Week6

Week6 's Menu

- パッケージ:package
- ラッパークラス:wrapper class
- 例外:Exception
- 標準入力:Standard Input
- スレッド:Thread

Week6 's Menu

- パッケージ:package
- ラッパークラス:wrapper class
- 例外:Exception
- 標準入力:Standard Input
- スレッド:Thread

パッケージ:Package

クラスやインターフェースをひとまとめにしたもの

パッケージ	クラス１
	クラス２
	インターフェース

パッケージ:Package

- C言語のヘッダーファイル（みたいなもの）
- import宣言でパッケージを使用する
- 標準で多種多様なパッケージがある
- 機能ごとにパッケージを作ると集団開発で便利

mathパッケージ :数学の関数が入っている
utilパッケージ: データ構造や便利なクラスが入っている
ioパッケージ:入出力が行えるクラスが入っている

パッケージ:Package

パッケージの作成方法

1. プロジェクトを右クリック
2. 「新規」→「パッケージ」をクリック
3. 「package-info.javaを作成する」の
   チェックを外す
4. 「packageA」という名前を付けて
   「完了」をクリック

パッケージ:Package

ClassA.java(テキスト２ページ)

```
package packageA;
public class ClassA {
    public void print(){
        System.out.println("This is classA in packageA.");
    }
}
```

パッケージ:Package

メイン関数(テキスト２ページ)

```
public static void main(String[] args) {
        ClassA classA = new ClassA();
        classA.print();
    }
```

パッケージ:Package

実行しようとすると…
エラーが発生する…

何故？

- packageAを使うと宣言していないから
- メイン関数はデフォルトパッケージ内に作成されているので宣言なしにpackageA内のクラスにアクセス出来ない

パッケージ:Package

パッケージのimport宣言

import パッケージ;

メイン関数(テキスト３ページ)

```
package week6;
import packageA.ClassA;    <-これを追加する


public class week6 {
```

パッケージ:Package

まだあるパッケージのメリット

もしかしたら
集団開発した時に同じ名前でクラスを定義してしまうかも（競合の可能性）

パッケージ:Package

まだあるパッケージのメリット

もしかしたら
集団開発した時に同じ名前でクラスを
定義してしまうかも（競合の可能性）

異なるパッケージ名同じクラス名を
同時使用することもできる！

パッケージ:Package

実際に同名クラスを
呼び出してみよう

- テキスト２ページを見てパッケージを作成
- 名前はpackageB

パッケージ:Package

まずは、パッケージ作成から

classB.java(テキスト４ページ)

```
package packageB;
public class ClassA {
    public void print(){
        System.out.println("This is ClassA in packageB");
    }
}
```

パッケージ:Package

次に、メインメソッドを変更

classB.java(テキスト４ページ)
※import宣言は必要ナシ

```
    public static void main(String[] args) {
        packageA.ClassA classA = new packageA.ClassA();
        packageB.ClassA classB = new packageB.ClassA();
        classA.print();
        classB.print();
    }
```

パッケージ:Package

２つの同名クラスにアクセスできた！

Output

This is classA in packageA .
This is classA in packageB .

パッケージ:Package

パッケージ作成時の注意点

アクセス修飾子を忘れずに！

アクセス修飾子	機能
private	同じクラスからのみアクセス可能
なし	同じパッケージからアクセス可能
protected	同じパッケージ、継承先からアクセス可能
public	全クラスからアクセス可能

Week6 's Menu

- パッケージ:package
- ラッパークラス:wrapper class
- 例外:Exception
- 標準入力:Standard Input
- スレッド:Thread

ラッパークラス
:Wrapper class

プリミティブ型をオブジェクト型に変更

ラッパークラス
:Wrapper class

プリミティブ型をオブジェクト型に変更

で？

あと、パッケージの時と画像変わっていない…

ラッパークラス
:Wrapper class

プリミティブ型をオブジェクト型に変更

プリミティブ型とは、
int,charなどのクラスではない型のこと

オブジェクト型（クラス型）とは、
クラスで定義した型のこと

ラッパークラス
:Wrapper class

つまり、ラッパークラスとは

変数をクラスとして扱えるように
ラッピングするクラス

ラッパークラス
:Wrapper class

ラッパークラス利用法①:型変換

```
// 文字列の数字を整数値に変換
int num = Integer.parseInt("1000");
// 実数値を文字列に変換
String num = Double.toString(123.456);
```

ラッパークラス :Wrapper class

ラッパークラス利用法②:ArrayList

```
package week6;
import java.util.ArrayList;
public class week6 {
    public static void main(String[] args) {
        ArrayList<Integer> arraylist = new ArrayList<Integer>();
        for(int i=0;i<10;i++)arraylist.add(i);
        for(int i :arraylist)System.out.println(i);
    }
}
```

Week6 's Menu

- パッケージ:package
- ラッパークラス:wrapper class
- 例外:Exception
- 標準入力:Standard Input
- スレッド:Thread

例外:Exception

例外とは

- プログラムの実行中にエラーが起きること
- コンパイル時にエラーがでないので厄介
- 例外の種類によっては強制終了する

例外:Exception

例外の具体例

- ArithmeticException:ゼロ除算等で発生
- NullPointerException
  :存在しない値(null)を参照しようとすると発生
- ArrayIndexOutOfBoundsException
  :配列の範囲外に参照しようとすると発生

例外:Exception

ぬるぽ(NullPointerException)が起きるソース

```
package week6;
public class week6 {
	public static void main(String[] args) {
		String str = null;
		System.out.println(str.length());
	}
}
```

例外:Exception

例外が発生したらどうすればいいの？

例外処理をする

例外:Exception

例外処理とは？

例外が発生した時だけ行われる処理を書くこと

これを書けば
　例外が発生しても
　　強制終了しない

例外:Exception

例外処理の書き方

try-catch-finallyを使う

例外:Exception

```
try{
例外が発生しそうなプログラムの処理を記述する
}
catch(発生した時に対策したい例外を指定する){
例外は発生した時の処理を記述する
}
finally{
    例外の発生に関係なく行う処理を記述する
    省略可能
}
```

例外:Exception

サンプルソースP.７を実行してみよう

例外:Exception

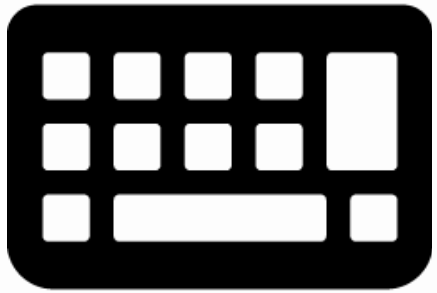
細かい注意

- throwsキーワードを使えば、例外を発生させるメソッドを作れる
- 例外処理しないと使えないクラスやメソッドが存在する

Week6 's Menu

- パッケージ:package
- ラッパークラス:wrapper class
- 例外:Exception
- 標準入力:Standard Input
- スレッド:Thread

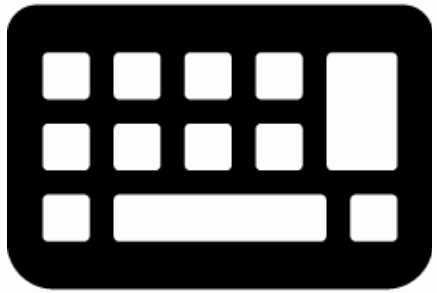
標準入出力
:Standard Input



いま標準入出力をやる理由

ストリームを理解する必要があるから

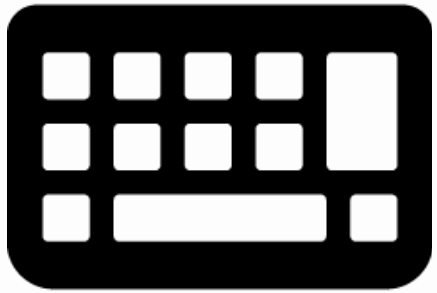
標準入出力
:Standard Input



ストリームとは

英語-> stream:流れ
Java -> stream:データの流れ

標準入出力
:Standard Input



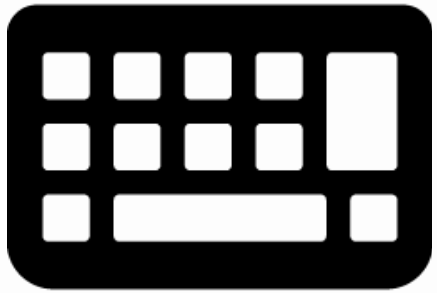
入力に使うストリーム

InputStream: 入力をバイトデータで受け取る

InputStreamReader: バイトストリームから
文字ストリームに変換する

BufferedReader: 文字をバッファリングして
テキストを効率よく読み込む

標準入出力
:Standard Input

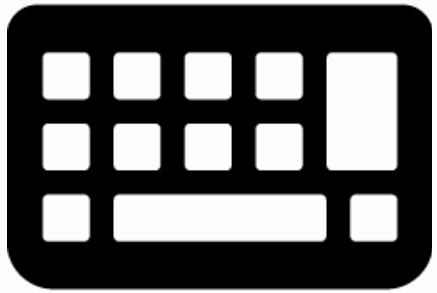


標準入力とは

InputStream->InputStreamReader->BufferedReaderすること

入力をバイトデータで受け取り
バイトデータを文字に変換し
文字をバッファし文字列にする
こと

標準入出力
:Standard Input



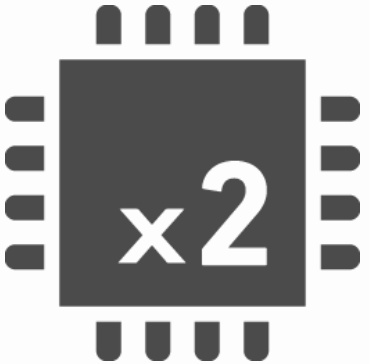
サンプルソースを実行してみよう(P.9)

※例外処理をしないと標準入力はできない
※コンパイルが通らない

Week6 's Menu

- パッケージ:package
- ラッパークラス:wrapper class
- 例外:Exception
- 標準入力:Standard Input
- スレッド:Thread

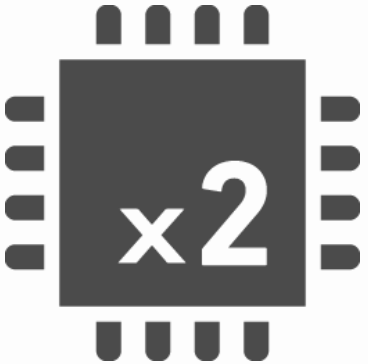
スレッド:Thread



スレッドとは

- プログラムの処理の単位
- 一つのスレッドをシングルスレッドという
- 普通のプログラムはシングルスレッド
- 複数スレッドの場合は、マルチスレッド

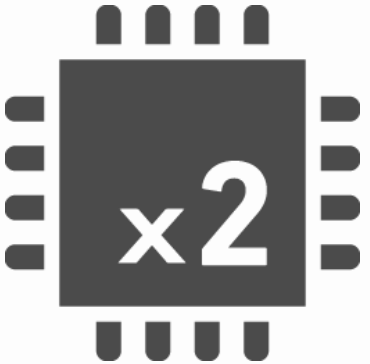
スレッド:Thread



マルチスレッドプログラミングとは

- 同時並行で異なる処理をさせる方法
- Javaではマルチスレッドプログラミング可能
  （擬似的なものだけど）

スレッド:Thread

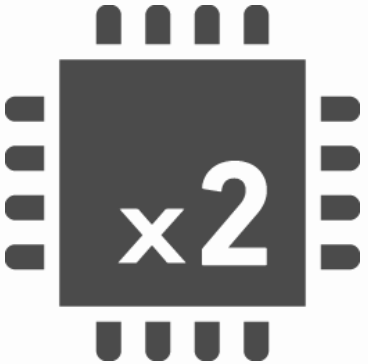


マルチスレッドプログラミング具体例

ゲームはマルチスレッドで動いている。
例えば、Loading画面なら

- 画面上にLoadingと表示するスレッド
- メモリにデータを格納するスレッド

スレッド:Thread

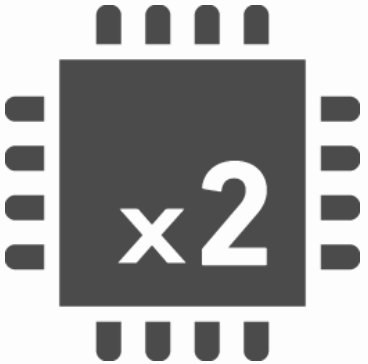


Javaでのマルチスレッド

Threadクラスを継承することで
　マルチスレッドプログラムが可能

こういうのがオブジェクト指向の強みだよね

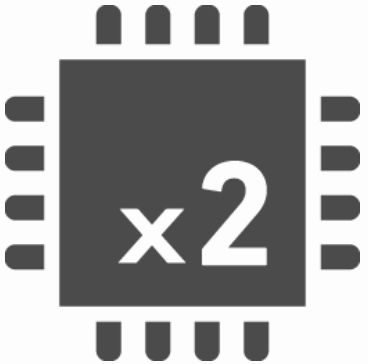
スレッド:Thread



サンプルソースを実行してみよう(P.10)

1.新しいクラスを作成（名前はProcess）
2.run()メソッドをオーバライド
3.メインメソッド内でstart()メソッドを使って、
　マルチスレッド開始

スレッド:Thread



サンプルソースを実行してみよう(P.10)

process1.start()メソッドの終了を待たずに
process2.start()が動いているのが確認できる

演習問題

問題１

キーボードから、数字を入力し、
その数字の二倍を出力するプログラムを作成せよ
(double型で)。

演習問題

問題２

キーボードから、
２つの数字と四則演算の符号を入力し、
その入力をもとに四則演算を行うプログラムを
作成せよ（double型）。
例外処理はしなくてもいい。