

君はTigerを見たか

- J2SE1.5 の新機能 -

横河電機
櫻庭 祐一



Agenda

- ◆ Tiger の概要
- ◆ Tigerのテーマ
- ◆ Ease of Development
- ◆ Reliability, Availability and Serviceability
- ◆ Desktop Client
- ◆ Performance and Scalability

Tiger の概要

- ◆ J2SE のメジャーバージョンアップ
 - ◆ Tiger はコードネーム
- ◆ リリースの予定は 2004 年夏 ?
- ◆ JCP により仕様が策定されている
 - ◆ JSR-176
 - ◆ メンバーは
Apache, Apple, Borland, Cisco, 富士通, HP, IBM, Macromedia, Nokia, Oracle, SAS, SAP, SavaJe, Osvaldo Doederlein, Juergen Kreieder
 - ◆ 現状は Community Draft Ballot を通過 もうすぐ Public Review
- ◆ 今までにないほどの変化
 - ◆ 言語仕様が大幅変更
 - ◆ 関連する JSR が 15
 - ◆ Bug Fix とパフォーマンスの向上

Tiger の概要 cont.

◆ Tiger に関連する JSR

- ◆ JSR-003 Java Management Extensions (JMX) Specification
- ◆ JSR-013 Decimal Arithmetic Enhancement
- ◆ JSR-014 Add Generic Types to the Java Programming Language
- ◆ JSR-028 Java SASL Specification
- ◆ JSR-114 JDBC Rowset Implementations
- ◆ JSR-133 Java Memory Model and Thread Specification Revision
- ◆ JSR-163 Java Platform Profiling Architecture
- ◆ JSR-166 Concurrency Utilities
- ◆ JSR-174 Monitoring and Management Specification for the Java VM
- ◆ JSR-175 A Metadata Facility for the Java Programming Language
- ◆ JSR-199 Java Compiler API
- ◆ JSR-200 Network Transfer Format for Java Archives
- ◆ JSR-201 Extending the Java Programming Language
with Enumerations, Autoboxing, Enhanced for loops and Static Import
- ◆ JSR-204 Unicode Supplementary Character Support
- ◆ JSR-206 Java API for XML Processing (JAXP) 1.3

- ◆ その他にも JSR-202 Class File Spec. Update, JSR-203 NIO2 など

Tiger's Themes

- ◆ Ease of Development
- ◆ Reliability, Availability and Serviceability
- ◆ Desktop Client
- ◆ Performance and Scalability



Tiger's Themes

- ◆ Ease of Development
 - ◆ Metadata
 - ◆ 拡張 for, enum, static import, autoboxing, varargs
 - ◆ Disconnected Rowset (JDBC)
- ◆ Reliability, Availability and Serviceability
 - ◆ Generics
 - ◆ いつでも Stuck Trace
 - ◆ JVM Monitoring
 - ◆ Profiling
 - ◆ Concurrency Utilities
- ◆ Desktop Client
- ◆ Performance and Scalability





- ◆ コンパイラやツールに対してヒントを与えるための機構

- ◆ C# のカスタム属性の Java 版
- ◆ メタデータをつけられるのは クラス、メソッド、フィールド

- ◆ 新しいキーワード @

- ◆ たとえば

```
public @Webmethod void foo() { ... }
```

- ◆ なんでもかんでも書けるわけではない

- ◆ あらかじめ定義する必要がある
- ◆ @interface を使用して定義する

```
public @interface RequestForEnhancement {  
    int id();  
}
```

- ◆ 何に使うか

- ◆ ソースコードの自動生成
 - RMI や JAX-RPC でインタフェースを自動生成
- ◆ web.xml や BeanInfo などの自動生成
- ◆ アスペクト指向のプログラミング



Ease of Development - Metadata - cont.

◆ 使い方

1 アノテーションの定義

定義済みのアノテーションを使用するならば省略

```
public @interface RequestForEnhancement {  
    int id();  
    String engineer();  
    String date();  
}
```

引数や使用できる型など制約あり

実際には `java.lang.annotation.Annotation` インタフェースの派生インタフェースになる

2 アノテーションの使用

```
@RequestForEnhancement(  
    id = 2868724;  
    engineer = "Poindexter";  
    date = "4/1/2004"; )  
public static void travelThroughTime(Date destination) { ... }
```



Ease of Development - Metadata - cont.

◆ 使い方

3 アノテーションの読み込み (ツールなどで)

リフレクションを利用する

```
Method m = TimeTravel.class.getMethod("travelThroughTime", new Class[] {Date.class});  
RequestForEnhancement rfe = m.getAnnotation(RequestForEnhancement.class);
```

```
int id = rfe.id();  
String engineer = rfe.engineer();  
String date = rfe.date();
```

◆ デフォルトで用意されているアノテーション

Documented
Inherited
Target
Visibility

仕様書にはその他に Retention, Overrides の記述があり
順次増加していくと思われる



- ◆ **拡張 for 文 = For Each**

- ◆ 要素がなくなるまでループする
- ◆ 使えるのは Collection, 配列

```
List list = new ArrayList();  
...  
for (Object o : list) {  
    System.out.println((Integer)o);  
}
```

- ◆ VM は変更なし コンパイラ (javac) のみで対応
- ◆ 実際には Iterator で書き直しているだけ

```
Object obj;  
for(Iterator iterator = list.iterator(); iterator.hasNext(); System.out.println((Integer)obj))  
    obj = iterator.next();
```

パフォーマンスが必要なときはやめたほうがいいかもしれない





◆ Effective Java の Typesafe Enum

- ◆ C の enum の問題をクリア enum の要素は int ではない
- ◆ 書き方

```
enum Number {ONE, TWO, THREE, FOUR}
```

行端のセミコロンはいらない。

- ◆ 比較 (equals, compareTo) が可能
- ◆ switch でも使える
- ◆ ユーザにやさしい toString
System.out.println(Number.ONE);

実行

```
>java XXXX
```




```
ONE
```

◆ java.lang.Enum クラスの派生クラスとして実装される

- ◆ JVM の変更はなし
- ◆ 便利なメソッドが用意されている value, compareTo, name など
- ◆ さまざまな拡張が可能

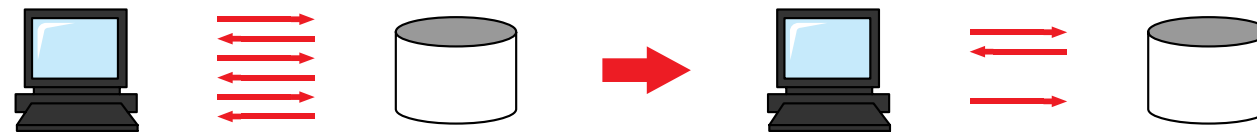
```
public enum Num {ONE, TWO, THREE, FOUR;  
    public String toString() {  
        return name().substring(0,1).toUpperCase() + name().substring(1).toLowerCase();  
    }  
}
```

Ease of Development - その他の仕様変更 -

- ◆ Autoboxing/Unautoboxing 
 - ◆ ラップクラスとプリミティブクラスを区別しないで扱える
Integer x = 10;
int y = new Integer(5);
 - ◆ コレクションで使うと便利 しかし、使いすぎはパフォーマンスダウン
- ◆ Static Import 
 - ◆ import 文を拡張して static 変数/static メソッドをインポートできるように
import static java.lang.Math.*;
...
double x = sin(PI / 180.0 * theta);
- ◆ 可変長引数 
 - ◆ フォーマットには欠かせない
 - ◆ しかし、それ以外の使い道は...



- ◆ J2SE 1.4 で提供されていたが...
 - ◆ ResultSet をラップしていただけ
- ◆ CachedRowSet DB との通信をキャッシュする
 - ◆ 大幅な通信量の削減が可能



- ◆ WebRowSet リモートとのRowSetのやり取り
 - ◆ リモートのクライアントは直接 DB にはアクセスしない
 - ◆ データは XML を使用





◆ Generics とはなにか

- ◆ 型によるクラス、インタフェース、メソッドのパラメタ化
- ◆ C++ のテンプレートのようなものだが、ちょっと違う
タイプチェックが行われる

◆ 例

OLD

```
List list = new ArrayList();  
list.add(new Integer(10));  
int x = ((Integer)list.get(0)).intValue(); // キャストが必要
```

NEW

```
List<Integer> list = new ArrayList<Integer>();  
list.add(new Integer(10));  
int x = list.get(0).intValue(); // キャストが不必要
```

◆ Integer でパラメタ化

List には Integer 以外は入れられなくなる





◆ Generics を使ったクラス定義

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

◆ ライブラリは何が変わった

- ◆ コレクションフレームワーク
- ◆ リフレクション
- ◆ ロギング API
 など





- ◆ JSR-174 軽い JVM のモニタリング
 - ◆ 24 時間週 7 日モニタリングできるように
 - ◆ SNMP のサポート
 - ◆ プロファイラより扱うデータを限定する
 - スレッドの一覧、スレッドのステータス、メモリの使用状況、GC

- ◆ JSR-163 Profiling JVMPI から JVMTI
 - ◆ デバッガ (JPDA) との統合
 - ◆ Out Process でのプロファイリング
 - ◆ 様々な情報の取得





- ◆ Doug Lea による並列プログラミングのユーティリティ
 - ◆ 非同期処理
 - ◆ スレッドプール
 - ◆ キュー、ブロッキングキュー
 - ◆ 時間表現
 - ◆ ロック
 - ◆ アトミック処理
 - ◆ 同期機構
- ◆ 非同期処理
 - ◆ `java.util.concurrent.Executor` と `java.util.concurrent.Executors`
 - ◆ `Runnable` と `Callable`
 - Callable は戻り値がある





```
Executor e = new XXXXExecutor();
Future<String> future = Executors.execute(e, new Callable<String>() {
    public String call() {
        ...
        return "Good";
    }
});

try {
    String result = future.get();
    System.out.println("CallableTask Execution Result: " + result);
} catch (Exception ex) {
    ex.printStackTrace();
}
```

◆ スレッドプール

- ◆ Executors のユーティリティメソッド
 - newCachedThreadPool
 - newFixedThreadPool
 - newSingleThreadExecutor





◆ ロック

- ◆ synchronized ではないロックの機構

```
Lock lock = new ReentrantLock();  
  
...  
lock.lock(); // block until condition holds  
try {  
    ...  
} finally {  
    lock.unlock()  
}
```

◆ 同期機構

- ◆ Semaphore
- ◆ CountdownLatch
- ◆ CyclicBarrier
- ◆ Exchanger



Conclusion

- ◆ 大幅な言語仕様の変更
 - ◆ 使いこなせば開発効率が向上
- ◆ その他の新機能も盛りだくさん
 - ◆ 今日紹介したのはほんの一部
- ◆ パフォーマンスの向上
 - ◆ HotSpot の更なるチューニング
 - ◆ 巨大な Heap をサポート
 - ◆ フットプリントの減少
 - ◆ 起動時間の短縮
- ◆ 完備されていないドキュメント
 - ◆ 使いたくても使い方がわからないものも多い
 - ◆ JavaDoc さえない
- ◆ Watch out for Tiger