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

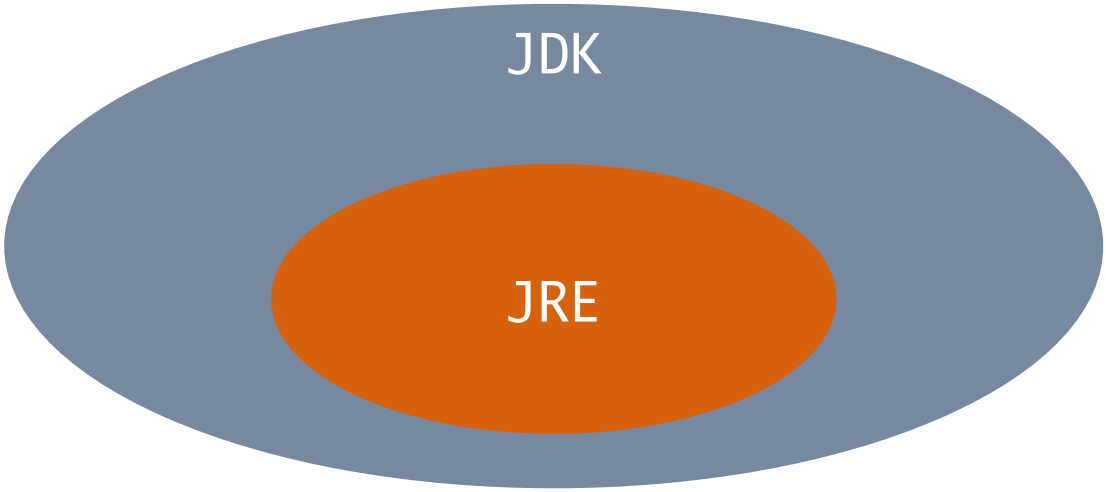
JRE と JDK の違いは？

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．

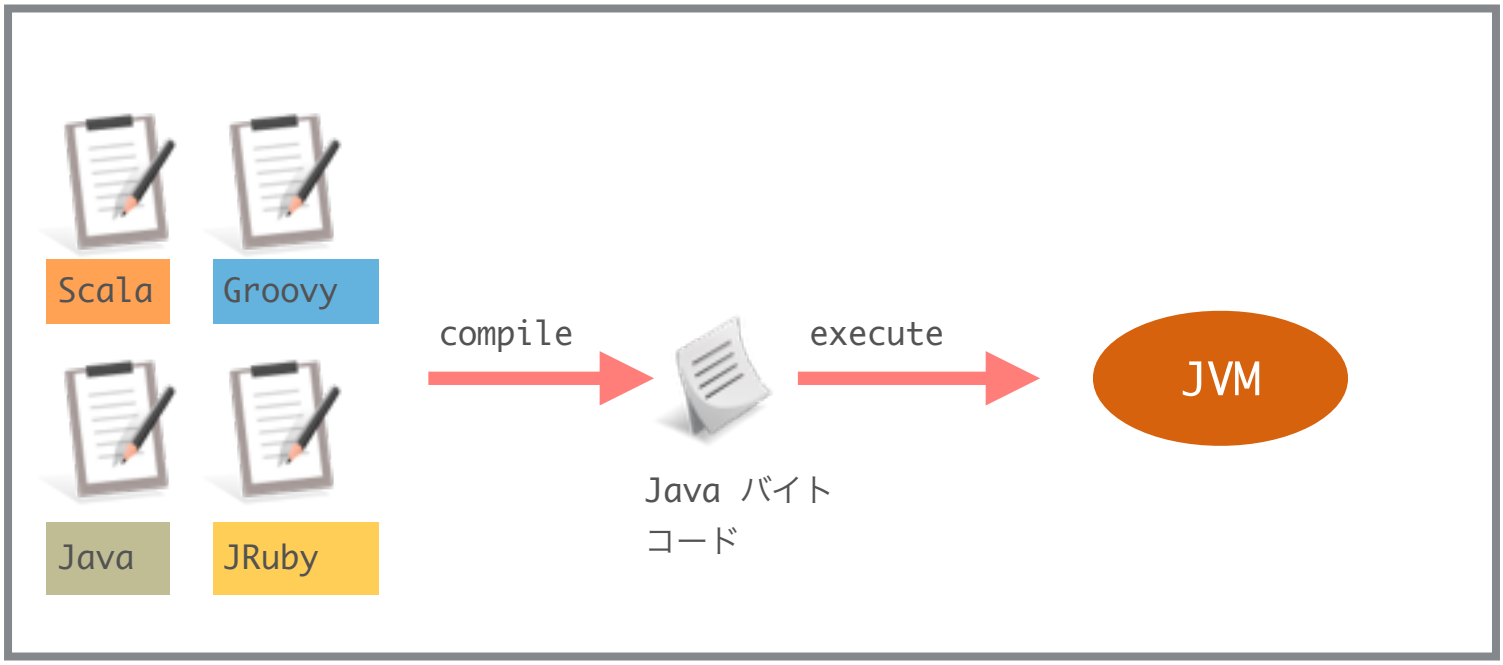
JVM は Java だけのものではない

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.

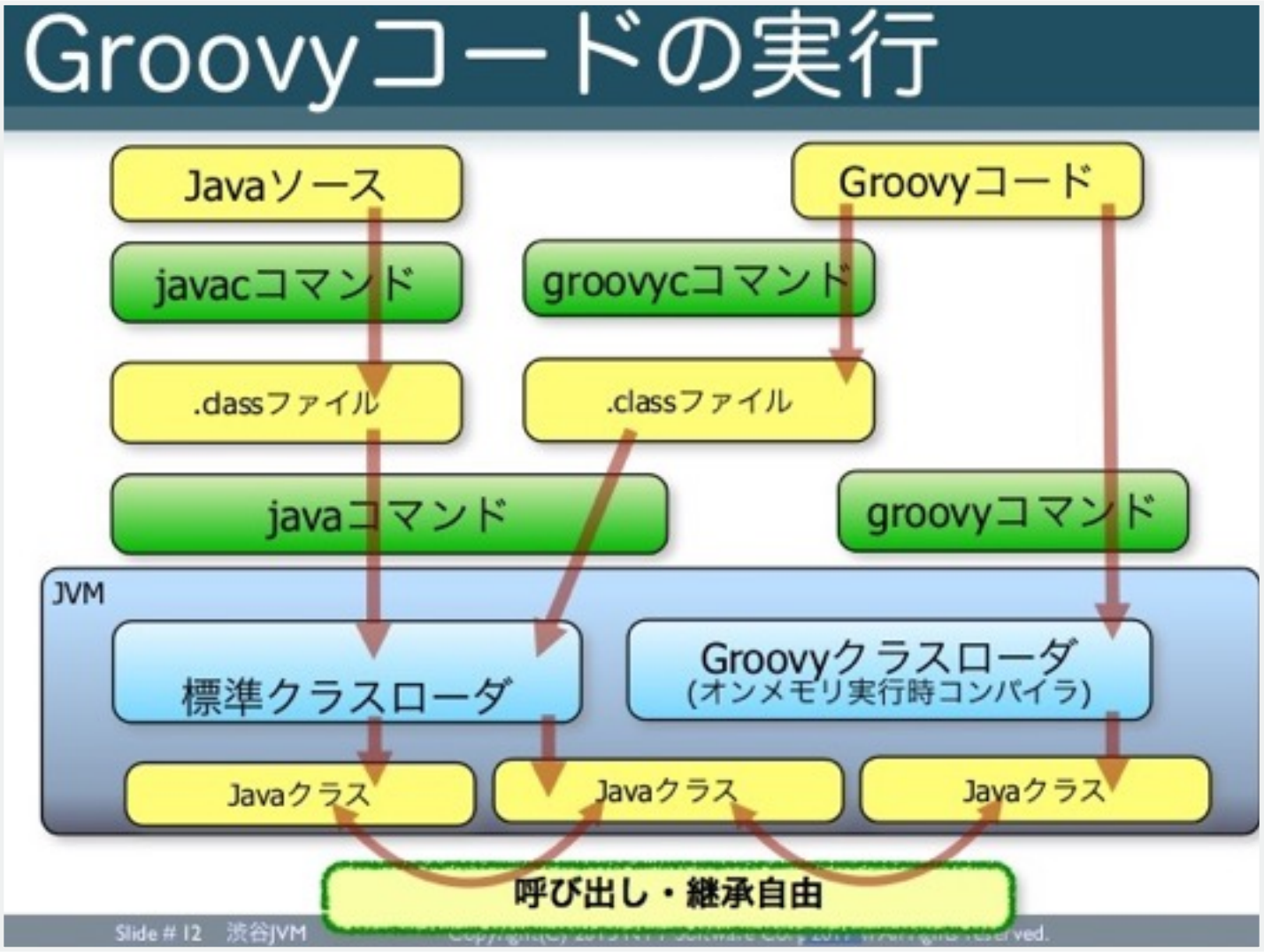
そのため、JVM 上でプログラムを動作させたい場合、Java バイトコードさえ作成できれば言語は Java である必要は無いということです.

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

```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.

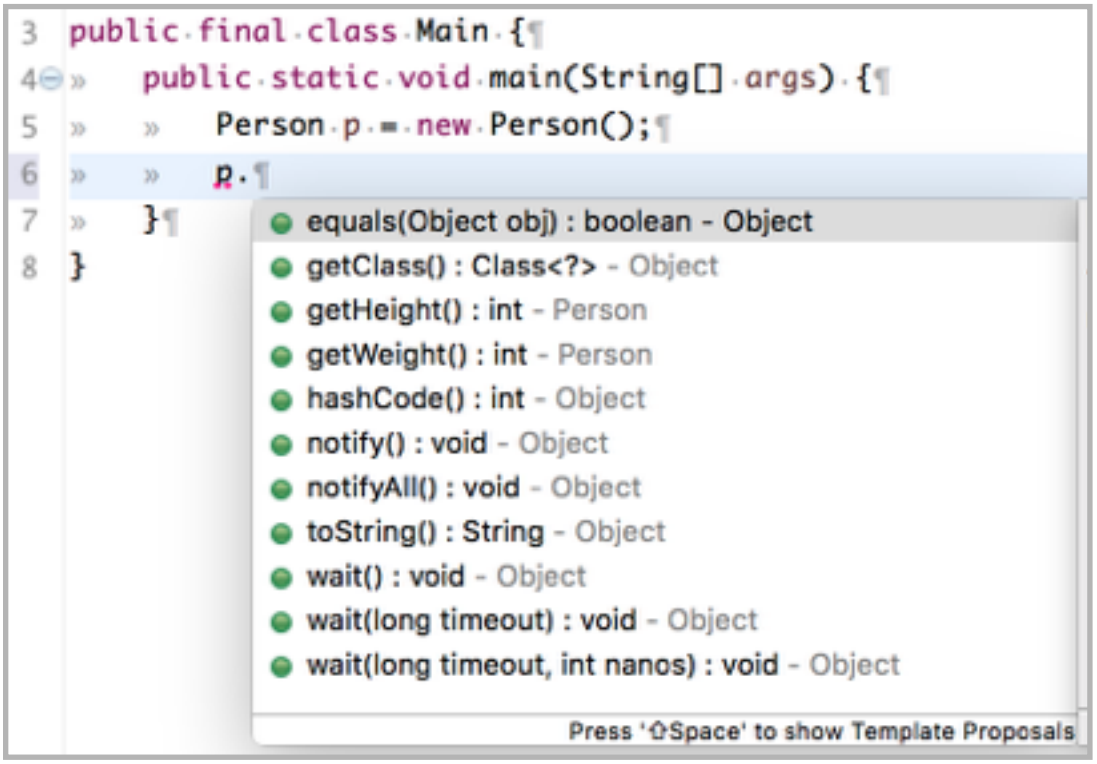
```
class ExPerson extends Person { }
 -> The type ExPerson cannot subclass the final class Person
```

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

スレッドダンプ＆GC

スレッドダンプ

スレッドダンプとは

ある時点の JVM 上で動作しているスレッドの情報が出力されたダンプデータです．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．

なぜスレッドダンプを解析するか

Java はマルチスレッドで動作するプログラムを作成することができますが、マルチスレッド

スレッドダンプの取得方法

スレッドダンプの見方

表 1 スレッドダンプの主な情報

名称		CPU
スレッド名		

スレッドダンプの解析方法

スレッドダンプの解析ツール

テキストデータのため、そのまま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それを使うのが良いと思います.

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、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、プログラムの処理で必要無くなったインスタンスは、どこからも参照されなくなり、ヒープ上に

ヒープとは

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.

- ヒープ領域
- 非ヒープ領域
- ネイティブメモリ
- C

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.

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
    - 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」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．

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

ヒープサイズの指定

GC に必要な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.

-Xms<値> …… ヒープ初期サイズ
-Xmx<値> …… ヒープ最大サイズ

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.

-XX:NewRatio=<値: デフォルト値=2>
-XX:NewSize=<値>
-XX:MaxNewSize=<値>

-Xmn<値>

young = ヒープ初期サイズ / (1 + NewRatio)

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

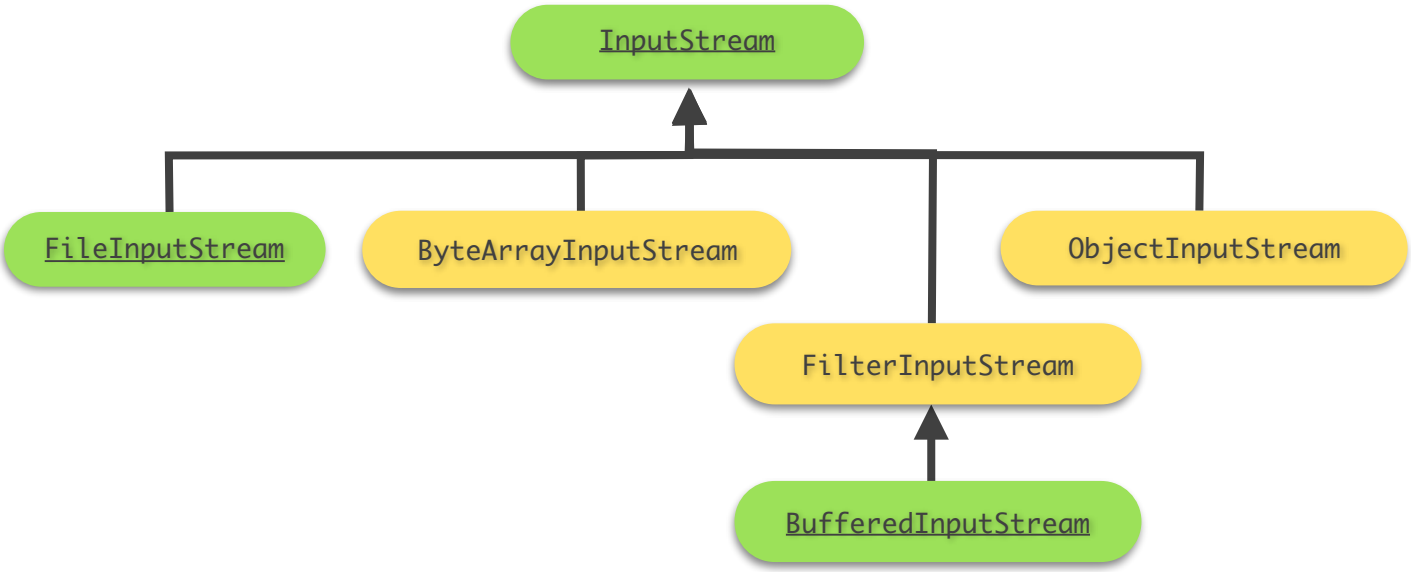
Java ではデータの入出力をストリームというデータの流れを表す概念として扱います. 入力元や出力先がディスクだけでなく、ネットワークなどであっても抽象的に入出力が扱うことができます.
大きく下記の種類に分けることができます.

- 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

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

関数型インターフェース

ラムダ式

Stream

メソッド参照

Date and Time API (JSR-310)