

# 上級プログラミング1(第9回)

---

工学部 情報工学科  
木村昌臣

# 今日のテーマ

---

- 入出力に関わるプログラミング
  - 例外処理
  - マルチスレッドプログラミング
-

# 入出力に関わるプログラミング

---

# TXTファイルからデータを読み出そう

```
import java.io.*;
class Smp10501 {
    public static void main(String[] args){
        BufferedReader bf=null;  String strData="";
        try{    FileInputStream fI= new FileInputStream("Sample.txt");
                InputStreamReader iS=new InputStreamReader(fI,"SJIS");
                bf=new BufferedReader(iS);
                while((strData=bf.readLine())!=null){
                    System.out.println(strData);
                }
            }catch(Exception e){
                e.printStackTrace();
            }finally{
                try {
                    bf.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

## 先週の復習(import文)

```
import java.io.*;
```

ファイル入出力用のクラスを使うのでそのパッケージをインポートしておく

```
class Smpl0501 {  
    public static void main(String[] args){
```

```
        . . . . .
```

ファイル入出処理の記述

```
        . . . . .
```

```
    }
```

```
}
```

# テキストファイルの読み出し

```
String line="";
```

バイナリデータ  
として読み出し

```
FileInputStream fi
```

```
= new FileInputStream("S
```

文字データ  
として読み出し

```
InputStreamReader is
```

```
=new InputStreamReader(fi,"SJIS");
```

```
BufferedReader br
```

```
=new BufferedReader(is);
```

```
while((line=br.readLine())!=null){
```

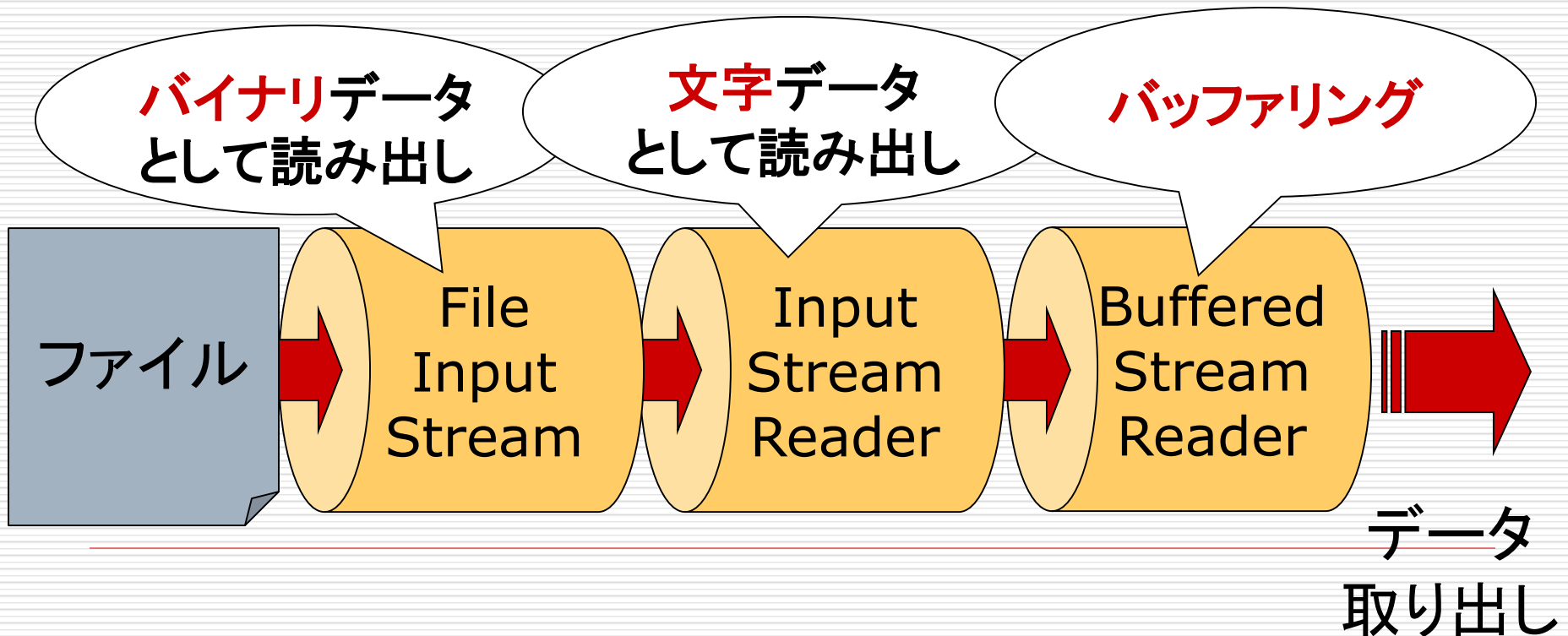
```
System.out.println(line);
```

バッファリング

```
}
```

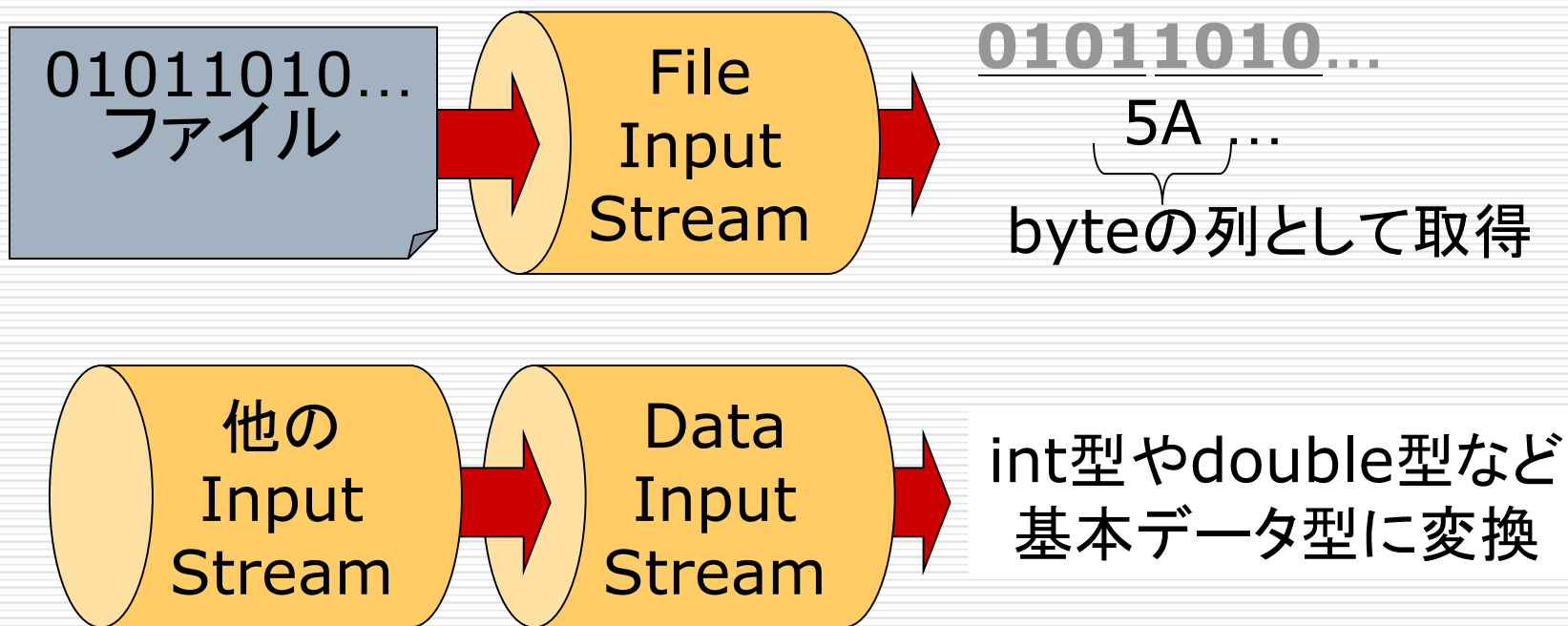
# テキストファイルの読み込み関係のクラス

テキストファイルからデータを読み込むときには、  
通常、三段構えで行う



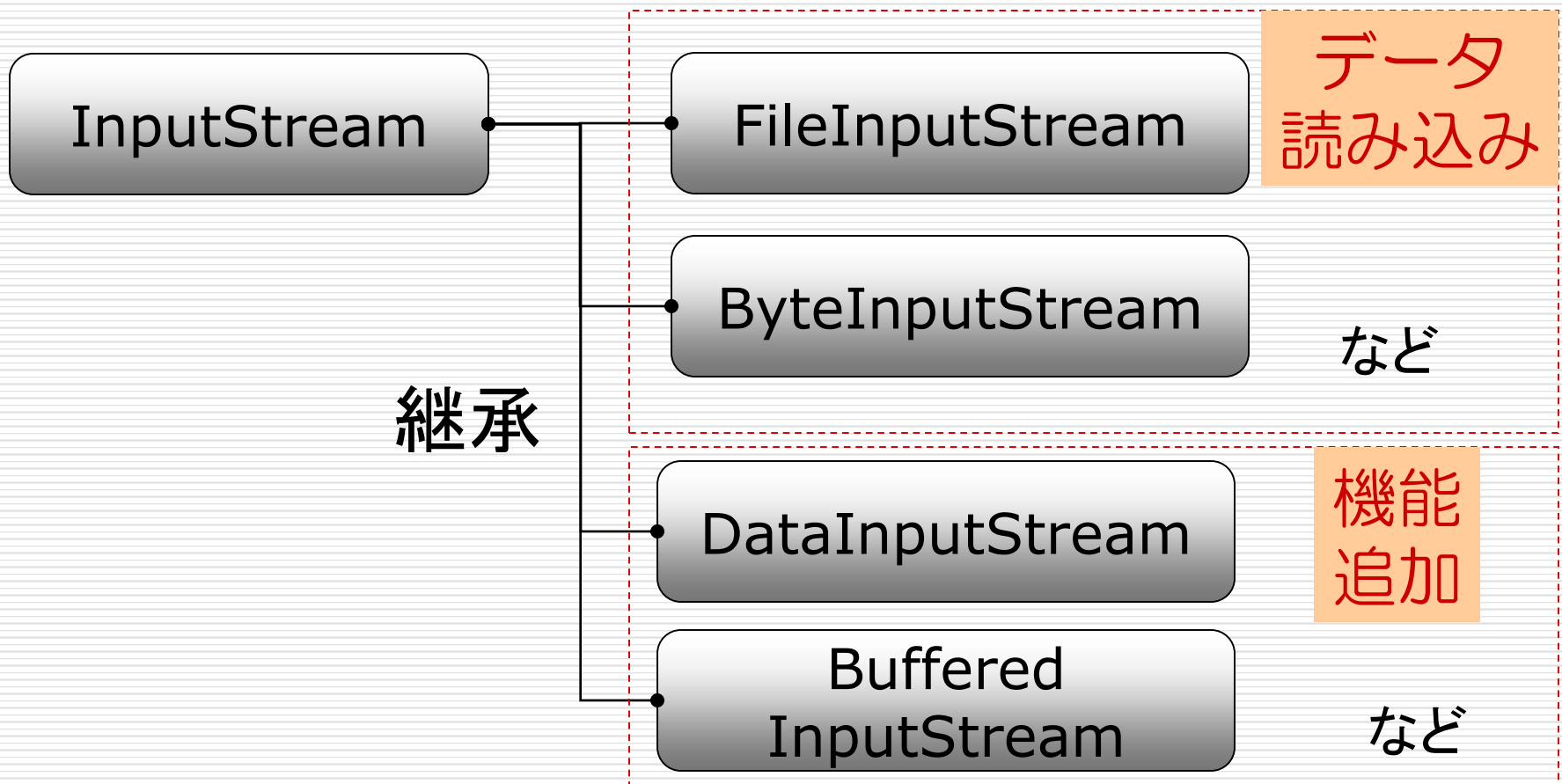
# バイト入力ストリーム

byte単位でプログラムに渡すデータの流れ



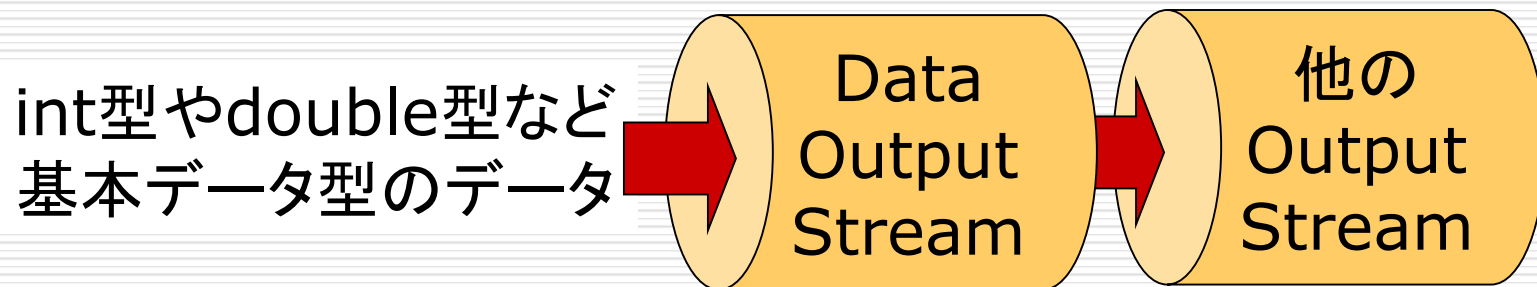
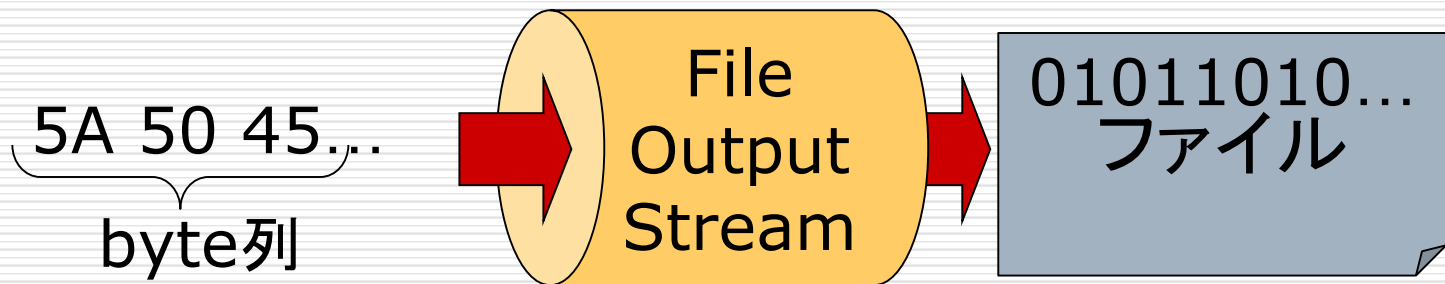


# バイト入力ストリーム

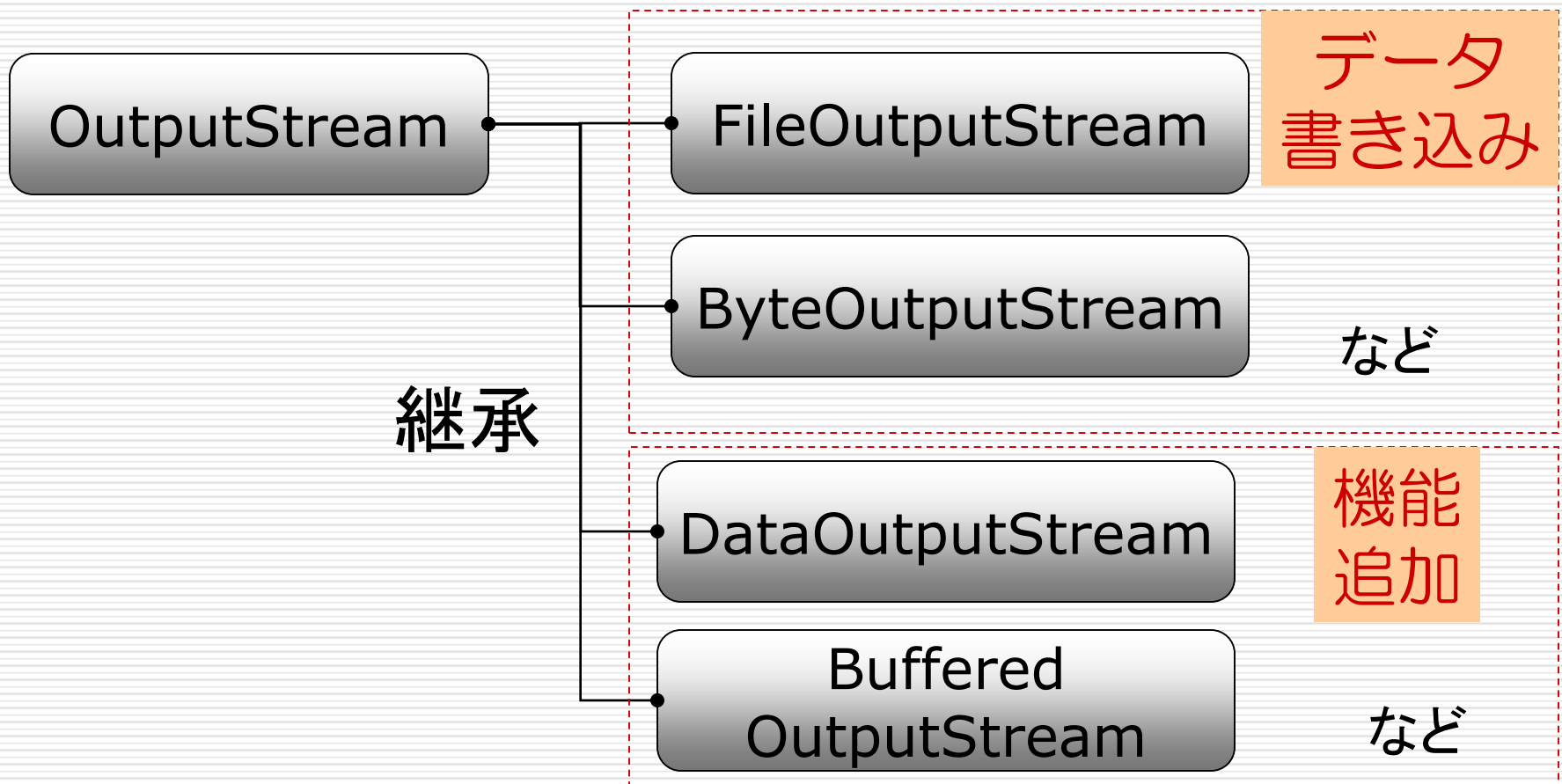


# バイト出力ストリーム

## byte単位で出力するデータの流れ

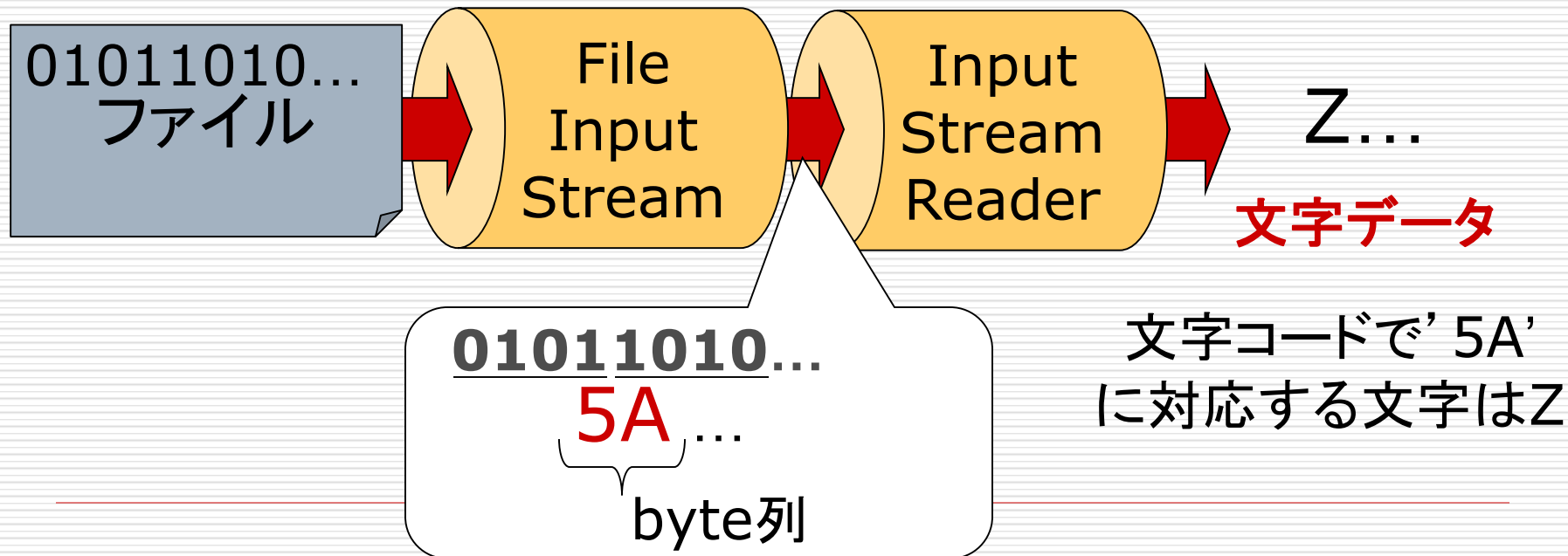


# バイト出力ストリーム



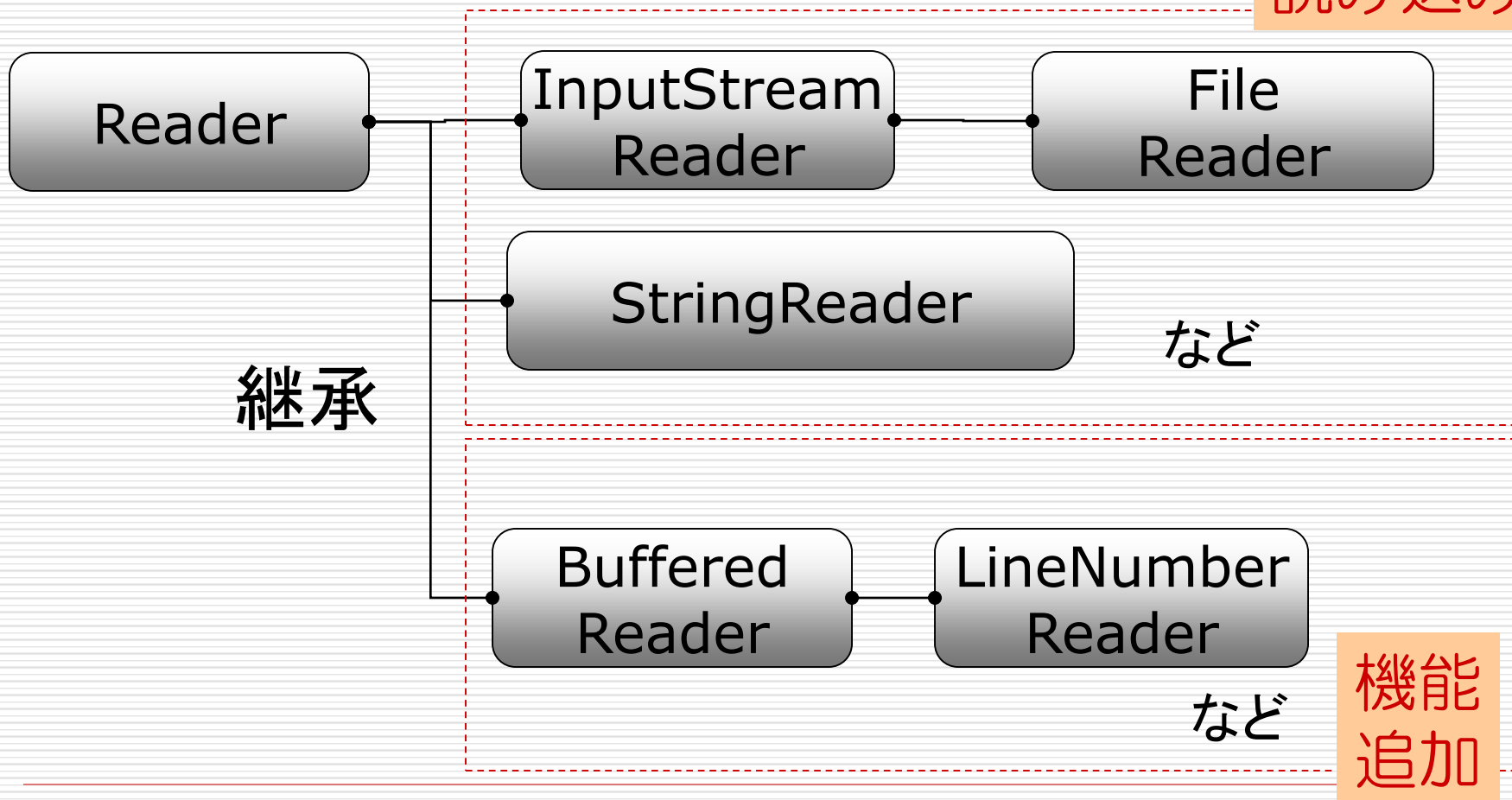
# 文字ストリーム

文字単位でプログラムに渡すデータの流れ



# Readerクラス

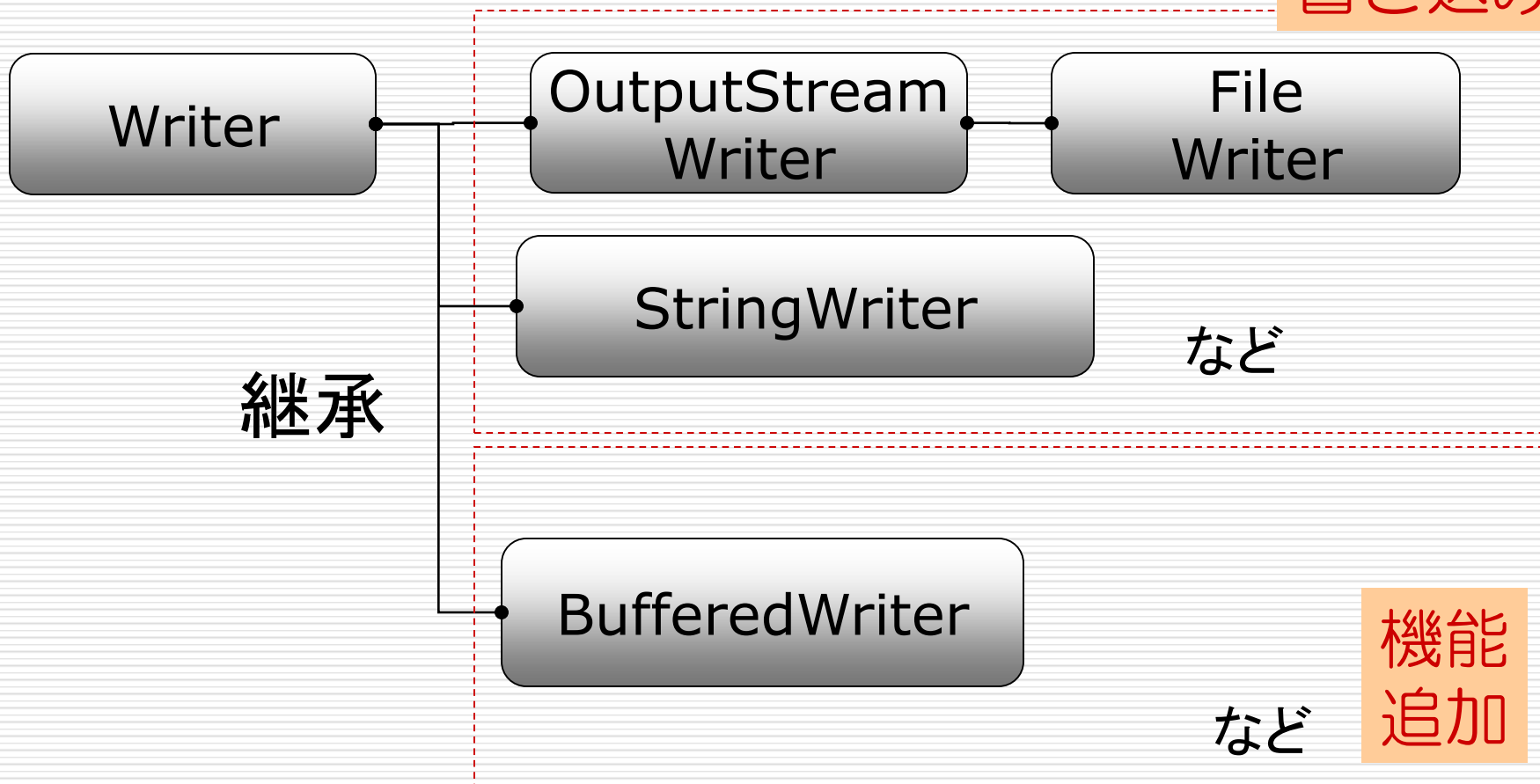
データ  
読み込み



※クラス名はスペースの都合上改行している

# Writerクラス

データ  
書き込み



※クラス名はスペースの都合上改行している

# 注意

---

ファイルを使ったら閉じるべし

**BufferedReaderをcloseするとiSやfIも内部的にcloseされる**

```
FileInputStream fI= new  FileInputStream("...");
InputStreamReader iS=new InputStreamReader(fI,"SJIS");
BufferedReader bf=new BufferedReader(iS);
    while((strData=bf.readLine())!=null){
        System.out.println(strData);
    }
bf.close();
```

---

# まとめ

---

- バイト列での入出力部分とそれを文字列として扱う部分を、プログラムに必要な機能に応じて組み合わせる
    - バイト列入出力部分: InputStream, OutputStreamとその子クラス
    - 文字列取り扱い部分: Reader, Writerの子クラス
-



# 例外处理

---

# TXTファイルからデータを読み出そう(再掲)

```
import java.io.*;
class Smpl0501 {
    public static void main(String[] args){
        BufferedReader bf=null;  String strData="";
        try{    FileInputStream fI= new FileInputStream("Sample.txt");
                InputStreamReader iS=new InputStreamReader(fI,"SJIS");
                bf=new BufferedReader(iS);
                while((strData=bf.readLine())!=null){
                    System.out.println(strData);
                }
            }catch(Exception e){
                e.printStackTrace();
            }finally{
                try {
                    bf.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

# ・・・こんなときどうしよう？

---

ファイルが  
ないよ！

ファイルはあるけど  
読み取れない！

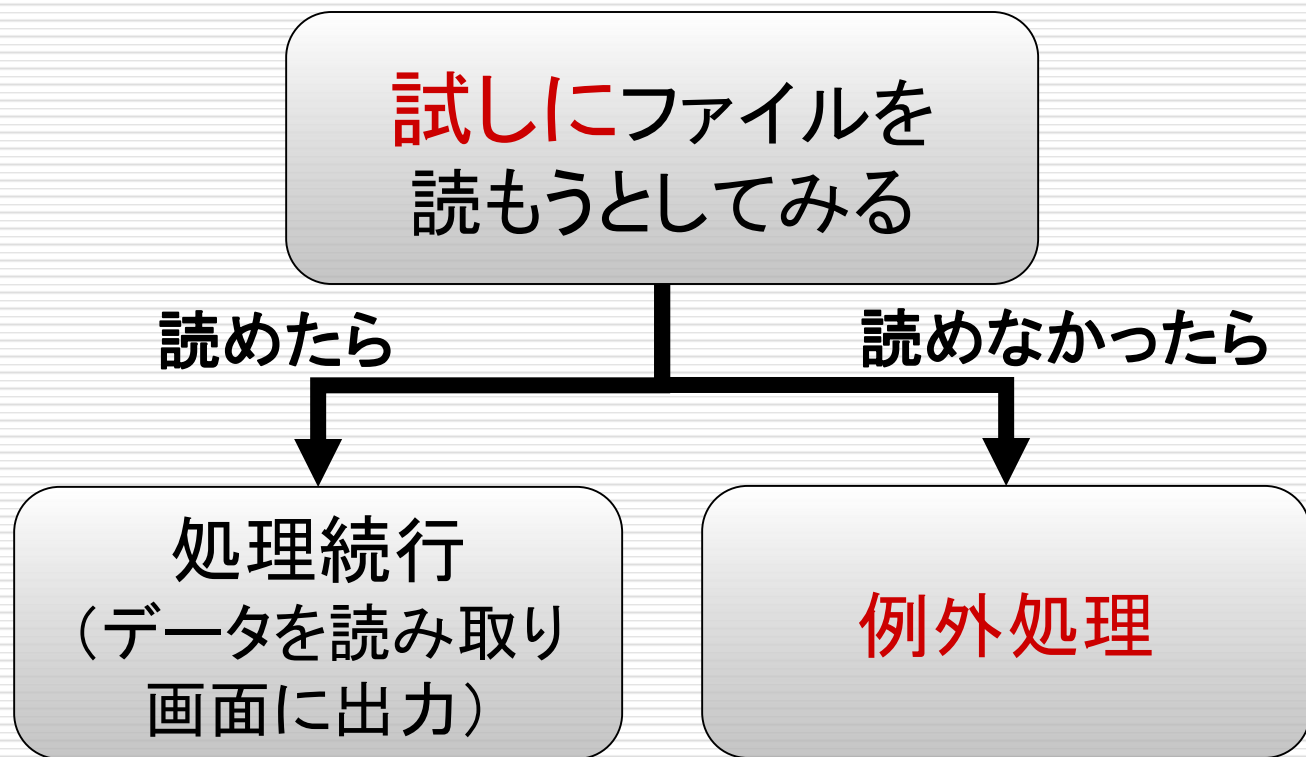


などなど

---

# こんなことができれば便利

---



うまくいかなかったこと(ファイルが開けなかった等)  
を**例外**という

# TXTファイルからデータを読み出そう(再掲)

```
public static void main(String[] args){  
    String strData="";    BufferedReader bf=null;  
  
    try{  
        FileInputStream fI  
            = new FileInputStream("Sample.txt");  
        InputStreamReader iS  
            =new InputStreamReader(fI,"SJIS");  
        bf=new BufferedReader(iS);  
        while((strData=bf.readLine())!=null){  
            System.out.println(strData);  
        }  
    }  
    catch(Exception e){  
        e.printStackTrace();  
    }  
    .....(略).....  
}
```

試しに

もし例外が発生したなら、その例外を捕えて  
{ }内の処理を行う

# 複数の例外に対して別の処理を行わせることも可能

```
try{
```

```
    ....ファイル読み出し処理....
```

```
}
```

```
catch(FileNotFoundException e){  
    System.out.println("そんなファイルはありません。");  
    e.printStackTrace();  
}
```

```
catch(IOException e){  
    System.out.println("入出力エラーが発生しました。");  
    e.printStackTrace();  
}
```

```
.....(略).....
```

```
}
```

# Finally文

- どんな例外処理をしても、正常処理であっても必ず実施する処理を記述する

```
try{ ...  
    bf=new BufferedReader(iS);  
    ...  
} catch(FileNotFoundException e){  
    System.out.println("そんなファイルはない。");  
} catch(IOException e){  
    System.out.println("入出力エラーが発生しました。");  
}finally{ ...  
    bf.close();  
    ...  
}
```

**必ず**  
入力ストリーム  
を閉じる

# 例外とエラーの違い

---

エラー

ハードウェア障害  
OSのバグによるエラー

システムエラー

例外

ゼロで除算している  
文法が間違っている  
ファイルがない  
入出力がうまくいかない  
など

プログラムエラー



# スレッド

---

# 普通のプログラム

---

```
double x=0.25;
```

```
x=x*(1-x);
```

```
x=Math.exp(-x);
```

```
System.out.println("結果は"+x);
```

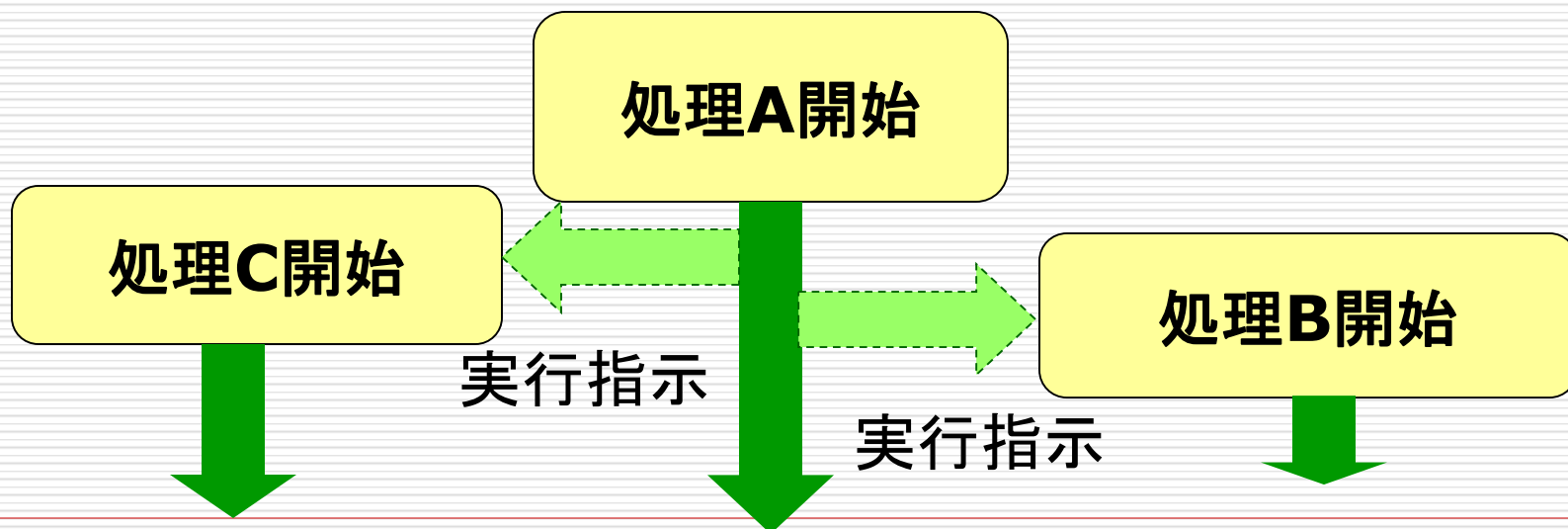
一連の  
処理が  
順に  
実行  
される

通常  
処理の流れは  
1つ

# マルチスレッド

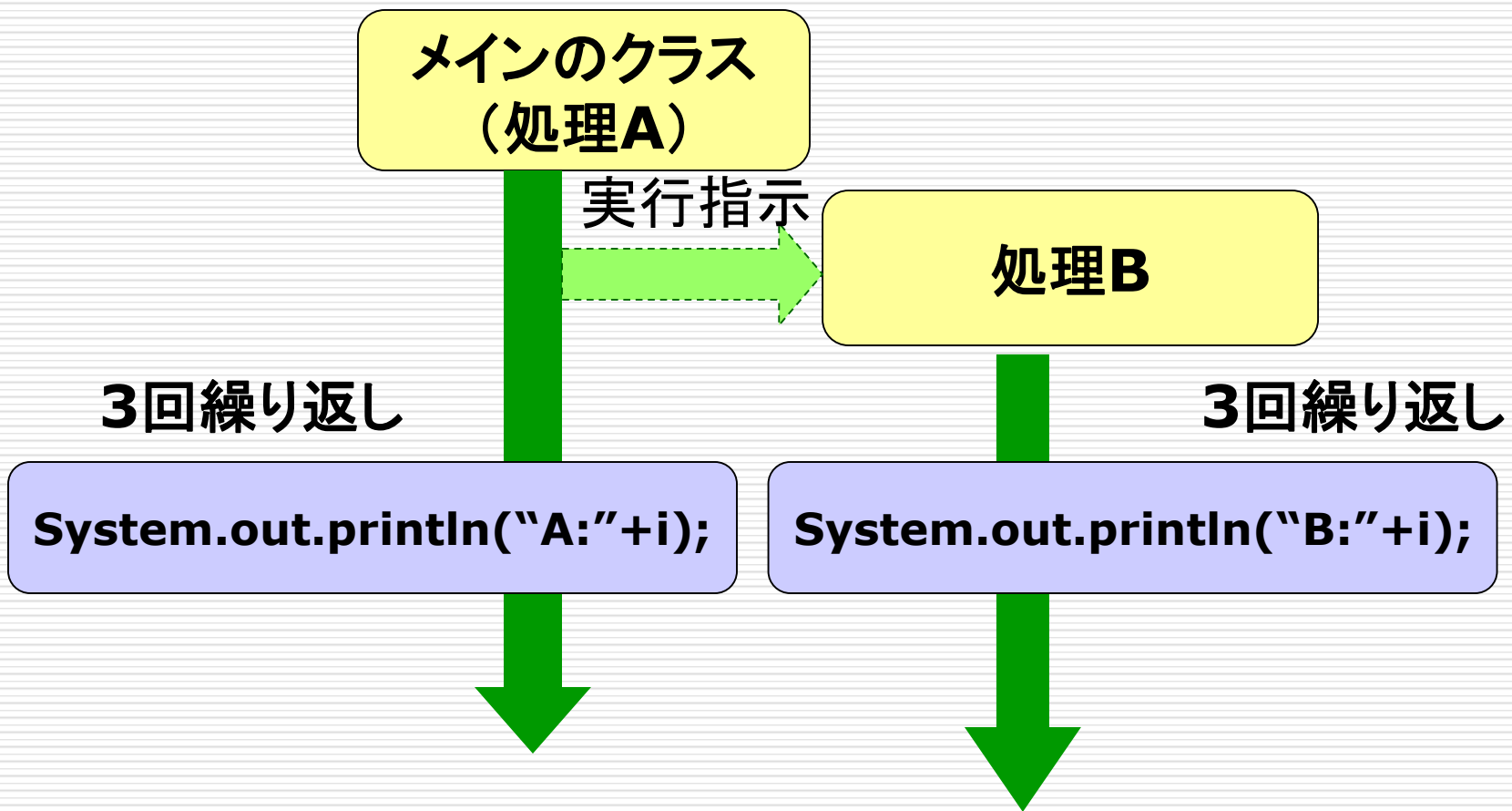
---

- 1つのプログラム(正確にはプロセス)の中で、処理の流れ(**スレッド**)を同時に複数実行するしくみ
- 複数のスレッドが同時に複数実行されるプログラムを**マルチスレッドプログラム**と呼ぶ(普通のプログラムはシングルスレッドプログラム)



# マルチスレッドプログラムの簡単な例

---



# ソースプログラム(1) 処理B

---

```
class ThreadSmpl extends Thread {  
    public void run(){  
        for(int i=0; i<3; i++){  
            System.out.println("B:"+i);  
        }  
    }  
}
```

- ① Threadクラスを継承する
- ② run()メソッドをオーバーライドし、同時実行させる処理を記述する。  
run()メソッドはstart()メソッドから呼び出される

## ソースプログラム(2) 処理A

---

```
class ThreadMain {  
    public static void main(String[] args) {  
        ThreadSmpl shoriB = new ThreadSmpl();  
        shoriB.start();  
        for(int i=0; i<3; i++){  
            System.out.println("A:"+i);  
        }  
        System.out.println("Done(main).");  
    }  
}
```

処理Bを新規スレッドとしてstart()メソッドで実行後、A:・・・を表示

# 結果

---

```
A:0  
A:1  
A:2  
Done(main).  
B:0  
B:1  
B:2
```

処理Aと処理Bの実行順は  
環境に依存するが、この結果では  
**Bの処理の前に  
Aの処理が終わっている  
ことに注意**

シングルスレッドでは、  
処理Bの実行前に  
処理Aが終わることはありえない

処理A(ThreadMain)のmainメソッドも  
Threadクラス(ThreadSmpl)のオブジェクトのrunメソッドも  
**同等のスレッドとして扱われる**

# 結果

---

**Done.**

**B:0**

**C:0**

**B:1**

**B:2**

**C:1**

**C:2**

} 同時

BとCのスレッドは同時に  
始まり、Bは1秒おき、  
Cは1.5秒おきに  
画面に表示する

それぞれのスレッドが独立に動いている  
ことがわかる