

# **Java Doでしよう #10**

## **Java9直前！最近のJava復習ハンズオン**

Javaエンジニアグループ北海道 "Java Do"  
山川 広人 @gishi\_yama



# ハンズオンの準備

- <https://goo.gl/Mxg7Dq> をブラウザで開く
- 画面右側の **Clone or Download** ⇒ Download ZIP でダウンロードする
- ダウンロードしたファイルを解凍する  
Windowsは右クリック⇒展開、Macはそのまま開く（ダブルクリック）
- 解凍したフォルダをIDEで開く（Mavenプロジェクトです）

**Eclipse** : ファイル→インポート→既存のMavenプロジェクト

**NetBeans** : ファイル→プロジェクトを開く

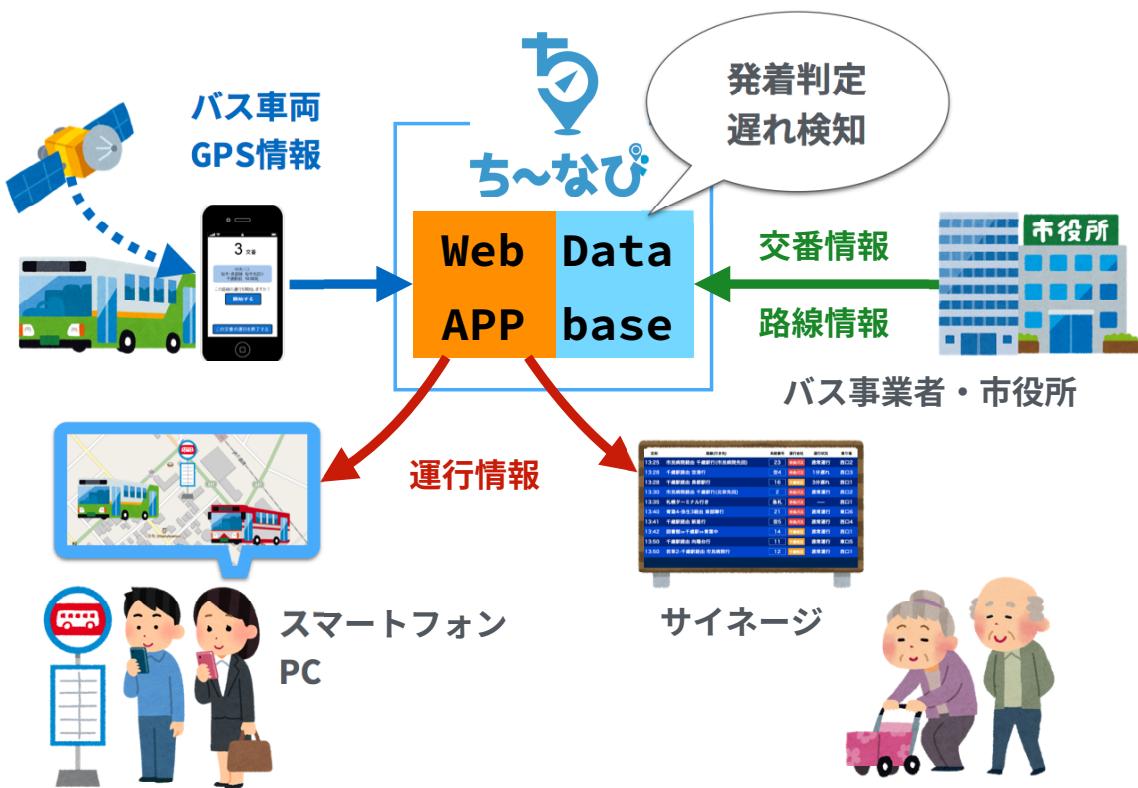
**IntelliJ IDEA** : Open

- src/test/AppTest.java を実行して、Webアプリが動作すること確認する



# 自己紹介

千歳市でIoV（バスロケーションシステム）の実用化研究をしています。  
(バックグラウンドにJavaを使ってます)



# Java9がそろそろですね

2018年9月21日（月）にリリース予定！新機能もたくさん！

- Milling Project Coin : 構文変更  
try-with-resourceの変更、匿名クラスのダイアモンド演算子...
- Collection Factory Methods 

```
List<String> list = List.of("A", "B", "C", "D");  
Map<String, Integer> map = Map.of("A", 10, "B", 20);
```
- JShell : REPL対話的な実行
- Jigsaw : モジュールシステム
- Http2クライアント（実験的導入）
- 他

参考 : <http://itpro.nikkeibp.co.jp/atcl/column/15/120700278/index.html>  
[https://www.slideshare.net/bitter\\_fox/10java9java8](https://www.slideshare.net/bitter_fox/10java9java8)



# ところで皆さん、普段はどのJavaを使ってるんです？

- Java 9 (Early-Access Builds)
- Java 8
- Java 7
- Java 6
- Java 5
- Java 1.4
- それ以下
- 普段は使ってない or はじめて



# 今日のゴール

「Java 7,8でどんなことが便利になったっけ？」という観点で復習をして、  
Java 9 のリリースに備えよう。

- Try-With-Resource (7)
- ファイル処理(7)
- Date Time API(8)
- Stream API/ラムダ式(8)
- そのほか(7,8)



# Try-with-Resource (Java 7)



# Try-with-Resource (Java 7)

Javaの例外ハンドラといえば、  
Try-Catch-Finally。  
だいたいI/OとかDB処理が伴うと、  
closeメソッドでfinallyが  
大変なことになる。

よく、Javaは冗長という例で  
出されてたDB接続のコード→

DBの検索結果をインスタンスに  
マッピングする場合は、さらに  
長くなることも....

```
Connection conn = null;
Statement stmt = null;
try {
    conn = DriverManager.getConnection("URL", "UserName", "Password");
    stmt = conn.createStatement();
    returning = stmt.executeUpdate(sql);
} catch (SQLException ex1) {
    ex1.printStackTrace();
} finally {
    try {
        if (stmt != null) {
            stmt.close();
        }
    } catch (SQLException ex2) {
        ex2.printStackTrace();
    }
    try {
        if (conn != null) {
            conn.close();
        }
    } catch (SQLException ex3) {
        ex3.printStackTrace();
    }
}
```

リソース解放のための  
ボイラープレートコード  
⇒Try-With-Resourceで  
解消される！



# Try-with-Resource (Java 7)

Closableインターフェースを継承したインスタンスをI/OやDB処理に使う場合、

**Try-with-Resourceを使えば closeメソッドを省略できる（自動的にcloseする）！**

Connection, PreparedStatement が Closable

```
try (Connection conn =
      DriverManager.getConnection("URL", "UserName", "Password")) {
    try (PreparedStatement stmt = conn.prepareStatement(sql)) {
        returning = stmt.executeUpdate();
    }
} catch (SQLException ex) {
    ex.printStackTrace();
}
```

自動でconn.close();

自動でstmt.close();

例外をスローするなら、catchブロックもいらない（最低6行まで短く）  
ひとつのcatch句で、複数の例外をキャッチすることもできる

※例として、一番基本的なDB接続コードについています。

Closableなクラスのインスタンスになっていれば、他フレームワークや  
ファイル操作を利用したI/Oであっても自動closeされます。



# やってみよう① (HandsOn7#TryWithResourceを使う)

```
@Test
public void TryWithResourceを使う() {
    System.out.println("TryWithResourceを使う -----");
    List<String> lines = new ArrayList<>();

    File file = new File(url.getPath());
    try (FileReader fr = new FileReader(file)) {
        BufferedReader bf = new BufferedReader(fr));
        while (true) {
            String line = bf.readLine();
            if (line == null) {
                break;
            }
            lines.add(line);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    for (String line : lines) {
        System.out.println(line);
    }
}
```



# ファイル処理 (Java 7)



# ファイル処理 (Java7)

NI02 (Path, Paths, Files) を使って、ファイル操作が容易になった！

Pathクラス：ファイルの場所を管理するクラス

Pathsクラス：Pathインスタンスを作成するファクトリクラス

Filesクラス：Pathインスタンスを用いたファイル操作クラス

```
// 例  
// ファイルの場所を特定  
Path path = Paths.get("ファイルパスやURIインスタンス");  
// テキストファイルの全行読み込み  
Files.readAllLines(path);
```

Filesには、ファイル・フォルダ操作（作成・削除・複製・書き込み）や、属性確認・変更といったメソッドも備えられている



## (参考) Filesのメソッド

```
// pathAのファイルを作成  
Files.createFile(pathA);  
  
// pathAのディレクトリを作成  
Files.createDirectories(pathA);  
  
// pathAからpathBにファイルを複製  
Files.copy(pathA, pathB);  
  
// pathAからpathBにファイルを移動  
Files.move(pathA, pathB);  
  
// pathAを削除  
Files.delete(pathA, pathB);
```

```
// pathAにテキストを書き込み  
List<String> out = ...  
Files.write(pathA, out);  
  
// pathAの属性を読み込み(例: ファイルサイズ)  
Files.size(pathA);  
  
// pathAの属性を読み込み(例: 実行可能か)  
Files.isExecutable(pathA);  
  
// pathAの属性を読み込み(例: 読み書き可能か)  
Files.isReadable(pathA);  
Files.isWritable(pathA);
```

このほかにも、フォルダツリーを再帰的に操作できる `FileVisitor` などもNIO2で準備されている



# やってみよう① (HandsOn7#NIO2を使う)

```
@Test
public void NIO2を使う() {
    System.out.println("NIO2を使う -----");
    List<String> lines = new ArrayList<>();
    URI uri = URI.create(url.toString());

    Path path = Paths.get(uri);
    try (BufferedReader bf = Files.newBufferedReader(path)) {
        while (true) {
            String line = bf.readLine();
            if (line == null) {
                break;
            }
            lines.add(line);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }

    for (String line : lines) {
        System.out.println(line);
    }
}
```



## やってみよう② (HandsOn7#NIO2を使う\_小さなファイル)

```
@Test
public void NIO2を使う_小さなファイル() {
    System.out.println("NIO2を使う_ファイルサイズが小さいとき -----");
    List<String> lines = null;
    URI uri = URI.create(url.toString());

    Path path = Paths.get(uri);
    try {
        lines = Files.readAllLines(path);
    } catch (IOException e) {
        e.printStackTrace();
    }

    for (String line : lines) {
        System.out.println(line);
    }
}
```



# Date Time API (8)



# Date Time API

DateやCalenderに比べて、日付や時刻を扱いやすくなった！

例)

- n日後、n日前といった日時の操作は?  
⇒ `LocalDate`, `LocalTime`, `LocalDateTime` などのクラスとメソッド
- 時間や年月の差や比較は?  
⇒ `Period`, `Duration` などのクラスとメソッド
- うるう年は?  
⇒ クラスやメソッドの中で自動考慮



# LocalDate, LocalTime, LocalDateTime

LocalDateは日付を、LocalTimeは時刻を、LocalDateTimeは日付と時刻をインスタンスで表せる

例) 2017年9月2日13:34:00.000だったとき、

```
LocalDateTime.now();           //2017-09-02 13:34:00.000  
LocalDate.now();              // 2017-09-02  
LocalTime.now();              // 13:34:00.000  
LocalDateTime.of(2016, 3, 1, 0, 0); // 2016-03-01 00:00:00.000
```

```
LocalDateTime ldt = LocalDateTime.now(); //2017-09-02 13:34:00.000  
ldt.getYear();                  // 2017  
ldt.getDayOfWeek();            // SATURDAY
```



# 日時の表示形式を変更する

画面やログに表示する際に、日時の表示形式を変更したい場合は、  
DateTimeFormatterを用いる

例) 2017年9月2日13:34:00.000だったとき、

```
LocalDateTime ldt = LocalDateTime.now();
System.out.println(ldt);                      //2017-09-02T13:34:00.000

DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy年MM月dd日h時m分s秒");
System.out.println(dtf.format(ldt));      // 2017年9月2日13時34分0秒
```

※利用できるパターン文字はJavaDocに掲載されている

<https://docs.oracle.com/javase/jp/8/docs/api/java/time/format/DateTimeFormatter.html>



# 日時の操作メソッド

日付を変えたインスタンスを取得するメソッドが用意されている  
(うるう年なども反映される)

```
LocalDateTime ldt = LocalDateTime.now(); //2017-09-02 13:34:00.000
ldt = ldt.plusDays(1); //2017-09-03 13:34:00.000
ldt = ldt.minusHours(1); //2017-09-02 12:34:00.000
ldt = ldt.withYear(2019); //2019-09-02 13:34:00.000
```

`plusXXX` メソッドは足し算、  
`minusXXX` メソッドは引き算、  
`withXXX` メソッドは上書き

をした新しいインスタンスが生成される。  
(元のインスタンスを変更するのではなく、新しいインスタンスを作成する  
イミュータブルなインスタンスであることに注意



# 時間や年月の差を求める

時間の差はDurationクラス、年月の差はPeriodインスタンスで求められる

```
LocalTime time1 = LocalTime.of(2, 48);
LocalTime time2 = LocalTime.now();
// time1とtime2の時間差をとる

Duration duration = Duration.between(time1, time2);
System.out.println(time1.toString() + "から" + duration.toMinutes() + "分経過");
```

```
LocalDate date1 = LocalDate.of(1980, 8, 20);
LocalDate date2 = LocalDate.now();
// date1とdate2の年月差をとる

Period period = Period.between(date1, date2);
System.out.println(date1.toString() + "から" + period.toTotalMonths() + "ヶ月経過");
```



# (参考) DB用のデータ型や、Date Time API間での変換

DB用のデータ型（JDBCのjava.sqlパッケージ）や、Date Time API間で変換できるメソッドが用意されている

```
// LocalDateTime から java.sql.Timestamp  
LocalDateTime ldt = LocalDateTime.now();  
java.sql.Timestamp sqlTime = java.sql.Timestamp.valueOf(ldt);
```

```
// LocalDate から java.sql.Date  
LocalDate ldt = LocalDate.now();  
java.sql.Date sqlTime = java.sql.Date.valueOf(ldt);
```

```
// java.sql.Timestamp から LocalDateTime  
LocalDateTime ldt = sqlTime.toLocalDateTime();
```

```
// java.sql.Date から LocalDate  
LocalDate date2 = sqlDate.toLocalDate();
```

```
// LocalDate, LocalTime を使って LocalDateTime  
LocalDateTime ldt = LocalDateTime.of(LocalDate.now(), LocalTime.now());
```

```
// LocalDateTime を使って、LocalDate, LocalTime  
LocalDate date = ldt.toLocalDate();  
LocalTime time = ldt.toLocalTime();
```



# やってみよう① (HandsOn8#今日の日付と時刻を作る)

```
@Test
public void 今日の日付と時刻を作る() {
    // 日付の作成
    LocalDate date = LocalDate.now();
    System.out.println(date);

    // 時刻の作成
    LocalTime time = LocalTime.now();
    System.out.println(time);

    // 今年は何年？
    int year = date.getYear();
    System.out.println("今年は" + year + "年");

    // 今は何時？
    int hour = time.getHour();
    System.out.println("今は" + hour + "時");

    // 今日は何曜日？
    DayOfWeek dow = date.getDayOfWeek();
    System.out.println("今日は" + dow);
}
```



## やってみよう②（HandsOn8#日時の表示方法を加工する）

```
@Test  
public void 日時の表示方法を加工する() {  
    LocalDateTime ldt = LocalDateTime.now();  
    System.out.println(ldt);  
    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy年MM月dd日h時m分s秒");  
    System.out.println(dtf.format(ldt));  
}
```



## やってみよう③ (HandsOn8#日時を操作する)

```
@Test
public void 日時を操作する() {
    // 2016年3月1日12時30分
    LocalDateTime ldt1 = LocalDateTime.of(2016, 3, 1, 12, 30);
    System.out.println(ldt1);

    // ldt1の一日前のインスタンスを取得
    LocalDateTime ldt2 = ldt1.minusDays(1);
    System.out.println("一日前は" + ldt2);

    // ldt1を2017年に上書きしたインスタンスを取得
    LocalDateTime ldt3 = ldt1.withYear(2017);
    System.out.println(ldt3);

    // ldt3の一日前のインスタンスを取得
    LocalDateTime ldt4 = ldt3.minusDays(1);
    System.out.println("一日前は" + ldt4);
}
```



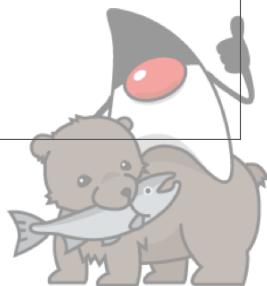
# やってみよう④ (HandsOn8#日時の差を計算する)

```
@Test
public void 日時の差を計算する() {
    LocalTime time1 = LocalTime.of(2, 48);
    LocalTime time2 = LocalTime.now();

    // time1とtime2の時間差をとる
    Duration duration = Duration.between(time1, time2);
    System.out.println(time1.toString() + "から" + duration.toMinutes() + "分経過");
    System.out.println(time1.toString() + "から" + duration.toHours() + "時間" + duration.toMinutes() % 60 + "分経過");

    // date1には、皆さんの誕生日を入れて下さい
    LocalDate date1 = LocalDate.of(1980, 8, 20);
    LocalDate date2 = LocalDate.now();

    // date1とdate2の年月差をとる
    Period period = Period.between(date1, date2);
    System.out.println(date1.toString() + "から" + period.toTotalMonths() + "ヶ月経過");
    System.out.println("あなたの年齢は" + period.toTotalMonths() / 12 + "歳");
}
```



# Stream API/ラムダ式 (Java 8)



# Stream API/ラムダ式

## Stream

配列、List、Set、Mapといったコレクションへの順次・並列処理を記述できるAPI  
(InputStream, OutputStreamといったxxxStreamとは違う)

## ラムダ式

関数型言語を意識して、Javaで関数のように処理を記述できるようにした式



# 例えば、Streamでリストの要素を表示する

List 22 10 19 38

```
List<Integer> list =  
    Arrays.asList(22, 10, 19, 38);  
  
for (Integer i : list) {  
    if (i % 2 == 0) {  
        System.out.println(i);  
    }  
}
```

要素*i*を順番に取り出して、要素分繰り返す

要素が偶数かどうかを判定する

偶数であれば、要素を表示する

List 22 10 19 38 Stream

```
List<Integer> list =  
    Arrays.asList(22, 10, 19, 38);  
  
list.stream()  
    .filter(i -> i % 2 == 0)  
    .forEach(i -> System.out.println(i));
```

listをstreamにする

streamの要素を偶数のものだけにする

残った要素を表示する

処理させたいことが増えても、より宣言的に・簡素に書ける！



# Stream API

```
list.stream()  
    .filter(i -> i % 2 == 0)  
    .forEach(i -> System.out.println(i))
```

← 生成処理  
← 中間処理  
← 終端処理

この部分がStreamAPI

StreamAPIは、

- ・**生成処理** (Streamを作成・生成する)
- ・**中間処理** (Streamの要素を抜き出し・フィルタ・変化させたり、Streamの設定を変える)
- ・**終端処理** (Streamの処理結果の形を決めて、処理を実行する)

の3つに分けられ、

生成処理.中間処理.中間処理....中間処理.終端処理 のようにつなげて宣言する。



# Streamの生成・中間・終端処理

staticメソッド	中間処理メソッド	終端処理メソッド	その他
<ul style="list-style-type: none"><li><a href="#">builder</a></li><li><a href="#">concat</a></li><li><a href="#">empty</a></li><li><a href="#">generate</a></li><li><a href="#">iterate</a></li><li><a href="#">of</a></li><li><a href="#">range</a></li><li><a href="#">rangeClosed</a></li></ul>	<ul style="list-style-type: none"><li><a href="#">asDoubleStream</a></li><li><a href="#">asLongStream</a></li><li><a href="#">boxed</a></li><li><a href="#">distinct</a> (<a href="#">distinct</a>)</li><li><a href="#">filter</a> (<a href="#">filter</a>)</li><li><a href="#">flatMap</a> (<a href="#">flatMap</a>) [ /2015-12-13 ]</li><li><a href="#">flatMapToDouble</a></li><li><a href="#">flatMapToInt</a></li><li><a href="#">flatMapToLong</a></li><li><a href="#">limit</a> (<a href="#">limit</a>)</li><li><a href="#">map</a> (<a href="#">map</a>)</li><li><a href="#">mapToDouble</a> (<a href="#">mapToDouble</a>)</li><li><a href="#">mapToInt</a> (<a href="#">mapToInt</a>)</li><li><a href="#">mapToLong</a> (<a href="#">mapToLong</a>)</li><li><a href="#">mapToObj</a></li><li><a href="#">onClose</a> [ /2015-12-13 ]</li><li><a href="#">parallel</a></li><li><a href="#">peek</a> (<a href="#">peek</a>)</li><li><a href="#">sequential</a></li><li><a href="#">skip</a> (<a href="#">skip</a>)</li><li><a href="#">sorted</a> (<a href="#">sorted</a>)</li><li><a href="#">unordered</a></li></ul>	<ul style="list-style-type: none"><li><a href="#">allMatch</a> (<a href="#">allMatch</a>)</li><li><a href="#">anyMatch</a> (<a href="#">anyMatch</a>)</li><li><a href="#">average</a></li><li><a href="#">collect</a> (<a href="#">collect</a>)</li><li><a href="#">count</a> (<a href="#">count</a>)</li><li><a href="#">findAny</a> (<a href="#">findAny</a>)</li><li><a href="#">findFirst</a> (<a href="#">findFirst</a>)</li><li><a href="#">forEach</a> (<a href="#">forEach</a>)</li><li><a href="#">forEachOrdered</a> (<a href="#">forEachOrdered</a>)</li><li><a href="#">iterator</a></li><li><a href="#">max</a> (<a href="#">max</a>)</li><li><a href="#">min</a> (<a href="#">min</a>)</li><li><a href="#">noneMatch</a> (<a href="#">noneMatch</a>)</li><li><a href="#">reduce</a> (<a href="#">reduce</a>) [ /2017-02-25 ]</li><li><a href="#">spliterator</a></li><li><a href="#">sum</a></li><li><a href="#">summaryStatistics</a></li><li><a href="#">toArray</a> (<a href="#">toArray</a>)</li></ul>	<ul style="list-style-type: none"><li><a href="#">close</a></li><li><a href="#">isParallel</a></li></ul>

引用：<http://www.ne.jp/asahi/hishidama/home/tech/java/stream.html>



# (個人的に) 特に良く使う処理

## staticメソッド

- [builder](#)
- [concat](#)
- [empty](#)
- [generate](#)
- [iterate](#)
- [of](#)
- [range](#)
- [rangeClosed](#)

## 中間処理メソッド

- [asDoubleStream](#)
- [asLongStream](#)
- [boxed](#)
- [distinct \(distinct\)](#)
- [filter \(filter\)](#)
- [flatMap \(flatMap\)](#) [/2015-12-13]
- [flatMapToDouble](#)
- [flatMapToInt](#)
- [flatMapToLong](#)
- [limit \(limit\)](#)
- [map \(map\)](#)
- [mapToDouble \(mapToDouble\)](#)
- [mapToInt \(mapToInt\)](#)
- [mapToLong \(mapToLong\)](#)
- [mapToObj](#)
- [onClose \[/2015-12-13\]](#)
- [parallel](#)
- [peek \(peek\)](#)
- [sequential](#)
- [skip \(skip\)](#)
- [sorted \(sorted\)](#)
- [unordered](#)

## 終端処理メソッド

- [allMatch \(allMatch\)](#)
- [anyMatch \(anyMatch\)](#)
- [average](#)
- [collect \(collect\)](#)
- [count \(count\)](#)
- [findAny \(findAny\)](#)
- [findFirst \(findFirst\)](#)
- [forEach \(forEach\)](#)
- [forEachOrdered \(forEachOrdered\)](#)
- [iterator](#)
- [max \(max\)](#)
- [min \(min\)](#)
- [noneMatch \(noneMatch\)](#)
- [reduce \(reduce\)](#) [/2017-02-25]
- [spliterator](#)
- [sum](#)
- [summaryStatistics](#)
- [toArray \(toArray\)](#)

## その他

- [close](#)
- [isParallel](#)

引用：<http://www.ne.jp/asahi/hishidama/home/tech/java/stream.html>



# (個人的に) 特に良く使う処理

staticメソッド	中間処理メソッド	終端処理メソッド	その他
<ul style="list-style-type: none"><li>• <a href="#">builder</a></li><li>• <a href="#">concat</a></li><li>• <a href="#">empty</a></li><li>• <a href="#">generate</a></li><li>• <a href="#">iterate</a></li><li>• <a href="#">of</a></li><li>• <a href="#">range</a></li><li>• <a href="#">rangeClosed</a></li></ul>	<p>複数のStreamを合成したStreamを作る</p> <ul style="list-style-type: none"><li>• <a href="#">distinct (distinct)</a></li><li>• <a href="#">filter (filter)</a></li><li>• <a href="#">flatMap (flatMap)</a> [/2015-12-12]</li><li>• <a href="#">map (map)</a></li><li>• <a href="#">mapToDouble (map.ToDouble)</a></li><li>• <a href="#">mapToInt (map.ToInt)</a></li><li>• <a href="#">mapToLong (map.ToLong)</a></li><li>• <a href="#">mapToObj (mapToObj)</a></li><li>• <a href="#"> onClose [/2015-12-13]</a></li><li>• <a href="#">parallel</a></li><li>• <a href="#">peek (peek)</a></li><li>• <a href="#">sequential</a></li><li>• <a href="#">skip (skip)</a></li><li>• <a href="#">sorted (sorted)</a></li><li>• <a href="#">unordered</a></li></ul>	<p>終端処理メソッド</p> <ul style="list-style-type: none"><li>• <a href="#">allMatch (allMatch)</a></li><li>• <a href="#">anyMatch (anyMatch)</a></li><li>• <a href="#">average (average)</a></li><li>• <a href="#">collect (collect)</a></li><li>• <a href="#">count (count)</a></li><li>• <a href="#">findAny (findAny)</a></li><li>• <a href="#">forEach (forEach)</a></li><li>• <a href="#">forEachOrdered (forEachOrdered)</a></li><li>• <a href="#">iterator (iterator)</a></li><li>• <a href="#">max (max)</a></li><li>• <a href="#">min (min)</a></li><li>• <a href="#">noneMatch (noneMatch)</a></li><li>• <a href="#">reduce (reduce) [/2017-02-25]</a></li><li>• <a href="#">spliterator (spliterator)</a></li><li>• <a href="#">sum (sum)</a></li><li>• <a href="#">summaryStatistics (summaryStatistics)</a></li><li>• <a href="#">toArray (toArray)</a></li></ul>	<p>その他</p> <ul style="list-style-type: none"><li>• <a href="#">close</a></li><li>• <a href="#">isParallel</a></li></ul>

引用：<http://www.ne.jp/asahi/hishidama/home/tech/java/stream.html>



# (個人的に) 特に良く使う処理

## staticメソッド

- builder
- concat
- empty
- generate
- iterate
- of
- range
- rangeClosed

## 中間処理

- asDoubleStream
- asLongStream
- boxed
- distinct (distinct)
- filter (filter)
- flatMap (flatmap)
- flatMapToDouble
- flatMapToInt
- flatMapToLong
- limit (limit)
- map (map)
- mapToDouble
- mapToInt (maptoint)
- mapToLong (maptolong)
- mapToObject
- onClose (onclose)
- parallel
- peek (peek)
- sequential
- skip (skip)
- sorted (sorted)
- unordered

プリミティブ型の要素をラッパー型の要素に変換する  
例) int のStream を Integer のStreamに

filter: 要素を抜き出す (if)

flatMap: 要素をネストする (forの入れ子)

要素の型を変換する 例) A型のStreamをB型のStreamに

プリミティブ型の要素を何らかの型の要素に変換する

例) int のStreamをA型のStreamに

Streamの中間処理・終端処理を並列処理にする

要素を並び替える

引用：<http://www.ne.jp/asahi/hishidama/home/tech/java/stream.html>



# (個人的に) 特に良く使う処理

## staticメソッド

- [builder](#)
- [concat](#)
- [empty](#)
- [generate](#)
- [iterate](#)
- [of](#)
- [range](#)
- [rangeClosed](#)

## 中間処理メソッド

- [asDoubleStream](#)
- [asLongStream](#)
- [boxed](#)
- [distinct \(distinct\)](#)
- [filter \(filter\)](#)
- [flatMap \(flatMap\)](#) [ /2015-12-13 ]
- [flatMapToDouble](#)
- [flatMapToInt](#)
- [flatMapToLong](#)
- [limit \(limit\)](#)
- [map \(map\)](#)
- [mapToDouble \(mapToDouble\)](#)
- [mapToInt \(mapToInt\)](#)
- [mapToLong \(mapToLong\)](#)
- [mapToObj](#)
- [onClose \[ /2015-12-13 \]](#)
- [parallel](#)
- [peek \(peek\)](#)
- [sequential](#)
- [skip \(skip\)](#)
- [sorted \(sorted\)](#)
- [unordered](#)

## 終端処理メソッド

- [allMatch \(allMatch\)](#)
- [anyMatch \(anyMatch\)](#)
- [average](#)
- [collect \(collect\)](#)
- [count \(count\)](#)
- [findFirst \(findFirst\)](#)
- [findFirst \(findFirst\)](#)
- [forEach \(forEach\)](#)
- [forEachOrdered \(forEachOrdered\)](#)
- [iterator](#)
- [max \(max\)](#)
- [min \(min\)](#)
- [noneMatch \(noneMatch\)](#)
- [reduce \(reduce\)](#)
- [spliterator](#)
- [sum](#)
- [summaryStatistics](#)
- [toArray \(toArray\)](#)

## その他

- [close](#)
- [isParallel](#)

Streamの要素からCollectionを作る

Streamの要素の最初の一つを貰う

Streamの要素すべてを使って何かを作る（例：合計、結合）

引用：<http://www.ne.jp/asahi/hishidama/home/tech/java/stream.html>



# ラムダ式

```
list.stream()  
.filter( i -> i % 2 == 0 )  
.forEach( i -> System.out.println(i) );
```

この部分がラムダ式

```
list.stream()  
.filter(i -> i % 2 == 0)  
.forEach(System.out::println);
```

メソッド参照

ラムダ式は、アロー演算子を使って

$x$  が●●のとき /  $x$ を○○にする といった式を

$x -> \bullet\bullet$  /  $x -> \circ\circ$  のように記述できる関数オブジェクト。

$x$ は、（直前の処理までが行われた）stream内の要素ひとつひとつを表す。

また、クラス（インスタンス）名::メソッド名 のように記述できる

メソッド参照 という記載方法もある。



# ラムダ式（の元となる関数型インターフェース）のパターン

**Supplier** (値の生成:引数なし, 返値あり)

式の形は  $x \rightarrow \circ\circ$

**Predicate** (条件判定:引数1つ, 返値はboolean)

式の形は  $x \rightarrow \circ\circ$

**Function** (値の変換:引数1つ, 返値あり)

式の形は  $x \rightarrow \circ$

**Consumer** (値の処理:引数1つ, 返値なし)

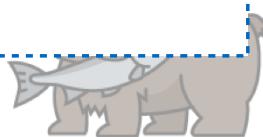
式の形は  $x \rightarrow \circ\circ$

filterで要素をしぼる時などに  
よく使われる

mapで要素の型を変更するときに  
よく使われる

Streamの結果で外の変数を変更したり、  
 $x \rightarrow \text{System.out.println}(x)$ のように  
Streamの結果の処理に利用する

ただし、Streamの外の変数をStreamの操作で変更するようなコードは、  
最終的に取り回しが悪くなる



# ラムダ式（の元となる関数型インターフェース）のパターン

BinaryOperator (値の変換:引数2つ, 返値あり)

式の形は  $(x, y) \rightarrow \text{○○}$  、Functionの派生

Streamの複数の結果を  
一つの要素にまとめる

## ラムダ式の代わりに使えるクラス（orインターフェース）

Collectors#toList, toSet, joining, toMap, groupingBy...

Stream#collectメソッドで、Streamの結果を新しいCollectionにする

Comparator#comparing, reversed...

Stream#sortedメソッドで、Streamの要素の並べ替え順を設定する



# やってみよう① (HandsOn8#Streamとこれまでの違い)

```
@Test
public void Streamとこれまでの違い() {
    List<Integer> nums = Arrays.asList(22, 10, 19, 38);

    for (Integer i : nums) {
        if (i % 2 == 0) {
            System.out.println(i);
        }
    }

    nums.stream()
        .filter(i -> i % 2 == 0)
        .forEach(i -> System.out.println(i));
}
```



## やってみよう②（HandsOn8#長さが5以上の単語だけをListに集める）

```
@Test  
public void 長さが5以上の単語だけをListに集める() {  
    List<String> filterdList = words.stream()  
        .filter(word -> word.length() >= 5)  
        .collect(Collectors.toList());  
  
    System.out.println(filterdList);  
}
```



## やってみよう③（HandsOn8#要素を単語の長さに変えてListに集める）

```
@Test  
public void 要素を単語の長さに変えてListに集める() {  
    List<Integer> lengthList = words.stream()  
        .map(word -> word.length())  
        .collect(Collectors.toList());  
  
    System.out.println(lengthList);  
}
```



## やってみよう④（HandsOn8#要素を単語の長さに変えて5以上の数字だけをListに集める）

```
@Test  
public void 要素を単語の長さに変えてListに集める() {  
    List<Integer> lengthList = words.stream()  
        .map(word -> word.length())  
        .collect(Collectors.toList());  
  
    System.out.println(lengthList);  
}
```



## やってみよう⑤ (HandsOn8#要素を文字のリストに変えてListに集める)

```
@Test  
public void 要素を文字のリストに変えてListに集める() {  
    List<String> chars = words.stream()  
        .map(word -> word.split(""))  
        .flatMap(word -> Arrays.stream(word))  
        .collect(Collectors.toList());  
  
    System.out.println(chars);  
}
```



# やってみよう⑥ (HandsOn8#単語と文字数のMapに集める)

```
@Test  
public void 単語と文字数のMapに集める() {  
    Map<String, Integer> map = words.stream()  
        .collect(Collectors.toMap(s -> s, s -> s.length()));  
  
    System.out.println(map);  
}
```



# やってみよう⑦（HandsOn8#一番短い単語を取得する）

```
@Test  
public void 一番短い単語を取得する() {  
  
    String defaultStr = "none.";  
    String ans = words.stream()  
        .sorted(Comparator.comparing(String::length))  
        .findFirst()  
        .orElse(defaultStr);  
  
    System.out.println(ans);  
}
```



## やってみよう⑧ (HandsOn8#一番長い単語を取得する)

```
@Test
public void 一番長い単語を取得する() {
    String defaultStr = "none.";
    String ans = words.stream()
        .sorted(Comparator.comparing(String::length).reversed())
        .findFirst()
        .orElse(defaultStr);

    System.out.println(ans);
}
```



## やってみよう⑨ (HandsOn8#単語の長さを合計する)

```
@Test
public void 単語の長さを合計する() {
    int defaultInt = 0;
    int ans = words.stream()
        .map(word -> word.length())
        .reduce(defaultInt, (x, y) -> x + y);

    System.out.println(ans);
}
```



## やってみよう⑩（HandsOn8#異なる処理をするStreamを合成して実行する）

```
@Test
public void 異なる処理をするStreamを合成して実行する() {
    Stream<String> s1 = words.stream()
        .filter(word -> word.length() >= 5)
        .map(String::toUpperCase);

    Stream<String> s2 = words.stream()
        .filter(word -> word.length() < 5)
        .map(String::toLowerCase);

    // ↑ どんなStreamを用意すればいいのか宣言のみ行われる

    List<String> ans = Stream.concat(s1, s2)
        .collect(Collectors.toList());

    // ↑ 終端処理(collect)でs1, s2の内容がすべて実行される

    System.out.println(ans);
}
```



# そのほか (Java 7, 8)



# そのほか、Java7,8で便利になったこと（の抜粋）

## nullチェック用のメソッド（7）

```
Objects.requireNonNull(str, "strがnull");
```

← strがnullの時、NPEをメッセージ付きで発生させる

## コンパレータ用のメソッド（7）

```
@Override  
public int compareTo(B other) {  
    // 小さい順に並び替える  
    return Integer.compare(x, other.getX());  
}
```

← ソート可能なクラス（Comparableインターフェースの実装クラス）で、並び替え順をメソッドで記述できる（Java7より前は、-1, 0, 1を返すわかりづらい形）

## 文字列結合メソッド（8）

```
List<String> strs = Arrays.asList("Hello", "Duke");  
System.out.println(String.join("_", strs)); //Hello_Duke
```

← Collectionから、接続文字を決めて文字列を結合できる



# そのほか、Java7,8で便利になったこと（の抜粋）

## equals, hashCode用の計算メソッド（7）

フィールドにx, yという変数があるようなクラスAがあるとき、equals, hashCode メソッドを用意することで、フィールドの値が一致 = インスタンスが同一 と判定できるようになる

### クラスAのコード（抜粋）

```
@Override  
public boolean equals(Object object) {  
    if (object instanceof A) {  
        A other = (A) object;  
        return Objects.equals(x, other.getX())  
            && Objects.equals(y, other.getY());  
    }  
    return false;  
}  
  
@Override  
public int hashCode() {  
    return Objects.hash(x, y);  
}
```

### 比較のコード例

```
A a1 = new A("hello", "world");  
A a2 = new A("hello", "world");  
assertTrue(a1.equals(a2)); // ← trueになる
```

←Java7より前は、もっと面倒な式が必要だった

参考：Javaの理論と実践：ハッシュの徹底 - IBM

<https://www.ibm.com/developerworks/jp/java/library/j-jtp05273/index.html>

Effective Java 第二版



# そのほか、Java7,8で便利になったこと（の抜粋）

## Optional : nullかもしれないインスタンス (8)

### コード例

```
String str1 = "a";
Optional<String> o1 = Optional.ofNullable(str1);
String str2 = null;
Optional<String> o2 = Optional.ofNullable(str2);

System.out.println("-----");
// nullじゃないときだけラムダ式が実行される
o1.ifPresent(s -> System.out.println(s));
o2.ifPresent(s -> System.out.println(s));

System.out.println("-----");
// nullの時は "nullです" を返す
System.out.println(o1.orElse("nullです"));
System.out.println(o2.orElse("nullです"));
```

### 実行結果

```
-----
a
-----
a
nullです
```

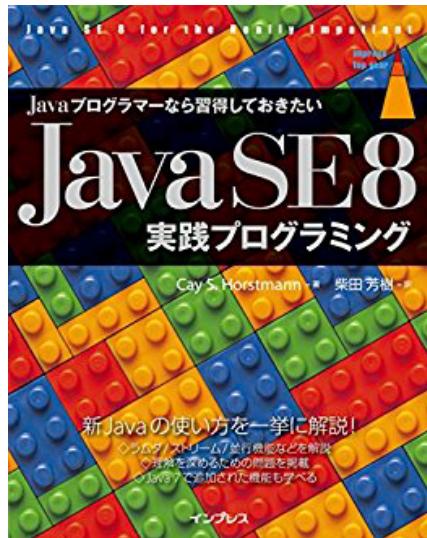
← Optionalには、nullになるかもしれない値を格納しておける。

ifPresent, orElseメソッドで、nullの時の動作を変更することができる



# 参考文献

Java7, Java8の変更点を  
より詳しく調べるなら



Cay S. Horstmann (著), 柴田 芳樹 (翻訳)  
Javaプログラマーなら習得しておきたい  
Java SE 8 実践プログラミング

Java7, Java8を使った  
モダンな書き方を身につけるなら



櫻庭祐一 (著)  
現場で使える [最新] Java SE 7/8 速攻入門

今回取り上げたものも、それ以外も、詳しく実用法が乗っています！

