中級者 になるための
Java勉強会
(学習->学習.日々学習());

@mkakimi

1. はじめに

タイトルは正しい Java8 の文法です

```
public class Main {
    public static void main(String[] args) {
        中級者 になるための = Java勉強会(学習->学習.日々学習());
    }

    public static 中級者 Java勉強会(Function<初級者, 中級者> c) {
        return c.apply(new 初級者());
    }
}
interface 級 { 級 日々学習(); }
class 初級者 implements 級 { public 中級者 日々学習() { return new 中級者(); } }
class 中級者 implements 級 { public 上級者 日々学習() { return new 上級者(); } }
class 上級者 implements 級 { public 上級者 日々学習() { return new 上級者(); } }
```

注意：中級者向け勉強会ではありません

初級者が中級者になれるといいな、という意図ですが本当は自分自身の勉強のためのアウトプット場所です.

※ちなみにJavaは識別子をUnicodeで扱うため、日本語で命名可能です. (Character#isJavaIdentifierPart())
通常のクラスやメソッド名に日本語を利用するのは避けるべきですが Junit のテストケース(メソッド)名などに利用する際は有効な場合もあります.

ちなみに Java7 までの書き方だと...

```
// Java8
中級者 になるための = Java勉強会(学習->学習.日々学習());

// Java7
中級者 になるための_= Java勉強会(new Function<初級者, 中級者>() {
    @Override
    public 中級者 apply(初級者 学習) {
        return 学習.日々学習();
    }
});
```

Java8は関数型言語の要素を取り入れたため、Java7までの匿名クラスの記述を簡素化できるようになりました．Java8での文法の変更は今までのJavaの歴史上、一番大きい変革かもしれません．

基礎編

2. 基礎編

Java について

この章では下記の事項を学ぶことができます.

- JVM の位置付け
- Java の基本的なプログラムに隠された暗黙的な約束事
- classpath を利用したコンパイル方法
- Jar ファイルの作成や解凍の方法

正しいつづりは「Java」

OK: JDK 9 / Java SE / Java EE / JavaFX
NG: JDK9 / JavaSE / J2EE / Java FX

現在の最新バージョンは JDK 8 (1.8)

2016-05-28 時点での最新版は JDK 8u92

JDK 7 (1.7) は 2015-04 で既にEOL

Java SE 7 以前のバージョンは既にEOLのため、セキュリティを考慮する場合は Java SE 8 以上を利用するようにしましょう.

JDK 9 (1.9) は現在開発中

時期バージョンである Java SE 9 は 2017年3月ごろリリース予定です. 以前より話題になっているプロジェクト Jigsaw などが盛り込まれる予定です.

バージョン番号の整理

JDK 1.0 -> JDK 1.1 -> J2SE 1.2 (J2と呼ばれ始める) -> J2SE 1.4 -> J2SE 5.0 (1.5ではない) -> Java SE 6 (表記が統一される) -> Java SE 7 -> Java SE 8 -> Java SE 9

この内、バージョン5と8で大きな変更が加えられています.

- Qiita 複雑怪奇なJavaの名称とバージョン番号を整理する: http://qiita.com/kinmojr/items/fb291da7c9b20906b083

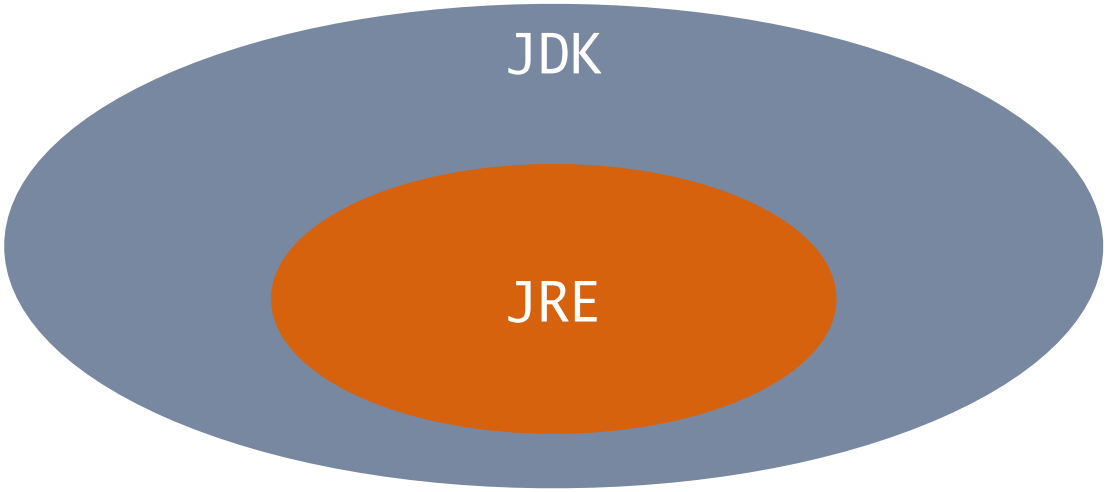
JRE と JDK の違いは？

JRE (Java Runtime Environment)

Java プログラムの実行環境(JVM)を提供する．java コマンドを実行することでJVMを起動し、指定された Java プログラムを実行する．
JVM や標準ライブラリ(rt.jar など)の実行用ファイルを含んでいます．

JDK (Java Development Kit)

JRE を含めた開発者向けセットです．
コンパイラ(javac)や開発に役立つツールなどを含んでいます．



そのため、これから Java を利用して開発する方は JDK を導入しコンパイル・実行環境を整えてください．
Java アプリケーションは JRE を導入すれば実行可能となりますが、Webアプリケーションなどを運用する環境の場合、JDK 付属のツールが役立つため、開発環境だけでなく本番環境でも JDK を導入すると良いです．

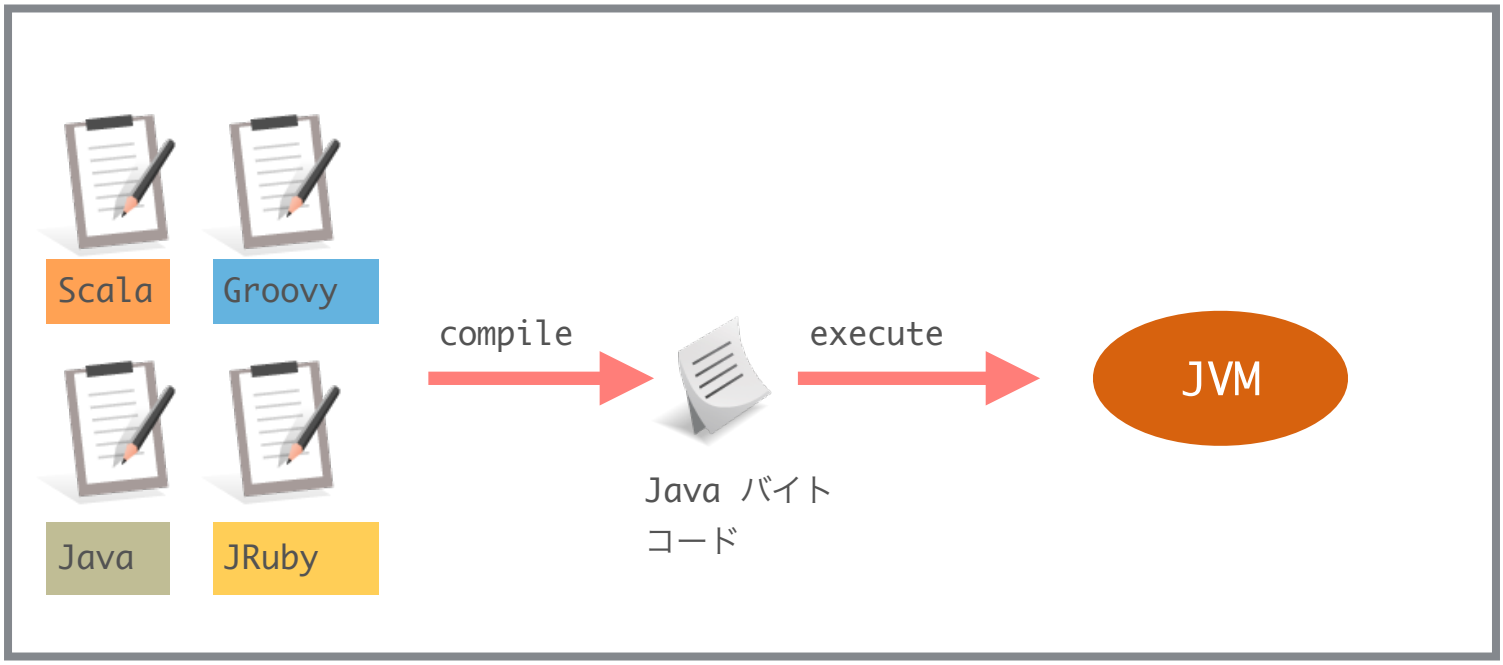
JVM は Java だけのものではない

JVM は中間言語である Java バイトコード(*.class)を解釈する

Java Virtual Machine (JVM) は *.java ファイルを解釈するのではなく、*.class ファイルである Java バイトコードを解釈してプログラムを実行します.

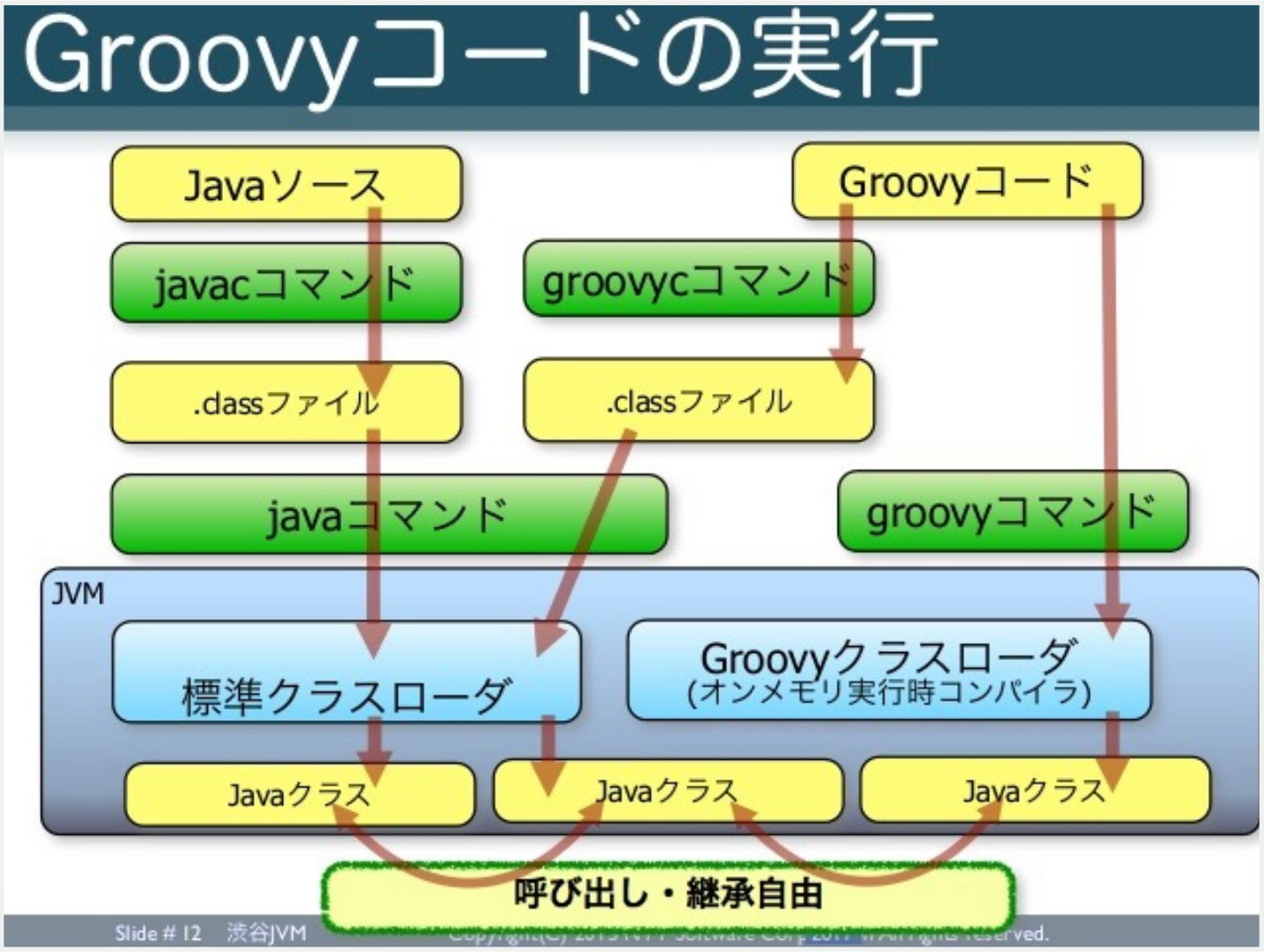
そのため、JVM 上でプログラムを動作させたい場合、Java バイトコードさえ作成できれば言語は Java である必要は無いということです.

Scala、JRuby、Groovy、Clojure などのプログラミング言語は、コンパイルすることで Java バイトコードを出力するため、これらの言語はJVM言語と呼ばれます.



そのため、Java の言語仕様だけでなく JVM の動作(GC やチューニングパラメータ)についても知ることで、その他の JVM 言語を利用した際にもその知識が役に立ちます.

Groovyの場合



(下記より引用)

- 今さら始めようGroovy: http://www.slideshare.net/uehaj/shibuya-jvm-groovy-20150418

Hello, World から読み解く Java

- /japs/src/main/java/org/japs/basic/compile/Main.java

```
package org.japs.basic.compile;

public class Main {
    public static void main(String[] args) {
        String hello = "Hello";
        String world = new String("World");
        System.out.println(hello + world);
    }
}
```

- プログラムは main() メソッドから開始する (public static void)
- java.lang パッケージには暗黙的に import される
- クラス(class)がファーストクラスオブジェクト
- public なクラスはファイル名と一致させる必要がある
- フォルダ構成とパッケージ構成をあわせる必要がある
- 変数の宣言、代入は「型 変数名 = 値;」
- インスタンスの生成は「new コンストラクタ()」
- 文字列の結合は「+」演算子で可能
- static 修飾子はフィールドやメソッドに付加できる (classも一部可)
- 「[]」は配列を表す

まずはコンパイルを行う

Java はコンパイラ言語です．JDK 付属の javac コマンドにより、コンパイルすることで中間言語(Java バイトコード)に変換されます．

```
*.java -> (javac) -> *.class ファイル
```

下記は japs/src/main/java/org/japs/basic/compile/Main.java ファイルをコンパイルする例です．
※ -d オプションにはコンパイルにより生成された class ファイルを出力するディレクトリを指定します．-cp オプションについては後述の「classpath について」の項で説明しますので、ここでは気にせずそのまま指定してください．

```
$ pwd
/Users/m-kakimi/Documents/workspace/japs
$ javac -d bin -cp src/main/java/ src/main/java/org/japs/basic/compile/Main.java
$ ls bin/org/japs/basic/compile/
Main.class
```

プログラムは main() メソッドから開始する

Java プログラムを実行する際はコンパイルによって作成された class ファイルを指定するわけではなく、実行対象とする main() メソッドを保持するクラス名を指定します．

```
$ ls bin/org/japs/basic/compile/
Main.class
$ java -cp bin org.japs.basic.compile.Main
HelloWorld
```

誤って class ファイルを指定して実行した場合は下記のような実行エラーとなります．

```
$ java bin/org/japs/basic/compile/Main.class
エラー: メイン・クラスbin.org.japs.basic.compile.Main.classが見つからなかったか ロードできませんでした
```

JVM は class ファイルを逐次解釈(インタープリット)しますが、常に同じように解釈し続けるのではなく、何度も呼び出されるような処理については JIT (Just In Time) コンパイラにより、プログラム実行中に内部的にコンパイルされ、高速化を図っています．

java.lang パッケージは暗黙的に import される

String 型は文字列を扱うクラスですが、完全修飾名は java.lang.String です. Java では、他パッケージのクラスを(パッケージ指定無しで)使用する場合には import する必要があります.

例: import jp.co.hoge;

但し、java.lang パッケージについては暗黙的にインポートされるため import 文に指定する必要はありません.

そのため、サンプルプログラム上でも、java.lang.String と記述するのではなく import 文も無しに String と直接記述しています.
java.lang 以外のパッケージに属するクラスを利用したい場合は、import 文を記述することで利用可能となります. import 文を記述しない場合は、下記のように完全修飾名でアクセスする必要があります.

パッケージを明示(完全修飾名で)指定する場合はクラス名の前にドット区切りで指定します.

```
public static void main(java.lang.String[] args) {
```

また、異なるパッケージで同一名称のクラス(例えば java.util.List と java.awt.List)を同時に扱いたい場合、import 文ではどちらか片方しか指定できないため、import されていない方のクラスについては完全修飾名で扱う必要があります.

```
import java.util.List;
...
List ul;
java.awt.List al;
```

クラス(class)がファーストクラスオブジェクト

```
public class Main { }
```

First-Class Object 第一級オブジェクト(以下FCO)とは、その単位で生成、代入、関数(メソッド)の引数への受け渡しが可能な単位のことです.

Java 言語での FCO はクラスのため、メソッドの引数にメソッドを渡すことなどは不可です.

例えば JavaScript では、関数(function)が FCO のため関数の引数に関数を渡すことが可能です.
Java8 では関数型言語的な記述が可能となりましたが、メソッドは FCO はでないことに変わりは無いため、内部的にはクラス(のインスタンス)が受け渡しされています.

public なクラスはファイル名と一致させる必要がある

- /japs/src/main/java/org/japs/basic/compile/Main.java

```
public class Main { }
```

public なクラスを定義する際は、ファイル名と一致させる必要があります．一致していない場合は下記のコンパイルエラーが発生します．

```
$ javac -cp .:bin -d bin src/main/java/org/japs/basic/compile/Main.java
src/main/java/org/japs/basic/compile/Main.java:3: エラー: クラスMainxはpublicであり、
ファイルMainx.javaで宣言する必要があります
public class Mainx {
       ^
エラー1個
```

public なクラスはファイル内に1つだけ定義可能です．public でないクラスは1ファイル内に複数定義可能です．
また、java ファイル内に public クラスの定義が無くてもよいです．

フォルダ構成とパッケージ構成をあわせる必要がある

- /japs/src/main/java/org/japs/basic/compile/Main.java

```
package org.japs.basic.compile;

public class Main { ...
```

class ファイルの配置階層とパッケージ構成は同一である必要があり、classpath(後述)を起点とした階層に配置します．
ソースファイルの配置階層についてはパッケージ構成と合わせる必要はありませんが、一般的には同一階層にします．
また、パッケージ名については一般的に jp.co.hoge のようにドメインを逆向きに命名し、全世界でユニークとなるようにします．

パッケージ構成や classpath の仕組みについては、JDK 9 で大きな変更が行われる可能性有るかもしれません．(Project Jigsaw)
10年近く前から JSR 277 や JSR 294 として紆余曲折しつつ進められています．現在は JSR 376 が Active 状態で進められているようです．

- Project Jigsaw: http://www.slideshare.net/skrb/module-programming-with-project-jigsaw
- OracleがJavaモジュールシステムの状況を報告: http://www.infoq.com/jp/news/2015/11/java-state-module-system

変数の宣言、代入は「型 変数名 = 値;」

```
        String hello = "Hello";
```

Java は型システムを持つ言語です．Java での「型」は大きく分けて基本的な「値」を格納する「プリミティブ型」とインスタンスへの「参照」を格納する「参照型」に分けることができます．

プリミティブ型

```
    int     a = 10;          // 32bit 整数 (default: 0)
    long    b = 10L;         // 64bit 整数 (default: 0L)
    short   c = 10;          // 16bit 整数 (default: 0)
    float   d = 10.0f;       // 32bit 浮動小数点 (default: 0.0f)
    double  e = 10.0;        // 64bit 浮動小数点 (default: 0.0)
    boolean f = true;        // 真偽値 (true or false) (default: false)
    char    g ='x';          // 16bit 整数 (default: \0)
    byte    h = 0x0a;        // 8bit 整数 (16進数記法) (default: 0)
    byte    i = 012;         // 8bit 整数 (8進数記法)
    byte    j = 0b0000_1010; // 8bit 整数 (2進数記法)
```

Java の数値型には C 言語のような unsighed 型はありません．(Java8 では、参照型として間接的に扱うことが可能になったようです)

-Java8 で入った unsigned 系メソッド: http://d.hatena.ne.jp/ta6ra/20141205

参照型(クラス型)

- 参照型はクラス型と呼ぶこともあります．
- クラス(java.lang.String など)全般が参照型となります．
- 参照型にはラッパークラスと呼ばれる、プリミティブ型のそれぞれに対応する型が java.lang パッケージに用意されています．

  int   -> Integer
  long   -> Long
  short   -> Short
  float   -> Float
  boolean -> Boolean
  char   -> Character
  byte   -> Byte
- new キーワードを利用して値を設定します．但し、String 型については言語仕様でダブルクォーテーション("")を用いたインスタンス作成をサポートしています．(詳細は後述)
- default 値は「null」となります．

上記記載のデフォルト値は、フィールドとして定義した場合の初期値です．ローカル変数の場合は、プリミティブ型も参照型も初期値は「不定」となり、初期化せず利用しようとするとコンパイルエラーとなります．

サロゲートペア

char 型は内部的に Unicode(UTF-16) で文字を表現していますが、前述の通り 2 バイト型のため「サロゲートペア」となる文字を格納することはできず、コンパイルエラーとなります.

```
        char c = '𩸽'; // ほっけ
```

```
src/main/java/org/japs/basic/compile/Main.java:9: エラー: 文字リテラルが閉じられていません
            char c = '𩸽'; // ほっけ
                     ^
src/main/java/org/japs/basic/compile/Main.java:9: エラー: '\ude3d'は不正な文字です
            char c = '𩸽'; // ほっけ
                      ^
src/main/java/org/japs/basic/compile/Main.java:9: エラー: 文字リテラルが閉じられていません
            char c = '𩸽'; // ほっけ
                       ^
エラー3個
```

インスタンスの生成は「new コンストラクタ()」

```
        String world = new String("World");
```

インスタンス

new + コンストラクタで生成され、メモリ上に格納されるクラスの実体です. クラス型にはこのインスタンスへの格納先を指し示す参照値が格納されます.

コンストラクタ

コンストラクタはクラス名と同名のメソッド定義です. 返り値は自身のインスタンスと決まっているため、通常のメソッドと異なり返り型は指定しません.

ファクトリメソッドパターン(デザインパターン)

Java に限ったことではありませんが、標準ライブラリの中にもファクトリメソッドで生成する場合も多く存在します. 内部的には new キーワードとコンストラクタで生成されていることに違いはありません.

```
        LocalDate now = LocalDate.now();
```

ファクトリメソッドパターンを使うことで下記のようなメリットがあります.

- コンストラクタの場合、同一シグニチャだとオーバーロードできませんが、ファクトリメソッドの場合、同一シグニチャでもメソッド名を変えることで複数定義可能です.
- 単なるメソッドのため、自由に命名可能です. 特定用途のインスタンスを生成する場合など、名前で表現することができます.
- シングルトンパターン(デザインパターン)などのように、インスタンス生成時の制御をすることが可能です.

- Examples of GoF Design Patterns in Java's core libraries: http://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries

文字列の結合は「+」演算子で可能

```
        System.out.println(hello + world);
```

文字列の結合は「+」演算子で可能ですが、結合の度に String のインスタンスが生成され、効率が悪いため、何度も文字列連結するような処理については内部的にバッファを利用する java.lang.StringBuilder を利用したほうが速度が早くなります.
最近はコンパイル時に自動的に「+」による文字列結合を StringBuilder に変換してくれるようですが、確実に速度を確保したい場合は明示的に実装するのがよいでしょう.

- /japs/src/main/java/org/japs/basic/string/builder/Main.java

```
        StringBuilder hw =new StringBuilder();
        hw.append("Hello");
        hw.append("World");
        System.out.println(hw);
```

また、StringBuilder と同様のクラスに java.lang.StringBuffer がありますが、こちらは各メソッドがスレッドセーフ(synchronized)実装となっているため、ロック処理が必要な場合以外は StringBuilder を利用しましょう. ローカル変数として定義する場合など、必ずシングルスレッドとして利用される場合はロックする必要は無いため、ほとんどの場合は StringBuilder で事が済みます.

「+」演算子を利用して結合する場合は、結合順序にも注意してください. 下記のような場合、左側から順に結合されるため意図した結果にならない場合があります.

- /japs/src/main/java/org/japs/basic/string/Main.java

```
        System.out.println(1 + "x");     // 1x
        System.out.println(1 + 2 + "x"); // 3x
        System.out.println("x" + 1 + 2); // x12
```

static 修飾子はフィールドやメソッドに付加できる

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

static を付加したフィールドをクラスフィールド、または静的フィールドと呼びます.
static を付加したメソッドをクラスメソッド、または静的メソッドと呼びます.
クラスメソッドやクラスフィールドはインスタンスではなく、クラスに対して紐づく定義となります.

C 言語などのようにローカル変数に static を付加することはできません.
一部の class (ネストクラス)に対しても static を付加することが可能です.

「[]」は配列を表す

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

「[]」で配列を表します.
配列を生成する場合は下記の方法があります.

- 配列生成1「new 型[サイズ]」
- 配列生成2「new 型[] = { 値1, 値2, ... };」
- 配列生成3「new 型[] = new 型[] { 値1, 値2, ... };」

「...(ドット3つ)」で可変長配列を利用することも可能です. 可変長配列は実引数を渡す場合に柔軟に指定することが可能です.
プリミティブ型の配列も、参照型扱いとなるため注意してください. 引数として渡した先で要素の値を変更した場合、渡し元側でも要素値が変更されていることになります.

- /japs/src/main/java/org/japs/basic/array/Main.java

```
package org.japs.basic.array;

public class Main {
	public static void main(String[] args) {
		int[] a = { 1, 2 };
		int[] b = new int[] { 1, 2 };
		//NG hello({ 1, 2 });
		hello(a); // 2 1
		hello(a); // 2 9
		world(1, 2); // 2 1
	}
	public static void hello(int[] x) {
		System.out.print(x.length);
		System.out.print(" ");
		System.out.println(x[0]);
		x[0] = 9;
	}
	public static void world(int... x) {
		System.out.print(x.length);
		System.out.print(" ");
		System.out.println(x[0]);
	}
}
```

classpath について

classpath 概要

classpath とは、コンパイル時(javac)や実行時(java)に、対象がコンパイルや実行に必要となるクラスファイルを検索するパスのことです．Java の基本クラス群(rt.jar)などは明示しなくても自動的に読み込まれます．指定する必要があるのは、自身で作成したクラスや、jar ファイル内のクラスを利用する際に指定する必要があります．また、classpath を指定しなかった場合はデフォルトでカレントディレクトリが対象となります．指定方法には2つの方法があります．

CLASSPATH環境変数を設定するか、または -classpath(-cp) オプションで指定します．コマンド実行毎に指定できるため -cp オプションによる指定が推奨です．

```
$ pwd
/Users/m-kakimi/Documents/workspace/japs

$ javac -d bin -cp bin: src/main/java/org/japs/basic/compile/ClasspathMain.java

$ java -cp bin: org.japs.basic.compile.ClasspathMain
Hello, Classpath A.
```

classpath に複数の値を指定する場合は区切り文字で区切って指定します．
※区切り文字: Win:[;(セミコロン)] Unix:[:(コロン)]

- Javaの道 クラスパス: http://www.javaroad.jp/java_basic2.htm

サンプルクラスと階層図

例として使用したクラスは下記の定義と配置になっています.

- /japs/src/main/java/org/japs/basic/compile/ClasspathMain.java

```
package org.japs.basic.compile;

public class ClasspathMain {
    public static void main(String[] args) {
        org.japs.basic.classpath.Classpath.print();
    }
}
```

- /japs/src/main/java/org/japs/basic/compile/ClasspathMain.java

```
package org.japs.basic.classpath;

public class Classpath {
    public static void print() {
        System.out.println("Hello, Classpath A.");
    }
}
```

```
/Users/m-kakimi/Documents/workspace/japs
|--bin
|--main
|  |--java
|  |  |--org
|  |  |  |--japs
|  |  |  |  |--basic
|  |  |  |  |  |--classpath
|  |  |  |  |  |  |--Classpath.java
|  |  |  |  |  |--compile
|  |  |  |  |  |  |--ClasspathMain.java
```

コンパイル時のクラス検索確認

コンパイル時のクラスファイル検索の様子を確認するため、javac コマンドに -verbose オプションを付加してみます.
例では classpath (-cp) に指定しているのは bin ディレクトリのみですが、実際には rt.jar などが読み込まれ、その中に格納されている java.lang.Object や java.lang.String クラスが読み込まれていることがわかります. JDK の基本となる検索パスの後に、javac コマンドで指定した bin ディレクトリが付加されています.

```
$ javac -d bin -cp bin: -verbose src/main/java/org/japs/basic/compile/
ClasspathMain.java
[RegularFileObject[src/main/java/org/japs/basic/compile/ClasspathMain.java]を構文解
析開始]
[14ミリ秒で構文解析完了]
[ソース・ファイルの検索パス: bin,.]
[クラス・ファイルの検索パス: /Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/
Contents/Home/jre/lib/resources.jar,/Library/Java/JavaVirtualMachines/
jdk1.8.0_66.jdk/Contents/Home/jre/lib/rt.jar,/Library/Java/JavaVirtualMachines/
jdk1.8.0_66.jdk/Contents/Home/jre/lib/sunrsasign.jar,/Library/Java/
JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/jre/lib/jsse.jar,/Library/Java/
JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/jre/lib/jce.jar,/Library/Java/
JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/jre/lib/charsets.jar,/Library/
Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/jre/lib/jfr.jar,/Library/
Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/jre/classes,/Library/Java/
JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/jre/lib/ext/cldrdata.jar,/
Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/jre/lib/ext/
dnsns.jar,/Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/jre/lib/
ext/jaccess.jar,/Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/
jre/lib/ext/jfxrt.jar,/Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/
Home/jre/lib/ext/localedata.jar,/Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/
Contents/Home/jre/lib/ext/nashorn.jar,/Library/Java/JavaVirtualMachines/
jdk1.8.0_66.jdk/Contents/Home/jre/lib/ext/sunec.jar,/Library/Java/
JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/jre/lib/ext/sunjce_provider.jar,/
Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/jre/lib/ext/
sunpkcs11.jar,/Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/jre/
lib/ext/zipfs.jar,/System/Library/Java/Extensions/MRJToolkit.jar,bin,.]
[ZipFileIndexFileObject[/Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/
Home/lib/ct.sym(META-INF/sym/rt.jar/java/lang/Object.class)]を読込み中]
[ZipFileIndexFileObject[/Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/
Home/lib/ct.sym(META-INF/sym/rt.jar/java/lang/String.class)]を読込み中]
[org.japs.basic.compile.ClasspathMainを確認中]
[ZipFileIndexFileObject[/Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/
Home/lib/ct.sym(META-INF/sym/rt.jar/java/io/Serializable.class)]を読込み中]
[ZipFileIndexFileObject[/Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/
Home/lib/ct.sym(META-INF/sym/rt.jar/java/lang/AutoCloseable.class)]を読込み中]
[RegularFileObject[bin/org/japs/basic/classpath/Classpath.class]を読込み中]
[RegularFileObject[bin/org/japs/basic/compile/ClasspathMain.class]を書込み完了]
[合計149ミリ秒]
```

Jar について

Jarとは

jarとは Java Archive の略で複数の class ファイルをまとめた(アーカイブした)ファイルのことです．多くのオープンソースライブラリやフレームワークはこの jar ファイル形式で提供されます．自身で作成したプログラムをライブラリとして提供する場合も、基本的にこの jar ファイル形式にして提供することになります．
特殊なファイルに思えますが、実は zip 形式で圧縮されているため、通常の zip 解凍ソフトや unzip コマンドで解凍可能です．

jar ファイルは下記のような特徴があります．

- /META-INF/MANIFEST.MF ファイルが格納される
- jar コマンドで作成可能
  - -c …… 新規作成
  - -f …… jar ファイル名指定

bin ディレクトリ配下を jar ファイル化する場合は下記のように行います．

```
$ pwd
/Users/m-kakimi/Documents/workspace/japs

$ jar -cf build/libs/japs-0.1.jar -C bin .

$ ll build/libs/japs-0.1.jar
-rw-r--r--  1 m-kakimi  staff  15853  3 21 15:06 build/libs/japs-0.1.jar

$ unzip -d build/libs/japs-0.1 build/libs/japs-0.1.jar
Archive:  build/libs/japs-0.1.jar
   creating: build/libs/japs-0.1/META-INF/
  inflating: build/libs/japs-0.1/META-INF/MANIFEST.MF
  creating: build/libs/japs-0.1/org/
...
```

classpath の指定順と Jar ファイルの関係

jar ファイル内のクラスを利用したい場合は classpath に、対象の jar ファイルを指定しますが、下記の点に注意してください.

classpathは記載した順にクラスを検索するため、同一パッケージの同一クラス名のクラスファイルが存在する場合、classpathの指定順で動作が変わる.

実際の動きを見てみましょう. 下記の例では japs-classpathA.jar には「Hello, Classpath A.」と表示するクラスが含まれており、japs-classpathB.jar には同一階層(パッケージ)に「Hello, Classpath B.」と表示するクラスが含まれているとします.

- /japs/build/libs/japs-classpathA.jar # /japs/src/main/java/org/japs/basic/classpath/Classpath.java

```
    public static void print() {
        System.out.println("Hello, Classpath A.");
    }
```

- /japs/build/libs/japs-classpathB.jar # /japs/src/main/java/org/japs/basic/classpath/Classpath.java

```
    public static void print() {
        System.out.println("Hello, Classpath B.");
    }
```

classpath に指定する順番を japs-classpathA.jar を先にした場合は、「Hello, Classpath A.」と表示されます.

```
$ java -cp build/libs/japs-classpathA.jar:build/libs/japs-classpathB.jar
org.japs.basic.compile.ClasspathMain
Hello, Classpath A.
```

次に classpath に指定する順番を japs-classpathB.jar を先にした場合は、「Hello, Classpath A.」と表示されます.

```
$ java -cp build/libs/japs-classpathB.jar:build/libs/japs-classpathA.jar
org.japs.basic.compile.ClasspathMain
Hello, Classpath B.
```

アスタリスクでワイルドカード指定した場合は、「Hello, Classpath A.」と表示されました.

```
$ java -cp 'build/libs/*' org.japs.basic.compile.ClasspathMain
Hello, Classpath A.
```

(参考) jarコマンドのヘルプ

```
$ jar
使用方法: jar {ctxui}[vfmn0PMe] [jar-file] [manifest-file] [entry-point] [-C dir]
files ...
オプション:
    -c  アーカイブを新規作成する
    -t  アーカイブの内容を一覧表示する
    -x  指定の(またはすべての)ファイルをアーカイブから抽出する
    -u  既存アーカイブを更新する
    -v  標準出力に詳細な出力を生成する
    -f  アーカイブ・ファイル名を指定する
    -m  指定のマニフェスト・ファイルからマニフェスト情報を取り込む
    -n  新規アーカイブの作成後にPack200正規化を実行する
    -e  実行可能jarファイルにバンドルされたスタンドアロン・
        アプリケーションのエントリ・ポイントを指定する
    -0  格納のみ。ZIP圧縮を使用しない
    -P  ファイル名の先頭の'/' (絶対パス)および\"..\" (親ディレクトリ)コンポーネントを保持する
    -M  エントリのマニフェスト・ファイルを作成しない
    -i  指定のjarファイルの索引情報を生成する
    -C  指定のディレクトリに変更し、次のファイルを取り込む
ファイルがディレクトリの場合は再帰的に処理されます。
マニフェスト・ファイル名、アーカイブ・ファイル名およびエントリ・ポイント名は、
フラグ'm'、'f'、'e'の指定と同じ順番で指定する必要があります。

例1: 2つのクラス・ファイルをアーカイブclasses.jarに保存する:
       jar cvf classes.jar Foo.class Bar.class
例2: 既存のマニフェスト・ファイル'mymanifest'を使用し、foo/ディレクトリの
           全ファイルを'classes.jar'にアーカイブする:
       jar cvfm classes.jar mymanifest -C foo/
```

ハンズ・オン

1. 作業に必要なリソースの取得

```
$ git clone https://github.com/hatimiti/japs.git
$ cd japs
```

2. javac コマンドによるコンパイル

- /japs/src/main/java/org/japs/basic/compile/Main.java をコンパイルしてみよう

```
$ javac -d bin src/main/java/org/japs/basic/compile/Main.java
$
```

3. java コマンドによる実行

- /japs/src/main/java/org/japs/basic/compile/Main.java を実行してみよう

```
$ java -cp bin org.japs.basic.compile.Main
HelloWorld
$
```

4. classpath の指定

- /japs/src/main/java/org/japs/basic/compile/ClasspathMain.java をコンパイルしてみよう

```
$ javac -d bin src/main/java/org/japs/basic/compile/ClasspathMain.java
src/main/java/org/japs/basic/compile/ClasspathMain.java:5: エラー: パッケージ
org.japs.basic.classpathは存在しません
                org.japs.basic.classpath.Classpath.print();
                                        ^
エラー1個
```

別のクラスを利用しているため classpath を指定しないとエラーとなります．先に「org.japs.basic.classpath.Classpath」クラスを bin ディレクトリへコンパイルし、ClasspathMain クラスをコンパイルする際に bin ディレクトリを classpath として指定しましょう．

```
$ javac -d bin src/main/java/org/japs/basic/classpath/Classpath.java
$ javac -d bin -cp bin src/main/java/org/japs/basic/compile/ClasspathMain.java
$
```

- /japs/src/main/java/org/japs/basic/compile/ClasspathMain.java を実行してみよう

```
$ java -cp bin org.japs.basic.compile.ClasspathMain
Hello, Classpath B.
$
```

5. jar ファイルの作成

- /japs/bin 配下を jar ファイルにしてみよう

```
$ ll *.jar
ls: *.jar: No such file or directory
$ jar -cf japs.jar -C bin .
$ ll *.jar
-rw-r--r--  1 m-kakimi  staff  46698  5 31 08:23 japs.jar
```

6. jar ファイルを利用して実行

- 作成した jar ファイルを利用して /japs/src/main/java/org/japs/basic/compile/ClasspathMain.java を実行してみよう

```
$ java -cp japs.jar org.japs.basic.compile.ClasspathMain
Hello, Classpath B.
```

クラス編

3. クラス編

クラスについて

この章では下記の事項を学ぶことができます.

- 基本的なクラスの作成方法
- 値クラスなどを作成する際に重要となる `java.lang.Object` のメソッド実装方法
- `assert` キーワードの利用方法

- /japs/src/main/java/org/japs/basic/type/_class/Person.java

```
package org.japs.basic.type._class;

/**
 * 人間の情報を表現するクラス.
 * @author m-kakimi
 *
 */
public final class Person {

    /** 身長 */
    private int height;
    /** 体重 */
    private int weight;

    public int getHeight() {
        return height;
    }

    public int getWeight() {
        return weight;
    }

}
```

クラスは class キーワードで定義

クラスを定義する場合は `class` キーワードを利用します. `class` キーワードの前に `final` を付加すると、このクラスの継承を抑制することが可能です.

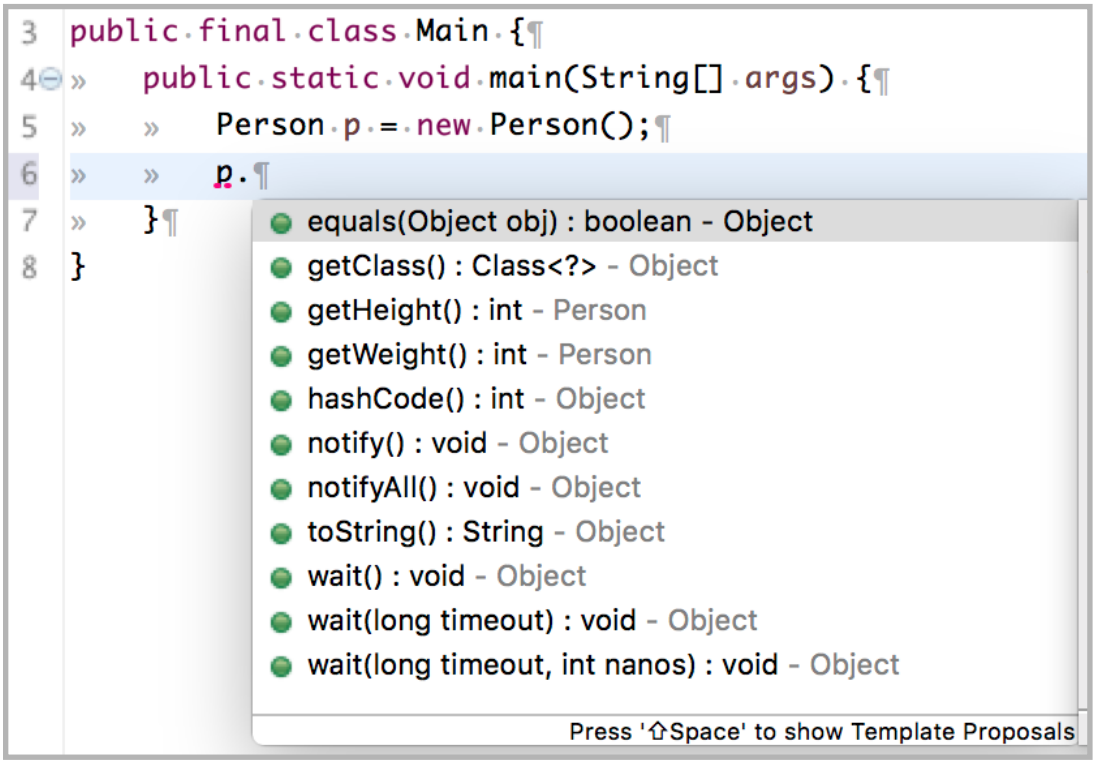
```
class ExPerson extends Person { }
 -> The type ExPerson cannot subclass the final class Person
```

暗黙的に java.lang.Object を継承している

クラスを継承する際には extends キーワードをクラス定義に付加するのですが、Java では、exntends 指定していない場合も java.lang.Object クラスを暗黙的に継承しています.
下記のように定義されているイメージです.

```
public final class Person extends java.lang.Object {
```

そのため、上記 Person クラスは java.lang.Object クラスで定義されているメソッドについても呼び出すことが可能です.
下記のように Person クラスのインスタンスを利用しようとした場合に、getHeight() や getWeight() メソッド以外も呼び出せることが確認できます.



java.lang.Object に定義されているメソッドの内、よく利用するものは下記です.

getClass() メソッド

```
    public final native Class<?> getClass();
```

getClass() メソッドは、当インスタンスの java.lang.Class クラスオブジェクトを取得します. 業務プログラム中で利用することは少ないかもしれませんが、フレームワークなどの共通処理を記述する際にクラス情報を扱いたい場合などに利用します. native(JNI)実装のため、具体的な実装は参照できません.
また、静的に特定クラスの Class インスタンスを取得したい場合は .class を利用します.

```
        System.out.println(p.getClass() == Person.class); // true
```

equals() メソッド

```
    public boolean equals(Object obj) {
        return (this == obj);
    }
```

equals() メソッドは、当インスタンスが、引数に指定された obj と「同一」とみなせるかを真偽値で返答するように実装します．それには何を以って「同一」かという定義を実装する必要があります．デフォルト(java.lang.Object)の実装では「==」で比較しているため、当インスタンスが指し示す参照先が同一かどうかで結果が確定します．
フィールドの値により同一性を表したい場合は、当メソッドをオーバーライドして再定義します．但し、equals() メソッドは javadoc に記載があるように、下記のルールに従いオーバーライドする必要があります．

1. 反射性(reflexive): null以外の参照値xについて、x.equals(x)はtrueを返します。
2. 対称性(symmetric): null以外の参照値xおよびyについて、y.equals(x)がtrueを返す場合に限り、x.equals(y)はtrueを返します。
3. 推移性(transitive): null以外の参照値x、y、およびzについて、x.equals(y)がtrueを返し、y.equals(z)がtrueを返す場合、x.equals(z)はtrueを返します。
4. 一貫性(consistent): null以外の参照値xおよびyについて、x.equals(y)の複数の呼出しは、このオブジェクトに対するequalsによる比較で使われた情報が変更されていなければ、一貫してtrueを返すか、一貫してfalseを返します。
5. null以外の参照値xについて、x.equals(null)はfalseを返します。

例えば、身長と体重が全く同じ Person クラスは同一とみなす場合は下記のようにオーバーライドする必要があります．

```
    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (!(obj instanceof Person)) {
            return false;
        }
        Person another = (Person) obj;
        return this.height == another.height
            && this.weight == another.weight;
    }
```

```
    public static void testPersonEquals() {
        Person x = new Person(170, 58);
        Person y = new Person(170, 58);
        Person z = new Person(170, 58);

        Person _ = new Person(171, 59);
        // 1. reflexive
        System.out.println("x eq x: " + x.equals(x)); // true
        // 2. symmetric
        System.out.println("x eq y: " + x.equals(y)); // true
        System.out.println("y eq x: " + y.equals(x)); // true
        System.out.println("_ eq x: " + _.equals(x)); // false
        // 3. transitive
        System.out.println("x eq y: " + x.equals(y)); // true
        System.out.println("y eq z: " + y.equals(z)); // true
        System.out.println("x eq z: " + x.equals(z)); // true
        // 4. consistent
        System.out.println("x eq y: " + x.equals(y)); // true
        System.out.println("x eq y: " + x.equals(y)); // true
        // 5. null
        System.out.println("x eq null: " + x.equals(null)); // false
    }
```

※注意：equals メソッドをオーバーライドする場合は、次に示す hashCode メソッドも適切にオーバーライドする必要があります．

hashCode() メソッド

```
    public native int hashCode();
```

hashCode() は、該当インスタンスのハッシュ値を取得します．Object クラスの hashCode() メソッドはnative(JNI)実装のため、具体的な実装は直接参照できません．
equals() メソッド同様、ハッシュコードをオーバーライドする場合にも守るべきルールが存在します．下記のように javadoc に記載があります．

1. Javaアプリケーションの実行中に同じオブジェクトに対して複数回呼び出された場合は常に、このオブジェクトに対するequalsの比較で使用される情報が変更されていなければ、hashCodeメソッドは常に同じ整数を返す必要があります。ただし、この整数は同じアプリケーションの実行ごとに同じである必要はありません。
2. equals(Object)メソッドに従って2つのオブジェクトが等しい場合は、2つの各オブジェクトに対するhashCodeメソッドの呼出しによって同じ整数の結果が生成される必要があります。
3. equals(java.lang.Object)メソッドに従って2つのオブジェクトが等しくない場合は、2つの各オブジェクトに対するhashCodeメソッドの呼出しによって異なる整数の結果が生成される必要はありません。ただし、プログラマは、等しくないオブジェクトに対して異なる整数の結果を生成すると、ハッシュ表のパフォーマンスが向上する場合があることに気付くはずです。

Person クラスに対して実装した場合の例です.
利用している Java のバージョン次第では、java.util.Objects#hash(Object... args) や java.util.Arrays#hashCode(Object[] args) が利用できます.

```
    @Override
    public int hashCode() {
        int result = 1;
        final int X = 31;
        result = result * X + this.height;
        result = result * X + this.weight;
        return result;
//      return java.util.Objects.hash(this.height, this.weight); // from 1.7
//      return java.util.Arrays.hashCode(
//          new Object[] { this.height, this.weight }); // from 1.5
    }
```

hashCode() は java.util.HashMap などの Key として利用されます. Map とは Key-Value が対となるデータ構造です. PHP でいうところの連想配列にあたります.
動作確認結果は下記です.

```
    public static void testPersonHashCode() {
        Map<Person, String> m = new HashMap<>();
        m.put(new Person(100, 20), "a");
        m.put(new Person(100, 21), "b");
        m.put(new Person(101, 20), "c");

        assert m.get(new Person(100, 20)).equals("a");
        assert m.get(new Person(100, 21)).equals("b");
        assert m.get(new Person(101, 20)).equals("c");
    }
```

上記例で利用している assert キーワードの使い道は、その時点でのデータ整合性のチェックポイントです. assert を実行時に有効化する場合は JVM 引数 -ea を付加して実行します.

```
$ javac -d bin -cp src/main/java:src/main/resources src/main/java/org/japs/basic/type/_class/Main.java

$ java -cp bin -ea org.japs.basic.type._class.Main
```

- java.lang.Object#hashCode()の性質: http://d.hatena.ne.jp/chiheisen/20120318/1332071962

toString() メソッド

```
    public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
    }
```

toString() メソッドは、当インスタンスの文字列表現を取得します.
デフォルト実装(java.lang.Object)では、上記のように「完全修飾名 + @ + ハッシュコード(16進数)」形式で返答されます.

デバッグの際などに、@以降のハッシュ値を確認して、インスタンスが同一かどうか確認するのに役に立つことがあります.

オブジェクトの文字列表現を変更したい場合は、当メソッドをオーバーライドします.

```
    @Override
    public String toString() {
          return new StringBuilder(Person.class.getName())
                      .append(" { height: ").append(this.height).append(",")
                      .append(" weight: ").append(this.weight)
                      .append(" } ").toString();
    }
```

System.out.println() の内部や「+」演算子による文字列連結では、この toString() が利用されます.

```
        Person p = new Person();
        String s = "Person: " + p;
        System.out.println(s);
        //-> Person: org.japs.basic.type._class.Person@1edf1c96
```

変数 s は「"Person: " + p.toString()」と同等の値となります.

自前のコンストラクタを定義していない場合はデフォルトコンストラクタが存在する

最初のコード例の Person クラスでは明示的にコンストラクタを定義していません．その場合、暗黙的に「デフォルトコンストラクタ」が作成されます．デフォルトコンストラクタは「public で引数の無い」コンストラクタです．

デフォルトコンストラクタの例：

```
    public Person() {}
```

そのため、明示的にコンストラクタを定義していなくても外部から Person クラスのインスタンスを生成することができます．但し、デフォルトコンストラクタに引数は渡せないため、Person クラスのようにフィールドが private 定義されている場合は、自前でコンストラクタを定義して値を設定する必要があります．

```
    public Person(int height, int weight) {
        this.height = height;
        this.weight = weight;
    }
```

アクセス修飾子は public を指定するか、もしくは何も指定しないことができる

トップクラスに対しては private や protected を指定することはできません．
public か、もしくはパッケージプライベート(アクセス修飾子に何も指定しない)が可能です．

```
public final class Person {...
```

※後述のネストクラスについては、private や protected も指定可能です．

ハンズ・オン

RFC821 を参考にメールアドレスを表現するクラスを作成します．クラス作成の練習のため、正確な仕様には沿っていないためご注意ください．

メールアドレス情報を表す MailAddress クラスの作成

```
package org.japs.basic.type._class;

public class MailAddress { }
```

フィールド定義

下記のフィールドを定義してください．外部に公開しない様、private で定義してください．

- localPart …… String 型
- domaon …… String 型

コンストラクタ定義

フィールドと同等の型を受け取るコンストラクタを実装してください．

```
public MailAddress(String localPart, String domain)
```

toString() メソッド定義

各フィールドを '@' で結合して返答するように実装してください．

```
public String toString()
```

equals() メソッド定義

各フィールドの同一性を以って実装してください．

```
public boolean equals(Object obj)
```

hashCode() メソッド定義

toString() の結果の hashCode() を返すように実装してください．

```
public int hashCode()
```

ローカルパートの妥当性チェック

ローカルパート(localPart)の妥当性をチェックするメソッド isValidLocalPart() を実装してください.

```
    /**
     * <special> ::= "<" | ">" | "(" | ")" | "[" | "]" | "\" | "."
     * | "," | ";" | ":" | "@"  """ | the control
     * characters (ASCII codes 0 through 31 inclusive and 127)
     * @return
     */
    public boolean isValidLocalPart() { ... }
```

ドメイン部分の妥当性チェック

ドメイン部分の妥当性をチェックするメソッド isValidDomain() を実装してください.

```
    /**
     * <let-dig-hyp> ::= <a> | <d> | "-"
     * <a> ::= any one of the 52 alphabetic characters A through Z
     *  in upper case and a through z in lower case
     * <d> ::= any one of the ten digits 0 through 9
     * @return
     */
    public boolean isValidDomain() { ... }
```

MailAddress クラスの動作確認

下記の main() メソッドを MailAddress クラスに実装し、動作確認を行ってください.

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

        assert new MailAddress("hatti33", "gmail.com").toString()
            .equals("hatti33@gmail.com");

        assert new MailAddress("hatti33", "gmail.com")
            .equals(new MailAddress("hatti33", "gmail.com"));

        assert !new MailAddress("hatti33", "gmail.com")
            .equals(new MailAddress("hatti333", "gmail.com"));

        assert new MailAddress("a", "b").isValidLocalPart();
        assert new MailAddress("-", "b").isValidLocalPart();
        assert !new MailAddress("@", "b").isValidLocalPart();

        assert new MailAddress("a", "b").isValidDomain();
        assert new MailAddress("a", "-").isValidDomain();
        assert !new MailAddress("a", "@").isValidDomain();

        assert new MailAddress("a", "b").hashCode()
            == new MailAddress("a", "b").hashCode();
        assert new MailAddress("a", "b").hashCode()
            != new MailAddress("b", "a").hashCode();

        Map<MailAddress, String> map = new HashMap<>();
        map.put(new MailAddress("x", "a"), "mkakimi");
        map.put(new MailAddress("y", "b"), "xxxxxxx");
        assert map.get(new MailAddress("x", "a")).equals("mkakimi");
    }
```

```
$ javac -cp src/main/java src/main/java/org/japs/basic/type/_class/MailAddress.java
-d bin
$ java -cp bin -ea org.japs.basic.type._class.MailAddress
```

interfaceについて

abstract classについて

enumについて

annotationについて

ハンズ・オン

スレッドダンプ＆GCログ編

スレッド

スレッドとは

Java は標準でマルチスレッドプログラミングをサポートしています．スレッドとは、1プロセス(JVM)内で「並列処理」が可能な仕組みです．複数の処理を同時に行いたい場合に利用します．
例えば、Java による Web アプリケーション（Servlet）では、アプリケーション・サーバー（Tomcat や Jetty など)により、要求されたリクエスト毎にスレッドが作成され、同時にリクエスト処理が行われます．

最近では CPU のコア数が増えたことにより、マルチスレッドにすることで複数の処理が各コアに割り当てられ、高速に処理できるようになるため、プログラムのマルチスレッド化が進んでいます．
今までは大きな処理単位で並列化することが多かったのですが、最近のプログラム言語はお手軽に小さな処理でもすぐに並列化できるように考慮されています．

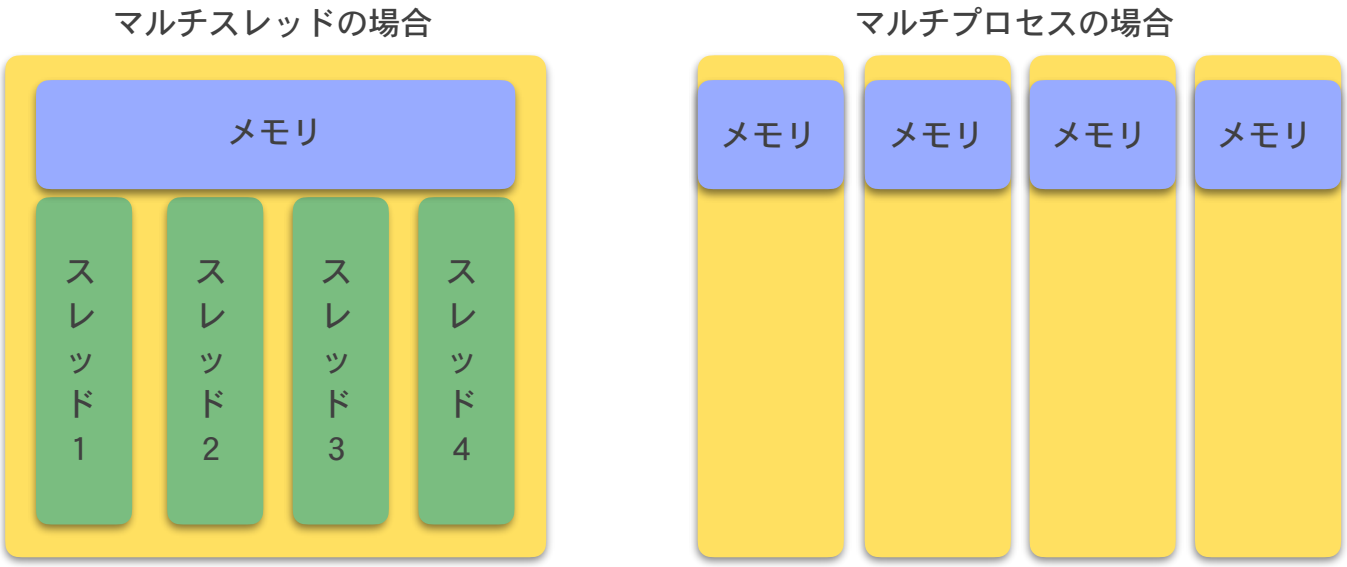
Java8 ではコレクション（java.util.List など）を簡単に並列処理できるようになりました．

```
package org.japs.java8.stream;

import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> list
            = Arrays.asList(new Integer[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 });
        list.parallelStream().forEach(System.out::print);
        // 7681359402 -> 出力順は都度変わる

    }
}
```

但し、単純に CPU にコア数が多いからといって parallelStream（並列処理)を利用すると早くなるかというとそうではありません．並列化するまでのスレッド生成などのオーバヘッドが通常より多くかかるため、並列化は処理対象件数がかなり多い場合などに有効になってくるかと思います．
利用する場合は、どちらが速いか測定しつつ導入するのが良いと思います．

マルチプロセスではだめ？

PHP なども複数の HTTP リクエストを同時に処理できますが、Servlet と異なるのは、PHP の場合はマルチスレッドではなく、マルチプロセスで処理されることです．ではマルチプロセスはマルチスレッドと何が違うのでしょうか？
マルチスレッドの場合、1つのプロセス(JVM)内で複数処理が分割されるため、JVM に割り当てられたメモリ領域(ヒープ)を各スレッドから同時に利用することができます．
しかし、マルチプロセスの場合は、それぞれのプロセス毎にメモリ領域が割り当てられるためお互いにその領域を共有することができません．



そのため、マルチスレッドの場合はそのメモリ領域を共通の作業場所として、スレッド間で並列処理をお互いに協調しながら動作させることが容易になります．マルチプロセスの場合は、OS が提供する機能(シェアドメモリ)やファイルI/Oなどによって制御する必要があります．

また、並列処理を行うために、新たなプロセスを立ち上げるよりも、同プロセス内でスレッドを立ち上げるほうがオーバヘッドが少ないため、並列化にかかるコストは少なくなります．

マルチスレッドのハマりどころ

これだけ見るとマルチスレッドを積極的に使いたくなるかもしれませんが、下記のような事項に注意する必要があります．

1. メモリ領域を共有するため、スレッド間のデータ競合が起きやすい
2. 特定のスレッドの異常に釣られて、プロセス(全てのスレッド)が終了してしまう場合がある
3. データ競合の整合性を保つためにプログラミングの難易度が高くなる

ハンズ・オン

1. 準備

```
$ git clone https://github.com/hatimiti/japs.git
$ cd japs
```

既に clone 済の場合は状態を最新化してください.

```
$ git pull origin master
```

2. Servlet の作成

後述するスレッドダンプ確認項では、マルチスレッドで動作するアプリケーションで確認を行うためため、ここでは Servlet プログラムを作成します. プログラムの実行環境となる AP サーバー(Servlet Container)については Tomcat を利用します.

事前に org.japs.web.JapsTomcat.JapsTomcat クラスを用意していますので、JapsTomcat に対して Servlet を設定してください.
JapsTomcat クラスのコンストラクタ第一引数には、自身のアカウント名にスラッシュを付加したもの、第二引数には生成した HttpServlet インスタンスを指定してください.

※ HttpSevlet#service(HttpServletRequest, HttpServletResponse) をオーバーライドすることで HTTP メソッド(GETやPOST)に関わらず動作します.

```
package org.japs.web.<アカウント>

...
public class <アカウント(1文字目大文字)>Servlet {
    public static void main(String[] args) throws Exception {
        new JapsTomcat("/<アカウント>", new HttpServlet() {
            // ここに Servlet を実装
        }).start();
    }
```

3. コンパイル

ハンズオン用プログラムでは、組込み Tomcat を利用しているため、ライブラリをダウンロードし、クラスパスに含める必要があります．また注意点としては、org.japs.web.JapsTomcat.JapsTomcat も利用しているため、あわせてコンパイル、クラスパスへの指定が必要となります．

```
$ mkdir lib
$ cd lib
$ curl -O http://central.maven.org/maven2/org/apache/tomcat/embed/tomcat-embed-core/8.0.20/tomcat-embed-core-8.0.20.jar
$ curl -O http://central.maven.org/maven2/org/apache/tomcat/embed/tomcat-embed-logging-juli/8.0.20/tomcat-embed-logging-juli-8.0.20.jar

$ cd ../
$ pwd
... /japs
$ javac -d bin -cp 'lib/*' src/main/java/org/japs/web/JapsTomcat.java
$ javac -d bin -cp bin:'lib/*' src/main/java/org/japs/web/hatimiti/HatimitiServlet.java
```

4. 実行確認

作成した Servlet が正常に動作しているか確認してください．
最後に表示されるポート番号で Tomcat が LISTEN しているため、そのアドレスに対して「http://localhost:< LISTEN ポート>/app/<アカウント>

```
$ java -cp bin:'lib/*' org.japs.web.hatimiti.HatimitiServlet
6 20, 2016 12:30:27 午前 org.apache.coyote.AbstractProtocol init
情報: Initializing ProtocolHandler ["http-nio-64045"]
6 20, 2016 12:30:27 午前 org.apache.tomcat.util.net.NioSelectorPool getSharedSelector
情報: Using a shared selector for servlet write/read
6 20, 2016 12:30:27 午前 org.apache.catalina.core.StandardService startInternal
情報: Starting service Tomcat
6 20, 2016 12:30:27 午前 org.apache.catalina.core.StandardEngine startInternal
情報: Starting Servlet Engine: Apache Tomcat/8.0.20
6 20, 2016 12:30:27 午前 org.apache.coyote.AbstractProtocol start
情報: Starting ProtocolHandler ["http-nio-64045"]
port: 64045 で LISTEN します。
```

```
$ curl http://localhost:64045/app/hatimiti
Hello, World
```

5. 繰り返し実行

```
$ watch -n 0.5 curl -s http://localhost:64045/app/hatimiti
```

6. 共有データを利用した Servlet の作成

次の点を満たすように実装してください.
クラスは org.japs.web.hatimiti.<アカウント>ShareDataServlet のように作成してください.

1. Servlet のインスタンスに private Integer フィールドを定義する.
2. そのフィールドをインクリメント操作する処理を Servlet 内で実行する.

インクリメント処理はアトミックな操作ではないため、スレッド間の共有データに対して操作を行うと、正常にインクリメントできない場合があります.

また、Servet はシングルインスタンスで動作するため、定義されたフィールドは、各リクエスト間で共有されることになります.

7. synchronized 化

インクリメント処理の部分を private メソッド increment に切り出し、synchronized キーワードによりアトミックに操作するように改修しましょう.

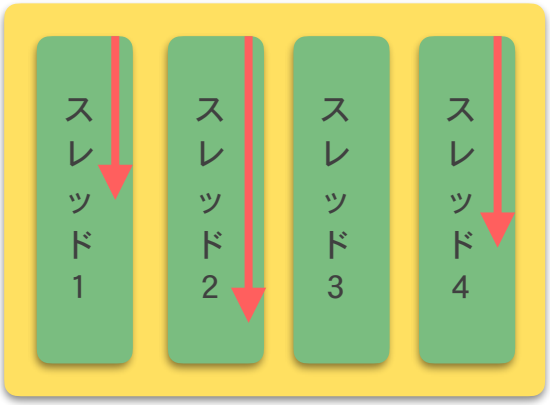
参考 URL

- TECHSCORE - スレッド: http://www.techscore.com/tech/Java/JavaSE/Thread/1/
- Qiita - スレッドとマルチプロセスの比較: http://qiita.com/shotaTsuge/items/0ad41fcee63a00a52f68
- もろず blog - イケてるエンジニアになろうシリーズ ～メモリとプロセスとスレッド編～: http://moro-archive.hatenablog.com/entry/2014/09/11/013520

スレッドダンプ

スレッドダンプとは

スレッドダンプとは、「ある時点」における JVM 上で動作しているスレッドの情報です．スレッドダンプにはスレッドIDやスレッド名、その時点のスレッドの状態が出力されています．
下記の図では、赤色の下矢印が特定の処理がどの段階まで進んでいるかを長さで表していいます．



この時点でスレッドダンプを取得した場合、スレッド1、スレッド2、スレッド4それぞれがどのメソッドの何行目まで実行されているか、各スレッドがどのような状態(ブロック状態や実行中など)か、をテキストデータとして取得することができます．

スレッドダンプの例

下記は Servlet を繰り返し実行している際に習得したスレッドダンプの一部です．他のスレッドについても情報が出力されますが、下記はその内のアプリケーションスレッドの1つを切り出したものです．
見方については後述しますので、ここではこのようなテキスト情報がスレッドダンプということを認識しておいてください．

```
"http-nio-65277-exec-9" #26 daemon prio=5 os_prio=31 tid=0x00007fea9a82e000 nid=0x6c03 waiting on condition
[0x000070000227b000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at org.japs.web.hatimiti.HatimitiShareDataServlet$1.service(HatimitiShareDataServlet.java:30)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:725)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:291)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:206)
        at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:219)
        at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:106)
        at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:501)
        at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:142)
        at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:79)
        at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:88)
        at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:516)
        at org.apache.coyote.http11.AbstractHttp11Processor.process(AbstractHttp11Processor.java:1086)
        at org.apache.coyote.AbstractProtocol$AbstractConnectionHandler.process(AbstractProtocol.java:659)
        at org.apache.coyote.http11.Http11NioProtocol$Http11ConnectionHandler.process(Http11NioProtocol.java:223)
        at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1558)
        at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.run(NioEndpoint.java:1515)
        - locked <0x000000076cad0418> (a org.apache.tomcat.util.net.NioChannel)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
        at java.lang.Thread.run(Thread.java:745)

   Locked ownable synchronizers:
        - <0x000000076c840060> (a java.util.concurrent.ThreadPoolExecutor$Worker)
```

なぜスレッドダンプを解析するか

マルチスレッドプログラムの場合によく問題になるのが、動作が不安定だったり処理が重たくなった場合にデバッグがしづらく、どのスレッドのどの箇所で問題が発生しているのかが分かりづらいことです.

そこで、そのような状態の時点のスレッドダンプを取得することで、どのスレッドが何を処理しているかという情報を取得し、負荷がかかっている箇所などの特定が可能となります.

スレッドダンプの取得方法

jps / jstack を利用する方法

スレッドダンプを取得する方法にはいくつかありますが、ここでは Java で用意されている標準的な操作で取得する方法を紹介します.
まず JDK に付属している jps コマンドを利用して、Java プロセスのプロセスIDを確認します.

```
$ jps
10382 HatimitiShareDataServlet
```

次に、同じく JDK 付属の jstack コマンドを利用してスタックトレースを取得します. 標準出力に出力されるため、リダイレクトなどを利用してログに書き出すとよいでしょう.
「-l」オプションで詳細情報が出力されるため付加しています.

```
$ jstack -l 10382 > japstack.log
```

jsack のヘルプ

```
$ jstack -h
Usage:
    jstack [-l] <pid>
        (to connect to running process)
    jstack -F [-m] [-l] <pid>
        (to connect to a hung process)
    jstack [-m] [-l] <executable> <core>
        (to connect to a core file)
    jstack [-m] [-l] [server_id@]<remote server IP or hostname>
        (to connect to a remote debug server)

Options:
    -F  to force a thread dump. Use when jstack <pid> does not respond (process is
hung)
    -m  to print both java and native frames (mixed mode)
    -l  long listing. Prints additional information about locks
    -h or -help to print this help message
```

標準コマンドを利用する方法

```
# java プロセスの標準出力をログへリダイレクト
java *** > redirect.log 2>&1
```

< linux の場合>

```
# PID を確認
ps aux | grep java

# 該当PIDのスレッドダンプをリダイレクトログへ出力
kill -3 (ここにPID)
```

< Windows の場合>

```
Ctrl+Breakキー
※Breakキーがないキーボードも存在する
```

スレッドダンプの見方

```
"http-nio-65277-exec-9" #26 daemon prio=5 os_prio=31 tid=0x00007fea9a82e000 nid=0x6c03 waiting on condition
[0x000070000227b000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at org.japs.web.hatimiti.HatimitiShareDataServlet$1.service(HatimitiShareDataServlet.java:30)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:725)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:291)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:206)
        at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:219)
        at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:106)
        at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:501)
        at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:142)
        at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:79)
        at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:88)
        at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:516)
        at org.apache.coyote.http11.AbstractHttp11Processor.process(AbstractHttp11Processor.java:1086)
        at org.apache.coyote.AbstractProtocol$AbstractConnectionHandler.process(AbstractProtocol.java:659)
        at org.apache.coyote.http11.Http11NioProtocol$Http11ConnectionHandler.process(Http11NioProtocol.java:223)
        at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1558)
        at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.run(NioEndpoint.java:1515)
        - locked <0x000000076cad0418> (a org.apache.tomcat.util.net.NioChannel)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
        at java.lang.Thread.run(Thread.java:745)

   Locked ownable synchronizers:
        - <0x000000076c840060> (a java.util.concurrent.ThreadPoolExecutor$Worker)
```

表 1 スレッドダンプの主な情報

名称	例示値	備考
スレッド名	http-nio-65277-exec-9	
スレッドID (tid)	0x00007fea9a82e000	
スレッドの状態	java.lang.Thread.State: TIMED_WAITING (sleeping)	
処理位置(現在)	at java.lang.Thread.sleep(Native Method)	スタックトレースの最終行
スタックトレース	処理位置以降の at …	処理の流れを確認可能
処理位置(開始)	at java.lang.Thread.run(Thread.java:745)	スタックトレースの一行目
ロック情報	- locked <0x000000076cad0418> (a org.apache.tomcat.util.net.NioChannel)	特定のオブジェクトをロック(synchronized)している情報

スレッドダンプの取得タイミング

スレッドダンプは以下の様なタイミングで取得／解析すると良いでしょう.

- システムがフリーズした時や、システム負荷が高い時
  正常なスレッドの状態は都度変化していきますが、問題となっているスレッド(フリーズや時間のかかっている処理)の場合は、スレッドの状態が変わらないことがあります.
  そのため、定期的(1秒毎など)にスレッドダンプを複数回取得し、正常にスレッドの状態が遷移しているかどうか判断します.

- 新しいライブラリや機能のバージョンアップをした時
  導入したライブラリや機能追加した場合に、それらの処理が負荷を高める原因になり得ます. スレッドダンプを取得し、正常に動作していることを確認しましょう.

- 負荷テスト実施時
  問題が発生する処理は、平常時のアクセス時は問題無いが、ピークアクセス時に処理し切れずに問題になることが多いです. そのような処理を発見するために、負荷テスト実施中にスレッドダンプを取得します.

スレッドダンプの解析手順

テキストデータのため、エディタなどでも参照することはできますが、便利な解析ツールが公開されていますので利用したほうが効率良く解析できて良いと思います.
ここでは、スレッドダンプの解析で有名なツールの「samurai」を利用したいと思います.

1. 対象プログラムを動作させます.
   ※ここでは watch コマンドにより定期実行しておきます. また、スレッドを競合させる場合は、下記コマンドを同時に複数のコンソールから実行しておきます.

```
$ watch -n 0.1 curl -s http://localhost:59889/app/hatimiti
```

2. jps コマンドなどを利用して、対象の Java プログラムのプロセスIDを確認します.

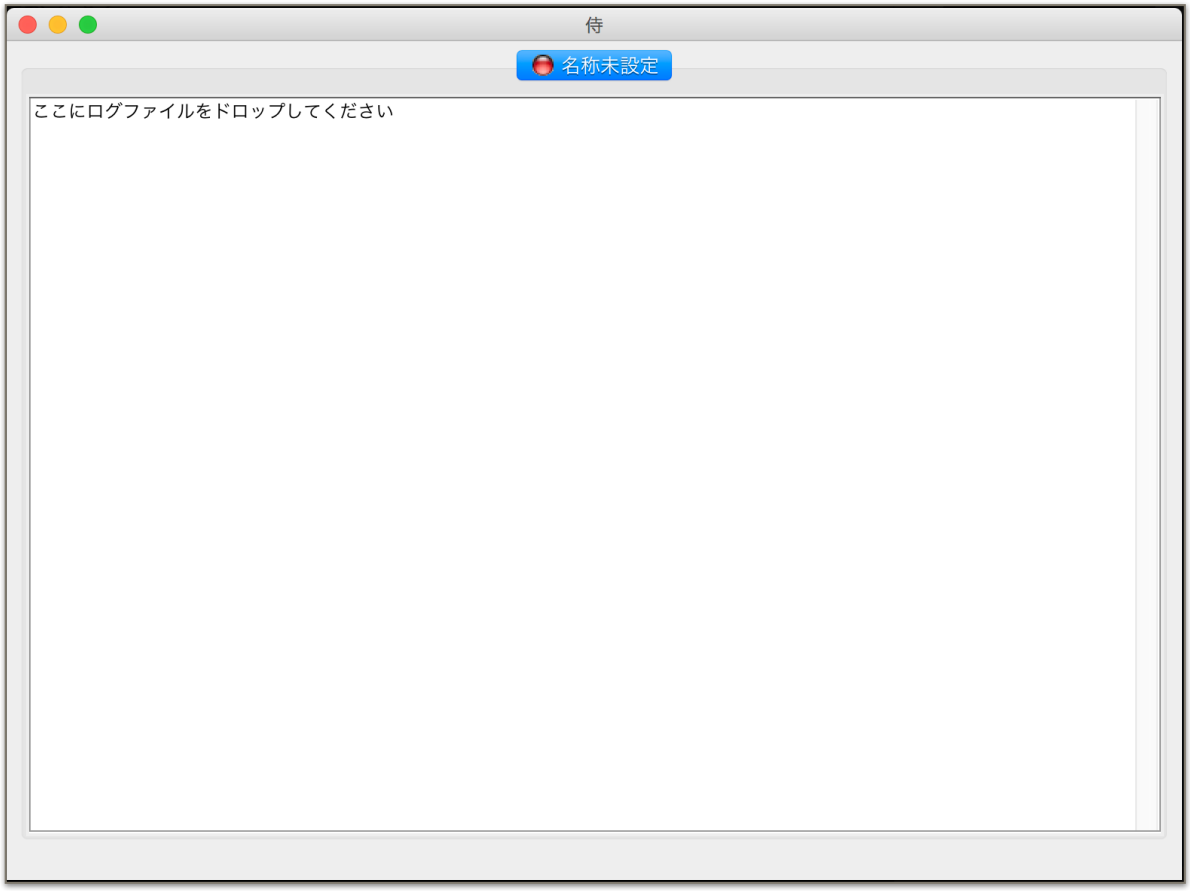
```
$ jps
2146 HatimitiServlet
2854 Jps
622
```

3. jstack コマンドを利用して、スタックトレースを取得します.
   ※ここでは 1 秒毎に jstack を実行し、 5 回分のスタックトレースを取得します.

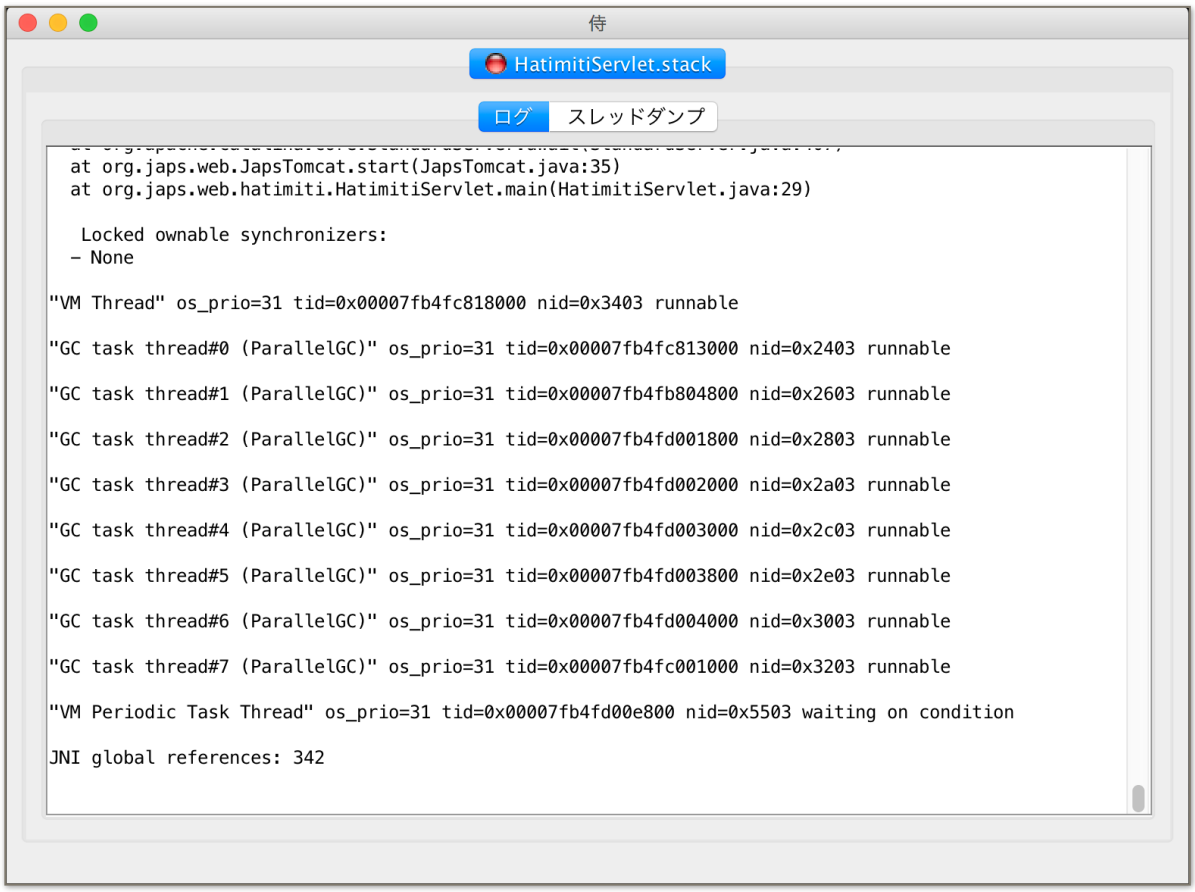
```
$ jstack -l 2146 >> ~/HatimitiServlet.stack
$ jstack -l 2146 >> ~/HatimitiServlet.stack
$ jstack -l 2146 >> ~/HatimitiServlet.stack
$ jstack -l 2146 >> ~/HatimitiServlet.stack
$ jstack -l 2146 >> ~/HatimitiServlet.stack
$
```

4. samurai を起動します.

```
$ java -jar /usr/local/samurai.jar &
```

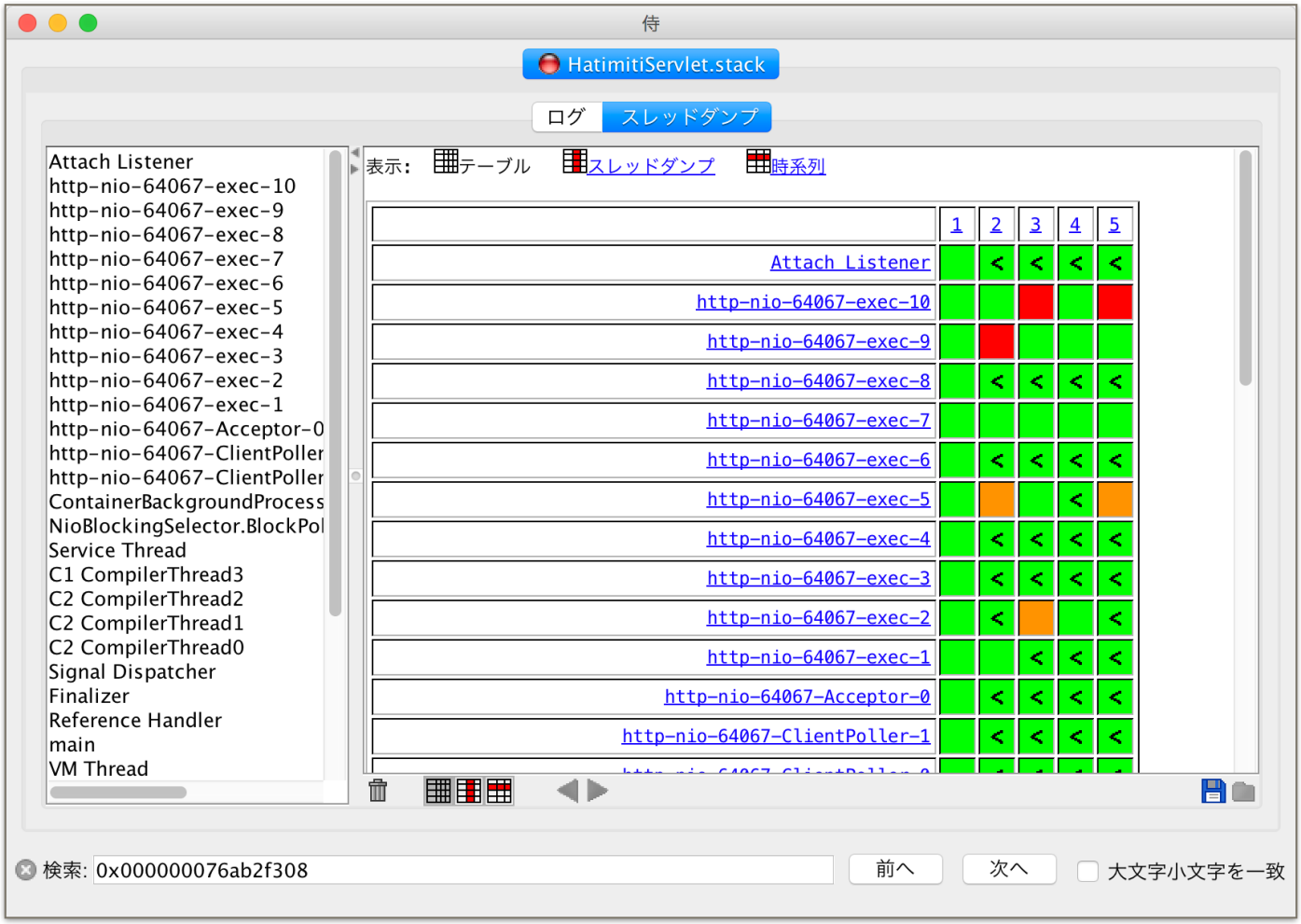


5. 取得したスタックトレースを samurai へドラッグ＆ドロップします.



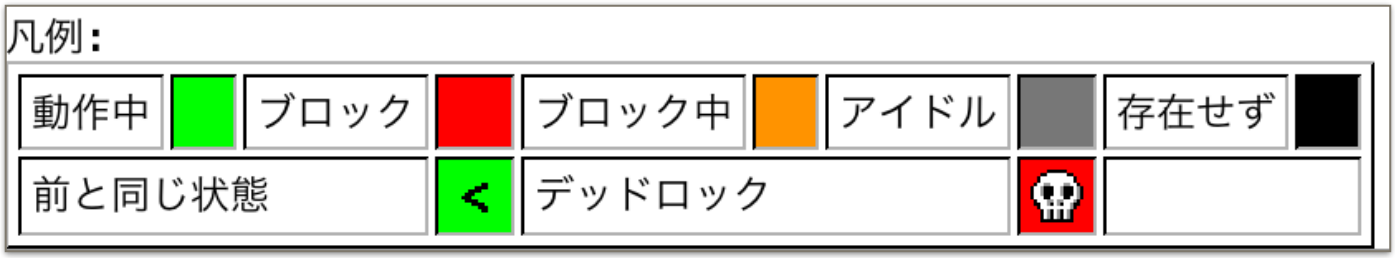
6. 「スレッドダンプ」タブを確認します
   左側レーンと右側レーンの最左列に表示されているのがスレッドの一覧です． Tomcat 上で単純な Servlet を動作させただけですが、こんなにも多くのスレッドが動作しているということになります．

   注意しないといけないのは、ここに表示されているスレッドは、アプリケーションスレッドだけでなく、JVM が利用するスレッドや Tomcat が利用するスレッドなども含まれていることです．普段からスレッドダンプを取得し、どれがアプリケーションスレッドかを把握しておきましょう．



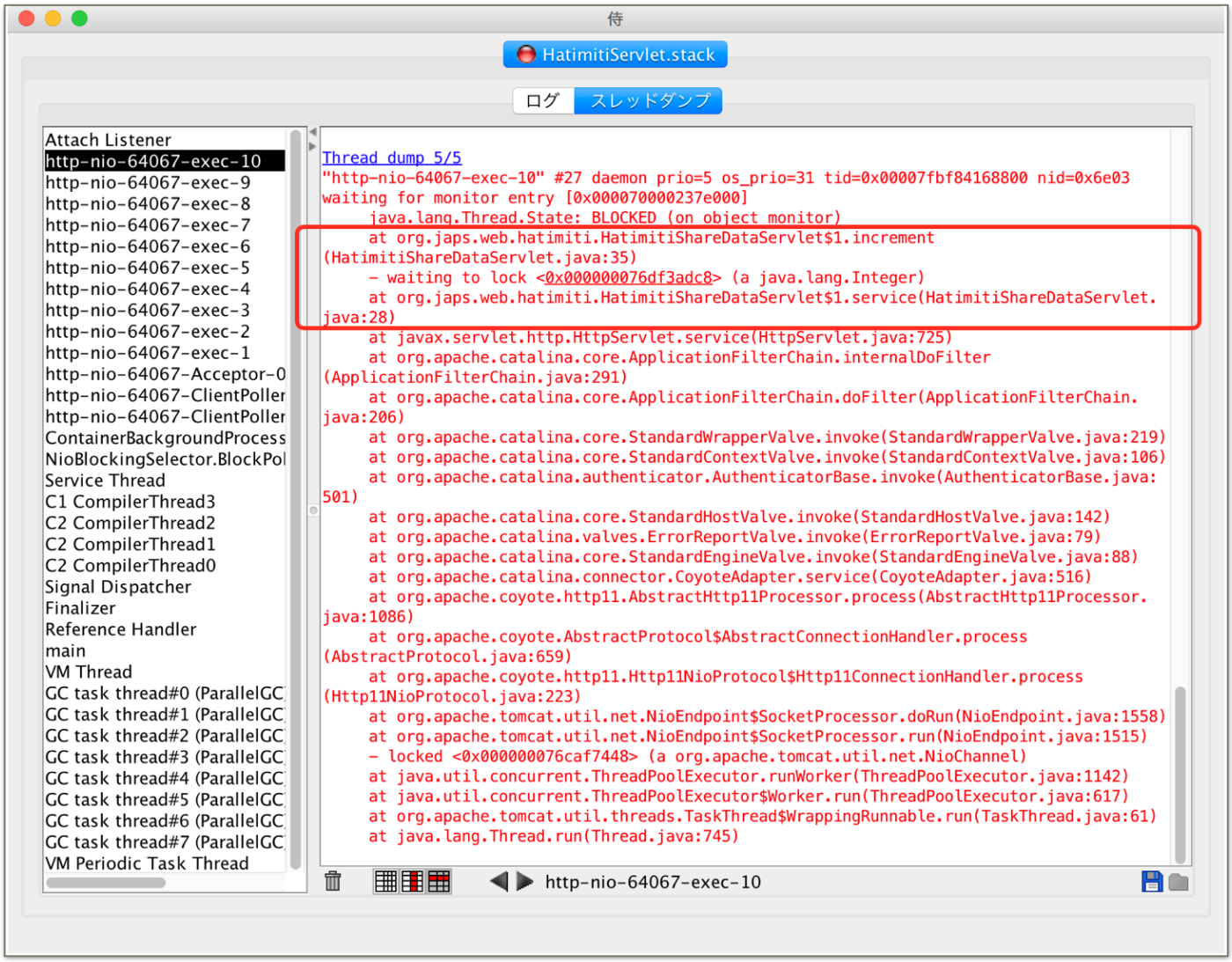
   今回は 1 秒毎に jstack を実行し、5 回分のスレッドダンプを取得したため、右側レーンの横軸には 1 ～ 5 までの表が表示されています．そのため、1 ～ 5 はそれぞれ、1 秒時点のスレッドの状態、2 秒時点のスレッドの状態と続き、5秒間のスレッドの状態の遷移を把握することができます．

7. アプリケーションスレッドの状態を確認する．
   下記のような凡例が samurai に表示されます．

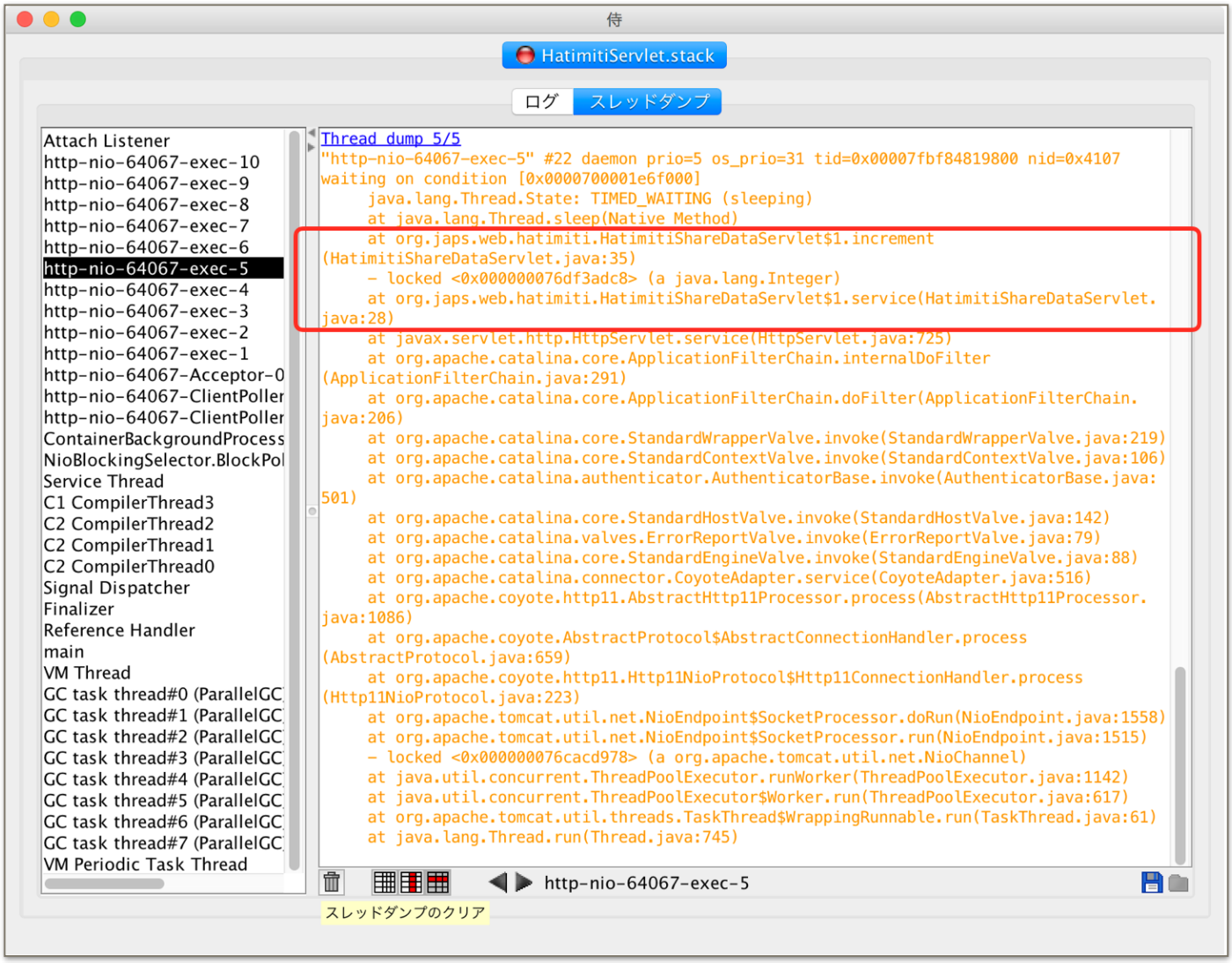


この中で、特に気にしないといけないのが「ブロック(赤)」の状態です．ブロックというのは、synchronized キーワードなどにより、他スレッドがロックしているオブジェクトの解放を待っている状態を含みます．このような状態が多く発生している(赤色の出現が多い)場合や、赤色で「前と同じ状態(<)」となっている場合は、待ち状態が多いためパフォーマンスの劣化に繋がります．

8. 赤色で表示されている枠をクリックします．
「- waiting to lock <0x...>」と表示されている箇所で、対象のオブジェクトの解放待ちをしていることになります．当サンプルでは Servlet のフィールドとして java.lang.Integer を競合させているため、オブジェクトID が 0x000000076df3abc8 の Integer オブジェクトの解放を待っているということになります．



9. ロック状態のオブジェクトIDをクリックします.
   こちらのスレッドが実際に Integer オブジェクトをロックしている側になります. ここの処理に問題が無いか見なおしてみましょう.



ハンズ・オン

1. samurai ツールのダウンロード

http://samuraism.jp/samurai/ja/index.html

2. スタックトレースの取得

3. samurai ツールでスタックトレースの確認

4. 処理を変更してスタックトレースの変更を確認

ガベージコレクション (GC)

ガベージコレクションとは

ガベージコレクション(以下 GC)とは、プログラム内で生成されたオブジェクトの内、どこからも参照さずに不要になったメモリ上のゴミ(Gabage)を掃除する処理です．Java では new キーワードを利用することで、ヒープ(後述)上にインスタンスが作成されますが、GC が実行されないと必要の無くなったインスタンスがヒープ上に残り続けて、メモリが足りなくなってしまいます．

C/C++ 言語などの場合は、GC の仕組みは用意されておらず、「delete」キーワードを利用して、自身でメモリ解放を意識しなければいけません．これらの言語でメモリ解放忘れが多く発生していたことも、Java で GC が理由の 1 つと考えられます．

GCの対象

GC の対象となる「不要なオブジェクト」とは下記のように、どこからも参照されなくなったオブジェクトです．

```
    Object obj = new Object();
    System.out.println(obj);
    obj = null;
```

もしくはローカル変数であれば、定義スコープ「{」から「}」を抜けた場合にも GC の対象となります．

ヒープとは

JVM のメモリ管理

java コマンドにより実行されたプログラムは、JVM プロセス上で動作します．JVM は自身のメモリ領域を下記のように分けて管理しています．
この内、new キーワードで生成したオブジェクトを確保しておく領域が「ヒープ領域」です．

- スタック領域
- Javaヒープ領域
- 非ヒープ領域
- Cヒープ(ネイティブメモリ)領域

ヒープの構造

JVM 上のヒープは GC の都合上、下記の領域に分けて管理されています.

- young (new)
  - eden
  - survivor (from / to)
- old / tenured

java コマンドにより実行されたプログラムは、JVM プロセス上で動作します. JVM は自身のメモリ領域を下記のように分けて管理しています.

GC の動作

GC はアプリケーションスレッドとは別にGC専用のスレッドを作成して処理を行いますが、アプリケーション側で利用しているメモリ空間にアクセスするため、GC中は一時的にアプリケーションスレッドが停止することがあります. その停止している時間を Stop The World と呼びます.

インスタンスは生成されるとまず young 領域に確保されます. そのままプログラム処理が続き young 領域がいっぱいになると最初の GC が実行され young 領域中の不要なインスタンスが除去されます.
これを「Minor GC」(または「Scavenge GC」)と呼びます.
継続的に Minor GC が行われると、利用中のインスタンスは from / to を行き来し、最終的に old 領域へと格納されます.

さらに old 領域がいっぱいになると、old 領域に対する GC が実行されます. これを「Full GC」と呼びます.

なぜ GC を意識する必要があるか

自動でメモリを解放してくれるのであれば、なぜ GC を意識する必要があるのでしょうか？
それは、GC はあくまでも「不要になったオブジェクトの解放」を行う仕組みであって、メモリリーク自体を防ぐ仕組みではないからです.
そして、前述したように「不要になった」という判断は、参照が途切れたオブジェクトであるため、システム仕様上不要になっているはずなのに、参照を保持したまま処理を続けてしまうと GC の対象とならず、いつかヒープを使い果たしてしまいます.
そのため、GC のログを確認し、正常にヒープがやりくりされていることを確認する必要があります.

GC ログの取得方法

GC の状況を確認するための 1 つの方法としては、GC ログを出力することです.
GC ログの設定を行う場合は、java コマンドの引数として下記のオプションを指定してください.

表 1 GC ログ出力のための JVM 起動オプション

オプション	説明	備考
-Xloggc:<GCログパス>	GC ログ・ファイルの出力先ファイル名	
-XX:+PrintGCDateStamps	GC 発生時間を絶対時間で出力する	指定しない場合は、JVM 起動後からの相対時間となるため読みづらい
-XX:+PrintGCDetails	詳細情報の表示	
-XX:+UseGCLogFileRotation	GC ログのローテーションを行う	JDK7u2 以降
-XX:NumberOfGCLogFiles=<ローテート数>	ローテートで残す世代数	JDK7u2 以降
-XX:GCLogFileSize=<ローテートサイズ>	ローテートタイミングとするログ・ファイルサイズ	JDK7u2 以降 0 を指定するとファイルサイズでのローテートは行わい.

ヒープ関連の調整

GC はヒープ上の掃除処理を行うため GC にかかる処理時間はヒープサイズに比例します. ヒープサイズの指定や、その他の領域のサイズは JVM の起動オプションで指定可能です.
ヒープのデフォルトサイズは環境により異なりますが、必要なヒープ値が分かっている場合は明示的に指定するのがオススメです.

表 1 ヒープサイズ設定のための JVM 起動オプション

オプション	説明	指定例	備考
-Xms<値>	初期ヒープサイズ	-Xms1024m	
-Xmx<値>	最大ヒープサイズ	-Xms1024m	
-XX:NewRatio=<値>	new 領域と old 領域の割合	-XX:NewRatio=2	デフォルト値: 2 new : old = 1 : 2
-XX:NewSize=<値>	初期 new 領域サイズ	-XX:NewSize=512m	-Xmn<値>と同様
-XX:MaxNewSize=<値>	最大 new 領域サイズ	-XX:MaxNewSize=512m	
-XX:SurvivorRatio=<値>	survivor 領域に対する new 領域の割合	-XX:SurvivorRatio=8	デフォルト値: 8 eden : survivor = 8 : 1

ヒープの初期サイズと最大サイズに同値を指定することで、処理中にヒープの拡張を行う必要が無くなるため効率的です.

young 領域のサイズについては「ヒープ初期サイズ / (1 + NewRatio)」となります.

非ヒープ関連の調整

補足情報ですが、一般的に非ヒープ領域についても JVM オプションを利用して調整します. 非ヒープ領域は、クラスの定義情報などが格納されるため、大規模なシステムになるにつれてクラス情報のサイズが膨れてメモリエラーとなります.

表 1 非ヒープサイズ設定のための JVM 起動オプション (Java7)

オプション	説明	指定例	備考
-XX:PermSize=<値>	初期 Permanent 領域サイズ	-XX:PermSize=256m	Java 8 以降は無視
-XX:MaxPermSize=<値>	最大 Permanent 領域サイズ	-XX:MaxPermSize=256m	Java 8 以降は無視

Java 8 以降は Perm 領域の役割は、ネイティブメモリの一部である Metaspace に取って代わりましたので、JVM オプションを指定する場合は注意してください.

表 2 非ヒープサイズ設定のための JVM 起動オプション (Java8)

オプション	説明	指定例	備考
-XX:MetaspaceSize=<値>	初期 Metaspace 領域サイズ	-XX:MetaspaceSize=256m	
-XX:MaxMetaspaceSize=<値>	最大 Metaspace 領域サイズ	-XX:MaxMetaspaceSize=1g	

GCの解析手順

スレッドダンプ同様テキストデータですが、スレッドダンプ以上に直接解析するのは難しいため、可視化できる「GCLogViewer」を利用したいと思います.

1. GC ログを格納するディレクトリを作成します.

```
$ sudo mkdir -p /var/log/japs
$ sudo chown <ユーザー名>:<グループ名> /var/log/japs
```

2. GC 確認用サーブレットを作成し、コンパイルします.
   サーブレットは、サーブレットのフィールドに、アクセスしたユーザーのセッションIDを格納する java.util.List<String> 型のフィールドを定義します. この時、List の実装クラスは java.util.ArrayList ではなく java.util.Vector 型を利用すると簡易に排他可能です.（実際の業務プログラムでは必要に応じて使い分けてください.）

```
$ pwd
/Users/m-kakimi/Documents/workspace/japs

$ javac -d bin -cp 'bin:lib/*' src/main/java/org/japs/web/hatimiti/
HatimitiGCServlet.java
```

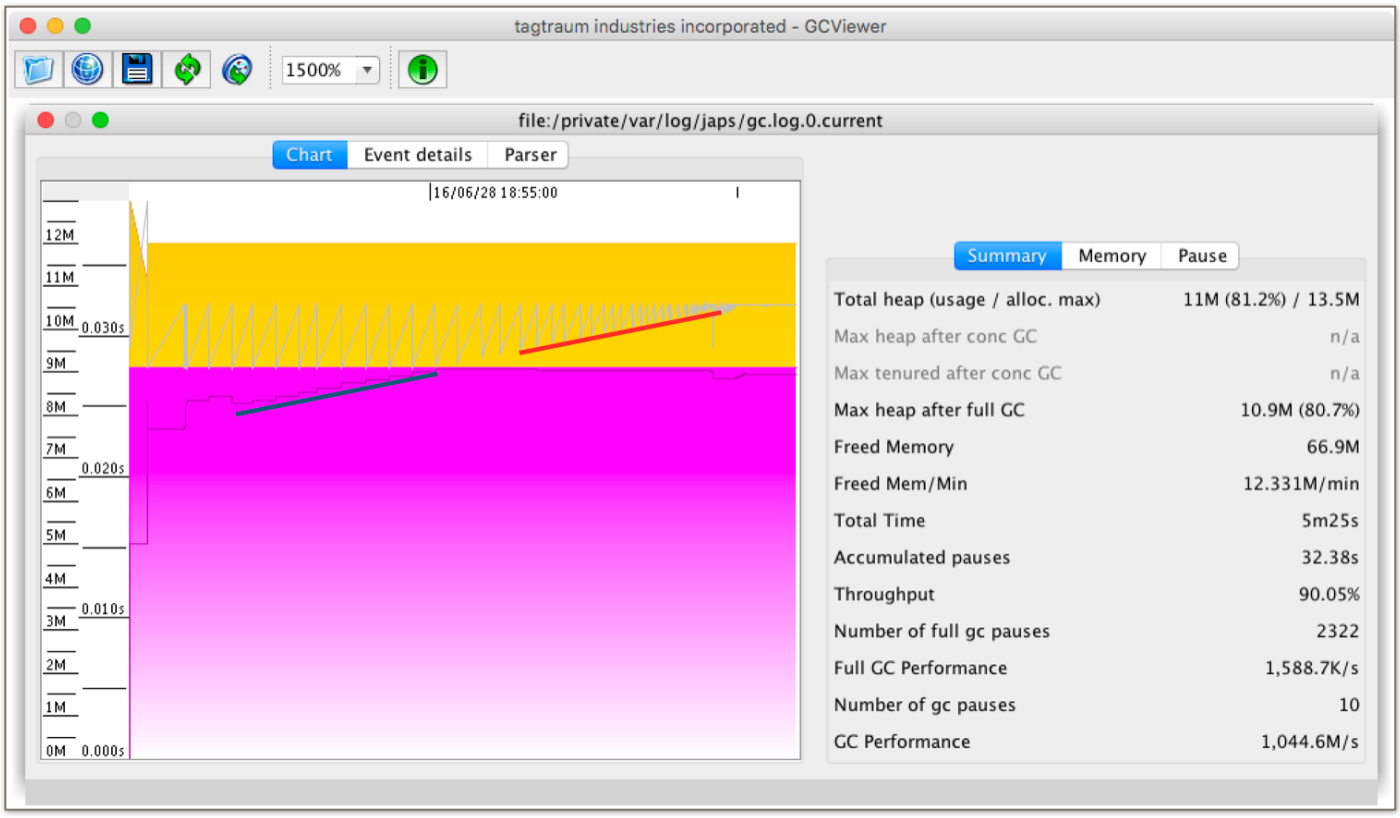
3. 対象プログラムに JVM オプションをして動作させます.
   GC を発生させやすくするために、ヒープサイズを少なめに確保します.（下記の -Xms12m -Xmx12m の指定）

```
$ java -cp 'bin:lib/*' -XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:/
var/log/japs/gc.log -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -
XX:GCLogFileSize=10M -Xms14m -Xmx14m  org.japs.web.hatimiti.HatimitiGCServlet
```

4. 起動させた Servlet に繰り返しリクエストし、しばらく放置します.

```
$ watch -n 0.1 curl http://localhost:58715/app/hatimiti
```

5. GC ログを GCLogViewer にドラッグ＆ドロップします.
   背景イエロー部分が young 領域です. 背景ピンク色が old 領域です.



このプログラムはメモリリークするように組んでいるため、young 領域と old 領域が徐々にあふれ、最終的には OutOfMemoryError となります．このように FullGC が発生しても、元の位置までヒープサイズが下がらない場合は、メモリリークとなりエラーとなります．

```
        new JapsTomcat("/hatimiti", new HttpServlet() {
            // ここに Servlet を実装
            private static final long serialVersionUID = 1L;

            private AtomicInteger i = new AtomicInteger(0);
            private List<String> strings = new Vector<>();

            @Override
            protected void service(
                    HttpServletRequest req,
                    HttpServletResponse resp)
                        throws ServletException, IOException {

                Writer w = resp.getWriter();
                strings.add(req.getSession().getId());
                w.write(strings.get(i.getAndIncrement()) + '\n');
                w.flush();
            }
        }).start();
```

6. 実際はこの時点でヒープダンプを取得し、どのオブジェクトによってメモリを圧迫しているかを確認しますが、ヒープダンプについては別記することとします．

ハンズ・オン

1. GCLogViewer ツールのダウンロード

https://github.com/chewiebug/GCViewer/wiki/Changelog

download / download mac version

2. GC ログの取得

前述の JVM オプションを参考に指定して、java プログラムを実行してください.
連続実行し、複数回 GC が実行されていること確認してください.

3. GCLogViewer ツールで GC ログの確認

出力されたログを GCLogViewer を利用して確認してください.

4. ヒープサイズを変更して GC の変化を確認

前述の JVM オプションを参考にヒープサイズの調整をし、GC の結果が変わることを確認してください.

参考 URL

- 最強のJVMチューニング・ツール: GCログを可視化するGCViewerとリモート接続でプロファイリング可能なVisualVM: http://x1.inkenkun.com/archives/780
- ITエンジニアとして生きる – JVMとGCのしくみ: http://d.hatena.ne.jp/ogin_s57/20120623/1340463194
- ITエンジニアとして生きる – JVMのチューニング: http://d.hatena.ne.jp/ogin_s57/20120709/1341836704
- 日記のような何か – Javaメモリ、GCチューニングとそれにまつわるトラブル対応手順まとめ: http://d.hatena.ne.jp/learn/20090218/p1
- 技術の犬小屋 – Javaのヒープ・メモリ管理の仕組みについて: http://promamo.com/?p=2828
- gihyo.jp – Javaはどのように動くのか〜図解でわかるJVMの仕組み: http://gihyo.jp/dev/serial/01/jvm-arc
-

GC の選択

GC のフェーズ

GC はメモリの解放処理を行いますが、その処理はいくつかのフェーズに分かれて処理しています.

1. 探索 …… 参照の途切れたインスタンスを探索する
2. 除去 …… 不要なインスタンスを除去する
3. 整理 …… メモリの断片化をコンパクションする

後述するように、GC にはいくつかの種類がありますが、それぞれの GC は上記フェーズ毎に動作が異なります.

GC の種類

GC のアルゴリズムには下記のようにいくつかの種類があり、システムの特性に合わせて自ら選択することができます.
各 GC アルゴリズムは基本的に、如何にして Full GC によるアプリケーション停止時間である Stop The World (STW) の時間を短くできるかを考えて用意されています.
Minor GC はどのアルゴリズムを選択しても、アプリケーションスレッドは停止されますが、その時間は短いため基本的に気にする必要はありません.

表 1 各GCのフェーズ毎の特性

名称	Minor	Full	探索	除去	整理	CPU	速度	備考
シリアル	STW	STW	STW/シングル	STW/シングル	STW/シングル	○	×	停止時間が多い
パラレル	STW	STW	STW/マルチ	STW/マルチ	STW/マルチ	△	△	
CMS	STW	-	- /マルチ	- /マルチ	- /シングル	×	○	・停止時間が短い ・90パーセンタイル
G1	STW	-	- /マルチ	- /マルチ	- / - *1	×	○	*1 頻度低

GC の選択

デフォルトでどの GC アルゴリズムが利用されるかは、プログラム実行環境毎に自動で選択されます.
明示的に GC アルゴリズムを選択する場合は、JVM オプションで指定します.

シリアル

単一 CPU、低メモリ環境で有効な GC です. Minor GC、Full GC 共に、アプリケーションスレッドを停止し、GC 用シングルスレッドで実行します.

【JVM 起動オプション】
-XX:+UseSerialGCflag

パラレル

複数 CPU 環境で有効な GC です．Minor GC、Full GC 共に、アプリケーションスレッドを停止し、GC 用マルチスレッドで実行します．

【JVM 起動オプション】
-XX:+UseParallelGC -XX:+UseParallelOldGC

CMS (Concurrent Mark Sweep)

複数 CPU 環境で、CPU 使用率に余裕がある場合に有効な GC です．Full GC をアプリケーションスレッドと平行でバックグラウンドスレッドで実行します(Concurrent)．old 領域のコンパクションを行う場合はアプリケーションスレッドを停止し、シングルスレッドで実行します．

【JVM 起動オプション】
-XX:+UseConcMarkSweepGC -XX:+UseParNewGC

G1 (Garbage 1st)

複数 CPU 環境、高メモリ環境(約 4 GB以上)で、CPU 使用率に余裕がある場合に有効な GC です．CMS 同様に動作(Concurrent)しますが、young / old 領域を複数のリージョンに分け、GC と同時に利用中のインスタンスを別のリージョンにコピーします．これがコンパクションの役割をすることで、G1 GC ではコンパクション処理の実行頻度が少なくなります．

※G1 は Java 7 時点では試験的な実装のため、利用する場合は Java 8 以降がよいかもしれません．

【JVM 起動オプション】
-XX:+UseG1GC

表 1 各GCのフェーズ毎の特性-1

名称	Minor	Full	探索	除去	整理	CPU	速度	備考

File入出力編

File 入出力

この章では下記の事項を学ぶことができます.

- 基本的なファイル入出力方法
- ストリームの概念

ストリーム

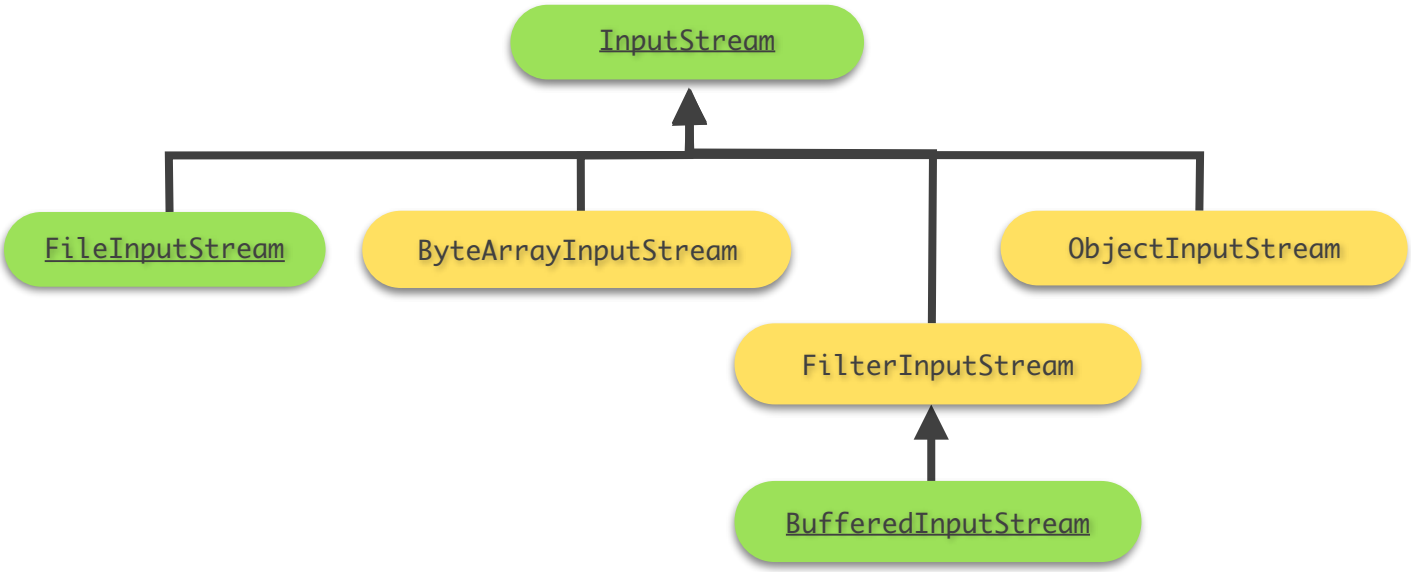
Java ではデータの入出力をストリームというデータの流れを表す概念として扱います. 入力元や出力先がディスクだけでなく、ネットワークなどであっても抽象的に入出力が扱うことができます.
大きく下記の種類に分けることができます.

- バイナリ入力
- バイナリ出力
- 文字入力
- 文字出力

```
- I/O Streams: http://msugai.fc2web.com/java/IO/index.html
```

バイナリ入力

バイナリデータの入力(読込)を行う場合は InputStream クラスを基点とするクラス群を利用します. 主な InputStream 系クラスは下記です. 下記のクラスはいずれも java.io パッケージ配下に属しています.



InputStream

```
public abstract class InputStream implements Closeable
```

InputStream の基点となるクラスです．abstract クラスのため、直接このクラスをインスタンス化することはせず、入力処理を抽象的に扱う型として利用します．

FileInputStream

```
public class FileInputStream extends InputStream
```

バイト単位で入力処理を行う際に利用可能です．
下記処理は、指定したテキストファイルから 3 バイトずつ読み込んで文字列として標準出力に表示しています．テキストファイルが UTF-8 の 3 バイト文字のみで構成されている前提の処理です．

● /japs/src/main/java/org/japs/basic/fileio/Main.java

```
class FileInputStreamTest {
    public void execute() throws IOException {
        try (InputStream is = new FileInputStream(
                new File("./src/main/resources/sample.txt"))) {
//      try (InputStream is = this.getClass()
//              .getResourceAsStream("/sample.txt")) {
//      try (InputStream is = this.getClass()
//              .getClassLoader().getResourceAsStream("sample.txt")) {

            byte[] bytes = new byte[3];
            while (is.read(bytes) != -1) {
                System.out.println(new String(bytes, Charset.forName("UTF-8")));
            }
        }
        /*
         * あ
         * い
         * う
         * え
         * お
         */
    }
}
```

● /japs/src/main/resources/sample.txt (UTF-8)

```
あいうえお
```

表 1 FileInputStream の主なメソッド

種別	定義	備考
コンストラクタ	FileInputStream(File file)	
1バイト単位の読込	int read()	
バッファ単位の読込	int read(byte[] b)	配列 b のサイズ(length)ずつ読込みます．ファイルの終わりに到達した場合は -1 を返します．
バッファ単位の読込	int read(byte[] b, int off, int len)	off 位置から len バイトのデータを b 配列へ読込みます．ファイルの終わりに到達した場合は -1 を返します．

BufferedInputStream

クラスパス上のリソースを読み込む場合

`classpath(-cp)` として `/japs/src/main/resources` が含まれている場合に、`/japs/src/main/resources/ClasspathMain` の `InputStream` を取得する場合は下記の方法で可能です．

```
InputStream is =
this.getClass().getClassLoader().getResourceAsStream("ClasspathMain"))
```

`this.getClass()` で、現在の処理(メソッド)が属するインスタンスを取得し、そのインスタンスが属しているクラスローダを `getClassLoader()` で取得します．そのクラスローダに定義されている `getResourceAsStream(String)` メソッドを利用することでクラスパス上のリソースを取得できます．

バイナリ出力

OutputStream

テキスト入力

Reader

テキスト出力

Writer

Java8編

1. Java8編

はじめに

Java のバージョン 8 は、2014-03-19 にリリースされました. Java 8 ではこれまでのバージョンアップの中でも一番と言ってよいほど多くの変更が加えられました.
「関数型」と呼ばれる考え方を導入し、Scala などの関数型オブジェクト指向言語の構文に近くなりました.

関数型インターフェース

関数型とは？

「関数型」言語の世界では、下記のようなワードがよく出現します.

- 高階関数
- 関数合成
- 副作用
- 参照透過性
- モナド

// TODO 詳細は学習中!!

Java 8 での「関数型」は、関数(メソッド)の単位で処理を受け渡しできるようになったことが大きいと考えられます.
第一級オブジェクト(FCO)を関数単位とみなして処理を扱えるようになり、ラムダ式(後述)や、メソッド参照(後述)によるコードの簡略化が図れます.

(※実際には内部的にクラス／インスタンス単位での処理が行われます)

関数型インターフェース

Java 8 から導入された「関数型インターフェース」とは、「実装すべき抽象メソッドを1つだけ保持したインターフェース」のことです.

下記は関数型インターフェースの定義例です.

```
interface SampleFuncIntf {
    int calc(int a, int b);
}
```

Java 7 までのインターフェースは、抽象メソッドの数は限定せず「特定の機能を有することを示す」ために、クラスに対して実装(implements)する用途が大半でした.

しかし、関数型インターフェースはクラスに対して実装(implements)することは基本的に無く、第一級オブジェクトとして、メソッドの引数として受け渡しすることが多くなります.

そのため、Java 8 以降の「インターフェース」は、今まで通りの「オブジェクト指向的に利用する」ものと、「関数型的に利用する」ものの 2 つに分けて定義／利用していくことになります.

※但し java.lang.Object で定義されているメソッド(toString()やequals()など)は抽象メソッドとして定義しても認められない

ラムダ式

java 8 では、関数型インターフェースを簡潔に記述するために「ラムダ式」が導入されました.
これにより今までの冗長だった記述を省くことができます.
「関数型インタフェースを引数にとるメソッド」は「実引数をラムダ式で記述可能」になります.

下記は、文字列型のリストをソートする処理ですが、Java 7 までは 6 行で記述していたものを、Java 8 では 1 行で記述できるようになりました. これは Collections#sort メソッドの第二引数である Comparator が関数型インターフェースであるため、ラムダ式を用いて記述することができるからです.

```
List<String> strings = Arrays.asList("a", "b", "c");

// Java 7
Collections.sort(strings, new Comparator<String>() {
      @Override
      public int compare(String o1, String o2) {
            return o1.compareTo(o2);
      }
});

// Java 8
Collections.sort(strings, (String o1, String o2) -> { return o1.compareTo(o2); });
```

ラムダ式の構文は下記の通りです.

```
(引数型1 引数名1, 引数型2 引数名2) -> {
    式1;
    式2;
    return 式3;
}
```

また、ラムダ式には省略記法があります.

- 引数が1つの場合は()を省略可能
- 型が推論できる場合は型指定を省略可能
- 式が1つの場合は `return` を省略可能

```
引数名1 -> 式1
```

- 引数が0個の場合は()は省略不可

```
() -> 式1
```

前述のコードサンプルのラムダ式を、省略して記述した場合は下記のようになります.
型指定や、{}、`return` を省略することができスッキリ記述できます.

```
Collections.sort(strings, (o1, o2) -> o1.compareTo(o2));
```

@FunctionalInterface アノテーション

インターフェース定義に対して `@java.lang.FunctionalInterface` アノテーションをつけることで関数型インターフェースとして正しいかどうかコンパイル時にチェック可能です.

```
@FunctionalInterface
interface SampleFuncIntf {
    int calc(int a, int b);
}
```

あくまでも形式チェック用のアノテーションのため、付加していなくても条件が揃っていれば関数型インターフェースとして利用可能です. ただ、関数型インターフェースとして明示できるため、基本的には付加するようにしましょう.

注意点は、インターフェースを定義した時点で抽象メソッドが 1 つだからと言って短絡的に `@FunctionalInterface` を付加してしまうことは避けましょう. そのインターフェースが本当に「関数型」として利用するものに対してのみ付加しましょう.

※ Java 7 までに標準で定義されていた、いくつかのインターフェースに対しても `@FunctionalInterface` アノテーションが付加されています.
`java.lang.Runnable`、`java.util.Comparator<T>` など

Java 8 で追加された標準関数型インターフェース

よく利用される関数パターンについては、標準で `java.util.function` パッケージ配下に関数型インターフェースが定義されています.

前述のように、自身で関数型インターフェースを定義することは可能ですが、可能な限りは以下に記述する標準で用意された関数型インターフェースを使いましょう.
標準関数型インターフェースで満たせない要件が発生した場合には自身で関数型インターフェースを作成する必要があります.

`Java 8` 以降の `Javadoc` にはこれらの標準関数型インターフェースが多数登場します.

代表的な標準関数型インターフェースについては用途を暗記してしまいしょう.

代表的な標準関数型インターフェース (引数 1 つ版)

表 1 Java 8 の標準関数型インターフェース

名称	戻り値	メソッド	説明	利用例	備考
Supplier	T	get()	引数無し、T型返り値	コンストラクタ new Object()、String.length()	
Consumer	void	accept(T t)	引数 1 つ、返り値無し	System.out.println("Hello ")	
Predicate	boolean	test(T t)	引数 1 つ、返り値 boolean	java.lang.Object# boolean equals(Object target)	
Function	R	apply(T t)	T型引数 1 つ、R型返り値	String# String[] split(String regex)	
UnaryOperator	T	apply(T t)	T型引数 1 つ、T型返り値		Function を継承しており、引数と戻り値の方が同一の Function と同等.

代表的な標準関数型インターフェース (引数 2 つ版)

引数が 2 つ受け取れるパターンの標準関数型インターフェースも用意されています. (Supplier 除く)

表 2 引数が 2 つの場合の標準関数型インターフェース

名称	戻り値	メソッド	説明	備考
BiConsumer	void	accept(T t, U u)	引数 2 つ、返り値無し	
BiPredicate	boolean	test(T t, U u)	引数 2 つ、返り値 boolean	
BiFunction	R	apply(T t, U u)	引数 2 つ、R型返り値有り	
BinaryOperator	T	apply(T t, T u)	引数 2 つ、T型返り値	UnaryOperator と同様の作りで、BiFunction を継承している.

プリミティブを扱う標準関数型インターフェース

さらに、プリミティブ型の Stream を扱うための Int***, Long***, Double***, Boolean*** が接頭辞の関数型インターフェースなどが用意されています.
(この辺りは、Java が参照型とプリミティブを分けて管理していることのデメリットかもしれません)

表 3 プリミティブ型を扱うための標準関数型インターフェース

名称	関数定義	ラムダ式例
int 型を扱うIF		
IntBinaryOperator	int applyAsInt(int left, int right);	(int a, int b) -> a + b
IntConsumer	void accept(int value);	(int a) -> System.out.println(a)
IntFunction	R apply(int value);	(int a) -> String.valueOf(a)
IntPredicate	boolean test(int value);	(int a) -> 0 < a
IntSupplier	int getAsInt();	() -> 10 * 10
IntToDoubleFunction	double applyAsDouble(int value);	(int a) -> a / 2.0
IntToLongFunction	long applyAsLong(int value);	(int a) -> a * a
IntUnaryOperator	int applyAsInt(int operand);	(int a) -> a + 1
long 型を扱うIF		
LongBinaryOperator	long applyAsLong(long left, long right);	※int 型の例と同様
LongConsumer	void accept(long value);	※int 型の例と同様
LongFunction	R apply(long value);	※int 型の例と同様
LongPredicate	boolean test(long value);	※int 型の例と同様
LongSupplier	long getAsLong();	※int 型の例と同様

名称	関数定義	ラムダ式例
LongToDoubleFunction	double applyAsDouble(long value);	※int 型の例と同様
LongToIntFunction	int applyAsInt(long value);	※int 型の例と同様
LongUnaryOperator	long applyAsLong(long operand);	※int 型の例と同様
double 型を扱うIF		
DoubleBinaryOperator	double applyAsDouble(double left, double right);	※int 型の例と同様
DoubleConsumer	void accept(double value);	※int 型の例と同様
DoubleFunction	R apply(double value);	※int 型の例と同様
DoublePredicate	boolean test(double value);	※int 型の例と同様
DoubleSupplier	double getAsDouble();	※int 型の例と同様
DoubleToIntFunction	int applyAsInt(double value)	※int 型の例と同様
DoubleToLongFunction	long applyAsLong(double value);	※int 型の例と同様
DoubleUnaryOperator	double applyAsDouble(double operand);	※int 型の例と同様
boolean 型を扱うIF		
BooleanSupplier	boolean getAsBoolean();	() -> Math.random() < 0.5
参照型 and プリミティブを扱うIF		
ObjIntConsumer	void accept(T t, int value);	(String a, int b) -> a + b
ObjLongConsumer	void accept(T t, long value);	※int 型の例と同様
ObjDoubleConsumer	void accept(T t, double value);	※int 型の例と同様
参照型 to プリミティブ型を扱うIF		
ToIntBiFunction	int applyAsInt(T t, U u);	(String a, String b) -> a.length + b.length
ToIntFunction	int applyAsInt(T value);	(String a) -> a.length
ToLongBiFunction	long applyAsLong(T t, U u);	※int 型の例と同様
ToLongFunction	long applyAsLong(T value);	※int 型の例と同様
ToDoubleBiFunction	double applyAsDouble(T t, U u);	※int 型の例と同様
ToDoubleFunction	double applyAsDouble(T value);	※int 型の例と同様

Stream API

Stream APIとは？

Stream API は、配列やコレクション(ListやMapなど)の集合情報を、パイプライン形式で処理できる API です．通常、これらの情報を扱う場合は iterate / while / for などのループ構文で処理して結果を得ますが、Stream を利用することでループ構文を使わずに結果を得ることができるようになります．そうすることで、手続き的なコードを宣言的なコードに置き換えることができ、可読性向上を図れます．また、関数型インタフェースで処理することを想定されているため、ラムダ式により簡潔に記述することが可能になります．

Stream API のコード例

下記の persons 変数は List<Person> 型で、Person 型は年齢(age)や給料(salary)などの情報を保持しています．この情報を基に「30 歳以上の給料の平均を取得」してみましょう．

Java 7 までであれば、このような処理は for 文、if文を用いて給料の合計を別変数に足しあわせておき、ループ外で対象人数で割る、というように手続き的に処理する必要がありました．

```
double result = 0.0;
int sum = 0;
int count = 0;
for (Person p : persons) {
    if (p.getAge() >= 30) {
        sum += p.getSalary();
        count++;
    }
}
if (count != 0) {
    result = ((double) sum) / count;
}
return result;
```

Java 8 では Stream API をこれをメソッドチェーンを用いて宣言的に結果を得ることができるようになりました．

```
return persons.stream()
    .filter(p -> p.getAge() >= 30)
    .mapToInt(p -> p.getSalary())
    .summaryStatistics().getAverage();
```

Stream は、SQL のように集合を扱う言語に頭を切り替えると理解しやすくなると思います.

```
SELECT
       AVG(p.getSalary())
  FROM persons p
 WHERE p.getAge() >= 30
```

※補足ですが、Java 7 までででもファイル入出力 API で「ストリーム」というワードが登場していましたが、その「ストリーム」と「Stream API」は関係無いため注意してください.

Stream の処理工程

Stream の処理は大きく「生成」→「中間操作」→「終端操作」の 3 つの工程に分けることができます. それぞれの工程で用意されているメソッドを用いて、集合を扱います.

生成

Stream を生成する工程です. インプットする情報によって、生成する方法が異なります.

表 1 Stream 生成

#	クラス	メソッド	静的	有限	備考
1	[参照型] java.util.stream.Stream [プリミティブ型] java.util.stream.IntStream java.util.stream.LongStream java.util.stream.DoubleStream * プリミティブ型 Stream の場合は右記メソッドの引数は対象のプリミティブ型で定義されています	of(T t)	○	○	
2		of(T... values)	○	○	
3		empty()	○	○	
4		iterate(final T seed, final UnaryOperator<T> f)	○		無限 Stream
5		generate(Supplier<T> s)	○		無限 Stream
6	java.util.stream.IntStream java.util.stream.LongStream * LongStream の場合は右記メソッドの引数は対象の long 型で定義されています	range(int startInclusive, int endExclusive)	○	○	
7		rangeClosed(int startInclusive, int endInclusive)	○	○	終了値含む
8	java.util.Collection (List, Map など)	stream()		○	
9		parallelStream()		○	並列 Stream
10	java.util.Arrays	stream(T[] array)	○	○	

```
Stream<String> s1  = Stream.of("Hello Stream");
Stream<String> s2  = Stream.of("Hello", "Stream");
Stream<String> s3  = Stream.empty();
IntStream      s4  = IntStream.iterate(1, n -> n + n); //1,2,4,8,16,...
DoubleStream   s5  = DoubleStream.generate(() -> Math.random()); // 乱数の無限Stream
IntStream      s6  = IntStream.range(1, 10); // 1〜9
IntStream      s7  = IntStream.rangeClosed(1, 10); // 1〜10
Stream<String> s8  = Arrays.asList("a", "b")/*List<String>*/.stream();
Stream<String> s9  = Arrays.asList("a", "b")/*List<String>*/.parallelStream();
Stream<String> s10 = Arrays.stream(new String[] { "Hello", "Stream"});
```

終端操作 (Terminal operation)

加工した情報を基に、結果を得る工程です．終端操作は 1 つの Stream に対して、1 回しか行うことができません．2 回以上終端操作を行った場合は例外が発生します．

表 3 java.util.stream.Stream<T> 終端操作 -基本操作-

#	メソッド	返答型	説明
1	forEach(Consumer<? super T> action)	void	各要素に対して action の処理を実施する．
2	forEachOrdered(Consumer<? super T> action)	void	各要素に対して action の処理を実施する． (処理順を保障する)
3	min(Comparator<? super T> comparator)	Optional<T>	渡された comparator で比較した結果の最小値を取得する．
4	max(Comparator<? super T> comparator)	Optional<T>	渡された comparator で比較した結果の最大値を取得する．
5	count()	long	要素数を取得する．
6	anyMatch(Predicate<? super T> predicate)	boolean	Predicate に適する要素が 1 つでも存在する場合は true を返す．
7	allMatch(Predicate<? super T> predicate)	boolean	Predicate に全ての要素が適する場合は true を返す．
8	noneMatch(Predicate<? super T> predicate)	boolean	Predicate に全ての要素が適さない場合は true を返す．
9	findFirst()	Optional<T>	最初の要素を取得する．
10	findAny()	Optional<T>	最初の要素を取得する． 但し、並列 Stream の場合は何れかの要素を取得する．
11	toArray()	Object[]	配列へ変換する．
12	toArray(IntFunction<A[]> generator)	A[]	配列へ変換する．(指定型) generator の引数には要素数が渡される．

```java
/*#1*/
Stream.of("a", "b", "c").forEach(e -> System.out.print(e)); //abc
System.out.println();
/*#2*/
Stream.of("b", "c", "a").parallel().forEachOrdered(e -> System.out.print(e)); //
bca
System.out.println();
/*#3*/
Optional<Integer> x = Stream.of(3, 1, 2).min((i, j) -> i.compareTo(j));
System.out.println(x); // Optional[1]
/*#4*/
Optional<Integer> y = Stream.of(3, 1, 2).max((i, j) -> i.compareTo(j));
System.out.println(y); // Optional[3]
/*#5*/
long z = Stream.of(0, 0, 0).count();
System.out.println(z); // 3
/*#6*/
boolean a = Stream.of(1, 2, 3).anyMatch(e -> e.equals(2));
System.out.println(a); // true
/*#7*/
boolean b = Stream.of(2, 2, 2).allMatch(e -> e.equals(2));
System.out.println(b); // true
/*#8*/
boolean c = Stream.of(2, 2, 2).noneMatch(e -> e.equals(3));
System.out.println(c); // true
/*#9*/
Optional<Integer> d = Stream.of(2, 1, 3).findFirst();
System.out.println(d); // Optional[2]
/*#10*/
Optional<Integer> e = Stream.of(2, 1, 3).findAny();
System.out.println(e); // Optional[2]
/*#11*/
Object[] f = Stream.of(2, 1, 3).toArray();
System.out.println(f); // [Ljava.lang.Object;@17f052a3
/*#12*/
Integer[] g = Stream.of(2, 1, 3).toArray(i -> new Integer[i]);
System.out.println(g); // [Ljava.lang.Integer;@685f4c2e
```

表 3 java.util.stream.Stream<T> 終端操作 -応用操作-

#	メソッド	返答型	説明
1	reduce(T identity, BinaryOperator<T> accumulator)	T	各要素に対して accumulator によるたたみ込み処理を行う. identity は初期値.
2	reduce(BinaryOperator<T> accumulator)	Optional<T>	各要素に対して accumulator によるたたみ込み処理を行う.

#	メソッド	返答型	説明
3	reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)	U	各要素に対して accumulator によるたたみ込み処理を行う. identity は初期値. combiner により、accumulator で算出した値を結合する.
4	collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner)	R	
5	collect(Collector<? super T, A, R> collector)	R	

中間操作 (Intermediate operation)

インプットされた情報に加工を施す工程です. 中間操作は行わない場合もあります.

表 2 java.util.stream.Stream<T> 中間操作

メソッド	返答型	説明
filter(Predicate<? super T> predicate)	Stream<T>	条件フィルタリング
map(Function<? super T, ? extends R> mapper)	Stream<R>	値変換(参照型→参照型)
mapToInt(ToIntFunction<? super T> mapper)	IntStream	値変換(参照型→int型)
mapToLong(ToLongFunction<? super T> mapper)	LongStream	値変換(参照型→long型)
mapToDouble(ToDoubleFunction<? super T> mapper)	DoubleStream	値変換(参照型→double型)
flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)	Stream<R>	値変換(参照型→Stream) 複数の Stream を 1 つの Stream へ平坦化する.
distinct()	Stream<T>	重複削除
sorted()	Stream<T>	ソーティング
peek(Consumer<? super T> action)	Stream<T>	デバッグ用の中間操作. Stream 処理中の値を System.out.println() などを利用して参照可能.
limit(long maxSize)	Stream<T>	指定要素数に限定する. 無限 Stream を有限化する際などに利用できる.
skip(long n)	Stream<T>	指定要素数分の処理をスキップする.

メソッド参照

メソッド参照は、

ハンズ・オン

1. 準備

```
$ git clone https://github.com/hatimiti/japs.git
$ cd japs
```

既に clone 済の場合は状態を最新化してください.

```
$ git pull origin master
```

2. コンパイルと実行

ファイル「japs/src/main/java/org/japs/java8/stream/HandsOn.java」をコンパイル

```
$ pwd
/Users/m-kakimi/Documents/workspace/japs
$ mkdir -p bin
$ javac -d bin -cp bin src/main/java/org/japs/java8/stream/Products.java
$ javac -d bin -cp bin src/main/java/org/japs/java8/stream/HandsOn.java
$ javac -d bin -cp bin src/main/java/org/japs/java8/stream/HandsOnAnswer.java
```

org.japs.java8.stream.HansOn クラスを実行

```
$ java -cp bin org.japs.java8.stream.HandsOn
```

3. Stream APIを使って書いてみましょう

ファイル「japs/src/main/java/org/japs/java8/stream/HandsOn.java」の
org.japs.java8.stream.HandsOn クラスのメソッドを完成させてください.

Stream 処理の対象とするデータは org.japs.java8.stream.Products#all() メソッドで取得できます.

表 1 Product 型の定義

フィールド型	フィールド名	用途	Getter	備考
java.lang.String	name	商品名	getName()	
int	value	商品価格	getValue()	

フィールド型	フィールド名	用途	Getter	備考
java.time.LocalDate	saleDay	発売日	getSaleDay()	
java.util.List<Company>	salesCompanies	販売会社(複数)	getSalesCompanies()	

表 2 Company 型の定義

フィールド型	フィールド名	用途	Getter	備考
java.lang.String	name	会社名	getName()	
java.lang.String	telNo	電話番号	getTelNo()	

※解答例は org.japs.java8.stream.HandsOnAnswer にあります.

参考 URL

- 社内Java8勉強会 ラムダ式とストリームAPI: http://www.slideshare.net/zoetrope/java8-lambdaandstream
- 試験対策とハンズオンでおぼえるラムダ式&Stream API: http://www.oracle.co.jp/jdt2016/pdf/HO-1.pdf

default メソッド

インターフェースに実装を持つメソッドを定義できるようになった

Java はバージョン間の下位互換性を保つことを重視しています．Java 7までは interface には抽象メソッドしか定義することができませんでしたが、Java 8では、Stream 処理(後述)を
実装をもつメソッドを定義するときには、キーワードdefaultをメソッドの宣言に付加する．

```
interface Movable {

	int walk();

	default int run() {
		return walk() * 2;
	}
}
```

抽象クラスとの違いは?

インスタンス変数が保持できるかどうか(インターフェースは保持できない)と、多重継承可能かどうか(インターフェースは多重実装可能)

staticメソッドも実装可能となった

java.lang.Objectクラスに定義されたメソッド(toString()など)のデフォルト実装は定義できない(コンパイルエラー)

→A default method cannot override a method from java.lang.Object

アクセス修飾子は public のみ(その他にするとコンパイルエラー)

→only public, abstract, default, static and strictfp are permitted

多重継承(ひし形継承)問題は、必ず子クラス側でのメソッドオーバーライドを強制することで回避している

親クラスがそのメソッドを既に実装済の場合は子クラス側でのオーバーライドは不要

Optional

Optional とは？

java.util.Optional とは、内部に値を 1 つだけ保持することができ、その値の有無を表現するために導入されました．NullPointerException(NPE) になる箇所を早期発見し対処可能になります．

※但し Optional 型自体は単なる参照型(クラス型)なので、Optional 型の変数自体に null を代入してしまうと意味が無いため注意しましょう．

Optional の利用場面は？

メソッドの作り手が、戻り値の型として Optional を利用します．
そのメソッドの利用者に Optional[empty](nullに相当) が返ることをコード上で明示可能となります．
今までは Javadoc に null が返ることを明記するしかなかったため、そもそも明記していなかったり javadoc を読まなかった場合にメソッド利用者が null の処理をせずにそのオブジェクトを次へ次へと渡して、その先で NPE が発生してしまいどこが例外の原因となっているか分かりづらかったのです．

メソッドの引数には利用しても、メソッドの利用者が null を渡さないことを強制できないため、あまり意味がありません．

Optional の種類と値

参照型用の Optional<T> とプリミティブ型用の Optional***(Int|Long|Double) が存在します．

主なファクトリメソッド

Optional<T> of(T v) …… Optionalを生成する。但し v が null の場合は例外発生(NullPointerException)
Optional<T> ofNullable(T v) …… Optionalを生成する。v が null の場合は Optional[empty]となる
Optional<T> empty() …… Optional[empty]を生成する。

主な値取得メソッド

T get() …… 保持している値を取得します．Optional[empty] の場合は例外が発生します．(java.util.NoSuchElementException: No value present)
T orElse(T v) …… 保持している値を取得する。Optional[empty]の場合は引数 v の値を返す。
T orElseGet(Supplier o) …… 保持している値を取得する。Optional[empty]の場合は引数 o のラムダ式結果を返す。

T orElseThrow(Supplier e) …… 保持している値を取得する。Optional[empty]の場合は引数 Supplier で生成した例外を throw します.

主な値操作メソッド

Optional<T> filter(Predicate predicate) …… 値のフィルタリング。フィルタ対象となった場合はOptional[empty]となる。
Optional<U> map(Function mapper) …… 値の変換T→U。nullを返した場合はOptional[empty]となる。
Optional<U> flatMap(Function mapper) …… mapと同じだが、入れ子のOptionalをフラットに扱える。
void ifPresent(Consumer consumer) …… 値が存在する場合のみ引数のラムダ式が実行される。

その他メソッド

boolean isPresent() …… 値が存在すれば true を返す。

Optional 実装例

```
public void xxx() {

    Optional<String> a = func("Hello");
    Optional<String> b = func("Hello, World");

    System.out.println(a); // Optional[!Hello!]
    System.out.println(b); // Optional.empty

    func("hello").orElseThrow(() -> new RuntimeException("Optional Empty"));
    func("hello").orElse("HELLO");

    Optional<Optional<String>> map1 = func("hello").map(s -> func(s + "W"));
    Optional<String> map2 = func("hello").flatMap(s -> func(s + "W"));

    System.out.println(map1); // Optional[Optional[!!hello!W!]]
    System.out.println(map2); // Optional[!!hello!W!]

    func("Hello").ifPresent(v -> {
        System.out.println(v); // !Hello!
    });
}

public Optional<String> func(String value) {
    return Optional.ofNullable(value)
            .filter(v -> v.length() < 10)
            .map(v -> "!" + v + "!");
}
```

Date and Time API (JSR-310)