

# Java 5 以降の Java

Yoshiori SHOJI

1000speakers

# 自己紹介

- ( r y
- 所属 java-ja

珍回答  
ズラリ!

街頭実験室② エンジンの整備、セキュリティエンジン...

お嬢様系女子大生を直撃☆  
SEの意味わかりますか？



[http://rikunabi-next.yahoo.co.jp/tech/docs/ct\\_s03600.jsp?p=000608](http://rikunabi-next.yahoo.co.jp/tech/docs/ct_s03600.jsp?p=000608)

珍回答  
ズラリ!

街頭実験室② エンジンの整備、セキュリティエンジン…

お嬢様系女子大生を直撃☆  
SEの意味わかりますか?



[http://rikunabi-next.yahoo.co.jp/tech/docs/ct\\_s03600.jsp?p=000608](http://rikunabi-next.yahoo.co.jp/tech/docs/ct_s03600.jsp?p=000608)

えみ

「今日の合コン、Javaを作ってるプログラマが来るらしいよ」

珍回答  
ズラリ!

街頭実験室② エンジンの整備、セキュリティエンジン…

お嬢様系女子大生を直撃☆  
SEの意味わかりますか?



[http://rikunabi-next.yahoo.co.jp/tech/docs/ct\\_s03600.jsp?p=000608](http://rikunabi-next.yahoo.co.jp/tech/docs/ct_s03600.jsp?p=000608)

えみ

「今日の合コン、Javaを作ってるプログラマが来るらしいよ」

みなみ

こういうの欲しいって言って、かわいいJavaとか作ってくれたらうれしいな♪

Java = 毛テ

みんな、ブログに

# みんな、ブログに

- こんな僕でも Java でモテました！！

# みんな、ブログに

- こんな僕でも Java でモテました！！
- 今まで自信の無かった僕ですが  
Java のおかげで彼女が出来ました！！

# みんな、ブログに

- こんな僕でも Java でモテました！！
- 今まで自信の無かった僕ですが  
Java のおかげで彼女が出来ました！！
- Java のおかげで合コンでも  
モテモテです！！

書くの良いよ

と言うわけで

# 1.4 時代と比べて

格好良くなるために

これだけ覚えておけば

明日からモテモテな

5つの事

# Generics

# Generics

# Generics

- 総称型, 型総称性

# Generics

- 総称型,型総称性
- 型の安全性を確保しつつ,汎用性も

# Generics(とりあえず使ってみる)

```
List<String> list = new ArrayList<String>();  
list.add("hoge");  
String hoge = list.get(0);
```

```
Map<String,Integer> map =  
    new HashMap<String,Integer>();
```

```
Set<String> set = new HashSet<String>();
```

# Generics(自分で作る)

```
public class Something<T> {  
    protected T t;  
  
    public Something(T t) {  
        this.t = t;  
    }  
  
    public T get() {  
        return t;  
    }  
}
```

**Generics**(もうちょっと)

# Generics(もうちょっと)

- ワイルドカードもある <?>

# Generics(もうちょっと)

- ワイルドカードもある <?>
- <? extends T> とか

# Generics(もうちょっと)

- ワイルドカードもある <?>
- <? extends T> とか
- <? super T> とか

# Generics(良く言われる)



# Generics(個人的感想)

# Generics(個人的感想)

- 最初に見たとき俺も思った。

# Generics(個人的感想)

- 最初に見たとき俺も思った。
- 使ってみると元には戻れない

# Generics(個人的感想)

- 最初に見たとき俺も思った。
- 使ってみると元には戻れない
- キャスト不要超便利！！

# Generics(個人的感想)

- 最初に見たとき俺も思った。
- 使ってみると元には戻れない
- キャスト不要超便利！！
- `Map<String,List<String>>` とかも書けるYO!

# Annotation

# Annotation

# Annotation

- いわゆるメタデータ

# Annotation

- いわゆるメタデータ
- @hogehoge っ て書く

# Annotation

- いわゆるメタデータ
- @hogehoge っ て書く
- 情報を付加する

# Annotation

- いわゆるメタデータ
- @hogehoge って書く
- 情報を付加する
- java.lang.reflect.AnnotatedElement  
(Class, Constructor, Field, Method, Package)

# Annotation(自分で作ってみる)

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
@interface TestAnnotation {
    String value() default "HelloWorld";
}
```

# Annotation (Retentionについて)

# Annotation (Retentionについて)

- SOURCE  
コンパイル時などに使う  
(バイトコードには含まれない)
- CLASS  
バイトコードに含まれる  
(javassistなどのバイトコード変換時に)
- RUNTIME  
実行時に見れる  
(リフレクションで取得できる)

# Annotation(リフレクションで取得)

```
@TestAnnotation("class HelloWorld")
class Test{
    @TestAnnotation("field HelloWorld")
    private String hoge;

    @TestAnnotation("constructor HelloWorld")
    public Test(){
        super();
    }

    @TestAnnotation("method HelloWorld")
    public String getHoge(){
        return hoge;
    }
}
```

# Annotation(リフレクションで取得)

```
import java.lang.annotation.Annotation;
import java.lang.reflect.AnnotatedElement;
```

```
public class AnnotationHelloworld {

    /**
     * @param args
     */
    public static void main(String[] args) {
        Class clazz = Test.class;
        printAnnotation(clazz);
        printAnnotation((AnnotatedElement[])clazz.getDeclaredConstructors());
        printAnnotation((AnnotatedElement[])clazz.getDeclaredFields());
        printAnnotation((AnnotatedElement[])clazz.getDeclaredMethods());
    }

    private static void printAnnotation(AnnotatedElement... elements){
        for(AnnotatedElement element :elements){
            for(Annotation annotation : element.getAnnotations()){
                System.out.println(annotation);

                // 実際に値を取得するときはこんな感じ
                //TestAnnotation foo = element.getAnnotation(TestAnnotation.class);
                //System.out.println(foo.value());
            }
        }
    }
}
```

# Annotation

(どこで使うか)

# Annotation

(どこで使うか)

- コンテナから Field に  
オブジェクトを入れたり

# Annotation

(どこで使うか)

- コンテナから Field に  
オブジェクトを入れたり
- メソッドにトランザクション範囲を  
指定したり

# Annotation

(どこで使うか)

- コンテナから Field に  
オブジェクトを入れたり
- メソッドにトランザクション範囲を  
指定したり
- 自分で書けば本当に使い道は無限



マヌケ  
が……

知っているが  
……

Pytho  
にも  
実力を  
示した

ド  
ド  
ド

ド  
ド  
ド

ド  
ド  
ド

# Autoboxing Auto-Unboxing

# Autoboxing

## Auto-Unboxing

# Autoboxing Auto-Unboxing

- プリミティブとラップクラスを自動変換

# Autoboxing Auto-Unboxing

- プリミティブとラップクラスを自動変換
- 意識しないで使えるけど考えずに使うと  
ダメダメ w

# Autoboxing Auto-Unboxing

```
private int i = 0;  
private Integer j = i; //こんな書き方が出来る  
private int k = j; //こんなのも出来る
```

# Autoboxing Auto-Unboxing

こうなってる

```
private int i = 0;  
private Integer j = Integer.valueOf(i);  
private int k = j.intValue();
```

# Autoboxing Auto-Unboxing 嵌りどころ

```
Integer i = 127;  
Integer j = 127;  
System.out.println(j == i); // true
```

```
i = 128;  
j = 128;  
System.out.println(j == i); // false
```

\*Sun の実装

なんで？

# Autoboxing Auto-Unboxing

## Integer#ValueOf()

```
public static Integer valueOf(int i) {  
    final int offset = 128;  
    if (i >= -128 && i <= 127) { // must cache  
        return IntegerCache.cache[i + offset];  
    }  
    return new Integer(i);  
}
```

# Autoboxing

# Auto-Unboxing

[http://java.sun.com/docs/books/jls/third\\_edition/html/conversions.html](http://java.sun.com/docs/books/jls/third_edition/html/conversions.html)

## Discussion

Ideally, boxing a given primitive value  $p$ , would always yield an identical reference. In practice, this may not be feasible using existing implementation techniques. The rules above are a pragmatic compromise. The final clause above requires that certain common values always be boxed into indistinguishable objects. The implementation may cache these, lazily or eagerly. For other values, this formulation disallows any assumptions about the identity of the boxed values on the programmer's part. This would allow (but not require) sharing of some or all of these references. This ensures that in most common cases, the behavior will be the desired one, without imposing an undue performance penalty, especially on small devices. Less memory-limited implementations might, for example, cache all characters and shorts, as well as integers and longs in the range of  $-32K - +32K$ .

じゃあ、  
全部ラッパーにして  
equals 使えば良くね？

# Autoboxing Auto-Unboxing 嵌りどころ

```
private static Integer i;  
private static Integer j = 1;  
public static void main(String[] args) {  
    System.out.println(i.equals(j));  
}
```

# Autoboxing Auto-Unboxing 嵌りどころ

ぬるぽ

```
private static Integer i;  
private static Integer j = 1;  
public static void main(String[] args) {  
    System.out.println(i.equals(j));  
}
```

# Autoboxing Auto-Unboxing

結論

# Autoboxing Auto-Unboxing 結論

- 便利だけどちゃんと意識しましょう

# Autoboxing Auto-Unboxing 結論

- 便利だけどちゃんと意識しましょう
- 意識しないで使うとホントダメダメw





Enhanced for loops

# Enhanced for loops

# Enhanced for loops

- 拡張 for 文 とか言われてる

# Enhanced for loops

- 拡張 for 文 とか言われてる
- foreach みたいなモノ

# Enhanced for loops

- 拡張 for 文 とか言われてる
- foreach みたいなモノ
- 配列とか List とか Set とかで使える  
(Iterable)

# Enhanced for loops

```
List<String> list = new ArrayList<String>();  
for(String text : list){  
}
```

```
//~~~~~
```

```
String[] list = new String[]{"h", "o", "g", "e"};  
for(String text : list){  
}
```

# Enhanced for loops

(内部的には)

# Enhanced for loops

(内部的には)

- Iteratorが使われてる  
(配列は普通に要素数にアクセス)

# Enhanced for loops

(内部的には)

- Iteratorが使われてる  
(配列は普通に要素数にアクセス)
- Iteratorが遅いは (ほとんど) 都市伝説  
実際に計測してみました↓

<http://yoshiori.org/blog/2006/12/javaiterator.php>

# Enhanced for loops

(内部的には)

- Iteratorが使われてる  
(配列は普通に要素数にアクセス)
- Iteratorが遅いは (ほとんど) 都市伝説  
実際に計測してみました↓  
<http://yoshiori.org/blog/2006/12/javaiterator.php>
- Iteratorパターンの肝は集合と反復子が別  
オブジェクトである所

# いまだに Java の Iterator が凄い遅いとか 勘違いしてる人が多いので

もう少し詳しく説明すると

ArrayList は AbstractList を継承していて

そこに Iterator メソッドがあるんだけど

そこで返してるのが AbstractList クラスの内部クラスの Itr ( Iterator を継承) なので

そのメソッド内で new Itr() してるコストがまずある。

( Iterator パターン)

で、Itr の next() メソッドは AbstractList の get(int) を呼んでるから  
そのコストが一つ

ちなみに拡張 for 文使うために

Iterable インターフェースも実装してるけど

そこで定義されているのは上記 Iterator メソッドのみ

# Enhanced for loops

(みんな使いましょう)

# Enhanced for loops

(みんな使いましょう)

- 反復の抽象化大事 (変更に対応できる)

# Enhanced for loops

(みんな使いましょう)

- 反復の抽象化大事 (変更に対応できる)
- 読みやすい

# Enhanced for loops

(みんな使いましょう)

- 反復の抽象化大事 (変更に対応できる)
- 読みやすい
- 型も安全

まだ  
拡張 for 文  
使ってないの？

もったいない！



# Enumerations

# Enumerations

# Enumerations

- TypeSafe Enum

# Enumerations

- TypeSafe Enum
- 定数などに

# Enumerations

- TypeSafe Enum
- 定数などに
- ぶっちゃけ内部的にはクラス

# Enumerations

(凄く簡単に)

```
public enum User {  
    AMACHANG,  
    NISHIO_HIROKAZU  
}  
  
public blog getUserBlog(User user){  
    //  
}
```

# Enumerations

(値を持たせたり)

```
public enum User {  
    AMACHANG("id:amachang"),  
    NISHIO_HIROKAZU("id:nishiohirokazu")  
  
    private String hatenaID;  
  
    private User(String hatenaID) {  
        this.hatenaID = hatenaID;  
    }  
}
```

# Enumerations

(switch文も書けたり)

```
switch (user) {  
  case AMACHANG:  
    //~~  
    break;  
  
  case NISHIO_HIROKAZU:  
    //~~  
    break;  
  
  default:  
    //~~  
    break;  
}
```

ん？

Switch？

プリミティブじゃ

ないのに？

Enumerations の

Switch 文

# Enumerations の Switch 文

- 内部的には ordinal 使ってます

# Enumerations の Switch 文

- 内部的には ordinal 使ってます
- つまり Switch 文でヌルポが出る可能性がある  
があるので注意

# Enumerations の Switch 文

- 内部的には ordinal 使ってます
- つまり Switch 文でヌルポが出る可能性がある  
があるので注意
- ちなみに内部的には \$SWITCH\_TABLE\*  
とかのメソッド出来てる



ドドドド

ド

Ennam  
カワイイヨ

落ちて書いて  
くださいッ!

Ennum!

Est  
nay  
uf  
mee  
!

どうか!  
どうか!  
落ちて書いて  
ください

# 可变引数

# 可變引數

# 可変引数

- 型宣言に「...」付けるだけ

# 可変引数

- 型宣言に「...」付けるだけ
- 受け取る方は配列として受け取る

# 可変引数

- 型宣言に「...」付けるだけ
- 受け取る方は配列として受け取る
- もちろん配列も渡せる

# 可變引數

```
String[] foo = {"h","o","g","e"};
```

```
function(foo); // OK
```

```
function("ho","ge") // OK
```

```
private static void function(String... strings){
```

```
    String[] bar = strings; // OK
```

```
    for(String text : strings){ // OK
```

```
        //~~
```

```
    }
```

```
}
```



Q. 特に何もありません

# まとめ

# まとめ

- Java 1.0 が<sup>ら</sup>出て既に12年

# まとめ

- Java 1.0 が<sup>で</sup>出て既に12 年
- JavaSE5 が<sup>で</sup>出たの4 年前

# まとめ

- Java 1.0 がでて既に12年
- JavaSE5 が出たの4年前
- もう、1/3 たってます！！！！

# まとめ

- Java 1.0 がでて既に12年
- JavaSE5 が出たの4年前
- もう、1/3 たってます！！！！
- ホントが一番面白いのは  
java.util.concurrent だったりしますw

ん？



- JavaSE6 - java version 1.6

- JavaSE6 - java version 1.6
- 5 - java version 1.5

- JavaSE6 - java version 1.6
- 5 - java version 1.5
- 4 - java version 1.4

- JavaSE6 - java version 1.6
- 5 - java version 1.5
- 4 - java version 1.4
- 3 - java version 1.3

- JavaSE6 - java version 1.6
- 5 - java version 1.5
- 4 - java version 1.4
- 3 - java version 1.3
- 2 - java version 1.2

- JavaSE6 - java version 1.6
- 5 - java version 1.5
- 4 - java version 1.4
- 3 - java version 1.3
- 2 - java version 1.2
- 1 - java version 1.1

- JavaSE6 - java version 1.6
- 5 - java version 1.5
- 4 - java version 1.4
- 3 - java version 1.3
- 2 - java version 1.2
- 1 - java version 1.1

- JavaSE6 - java version 1.6
- 5 - java version 1.5
- 4 - java version 1.4
- 3 - java version 1.3
- 2 - java version 1.2
- 1 - java version 1.1

あれ？

- JavaSE6 - java version 1.6
- 5 - java version 1.5
- 4 - java version 1.4
- 3 - java version 1.3
- 2 - java version 1.2
- 1 - java version 1.1

あれ？

java version 1.0 は Java 0 ??

ま、

色々あって

Java は

サバよんでるっぽい

ナンバーリング

になってます w

# ちなみに今日の内容

# ちなみに今日の内容

- Generics

# ちなみに今日の内容

- Generics
- Annotation

# ちなみに今日の内容

- Generics
- Annotation
- AutoboxingAuto-Unboxing

# ちなみに今日の内容

- Generics
- Annotation
- AutoboxingAuto-Unboxing
- Enhanced for loops

# ちなみに今日の内容

- Generics
- Annotation
- AutoboxingAuto-Unboxing
- Enhanced for loops
- Enumerations

# ちなみに今日の内容

- Generics
- Annotation
- AutoboxingAuto-Unboxing
- Enhanced for loops
- Enumerations
- 可変引数

0 から数えるよね！

# 参考

- Java in the Box  
<http://www.javainthebox.net/>
- JSR-176 J2SE 5.0 Release Contents  
<http://jcp.org/en/jsr/detail?id=176>

おわり