

3.3 Java 補足

- 修飾子
- ファイル入出力
- スレッド
- 正規表現
- その他

目次

- 1 修飾子
- 2 ファイル入出力
- 3 スレッド
- 4 正規表現
- 5 その他

修飾子

- **修飾子**^[Modifier]は変数、メソッド、クラス（またはブロック）を定義する際に、特定の特性を追加するために使います。
- 例えば、以前学んだ public と private は**アクセス修飾子**と、static は**静的修飾子**と呼ばれる。宣言の前にこれらの修飾子を書くことで、その特徴を示すことができます。
- Java でよく使われる修飾子は以下の通り：
 - アクセス修飾子：private、protected、public
 - abstract
 - static
 - final

Note

同じ修飾子であっても、修飾する対象によって異なる効果を持つ場合があります。

abstract 修飾子

- **abstract** 修飾子は修飾する対象が「**抽象的**」であることを表現するために用います。
- abstract 修飾子がクラスを修飾する場合、そのクラスは**抽象クラス**で、インスタンス化できず、いくつかのメソッドはサブクラスで実装する必要があることを表します。
- abstract 修飾子がメソッドを修飾する場合、そのメソッドが実装を持たない**抽象メソッド**であり、オーバーライドするサブクラスが必要であることを表します。
- abstract は変数を修飾**できません**。

static 修飾子

- **static** 修飾子は修飾する対象が「**静的**」であることを表現するために用います。
- static 修飾子がメンバ変数、メソッド、内部クラスを修飾する場合、それらが特定のオブジェクトではなく、**クラスに属する**静的な変数、メソッド、クラスであることを表します。
- static 修飾子はローカル変数には**使えません**。
- static 修飾子で修飾されたクラス内のブロックはクラスが最初にアクセスされたときに実行されます。

final 修飾子

- **final** 修飾子は、修飾する対象が「**不変**」であることを表現するために用います。
- final 修飾子で変数を修飾した場合、**1 回**のみ代入が可能です。再度代入しようとする、構文エラーが発生します。
 - しかし、ローカル変数、メンバ変数、静的変数に作用するときの効果には細かい違いがあるので、後ほど説明します。
- final 修飾子がメソッドを修飾する場合、そのメソッドはサブクラスで**オーバーライドできません**。
 - final は抽象的なメソッドを修飾できません。
- final 修飾子がクラスを修飾している場合、そのクラスを**継承できません**。

final なローカル変数

- final になったローカル変数は、宣言時の初期化を含めて、**1 回**しか代入できません。
 - 宣言時に初期化された場合、それ以降は代入できません。
 - 宣言時に初期化されていなら、以降は 1 回のみ代入できます。
- 既に値が割り当てられている final 変数を変更しようとする
と、構文エラーが発生します：

```
1 final int a = 10;  
2 a = 20; // => エラー  
3 final int b;  
4 b = 20;  
5 b = 3; // => エラー
```

final なインスタンス変数

- final になったインスタンス変数は、ローカル変数のように 1 回しか代入できないことに加え、**コンストラクタの終了前に代入しなければならない**という特別な条件があります。
- つまり、初期化時に値を決めるか、**すべてのコンストラクタ**で値を代入する必要があります。そうしなければエラーが発生します：

```
1 class A {  
2     final int a;  
3  
4     A() {  
5         a = 0; // このラインを抜けばエラーになる  
6     }  
7 }
```


final なクラス変数

- final なクラス変数は変更できないことに加え、初期化時（または静的ブロック内）に値を代入する必要があります。
- Java では、final static の変数は、よく**定数**[Constant]を表すのに使われます：

```
1 class Math {  
2     public final static PI = 3.14159;  
3     public final static E = 2.71828;  
4 }
```

Note

定数は大文字のスネークケース
にネーミングすべき。

その他の修飾子

修飾符	修飾対象	効果
strictfp	クラス、メソッド	浮動小数点数は、IEEE754 標準で厳密に計算
transient	インスタンス変数	シリアライズ化の対象から除く
synchronized	クラス、メソッド、ブロック	コードが複数のスレッドで同時に実行されることはできない
volatile	インスタンス変数	変数はキャッシュされない。
native	メソッド	メソッドは他の言語で実装されている
default	インタフェース内のメソッド	デフォルトの実装を定義している

修飾子の順番

- 宣言の前に複数の修飾子を書くとき、**最初はアクセス修飾子**を書きます。他の修飾子はどんな順番でも構いませんが、同じ修飾子を何度も書くことができません。

Example

```
private final static int a;  
private static final int b;  
public strictfp abstract void c();
```

Example

```
static public int a;  
final static final int b;
```

Q&A

目次

- 1 修飾子
- 2 **ファイル入出力**
- 3 スレッド
- 4 正規表現
- 5 その他

ファイルの読み出しと書き込み

- 前にコンソールでの出力（println() メソッド）を学びました。しかし、実際のアプリケーションでは、ユーザーがコンソールを使うことはほとんどありません。ハードディスクから直接**ファイルを読み書き**することが多いです。
- Java はファイルの入出力のための API を提供しています。以下の 2 つのパートに分けて学びましょう：
 - **ファイルを管理**し、特定のドライブにあるファイルを検索して開くための API：
 - java.io.File。
 - 次に、開いているファイルへの**書き込み**や、ファイルからデータの**読み出し**に使用する入出力 API：
 - java.io.InputStream、java.io.OutputStream。

File

- Java ではファイルを表すクラスとして **File** クラスが用意されています。
- File クラスは、ハードディスク上のディレクトリ（**フォルダ**）を表現するために使用することもできます。
- ファイルの作成、検索、削除など、基本的なファイル操作機能を提供します。

File オブジェクトの作成

- ファイルオブジェクトを作成するには、File クラスのコンストラクタが使用できます：

```
File file = new File("test.txt");
```

- 引数はファイル（パス）を表す文字列になります。
- **絶対パス**、**相対パス**のいずれでも使えます。

絶対パスと相対パス

- **絶対パス**とは、**ディスクレター**（Windows）または**ルートディレクトリ**（macOS）から始まるファイルのフルパスです：
 - D:\class\java\test.txt
 - ~\class\java\test.txt（または「/class/java/test.txt」）
- **相対パス**は、**現在のプログラム**のファイルパスから（Eclipse では、**プロジェクトルート**から）見るパスです：
 - test.txt
 - java\test.txt
 - ../test.txt

Note

Java 文字列にある「\」符号はエスケープする必要があることを忘れないでください。

相対パス

Case 1

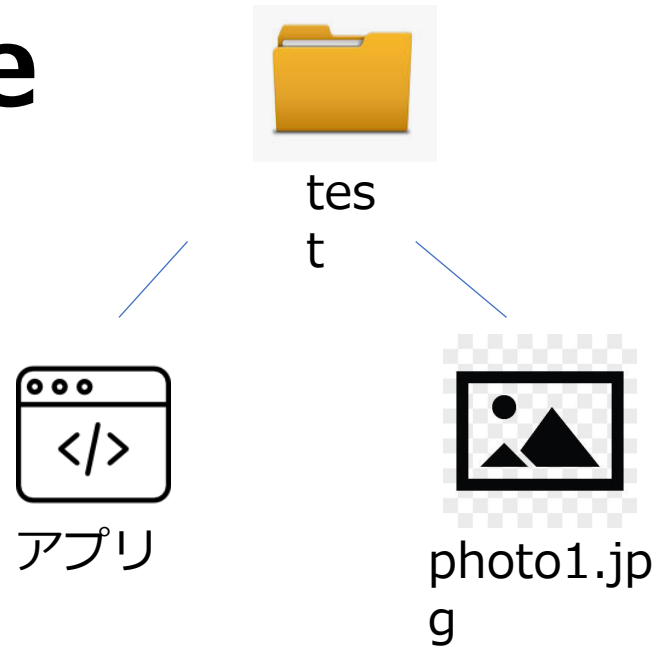


photo1 の相対パス :
photo1.jpg

Case 2

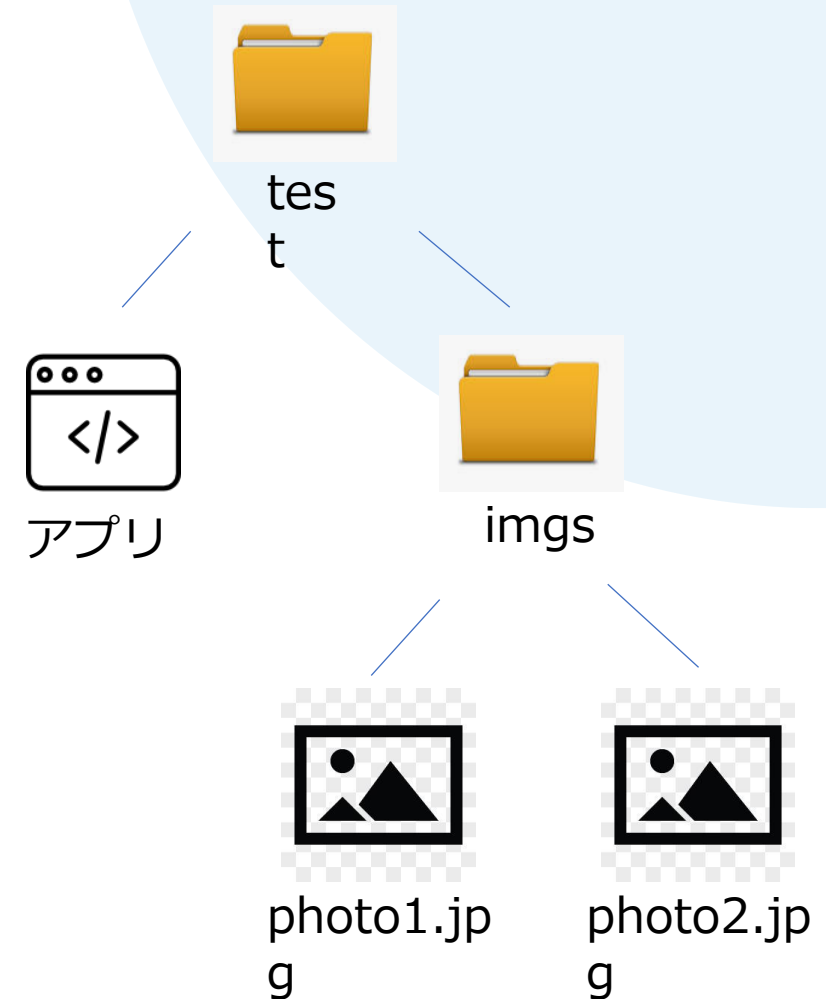


photo1 の相対パス :
imgs\photo1.jpg

Case 3

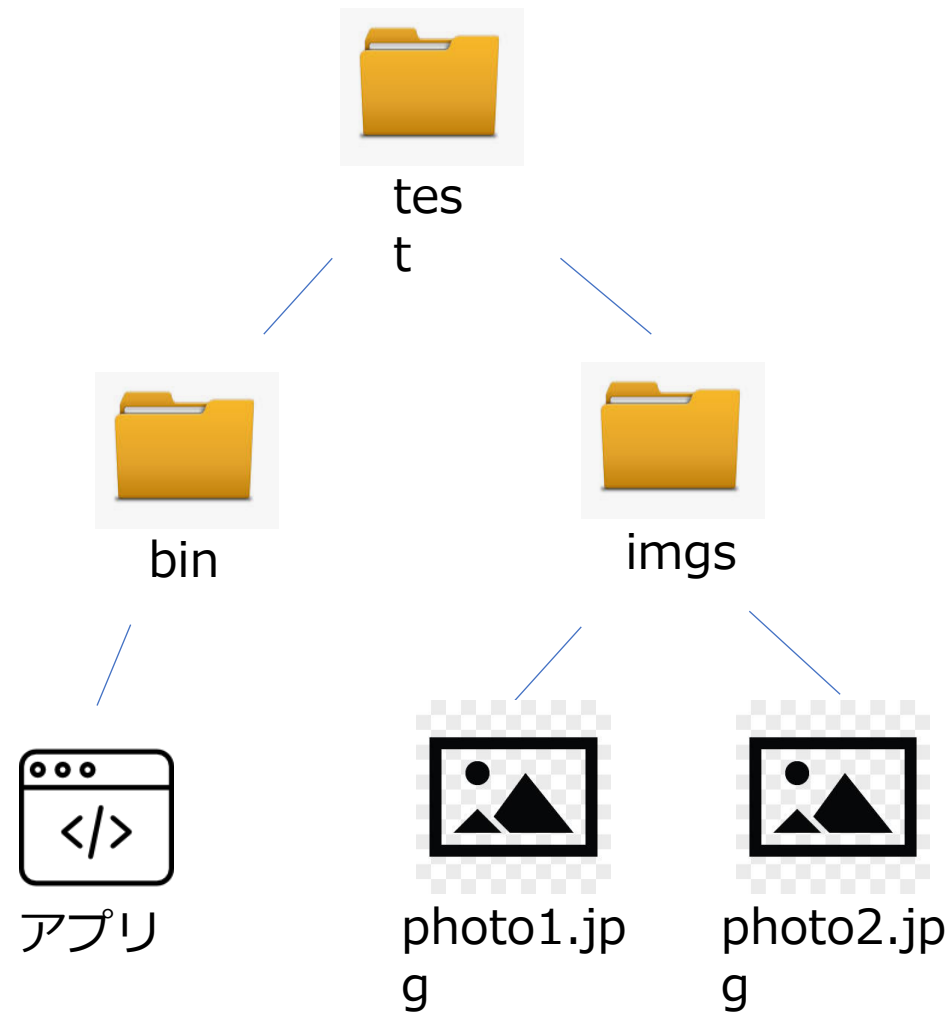


photo1 の相対パス:
`..\imgs\photo1.jpg`

Case 4

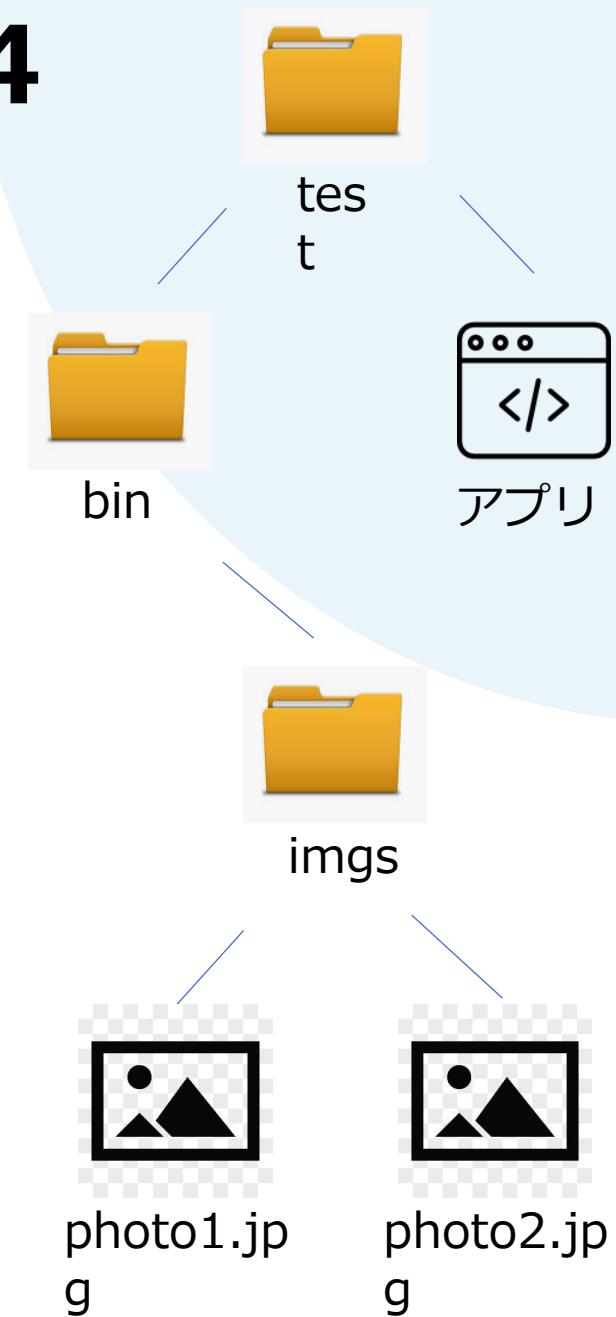


photo1 の相対パス : ?

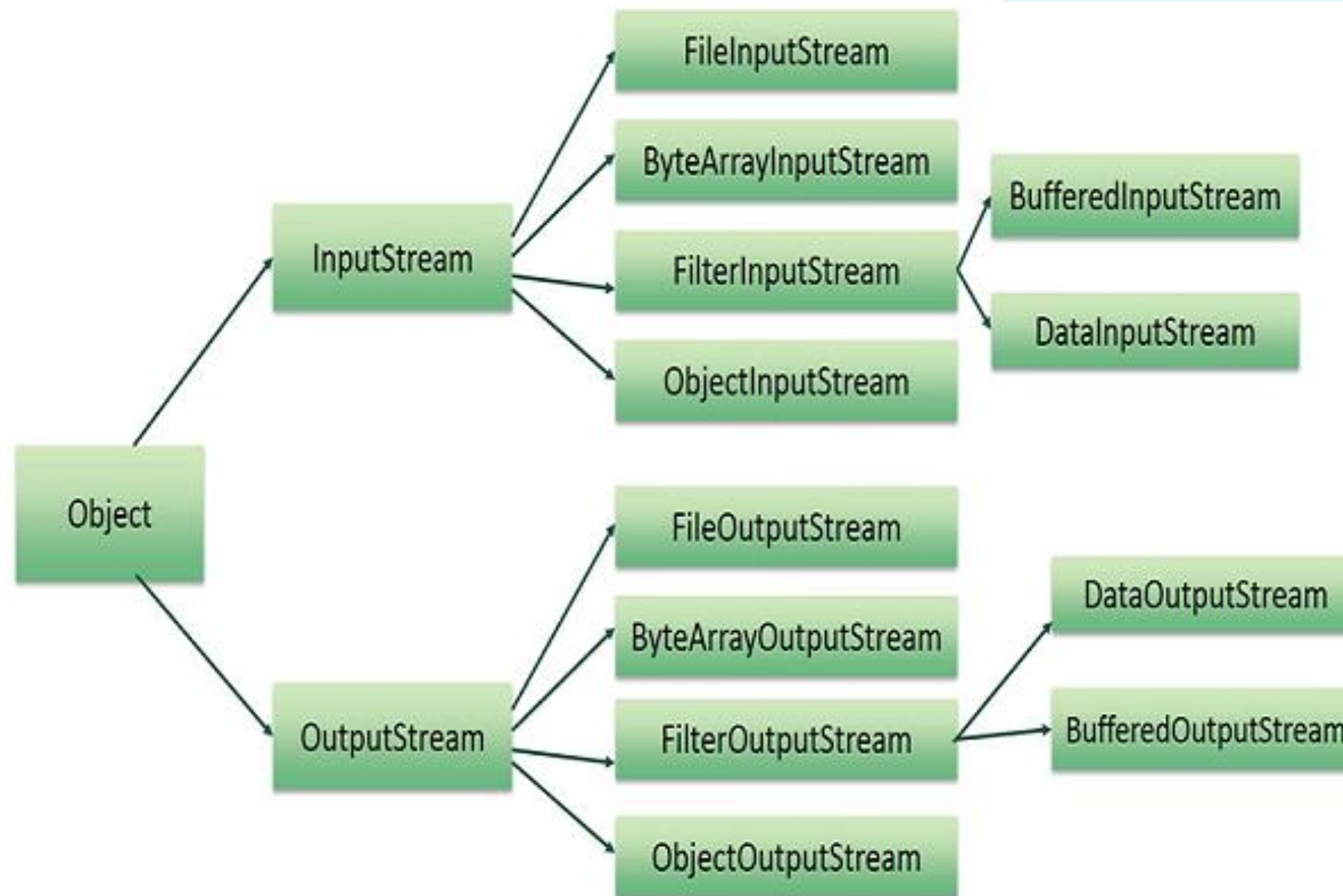
ファイルのメソッド

メソッド	機能
exists()	ファイル（フォルダ）が存在するかどうかを判断
createNewFile()	ファイルを作成
mkdir()	フォルダを作成
delete()	ファイル（フォルダ）を削除
getName()	ファイル名を取得
getAbsolutePath()	絶対パスを取得
length()	ファイルサイズを取得
list()	フォルダー内のファイル（フォルダー）のリストを取得
canRead()、 canWrite()	ファイルが読み取り可能か、書き込み可能かの判定



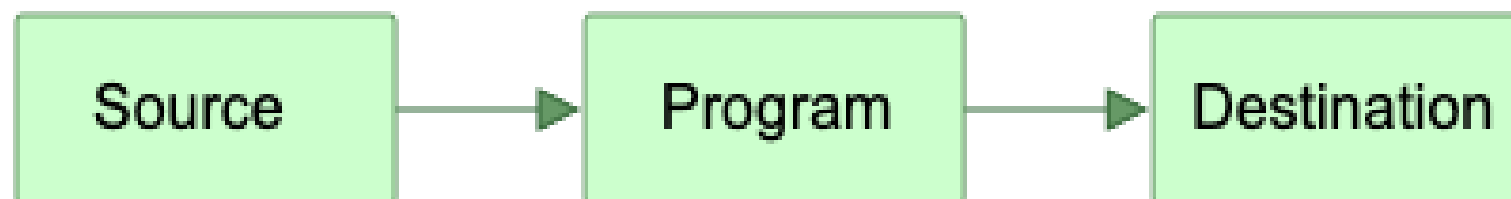
Java I/O API

- **Java I/O** (Input / Output) API は、あらゆる入出力のために使われます。



I/O APIの機能

- I/O API は、ファイルの読み書きだけでなく、様々なデータソースからデータを読み込んだり、データ**ターゲット**にデータを書き込んだりできるクラスを提供しています。
- 代表的なデータソースとターゲットは以下の通り：
 - ファイル、
 - パイプ [\[Pipe\]](#)、
 - ネットワーク通信、
 - コンソール（System.in、System.out、System.err など）。



I/O Streams

- **I/O Stream** は Java I/O の重要な一部ではあります。
- 簡単にいうと、入出力を「水の流れ（**ストリーム**）」のように行うものです。
- Stream クラスは 2 種類あります：
 - **Byte Stream** : InputStream、OutputStream のサブクラス
 - バイトの単位で入出力
 - 一般的に画像、音声、ビデオなどの非テキストファイル进行处理
 - **Char Stream** : Reader、Writer のサブクラス
 - 文字（キャラクター）単位で入出力
 - 一般的にテキストファイル进行处理
- ここでは Char Stream だけについて紹介します。

System.out.println() を理解しよう

System.*out*.println(xx)

System クラス

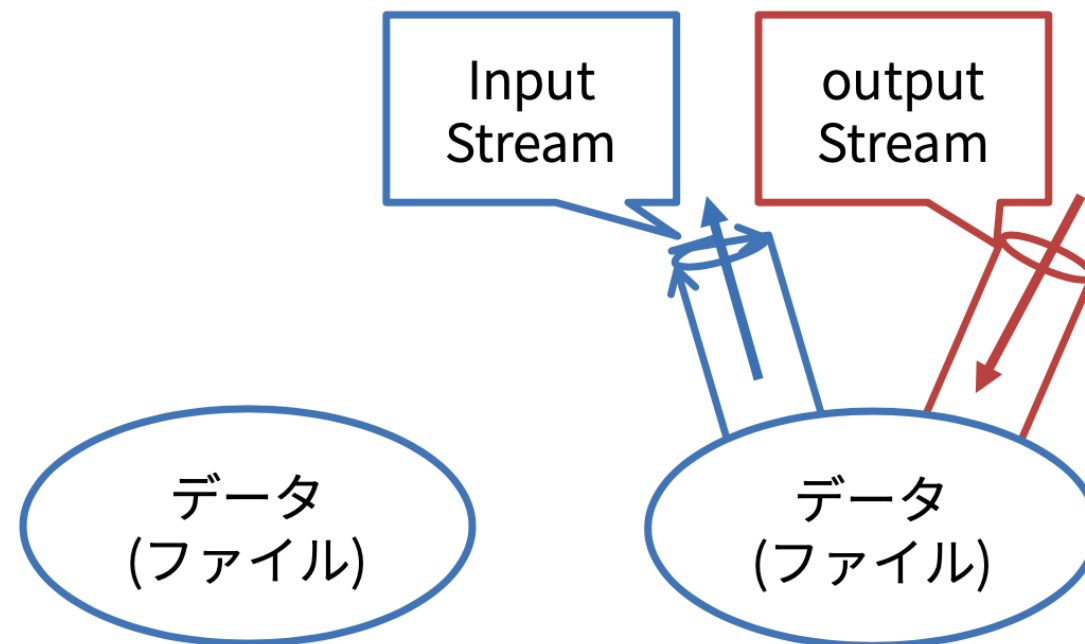
クラス変数 out には
PrintStream の オ
ブジェクトがある

標準出力に繋ぐ OutputStream
オブジェクトがその中にある

println()
println(String s)
println(int i)
などの PrintStream のメソッド

I/O Stream の使用の流れ

1. ファイルに対応する **File オブジェクト** を生成する。
2. ファイルを処理する **Stream のオブジェクト** を作成する。
3. Stream のメソッドによる**読み込み（書き込み）** 。
4. close() メソッドで Stream オブジェクトを**閉じます**。



入出力に必要なクラス

- 最もシンプルなファイルの入出力処理では、主に以下の 3 種類のクラスを使用します：
 - File クラス：入出力の対象となるファイルを指定
 - FileReader・FileWriter クラス：ファイルに直接**文字を入出力**
 - BufferedReader・BufferedWriter クラス：入出力の効率を向上させ、かつより便利にするメソッドが追加



Reader / Writerの作成

- FileReader か FileWriter を直接に使えば、一度に 1 文字しか読み書きできないため、それぞれ **BufferedReader** と **BufferedWriter** に配置して、行単位での入出力を簡単に実現します。
- BufferedReader を例にとると：

```
1 File file = new File("test.txt");
2 FileReader fileReader = new FileReader(file);
3 BufferedReader reader = new BufferedReader(fileReader);
```

- この場合、FileReader のコンストラクタが FileNotFoundException という例外をスローすることもあり、その対処が必要になります。

BufferedReader を使用

- BufferedReader が提供しているメソッドは以下の通り：

メソッド	機能
<code>read()</code>	文字を読む
<code>readLine()</code>	一行の文字列を読む
<code>lines()</code>	複数行の文字列を読む。戻り値は、リストに変換可能な文字列のストリームオブジェクト
<code>reset()</code>	もう一度最初から読む
<code>skip(long i)</code>	指定した文字数をとばす

close() メソッドと例外処理

- Reader と Writer を使用後、必ず **close() メソッドで全ての Reader・Writer を閉じてください**：

```
fileReader.close();  
reader.close();
```

- ただし、例外処理のために、入出力のコードを try-catch 文で囲むことがあります。例外が発生した場合、catch 文は close() 文を無視してメソッドを終わらせる恐れがあります。
- close() メソッドがこれらのクラスを確実に閉じるようにするには、**finally** 文や、**try-with-resource** 文を使用できます。

BufferedWriter を使用

- BufferedWriter が提供しているメソッドは以下の通り：

メソッド	機能
<code>write(int c)</code>	文字を書く
<code>write(String s)</code>	文字列を書く
<code>newLine()</code>	改行する
<code>flush()</code>	出力キャッシュをリフレッシュ

try-with-resource 文による例外処理

- try-with-resource 文を使用すると、入出力時の例外を簡単に処理できます。Java は例外の発生に関わらず、try ブロックの終了時に宣言された全ての Reader・Writer を自動的に閉じます：

```

1 try (
2     FileReader fileReader = new FileReader(file);
3     BufferedReader reader = new BufferedReader(fileReader)
4 ) {
5     System.out.println(reader.readLine());
6 } catch (Exception e) {
7     System.out.println("Exception occurred while reading the file: " + file);
8     e.printStackTrace();
9 }

```



Q&A

目次

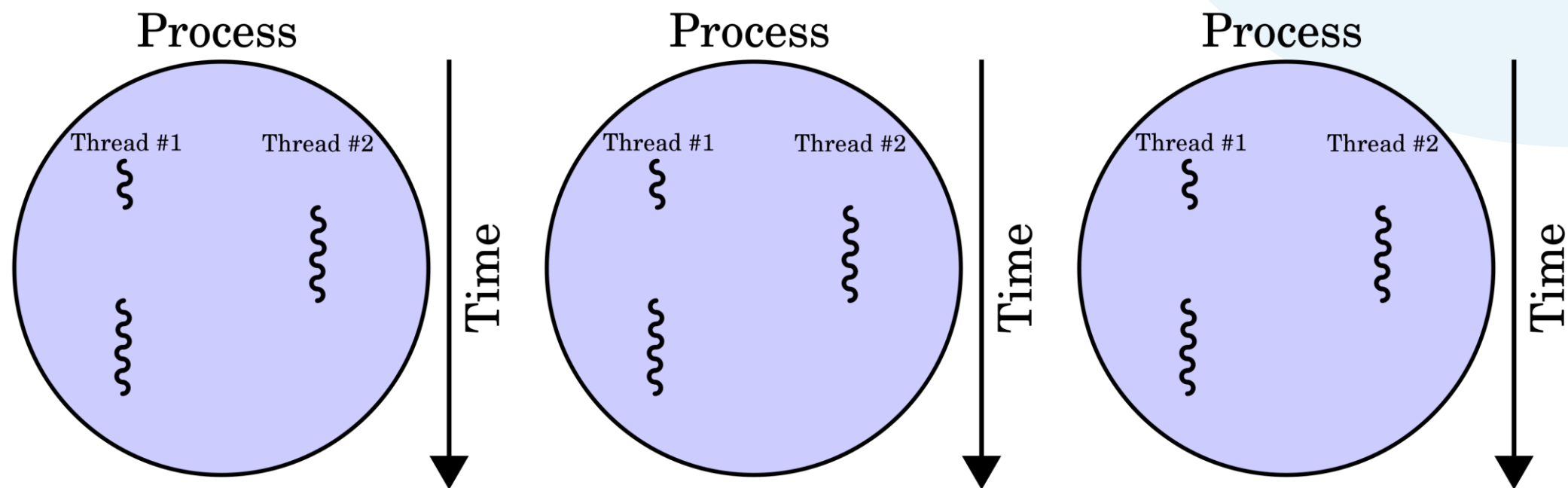
- 1 修飾子
- 2 ファイル入出力
- 3 スレッド
- 4 正規表現
- 5 その他

プロセス

- コンピュータでは、あるプログラムを実行しようとする
と、オペレーティングシステムが、そのプログラムに 1 つ
の**プロセス**[Process]を割り当てます。
- 複数のプロセスを同時に実行することができ、OS が自動的にそれらの実行時間を調整して、同時に実行できるようにしてくれます。

スレッド

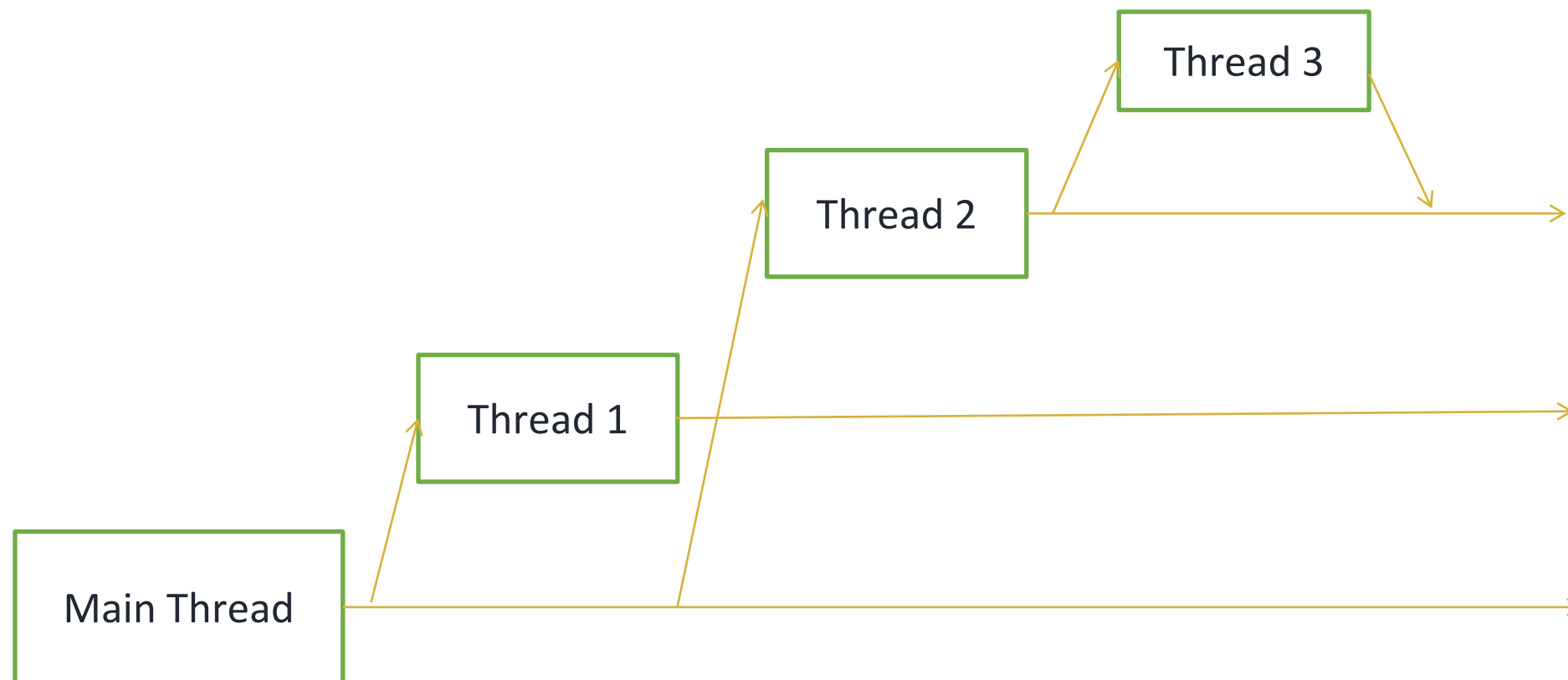
- しかし、時には同じプログラムの中で複数の作業を同時に行う必要があります。この時は、**スレッド**^[Thread]を使うべきです。プロセスと同様に、OS は複数のスレッドを同時に実行できるように、実行時間を自動的に割り当てることができます。



- 複数のスレッドを同時に実行させることは、**マルチスレッド**^[Multithread]と言います。マルチスレッドを使うケースを考えましょう。

Java のスレッド

- プログラムが実行され始めると、すべてのコードは**メインスレッド**と呼ばれるデフォルトのスレッドで実行されます。
- メインスレッドと同時にタスクを実行させたい場合は、新しい**スレッドを作成**し、そのスレッドに**タスクを与えて実行させる**ことが必要です。



スレッドの作成


- **Thread** クラスは、Java でスレッドを表すクラスです。
- スレッドを作成するには、Thread のコンストラクタでそのオブジェクトを作成します。ドキュメントを見ると、Thread のコンストラクタは Runnable 型の引数を受け取ります：

`Thread(Runnable target)`

Allocates a new Thread object.

- この **Runnable** は、**run()** というメソッドだけを持つ関数型インタフェースです：

```
1 package java.lang;
2
3 @FunctionalInterface
4 public interface Runnable {
5     public abstract void run();
6 }
```

- つまり、スレッドを生成するためには、Runnable インタフェースを実装したクラスを用意し、それをインスタンス化し、Thread のコンストラクタに渡す必要があるのです。
- ただし、Runnable は**関数型インタフェース**なので、ここは **lambda 式**が使えます ( § 3.2.2) :

```
1 Thread thread = new Thread(() -> {  
2     System.out.println("Do something 1");  
3     System.out.println("Do something 2");  
4     System.out.println("Do something 3");  
5 });
```

スレッドの実行

- 今のところ、スレッドは作成できましたが、まだ実行されていません。スレッドを実行するには、その **start()** メソッドを呼び出す必要があります：

```
1 Thread thread = new Thread( () -> {  
2     System.out.println( "Do something 1" );  
3     System.out.println( "Do something 2" );  
4     System.out.println( "Do something 3" );  
5 } );  
6 thread.start(); // Do something 1, 2, 3
```

複数のスレッドの実行

- スレッドを使用する主な理由は**複数のタスクの同時実行**です。異なるスレッドを作成して、それらが同時に実行されるかどうかを確認しましょう。

Try 

MultiThread.java

Note 

スレッドの実行順番は OS に決められて、
start() の順番と**必ずしも同じわけではない。**

スレッドのメソッド

- start() メソッド以外にも、スレッドの動作を制御するための様々なメソッドが提供されています。よく使われるのは：

メソッド	機能
start()	スレッドを実行
setPriority()	スレッドの優先順位を設定
join()	スレッドが終わるのを待つ
interrupt()	スレッドを割り込み
isAlive()	スレッドが実行中かどうかを判断
sleep()	現在のスレッドを一定時間で中断する

並行処理の問題

- 実際のスレッドの順番は操作システムによって自動的に決定されるため、複数のスレッドを同時に実行すること、いわゆる**並行処理**[Concurrent Processing]が多くあります。並行処理は、予期せぬ多くの問題を引き起こす可能性があります。
- 例えば、同じ変数の値を異なるスレッドで変更しようとする、期待するような結果が得られない恐れがあります。



- なぜそのような結果が出るのかを考えましょう。

不可分性

- スレッドが期待通りに実行されない理由の大部は、コードが**不可分性**^[Atomicity]（**アトミック性**、**原子性**ともいう）を満たしていないためです。不可分性とは、あるコードのブロックが常に連続的に実行され、他のスレッドより介入できないことを意味します。
- 例えば、先ほどの例では、変数がインクリメントされるたびに他の変数に影響があるコードは実行されないようにしたいです。

synchronized キーワード

- Java では、**synchronized** キーワードを使って、あるコードが同時に 1 つのスレッドしか実行できないように制御することができます。
- synchronized で修飾された同一オブジェクトのメソッドは、同時に 1 スレッドのみが実行可能です。

Try 01011
11010
01011
Parallel2.java

synchronized の他の使用法

- synchronized キーワードはメソッドを修飾する以外にも使い方がありますが、ここでは概要だけ紹介しましょう。

- **コードブロック**の修飾：

```
1 synchronized(obj) {  
2     // codes  
3 }
```

このコードブロックを「ロック」するには、オブジェクトを使用します。同じオブジェクトにロックされたメソッドは、同時に 1 つしか実行できません。最も一般的な使用法は、「**this**」または「**クラス名.class**」をロックとして使います。

- **クラスメソッド**も修飾できます。そのクラスメソッドは一度に 1 つのスレッドしか実行できません。

Q&A

目次

- 1 修飾子
- 2 ファイル入出力
- 3 スレッド
- 4 正規表現
- 5 その他

正規表現

- **正規表現**[\[Regular Expression, Regex\]](#)とは特定の条件を満たす文字列を確認する（**マッチ**[\[Match\]](#)という）ために使用できる特殊な言語であります。
- すべての正規表現は**文字列**であります。例えば、以下の文字列は、メールアドレスのフォーマットを満たす文字列（ABC@XXX.YZ）をマッチします：


```
[\\w\\-\\.\\_]+@[\\w\\-\\.\\_]+\\. [A-Za-z]+
```
- 正規表現の実用的な使い方は主に以下の 2 つです：
 - 文字列が条件を満たすかどうかを判定：**フォーム検証**[\[Validation\]](#)など
 - 文字列から条件を満たす部分文字列を検索：**クローラ**[\[Crawler\]](#)など

正規表現の基本文法

- 特殊文字を含まなければ、正規表現はそれ**自身**にマッチ：

Example



正規表現：

Apple

マッチ	マッチしない
Apple	Banana
	apple
	chocolate

特殊文字：「.」

- ワイルドカード符号「.」は**任意の 1 文字**にマッチ：

Example ✓

正規表現： `Ap.le`

マッチ	マッチしない
Apple	Banana
Apkle	apple
Ap3le	Aple
Ap~le	Appple

特殊文字：「*」

- 「a*」 は、**0 個以上**の「a」文字にマッチ：

Example✓

正規表現： `Ap*le`

マッチ	マッチしない
Apple	Banana
Aple	apple
Ale	Apkple
Appppple	AppleE

特殊文字の組合せ

- それらの文法を組み合わせたことができます：

Example

正規表現：`A.*le`

マッチ	マッチしない
Apple	Banana
Abbble	apple
Aabcle	AppleE
Ale	pppAle

特殊文字：「()」

- 小括弧「**()**」は他の記号と組み合わせて使用することができます。例えば、「(abc)*」は 0 個以上の「abc」にマッチ：

Example

正規表現：

`A(pp)*le`

マッチ	マッチしない
Apple	Banana
Ale	apple
Appppppple	Appple

特殊文字：「+」

- 「a+」は **1つ以上**の「a」文字にマッチ：

Example

正規表現：

A(pp)+le

マッチ	マッチしない
Apple	Banana
Appppple	apple
Appppppple	Ale

特殊文字：「|」

- 「文字列a|文字列 b」 は、指定した**いずれかの文字列**にマッチ

Example

正規表現： `A(pp|qq|rr)*le`

マッチ	マッチしない
Apple	Banana
Aqqle	apple
Arrle	Assle
Ale	Apqrsle

特殊文字：「[]」

- 「[abc]」は「a」「b」「c」の**いずれかの文字**にマッチ：

Example✓

正規表現：`A[pqrst]+le`

マッチ	マッチしない
Apple	Banana
Apppppple	apple
Asle	Ale
Aqrtle	Awwwle

「[]」の簡略表記

- 「[a-z]」は**小文字**に、「[A-Z]」は**大文字**に、「[0-9]」は**数字**にマッチ：

Example ✓

正規表現：

`A[a-z0-9]*le`

マッチ	マッチしない
Apple	Banana
Ale	apple
A123abcle	APPlE

特殊文字：「\」

- Java と同様、「\」はエスケープ文字で、特殊なマッチを表現したり、直接に書けない文字にマッチしたりします：

エスケープシーケンス	マッチする文字
\n	改行
\w	英文字と数字
\d	数字
\s	空白文字（空白、タブ、改行）
\.	「.」
\+	「+」
\(、\)	「(」 「)」
\\	「\」

その他の正規表現文法

- 正規表現の文法一覧：
 <https://quickref.me/regex>
- 正規表現を可視化：
 <https://regexper.com/>

Java での正規表現

- **Pattern** クラスは、正規表現のためのメソッドを提供。
- 文字列が正規表現にマッチするかどうかを判断するには、`Pattern.matches()` メソッド（または **String** の **`matches()`** メソッド）は使えます：

```
1 // => true
2 System.out.println(Pattern.matches("A[qr]+le", "Apple"));
3 // => false
4 System.out.println(Pattern.matches("A[qr]+le", "Awle"));
```

- 文字列から一致するすべての部分文字列を見つけるのは少し複雑で、`Matcher` クラスの **`find()`** メソッドと **`group()`** メソッドを使わなければなりません。

Try  **Regex.java**

「\」に関する注意事項

- エスケープシーケンス処理の場合、「¥¥」と表現します。
- 正規表現の場合、JAVA文字列内でのエスケープシーケンスを含める必要があるので、正規表現パターン内でバックスラッシュを表現するには、「¥¥¥¥」と書く必要があります。
- +等の特殊記号を正規表現でパターンで記載する場合は、「¥¥+」と書く必要があります。
- つまり、特殊文字を正規表現を使用してマッチさせるためには、エスケープシーケンス処理の「¥」と正規表現パターンのエスケープ処理「¥」が必要となります。

Artist: xkcd

エスケープシーケンス	マッチする文字	正規表現パターン
\n	改行	\\n
\w	英文字と数字	\\w
\d	数字	\\d
\s	空白文字（空白、タブ、改行）	\\s
\.	「.」	\\.
\+	「+」	\\+
\(、\)	「(」 「)」	\\(,\\)
\\	「\」	\\\\

Q&A

目次

- 1 修飾子
- 2 ファイル入出力
- 3 スレッド
- 4 正規表現
- 5 その他

Math

- **Math** クラスは**数学的**な計算に関連する多くの変数とメソッドを提供します：

変数	意味
PI	円周率
E	自然対数の底


Try

01011
11010
01011

MathEx.java

メソッド	意味
max(), min()	最大値・最小値
abs()	絶対値
cos(), sin() 等	三角関数
ceil(), floor()	切り上げ・切り下げ
pow()	累乗を求める
exp(), log()	自然対数のべき乗・対数
random()	ランダムに数を生成

- その他のメソッドについては、ドキュメントを参照してください：

 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Math.html>

時間 API

- **java.time** パッケージは、**時刻（日付）**に関連する機能を提供します。
- 例えば、**LocalDateTime** クラスが提供する機能は：

メソッド	機能
of	さまざまな形式の入力から、日付と時刻を表すオブジェクトを作成
now	現在の日付・時刻を取得
get	各種日付・時刻情報を取得する
plus	各種単位で日付・時刻を変化
with	各種単位で日付・時刻の値を設定



- ドキュメントには他の機能もたくさん：

 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/time/package-summary.html>

Scanner

- **Scanner** クラスは、データを簡単に入力する方法を提供しています。最も一般的な用途は、コンソールアプリケーションの開発や、テストのためのコンソール入力の処理です。
- Scanner を使ってコンソール入力を処理するには、Scanner のコンストラクタに **System.in** を指定します：

```
Scanner scanner = new Scanner(System.in);
```

Note

Scanner も I/O ストリーム的一种で、使い終わったら `close()` メソッドで閉じるのを忘れないように。

Scanner のメソッド

- Scanner でのデータ入力には以下のメソッドがあります：

メソッド	機能
next()	次の単語（文字列）を入力
nextLine()	この行の最後まで入力
nextInt()	次の整数を入力
nextDouble()	次の小数を入力
nextBoolean()	次のブール値を入力



- 注：複数行のデータを入力させる場合は、改行するたびに `nextLine()` メソッドを呼び出すのがよいでしょう。

文字列操作の問題

- プログラムで特定の文字列の値を繰り返して変更することが必要な場合があります：

```
String str = "";
for (int i = 0; i < 10; i++) {
    str += i + " ";
}
```

- Java の文字列はそもそも変化できないので、実際には**新しい文字列オブジェクトを繰り返して作成**することになり、無駄なリソースを消費してしまいます。
- 文字列を何回も変更する必要があるなら、**StringBuilder** クラスの使用を考慮してください。

StringBuilder

- **StringBuilder** は、文字列を保持するクラスで、新しいオブジェクトを作成せずに既存の文字列を変更できます。
- 先ほどのコードを、StringBuilder を使って書き直すとこのようになります：

```
1 StringBuilder sb = new StringBuilder();
2 for (int i = 0; i < 10; i++) {
3     sb.append(i).append(" ");
4 }
5 String str = sb.toString();
```

- この `append()` メソッドは、既存のオブジェクトを変更するだけで、新しいオブジェクトは作成**されません**。

StringBuilder のメソッド

- StringBuilder は他の便利なメソッドも提供しています：

メソッド	機能
charAt()	指定位置の文字を取得
indexOf()	部分文字列の位置を探す
insert()	文字列を挿入
delete()	文字列を削除
replace()	文字列を置き換え
reverse()	文字列を反転

Try 01011
11010
01011
DynamicString.
java

- その他のメソッドは、ドキュメントを参照してください：

 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/StringBuilder.html>

Optional


- **Optional** クラスは、**null** になるの可能性があるオブジェクトを格納するために使用されます：

```
Optional<String> data = Optional.ofNullable(str);
```

- Optional クラスを使用することで、null の可能性があるオブジェクトを操作するコードを簡単に書くことができます：

```
1 // 只在数据不为空时输出它
2 data.ifPresent(s -> System.out.println("str is " + s));
3 // 输出数据, 如果数据为空, 输出 "No data"
4 System.out.println(data.orElse("No data"));
```

- ドキュメントも参照：

 <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Optional.html>

Try 
OptionalTest.java

Q&A

まとめ

Sum Up



- 1.修飾子 : static、abstract、final。
- 2.ファイルの入出力する方法。
- 3.スレッドの基本概念とその使い方。
- 4.正規表現の基本文法と使い方。
- 5.その他の API : Math、LocalDateTime、StringBuilder、Scanner、Optional。



Light in Your Career.
THANK YOU!