を投げる。この例外は例外処理を強制されるので、例外処理を書かないとコンパイルエラーとなる。
- 同様に、readLineというメソッドは、IOExceptionという、例外処理を書かなければいけない例外のため、こちらもキャッチする必要がある。

# ファイルの書き込み



```
PrintWriter writer = new PrintWriter(new BufferedWriter(new FileWriter(filename)));
```

- BufferedWriterは書き込みへのバッファリングを行うクラスである。
- PrintWriterクラスはprintlnメソッドを持っているクラスである。  
(つまり, System.outで実装されているPrintStreamクラスと同じメソッドが実装されている)
- ファイルへの書き込みを行うFileWriterクラスを使う。

# ファイル書き込みの方法

ファイル書き込みためのオブジェクトとメソッド

作成するオブジェクト

`PrintWriter writer =`

`new PrintWriter(new BufferedWriter(new FileWriter(filename)));`

文字列をファイルに書き込むメソッド

`writer.println(line);`

# ファイル書き込みのプログラム例

```
import java.io.*;
public class WriteFile1 {
    public static void main(String[] args) {
        String filename = args[0];
        try{
            //標準入力からの読み取り
            BufferedReader reader = new BufferedReader(
                new InputStreamReader(System.in));
            //ファイルへの書き出し
            PrintWriter writer = new PrintWriter(
                new BufferedWriter(new FileWriter(filename)));
            String line;
            while(!(line=reader.readLine()).equals("EOF")) {
                writer.println(line);
            }
            reader.close(); writer.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

最後にcloseメソッドを忘れずに、ファイルを閉じるのを忘れると、書き込んだはずのものがファイルに書き込まれていない、といったことが起こる。

# ファイル操作

ファイルに関するさまざまな操作はFileクラスで実装されている.

操作	メソッド名	戻り値
ファイルの削除	<code>delete()</code>	<code>boolean</code>
ファイル名変更	<code>renameTo(ファイルオブジェクト)</code>	<code>boolean</code>
ファイルの存在確認	<code>exists()</code>	<code>boolean</code>
ディレクトリ内の一覧を得る	<code>list()</code>	<code>String[]</code>
ディレクトリの作成	<code>mkdirs()</code>	<code>boolean</code>



# ファイルの削除, 名前変更, 存在確認

```
import java.io.*;
public class FileManipulation {
    public static void main(String[] args) {
        String filename = args[0];
        String newfilename = args[1];
        File file = new File(filename); //ファイルの作成
        File nfile = new File(newfilename); //ファイルの作成
        //ファイル名変更
        if (file.renameTo(nfile)) {
            System.out.println(filename+"を"+newfilename+
                               "に変更しました");
        } else {
            System.out.println(filename+"を"+newfilename+
                               "に変更できません");
        }
    }
    . . . つづく
```

# ファイルの削除, 名前変更, 存在確認 (Cont.)

... つづき

```
//ファイル削除
if (file.delete()) {
    System.out.println(filename+"を削除しました");
} else {
    System.out.println(filename+"を削除できません");
}
//ファイルの存在確認
if (!file.exists()) {
    System.out.println(filename+"は存在しません");
} else {
    System.out.println(filename+"は既に存在します");
}
}
```

# Javaでの文字コードの扱い

- Javaの内部では, Unicodeという16ビットの文字コードを使っている. 従って, Javaのchar型も16ビットとなっている.
- Javaのソースファイルの文字コードはOSによって異なっている.
  - (例) Windows: Shift JIS, Unix: 日本語EUC
- javaのコンパイラ(javac)がコンパイルを始める前に, OSに依存した文字コードからUnicodeに自動的に変換を行う. 従って, ソースファイルがUnicodeでなくても問題ない.

# クラスライブラリのマニュアルを読む

- FileReaderについて, クラスライブラリを調べてみる.

```
java.io.Reader (implements java.io.Closeable, java.lang.Readable)
    java.io.BufferedReader
        java.io.LineNumberReader
    java.io.CharArrayReader
    java.io.FilterReader
        java.io.PushbackReader
    java.io.InputStreamReader
        java.io.FileReader
    java.io.PipedReader
    java.io.StringReader
```

java.io.BufferedReaderをクリックしてみると

java.io

## クラス **BufferedReader**

[java.lang.Object](#)

[java.io.Reader](#)

**java.io.BufferedReader**

すべての実装されたインタフェース:

[Closeable](#), [Readable](#)

直系の既知のサブクラス:

[LineNumberReader](#)

### ☆クラスの説明

```
public class BufferedReader  
extends Reader
```

### ☆コンストラクタの概要

#### コンストラクタの概要

[BufferedReader](#)([Reader](#) in)

デフォルトサイズのバッファでバッファリングされた、文字型入力ストリームを作成します。

[BufferedReader](#)([Reader](#) in, int sz)

指定されたサイズのバッファでバッファリングされた、文字型入力ストリームを作成します。

## ☆メソッドの概要

戻り値の型

引数の型, 機能の説明

<a href="#">String</a>	<a href="#">readLine()</a> 1 行のテキストを読み込みます。
Long	<a href="#">skip</a> (long n) 文字をスキップします。

## ☆readLine()の詳細

readLine

public [String](#) **readLine()** throws [IOException](#)

1 行のテキストを読み込みます。1 行の終端は、改行（「**¥n**」）か、復帰（「**¥r**」）、または復行とそれに続く改行のどれかで認識されます。

戻り値:

行の内容を含む文字列、ただし行の終端文字は含めない。ストリームの終わりに達している場合は **null**

例外:

[IOException](#) - 入出力エラーが発生した場合

# Splitメソッド

- Stringクラスに実装されているメソッド
- 文字列を指定された正規表現に一致した位置で分割する.
- 分割した結果は文字列の配列として返される

```
String[] result = "this,is,a,test".split(",");  
for (int x=0; x<result.length; x++)  
    System.out.println(result[x]);
```