効率化Javaプログラミング

～意外と知られていないJavaの便利機能～

株式会社エスプランニング
本多 鉄兵
前田 敬

自己紹介(前田)

- (株)エスプランニング ビジネスソリューションGr主任
- 28歳
- 2年前に札幌にUターン。ゴキブリが怖い…
- 最近はSwingでの業務系アプリ開発に従事
- Scalaに興味津々
- 札幌Javaコミュニティに参加
- 趣味は万年筆集め

自己紹介(本多)

- (株)エスプランニング 取締役
- 33歳
- 業務経験は10年程度、初めの開発業務でJava/C++を扱い、以降オブジェクト指向にはまる
- 現在は内部監査など作業をし、あまり現場作業を出来ない
- 面白いUIのアプリケーションを作りたい
- 趣味は漫画
- お金持ちになりたいです

はじめに

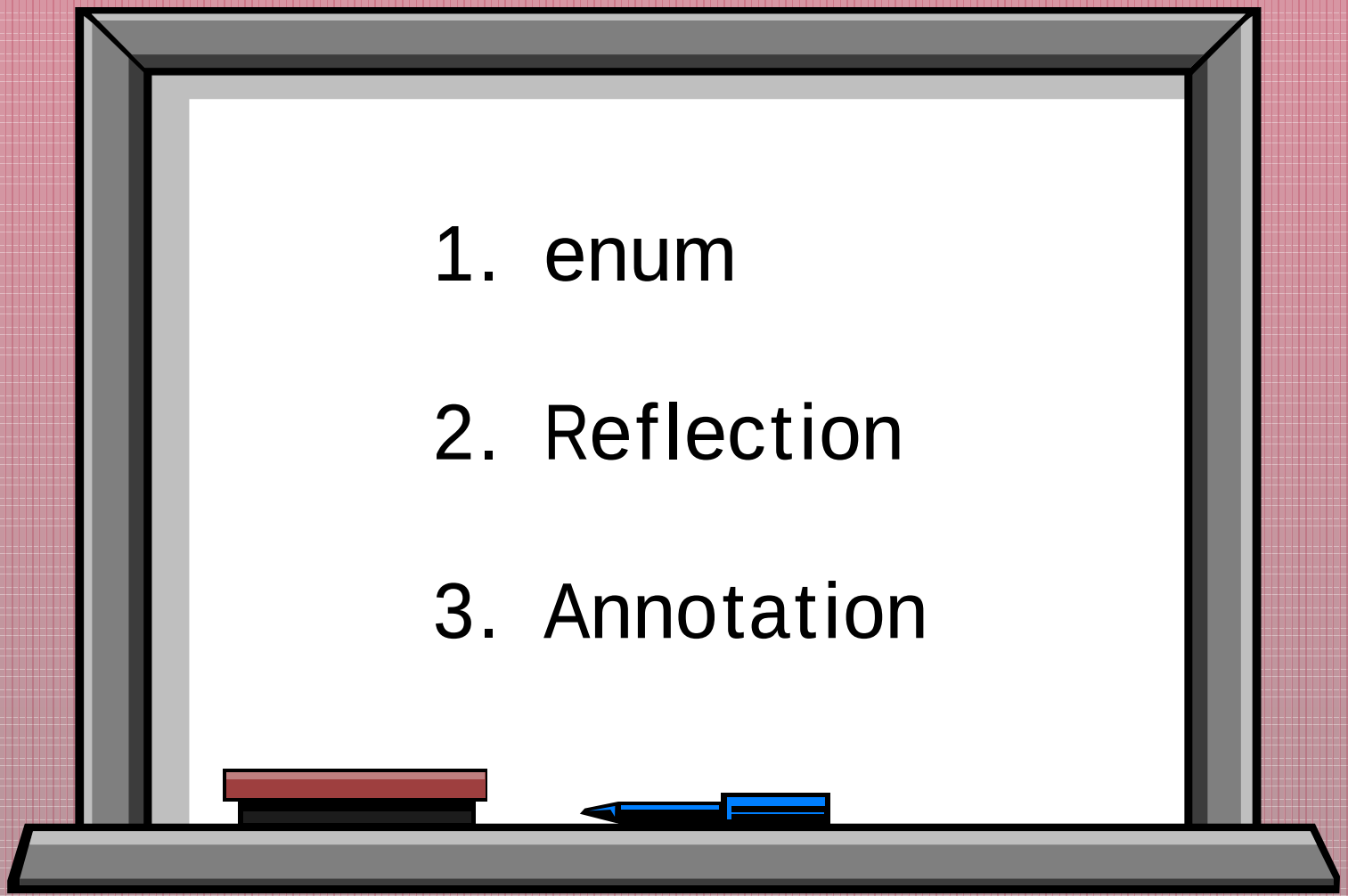
Javaプログラミングを行ううえで、効率化を考えた場合、既存フレームワークやライブラリの使用、IDEの選択/設定からコーディング規約の策定など、その方法は多岐に渡ります。

本セミナーではコーディングレベルで簡単に出来る効率化というところに焦点を当て、特に触れる機会の少ない（？）列挙体やアノテーション、リフレクションの使い方について、実践で培ったノウハウを元に説明いたします。

本セミナーがここにいる皆様方の一助となれば幸いです。

本日最後の70分間、お疲れのこととは思いますが、どうぞ最後までお付き合いください。

アジェンダ



enum

レジュメ

1. public static finalな定数クラスに別れを告げる
2. enumのポリモーフィズムを体感する
3. enumでデザインパターン
4. まとめ

よく見る定数クラス

```
public class Constraints {
    public static final char A = 'A';
    public static final char B = 'B';
    public static final char C = 'C';
    public static final char D = 'D';
    public static final char E = 'E';
    // ...
}
```

定数クラスを使ってみる

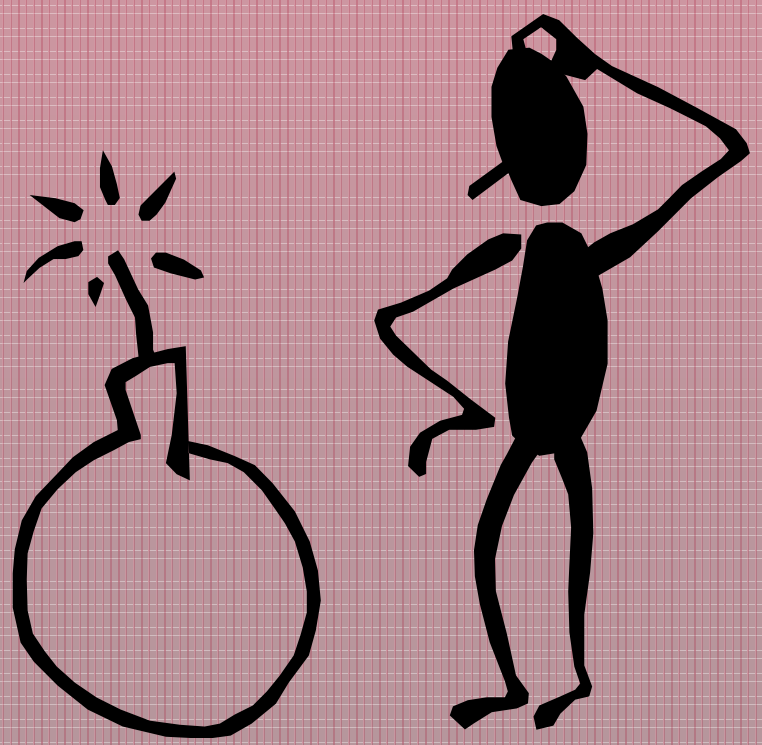
```
public class Student {

    private String name;
    private char rank;

    public Student(String name, char rank) {
        this.name = name;
        this.rank = rank;
    }
    //...
}
```

定数クラスの問題点

```
// 正常
Student teppei = new Student(“Teppei”, Constraints.A);

// 異常な指定だがコンパイルエラーにならない！
Student shinichi = new Student(“Shinichi”, 'F');
```

public static finalな定数クラスその他の問題

- ファイルの肥大化
- 定数の意味を変数名でしか定義できない
- static finalフィールドのインライン化



Type Safe Enumパターン

```
public class Rank {
    private final char rank;
    public static final Rank A = new Rank('A');
    public static final Rank B = new Rank('B');
    public static final Rank C = new Rank('C');
    public static final Rank D = new Rank('D');

    private Rank(char rank) {
        this.rank = rank;
    }
    public char getValue() {
        return this.rank;
    }
```

冗長過ぎる！

enumを使おう

```
public enum Rank {
    A('A'), B('B'), C('C'), D('D')
    ;
    private char rank;
    private Rank(char rank) {
        this.rank = rank;
    }
    public char getValue() {
        return this.rank;
    }
}
```

クラスファイルを見てみよう

```
 8  public final class Rank extends Enum {
 9
10      public static final Rank A;
11      public static final Rank B;
12      public static final Rank C;
13      public static final Rank D;
14      private char rank;
15      private static final Rank ENUM$VALUES[];
16
17      private Rank(String s, int i, char rank) {
18          super(s, i);
19          this.rank = rank;
20      }
21
22      public char getValue() {
23          return rank;
24      }
25
26      public static Rank[] values() {
27          Rank arank[];
28          int i;
29          Rank arank1[];
30          System.arraycopy(arank = ENUM$VALUES, 0, arank1 = new Rank[i = arank.length], 0, i);
31          return arank1;
32      }
33
34      public static Rank valueOf(String s) {
35          return (Rank) Enum.valueOf(constraint / Rank, s);
36      }
37
38      static {
39          A = new Rank("A", 0, 'A');
40          B = new Rank("B", 1, 'B');
41          C = new Rank("C", 2, 'C');
42          D = new Rank("D", 3, 'D');
43          ENUM$VALUES = (new Rank[] { A, B, C, D });
44      }
45  }
```

java.lang.Enumのメソッド一覧

メソッド	説明
public final String name()	列挙子の名前を返す
public final int ordinal()	列挙子の順番を返す
public final int compareTo(列挙子)	列挙子の順番で比較
public String toString()	文字列表現を返す(name()と同じ)
public static 列挙名 valueOf(String name)	文字列に該当する列挙子を返す
public static 列挙名[] values()	列挙子の配列を返す
public final Class<列挙名> getDeclaringClass()	列挙子のクラスを取得

潜在的な問題が含まれるコード

```
 3  public enum Rank {
 4
 5      A('A', 90.0d, 100.0d),
 6      B('B', 80.0d, 90.0d),
 7      C('C', 60.0d, 80.0d),
 8      D('D', 0.0d, 60.0d)
 9      ;
10      private char rank;
11      private double from;
12      private double to;
13
14      private Rank(char rank, double from, double to) {
15          this.rank = rank;
16          this.from = from;
17          this.to = to;
18      }
19      public char getValue() {
20          return this.rank;
21      }
22      public boolean includes(double score) {
23          return from <= score && score < to;
24      }
25  }
```

潜在的な問題が含まれるコード

```
3  public enum Rank {
4
5      A('A', 90.0d, 100.0d),
6      B('B', 80.0d, 90.0d),
7      C('C', 60.0d, 80.0d),
8      D('D', 0.0d, 60.0d)
9      ;
```

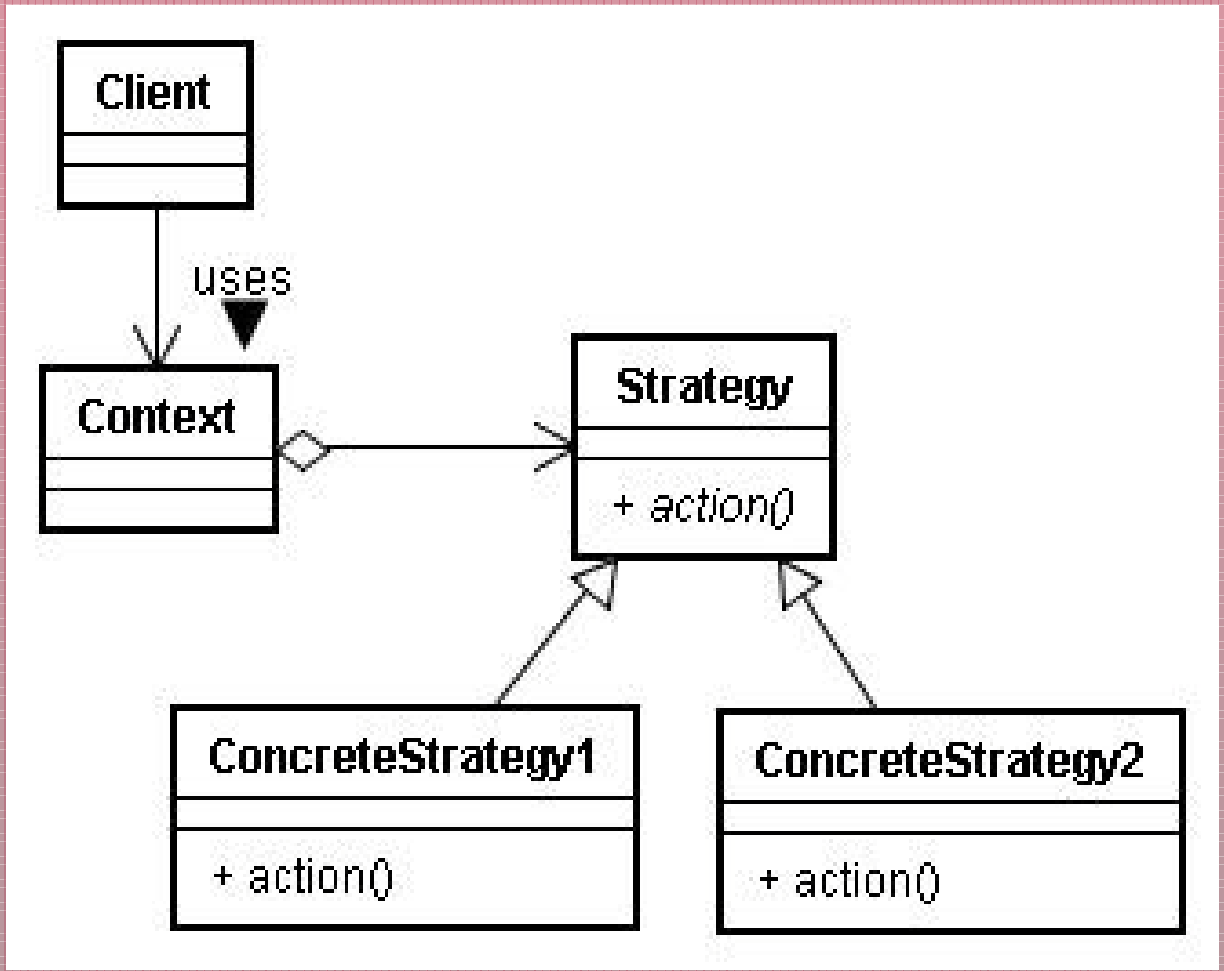
```
public boolean includes(double score) {
    return from <= score && score < to;
}
```

```
3  public class Main {
4
5      public static void main(String[] args) {
6          boolean result = Rank.A.includes(100.0d);
7          System.out.println(result); // false !
8      }
9  }
```

多態性を利用した解決方法

```
3  public enum Rank {
4
5      A('A', 90.0d, 100.0d) {
6          @Override
7          public boolean includes(double score) {
8              return super.includes(score) || score == to;
9          }
10     },
11     B('B', 80.0d, 90.0d),
12     C('C', 60.0d, 80.0d),
13     D('D', 0.0d, 60.0d)
14     ;
15     private char rank;
16     private double from;
17     protected double to;
18
19     private Rank(char rank, double from, double to) {
20         this.rank = rank;
21         this.from = from;
22         this.to = to;
23     }
24     public char getValue() {
25         return this.rank;
26     }
27     public boolean includes(double score) {
28         return from <= score && score < to;
29     }
30 }
```

Strategyパターンをenumで表現してみよう



じゃんけんの振舞いを抽象化したインターフェース

```
3  /**
4   * じゃんけんの振る舞い（グー、チョキ、パー）を抽象化した
5   * インターフェース.
6   */
7  public interface JankenStrategy {
8
9       /**
10       * 手を出す.
11       */
12      void hand();
13 }
```

JankenStrategyの実装クラス

```
3  public class GuStrategy implements JankenStrategy{
4
5      @Override
6      public void hand() {
7          System.out.println("ぐー");
8      }
9  }
```

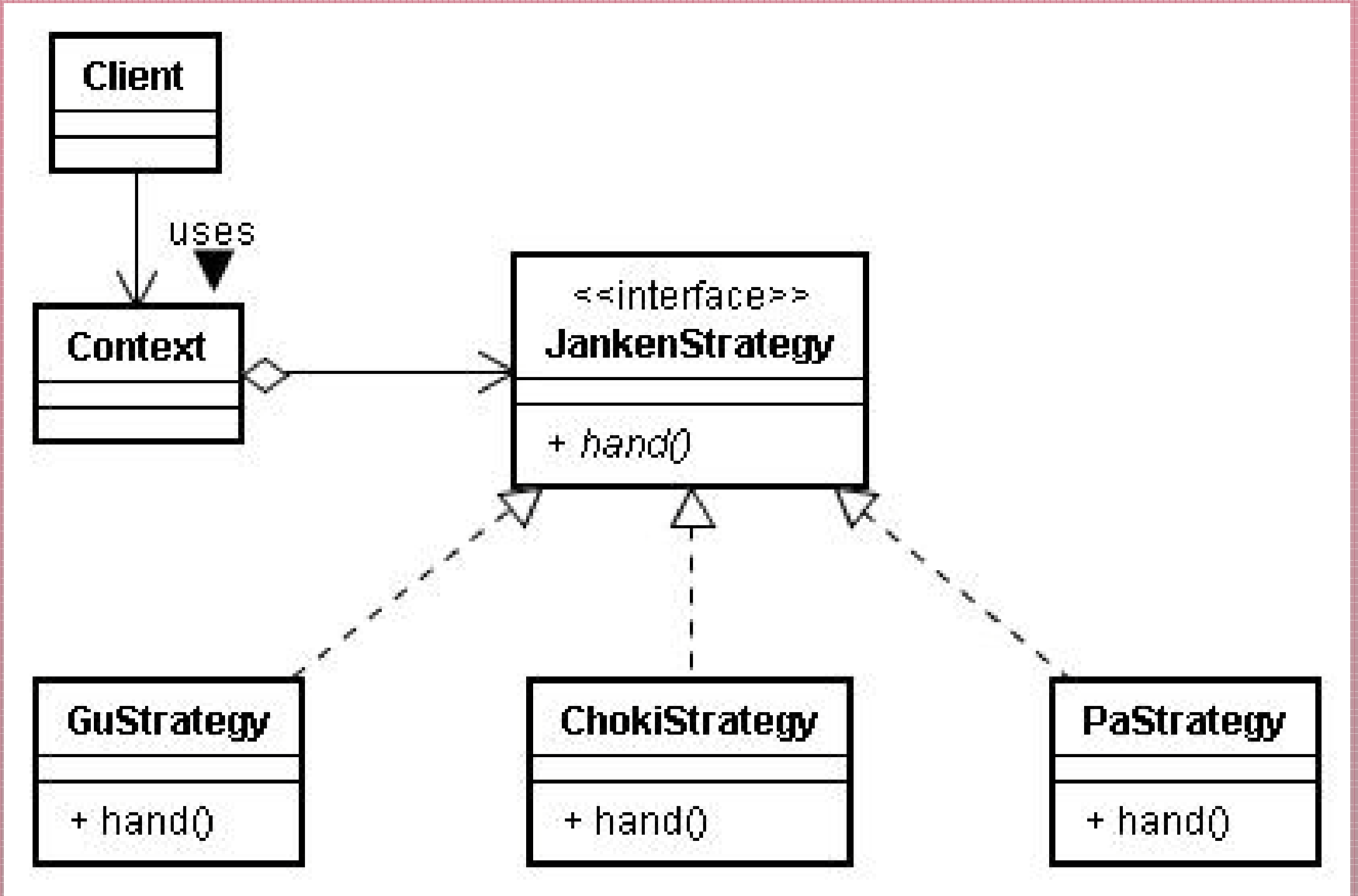
```
3  public class ChokiStrategy implements JankenStrategy {
4
5      @Override
6      public void hand() {
7          System.out.println("ちょき");
8      }
9  }
```

```
3  public class PaStrategy implements JankenStrategy {
4
5      @Override
6      public void hand() {
7          System.out.println("ぱー");
8      }
9  }
```

条件を判定するContextクラス

```
 6  public class JankenContext {
 7
 8      private static final Map<String, JankenStrategy> strategyMap;
 9
10      static {
11          strategyMap = new HashMap<String, JankenStrategy>();
12          strategyMap.put("GU", new GuStrategy());
13          strategyMap.put("CHOKI", new ChockiStrategy());
14          strategyMap.put("PA", new PaStrategy());
15      }
16      public JankenContext() {
17      }
18      public void hand(String kind) {
19          strategyMap.get(kind).hand();
20      }
21  }
```

クラス図


Client
uses
Context
<<interface>>
JankenStrategy
+ hand()
GuStrategy
+ hand()
ChokiStrategy
+ hand()
PaStrategy
+ hand()

JankenStrategyImplを作成

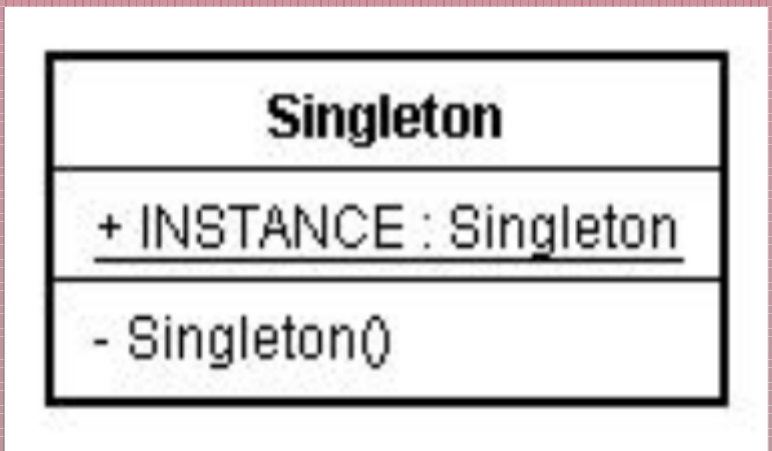
```
3  public enum JankenStrategyImpl implements JankenStrategy {
4
5      GU {
6          @Override
7          public void hand() {
8              System.out.println("ぐー");
9          }
10     },
11
12     CHOKI {
13         @Override
14         public void hand() {
15             System.out.println("ちょき");
16         }
17     },
18
19     PA {
20         @Override
21         public void hand() {
22             System.out.println("ぱー");
23         }
24     }
25 }
```

Contextも大幅にコード量減

```
3  public class JankenContext {
4
5      public void hand(String kind) {
6          JankenStrategy strategy = JankenStrategyImpl.valueOf(kind);
7          strategy.hand();
8      }
9  }
```

Singletonとは？

- あるクラスについて、そのインスタンスが単一であることを保証するためのデザインパターン



Singletonの作り方

1.自身のインスタンスをstatic finalなフィールドで生成

2.コンストラクタをprivateにする

3.Serializableをimplementsする

4.serialVersionUIDを定義

5.インスタンスフィールドをtransientにする

6.唯一のインスタンスを返すreadResolve()メソッドを定義

Singletonのサンプル

```java
/**
 * Singletonのサンプル.
 *
 * @author tmaeda
 */
public class Singleton implements Serializable {

    /** serialVersionUID. */
    private static final long serialVersionUID = 1L;

    /** 唯一のインスタンス. */
    public static final Singleton INSTANCE = new Singleton();

    /** インスタンスフィールドにはtransient修飾子. */
    transient private String str;

    /**
     * 外部からアクセスできないよう、コンストラクタはprivate.
     */
    private Singleton(){};

    /**
     * ディシアライズによるインスタンス生成を防御.
     */
    private Object readResolve() throws ObjectStreamException {
        return INSTANCE;
    }
```

enumを使用したSingletonのサンプル

Singletonインスタンスを示すenumを宣言するだけ！

```
3  /**
4   * enumのSingletonサンプル.
5   *
6   * @author tmaeda
7   */
8  public enum EnumSingleton {
9
10     /** 唯一のインスタンスを示す列挙子. */
11     INSTANCE,
12     ;
13     private String str;
```

SingletonをenumでかくことのメリットSingletonをenumで書くことのメリット

従来のSingletonに比べ、考慮しなければならないことがほとんどない。

	従来のSingleton	enumによるSingleton
自身のインスタンスをstatic finalなフィールドに定義	要	不要
コンストラクタをprivateにする	要	不要
SerializableをimplementsSerializableをimplements	要	不要
serialVersionUIDを定義	要	不要
インスタンスフィールドにtransient	要	不要
readResolveメソッドを定義	要	不要

enumのまとめ

- public static finalな定数クラスはenumにしよう！
- Javaのenumは多態性を表現できる！
- Singletonはenumで書くのがベスト・プラクティス！

Reflection

レジュメ

1. Reflectionのおさらい
2. 動的にインスタンスを生成するファクトリを作る
3. インスタンス生成時のパフォーマンス
4. テストで使える汎用的なtoStringメソッドを作る
5. まとめ

Reflectionのおさらい
-Reflectionで出来ること-

- プログラムの内部構成の情報取得
  - 型の判別
  - 修飾子、アノテーションの情報取得
  - フィールド、メソッド、コンストラクタの情報取得
  - クラス名、パッケージ名の取得
- 動的実行
  - 動的なインスタンス生成
  - 動的なメソッド呼び出し
  - 動的プロキシ生成
- etc・・・

Reflectionのおさらい
-Reflectionのサンプル-

java.lang.Enumクラスのメソッドを全て取得して表示する

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

    try {
        Class<?> clz = Class.forName("java.lang.Enum");
        Method[] mAry = clz.getDeclaredMethods();
        for (Method m : mAry) {
            System.out.println(m.toString());
        }

    } catch (Throwable e) {
        e.printStackTrace();
    }
}
```

```
public final java.lang.String java.lang.Enum.name()
protected final void java.lang.Enum.finalize()
public final int java.lang.Enum.hashCode()
protected final java.lang.Object java.lang.Enum.clone() throws java.lang.CloneNotSupportedException
public final int java.lang.Enum.compareTo(java.lang.Enum)
public int java.lang.Enum.compareTo(java.lang.Object)
public final boolean java.lang.Enum.equals(java.lang.Object)
public java.lang.String java.lang.Enum.toString()
public static java.lang.Enum java.lang.Enum.valueOf(java.lang.Class,java.lang.String)
public final java.lang.Class java.lang.Enum.getDeclaringClass()
private void java.lang.Enum.readObject(java.io.ObjectInputStream) throws java.io.IOException,java.lang.ClassNo
public final int java.lang.Enum.ordinal()
private void java.lang.Enum.readObjectNoData() throws java.io.ObjectStreamException
```

動的にインスタンスを生成するファクトリ

Classクラスのnewinstanceメソッドで引数で渡された型のインスタンスを動的に生成

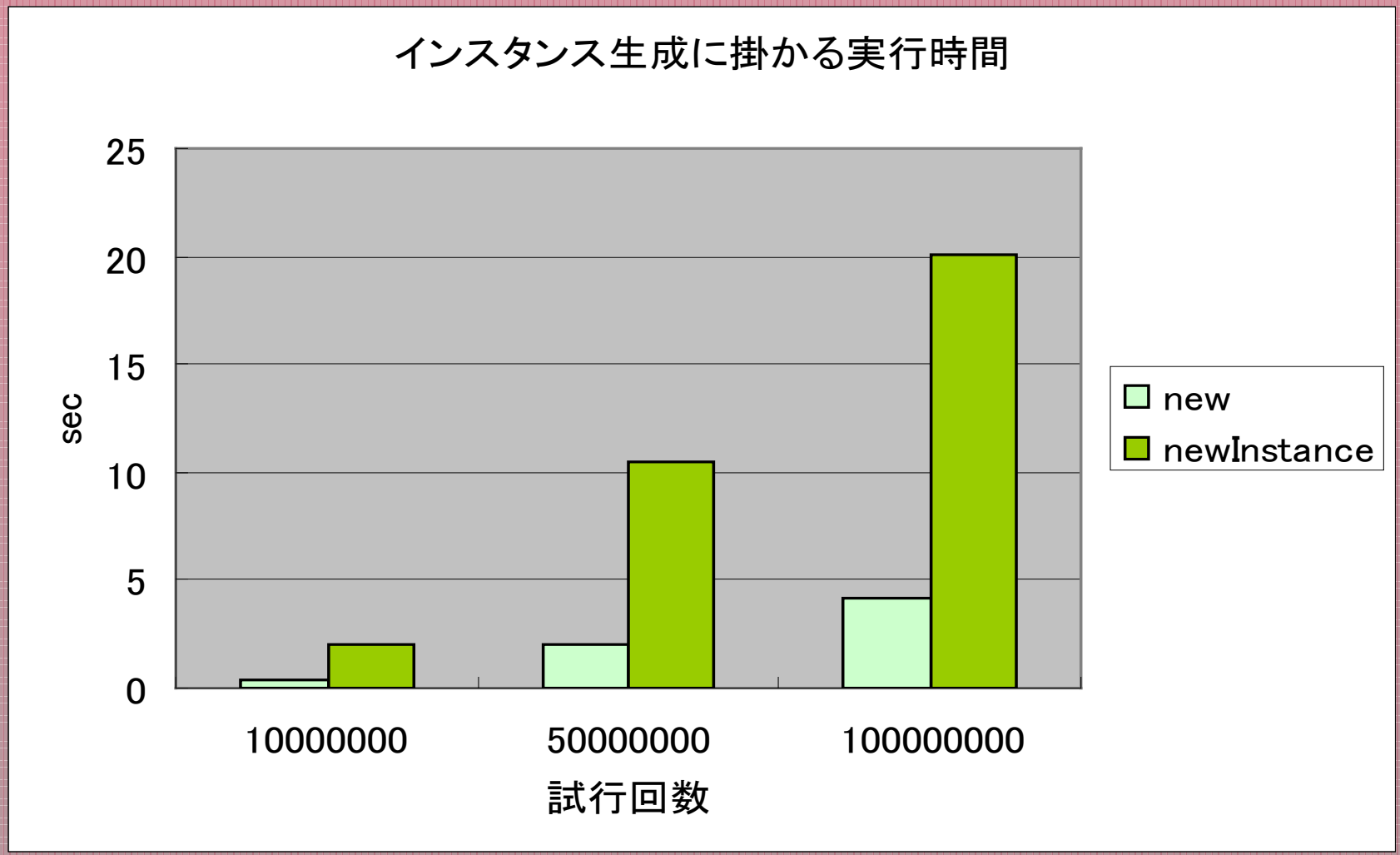
```
6  public class AnimalFactory {
7
8      public static final AnimalFactory INSTANCE = new AnimalFactory();
9
10     private static Map<Class<? extends Animal>, Animal> animalMap
11                     = new HashMap<Class<? extends Animal>, Animal>();
12
13     private AnimalFactory(){}
14
15     public Animal getAnimal(Class<? extends Animal> animalType) {
16         try {
17             Animal animal;
18             if ((animal = animalMap.get(animalType)) == null) {
19                 animal = animalType.newInstance();
20                 animalMap.put(animalType, animal);
21             }
22             return animal;
23         } catch (Exception e) {
24             e.printStackTrace();
25             throw new RuntimeException();
26         }
27     }
28 }
```

インスタンス生成時のパフォーマンス比較
-newとnewInstance-

```
7     public static void main(String[] args) throws Throwable {
8
9         long start = System.currentTimeMillis();
10        Class<?> clz = Class.forName("java.util.ArrayList");
11
12        for (int i=0; i < 100000000; i++) {
13    //          new ArrayList<Object>();
14            clz.newInstance();
15        }
16
17        System.out.println(System.currentTimeMillis() - start);
18    }
```

Windows XP
Intel® Core 2 Duo T7250
1.99GHz, 1.99GB RAM
JRE build 1.6.0_07-b06

インスタンス生成時のパフォーマンス比較
-newとnewInstance-



汎用的なtoStringメソッド

getDeclaredFieldsメソッドを使用し、そのインスタンスが保持するすべてのフィールドを取得している。

```
6  public class ToStringUtil {
7
8      public static <T> String toString(T instance) {
9
10         StringBuilder builder = new StringBuilder();
11         Class<?> clz = instance.getClass();
12         builder.append(clz.getName()).append("[");
13
14         Field[] fields = clz.getDeclaredFields();
15         for (Field field : fields) {
16             field.setAccessible(true);
17
18             String fieldName = field.getName();
19             Object fieldValue = null;
20
21             try {
22                 fieldValue = field.get(instance);
23             } catch (IllegalArgumentException e) {
24                 fieldValue = "*IllegaArgumentException*";
25             } catch (IllegalAccessException e) {
26                 fieldValue = "*IllegalAccessException*";
27             }
28             builder.append(fieldName + "=" + fieldValue + ", ");
29         }
30         return builder.append("]").toString();
31     }
```

まとめ

＜メリット＞

- デバッグ用、テスト用のツールとして使うと便利
- 動的にいろいろ出来るので、使いようによってはコーディングを量を減らすことができる

＜デメリット＞

- 設計を破壊してしまう可能性
- パフォーマンス・コストは良くない
- (IDEなどで)ソースが追いにくい

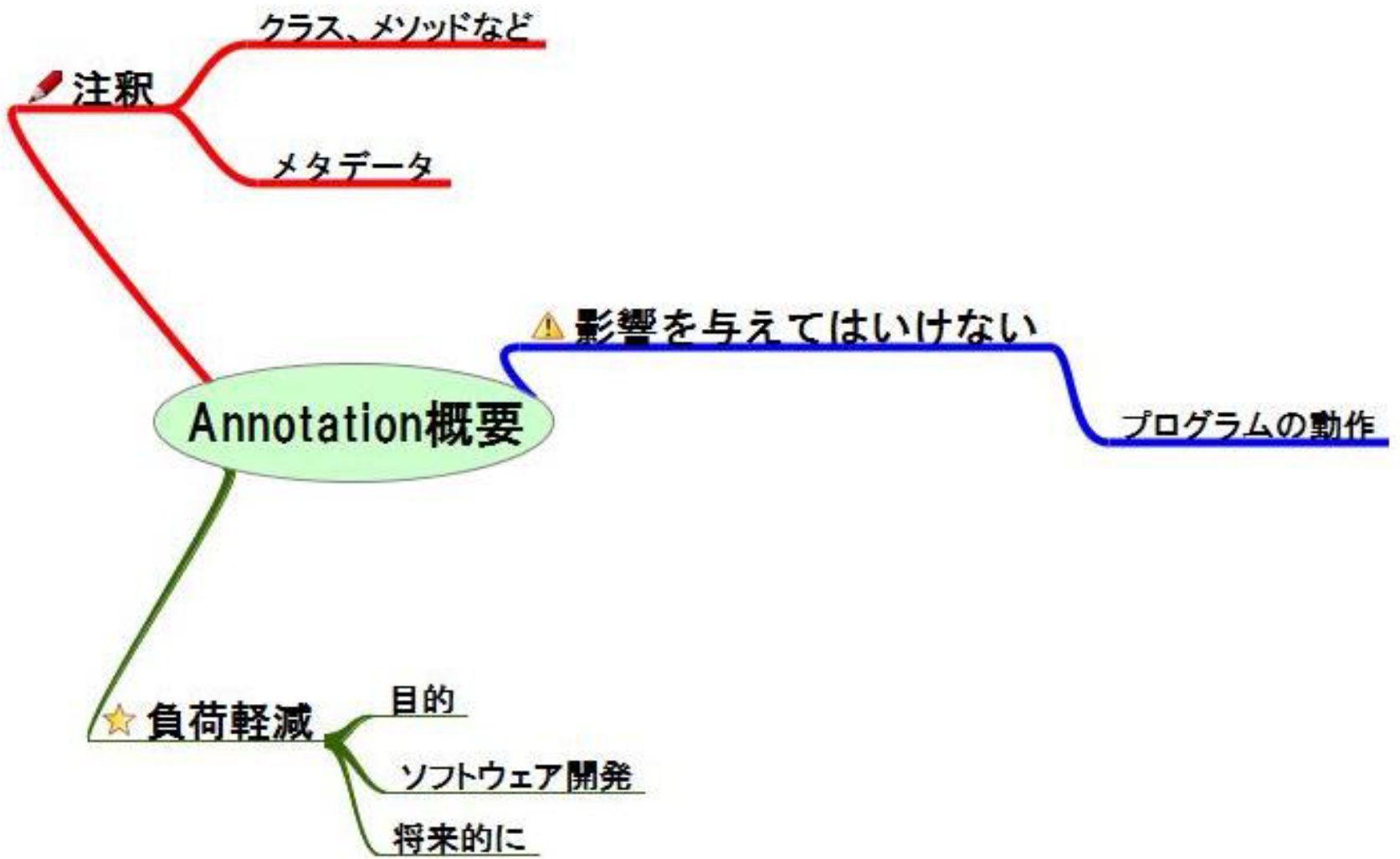
Reflectionは正しく使えば強力なツールになります。
特性を理解し、しっかりと検討したうえで有効に使いましょう。

Annotation

Annotation
-レジュメ-

- Annotationのおさらい
- Annotationでビルドチェック
- Annotationとリソース
- Annotation マーキング
- まとめ

Annotationのおさらい
-概要-



Annotationのおさらい
-分類(1)-

- マーカAnnotation

```
/**
 * サンプルです
 */
@Singleton
@ToDo("サンプルです")
@ResourceBundle(pkg="samplecollection", file="Sample.properties")
public class Sample {

    private static Sample _instance;

    public static Sample getInstance() {

        return _instance;
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
    }


    @ResourceElement(key="strValue", value="test")
    private String _strValue;

}
```

Annotationのおさらい
-分類(2)-

- 単一値Annotation

```
/**
 * サンプルです
 */
@Singleton
@ToDo("サンプルです")
@ResourceBundle(pkg="samplecollection", file="Sample.properties")
public class Sample {

    private static Sample _instance;

    public static Sample getInstance() {

        return _instance;
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
    }


    @ResourceElement(key="strValue", value="test")
    private String _strValue;

}
```

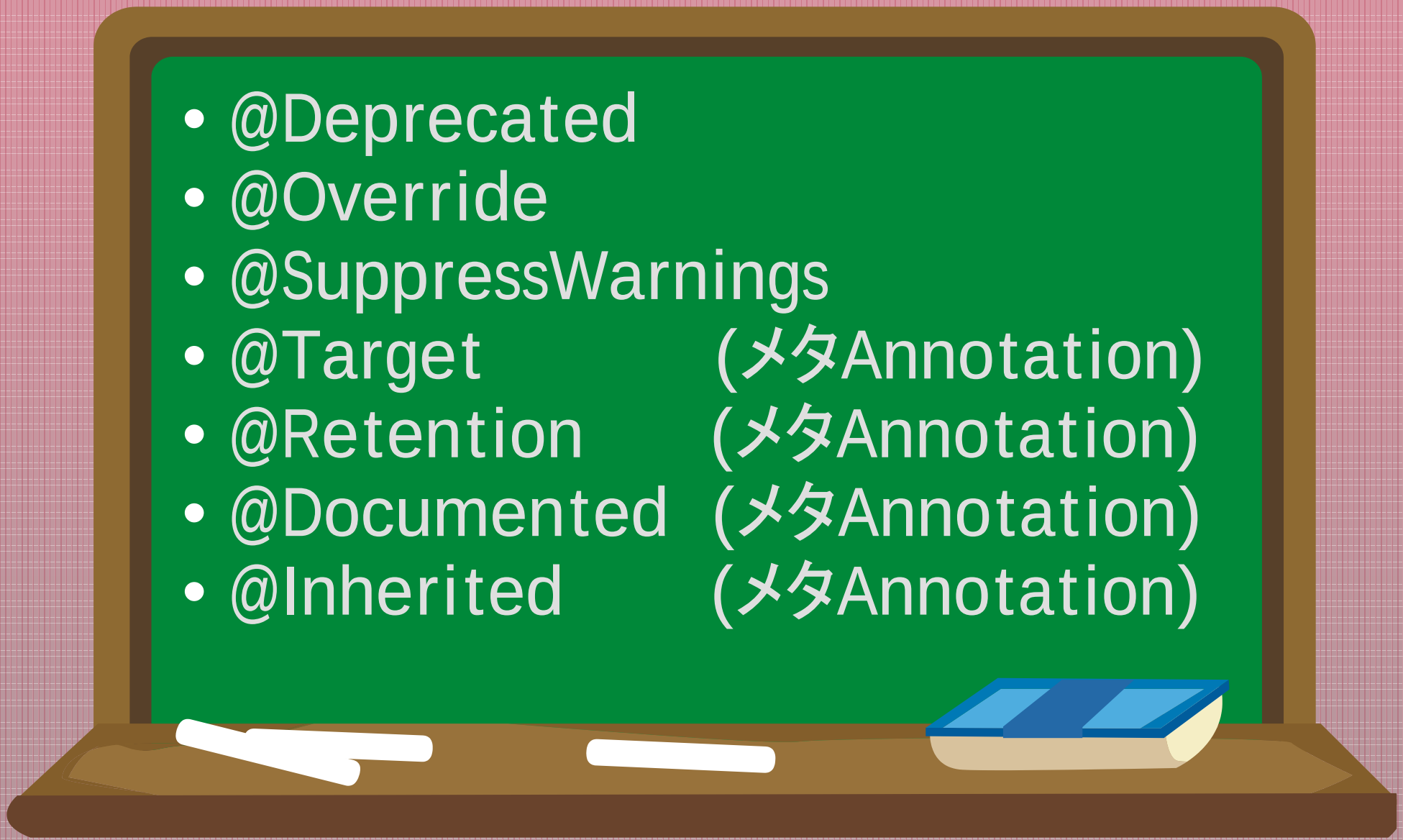
Annotationのおさらい
-分類(3)-

- フルAnnotation

```
/**
 * サンプルです
 */
@Singleton
@ToDo("サンプルです")
@ResourceBundle(pkg="samplecollection", file="Sample.properties")
public class Sample {

    private static Sample _instance;

    public static Sample getInstance() {

        return _instance;
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
    }


    @ResourceElement(key="strValue", value="test")
    private String _strValue;

}
```

Annotationのおさらい

-Java標準のAnnotation-



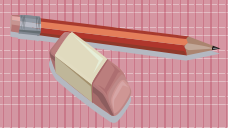
Annotationのおさらい
-独自のJavaアノテーション定義-

```
@Target(ElementType.XXX)
@Retention(RetentionPolicy.XXX)
public @interface Annotation名 {

    要素の型 要素名() default デフォルト値;
}
```

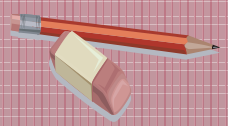
Annotationのおさらい
-Annotationを処理するツール-

- Javadoc
- apt(Annotation Processing Tool)コマンド(javacでも可)
- リフレクション

Annotationのおさらい
-なぜAnnotationを使用する？-



ソースに表現できない情報を提供したい



外部からAnnotationを解釈する必要がある

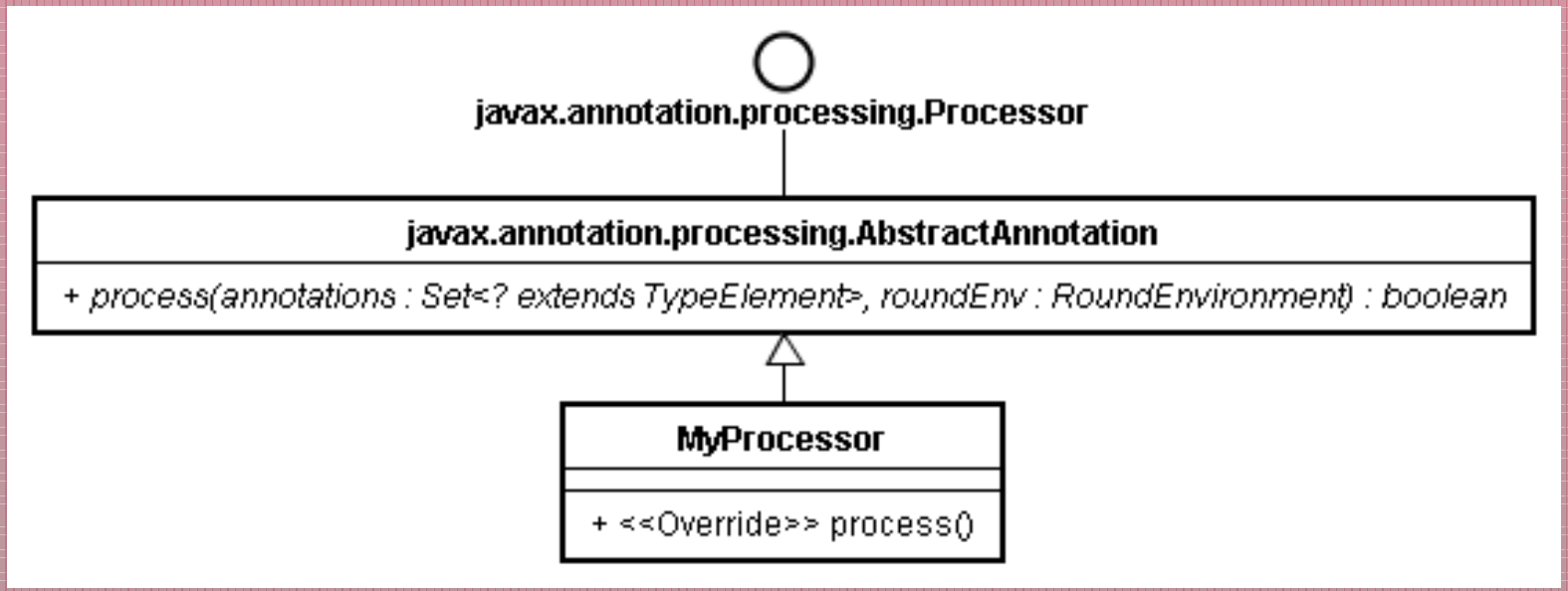
Annotationでビルドチェック

@ToDoと@Singleton

Annotationでビルドチェック
-Annotationを使用してビルドチェックするには-

- javax.annotation.processing.Processorを使用する

実際にはjavax.annotation.processing.AbstractProcessorを使用する。



- JavacコマンドにProcessorを渡す

```
C:¥>java -cp EsAnnotation.jar -processor jp.co.esplanning.annotation.processor.MyProcessor Sample.java
```

Annotationでビルドチェック
-@ToDo ～ 概要-

ビルド時に@ToDoが残っている場合、
その内容をWarningとして出力する。

@ToDo定義

```
1  package jp.co.esplanning.annotation;
2
3  import java.lang.annotation.Retention;
5
6  /**
7   * ToDoアノテーションです.最終的に解消しなければビルド時に警告を出します.
8   * @author honda
9   */
10 @Retention(RetentionPolicy.SOURCE)
11 public @interface ToDo {
12
13     public String value() default "ToDo箇所";
14 }
```

Annotationでビルドチェック
-@ToDo ～ ToDoProcessorの実装-

```java
/**
 * 「@ToDo」アノテーションを検知し、解消されていない場合は警告を出します.
 * @author honda
 */
@SupportedSourceVersion(SourceVersion.RELEASE_6)
@SupportedAnnotationTypes({"jp.co.esplanning.annotation.ToDo"})
public class ToDoProcessor extends AbstractProcessor {

    @Override
    public boolean process(Set<? extends TypeElement> annotations,
            RoundEnvironment roundEnv) {

        Messager msgr = super.processingEnv.getMessager();

        for (TypeElement annotation : annotations) {

            Set<? extends Element> elements = roundEnv.getElementsAnnotatedWith( annotation );
            for (Element element : elements) {

                ToDo todo = element.getAnnotation( ToDo.class );

                if (todo != null) {

                    msgr.printMessage( Kind.WARNING,
                            "ToDoが未解消です¥n¥t" + todo.value(), element );
                }
            }
        }

        return false;
    }
}
```

Annotationでビルドチェック
-@ToDo ～ Sample作成／実行-

```
5  public class Sample extends JFrame {
6
7      @ToDo("処理を書いて下さい。")
8      public static void main(String[] argv) {
9      }
10 }
```

```
C:¥MyWork¥MyLibrary¥Sample Collection¥Java¥Annotation>javac -cp EsAnnotation.jar Sample.java
Sample.java:8: 警告:ToDoが未解消です
        処理を書いて下さい。
        public static void main(String[] argv) {
                           ^

C:¥MyWork¥MyLibrary¥Sample Collection¥Java¥Annotation>
```

Annotationでビルドチェック
-@ToDo ～ 修正して実行-

```
5 public class Sample extends JFrame {↵
6 ↵
^
7 ^    public static void main(String[] argv) {↵
8 ^    }↵
9 }↵
```

```
C:¥MyWork¥MyLibrary¥Sample Collection¥Java¥Annotation>javac -cp EsAnnotation.jar Sample.java

C:¥MyWork¥MyLibrary¥Sample Collection¥Java¥Annotation>
```

Annotationでビルドチェック
-javacへの自作Processor指定-

- 「-processor XXXProcessor」

- 「META-INF/services」へのファイル格納

XXX.jar

META-INF/service

ファイル：javax.annotation.processing.Processor

```
1 jp.co.esplanning.annotation.processor.ToDoProcessor
2 jp.co.esplanning.annotation.processor.SingletonProcessor
3 jp.co.esplanning.annotation.processor.ResourceBundleProcessor
```

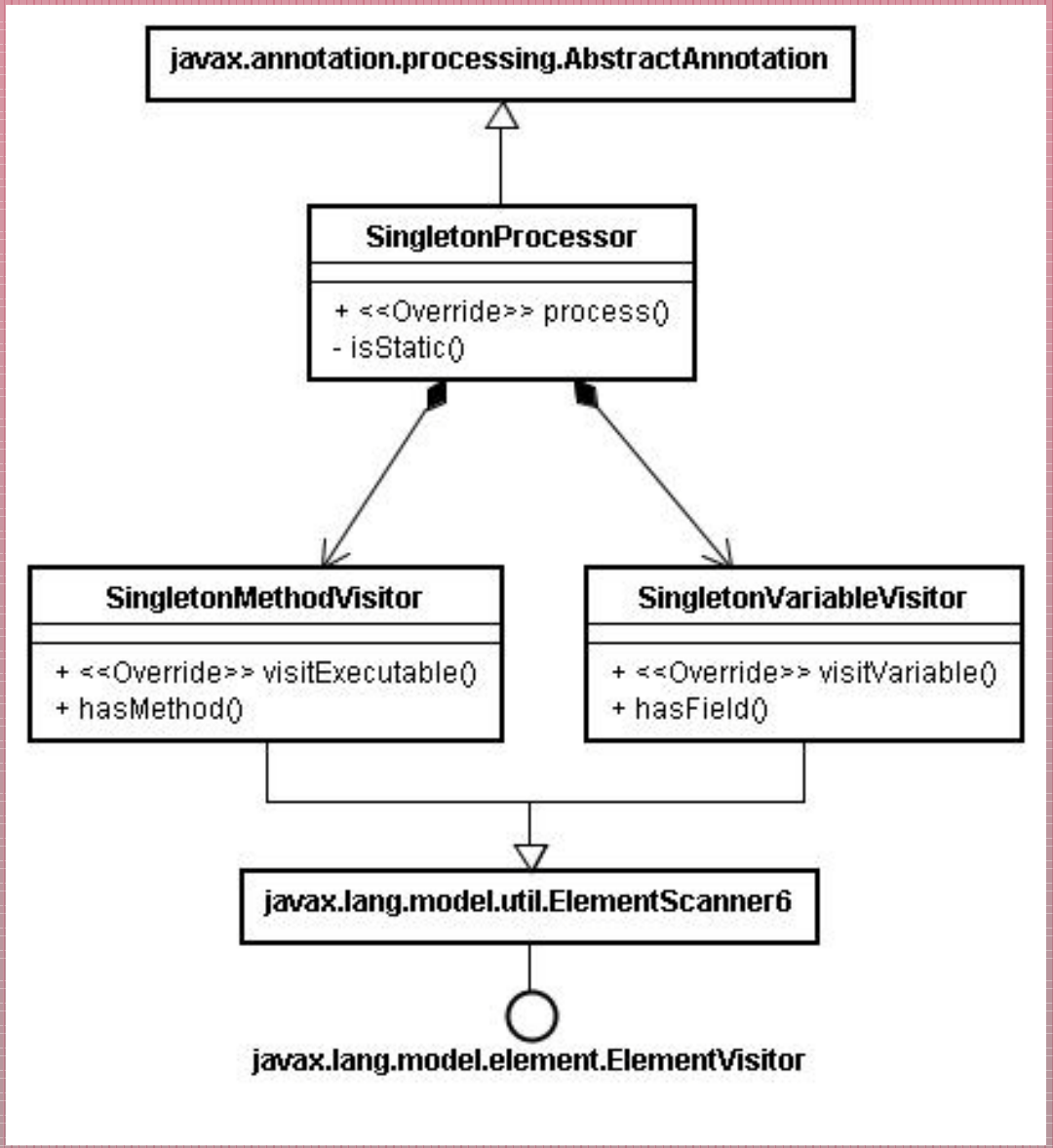
Annotationでビルドチェック
-@Singleton ～ 概要-

- static変数チェック(自身の型)
- staticメソッドチェック(自身の型が戻り値)

@Singleton定義

```
1  package jp.co.esplanning.annotation.designpattern;
2
3  import java.lang.annotation.Documented;
8
9  /**
10  * デザインパターンSingleton(GOF, 分類：生成)を表わすマーカアノテーション
11  * @author honda
12  */
13 @Documented
14 @Retention (RetentionPolicy.SOURCE)
15 @Target (ElementType.TYPE)
16 public @interface Singleton {
17 }
```

Annotationでビルドチェック
-@Singleton ～ SingletonProcessor-



```java
/**
 * Singletonアノテーションを処理するアノテーションプロセッサ.<br>
 * クラス内にそのクラス自信のstaticなFieldとそれを取得するMethodを持っていることチェックする
 * また、コンストラクタがprivateで隠蔽されている事
 */
@SupportedSourceVersion(SourceVersion.RELEASE_6)
@SupportedAnnotationTypes({"jp.co.esplanning.annotation.designpattern.Singleton"})
public class SingletonProcessor extends AbstractProcessor {

    @Override
    public boolean process(Set<? extends TypeElement> annotations,
            RoundEnvironment roundEnvironment) {

        // annotationが使用されている要素を取得
        for (Element element : roundEnvironment.getRootElements()) {

            // ルート要素がクラスでありかつ、
            // Singletonアノテーションで修飾されているかどうかを探索
            if (element.getKind() == ElementKind.CLASS &&
                    element.getAnnotation( Singleton.class ) != null) {

                SingletonVariableVisitor field =
                    new SingletonVariableVisitor( element.getSimpleName().toString() );
                SingletonMethodVisitor method =
                    new SingletonMethodVisitor( element.getSimpleName().toString() );

                field.scan( element );
                method.scan( element );

                if (!field.hasField() || !method.hasMethod()) {

                    // Elementを渡すとソース名と行番号が表示される。
                    Messager messager = super.processingEnv.getMessager();
                    messager.printMessage( javax.tools.Diagnostic.Kind.WARNING,
                            "Singletonにはstaticなフィールド、メソッドが必要です",
                            element );
                }
            }
        }

        return true;
    }
```

```
64    /**
65     * 指定要素がstaticかどうかの判定
66     * @param e 要素
67     * @return trueの場合はstatic
68     */
69    private static boolean isStatic(Element e) {
70
71        for (Modifier mod : e.getModifiers()) {
72
73            if (mod == Modifier.STATIC) {
74
75                return true;
76            }
77        }
78
79        return false;
80    }
```

```
 83     * Singletonのフィールドがあるかどうかを判定します.
 84     */
 85    private class SingletonVariableVisitor extends ElementScanner6<Void, Void> {
 86
 87        private boolean _hasField;
 88        private String _className;
 89
 90        public SingletonVariableVisitor(String className) {
 91
 92            _hasField = false;
 93            _className = className;
 94        }
 95
 96        public boolean hasField() {
 97
 98            return _hasField;
 99        }
100
101        @Override
102        public Void visitVariable(VariableElement e, Void p) {
103
104            // staticフィールドでありかつ、
105            // 自身の型があるか
106            if (SingletonProcessor.isStatic( e ) &&
107                    _className.equals( e.asType().toString() )) {
108
109                _hasField = true;
110            }
111
112            return super.visitVariable( e, p );
113        }
114    }
```

```
126    /**
127     * Singletonのメソッドがあるかどうかを判定します.
128     */
129    private class SingletonMethodVisitor extends ElementScanner6<Void, Void> {
130
131        private boolean _hasMethod;
132        private String _className;
133
134        public SingletonMethodVisitor(String className) {
135
136            _hasMethod = false;
137            _className = className;
138        }
139
140        public boolean hasMethod() {
141
142            return _hasMethod;
143        }
144
145        @Override
146        public Void visitExecutable(ExecutableElement e, Void p) {
147
148            // staticメソッドでありかつ、
149            // 自身の型が戻り値であるか
150            if (SingletonProcessor.isStatic( e ) &&
151                    _className.equals( getClassName( e.getReturnType().toString() ) ))
152
153                _hasMethod = true;
154            }
155
156            return super.visitExecutable( e, p );
157        }
158    }
159 }
```

Annotationでビルドチェック
-@Singleton ～ Sample作成／実行-

```
 5 @Singleton↵
 6 public class Sample extends JFrame {↵
 7 ↵
 8 ^    public static void main(String[] argv) {↵
 9 ^    }↵
10 }↵
```

```
C:¥MyWork¥MyLibrary¥Sample Collection¥Java¥Annotation>javac -cp EsAnnotation.jar Sample.java
Sample.java:6: 警告:Singletonにはstaticなフィールド、メソッドが必要です
public class Sample extends JFrame {
       ^

C:¥MyWork¥MyLibrary¥Sample Collection¥Java¥Annotation>
```

Annotationでビルドチェック
-@Singleton ～ 修正して実行-

staticのフィールドとメソッドを追加して実行する

```
 5 @Singleton
 6 public class Sample extends JFrame {
 7
 8     private static Sample _instance;
 9
10     public static Sample getInstance() {
11
12         return _instance;
13     }
14
15     public static void main(String[] argv) {
16     }
17 }
```

```
C:¥MyWork¥MyLibrary¥Sample Collection¥Java¥Annotation>javac -cp EsAnnotation.jar Sample.java

C:¥MyWork¥MyLibrary¥Sample Collection¥Java¥Annotation>
```

Annotationとリソース

@ResourceBundle

Annotationとリソース
-@ResourceBundle ～ 概要-

- リソースファイルの自動出力
(使用：@ResourceBundle)

- 「キー=値」の自動出力
(使用：@ResourceElement)

- リフレクションでリソースファイルの読み込み
(使用：@ResourceBundle、@ResourceElement)

Annotationとリソース
-@ResourceBundle ～ 定義(1)-

@ResourceBundle定義

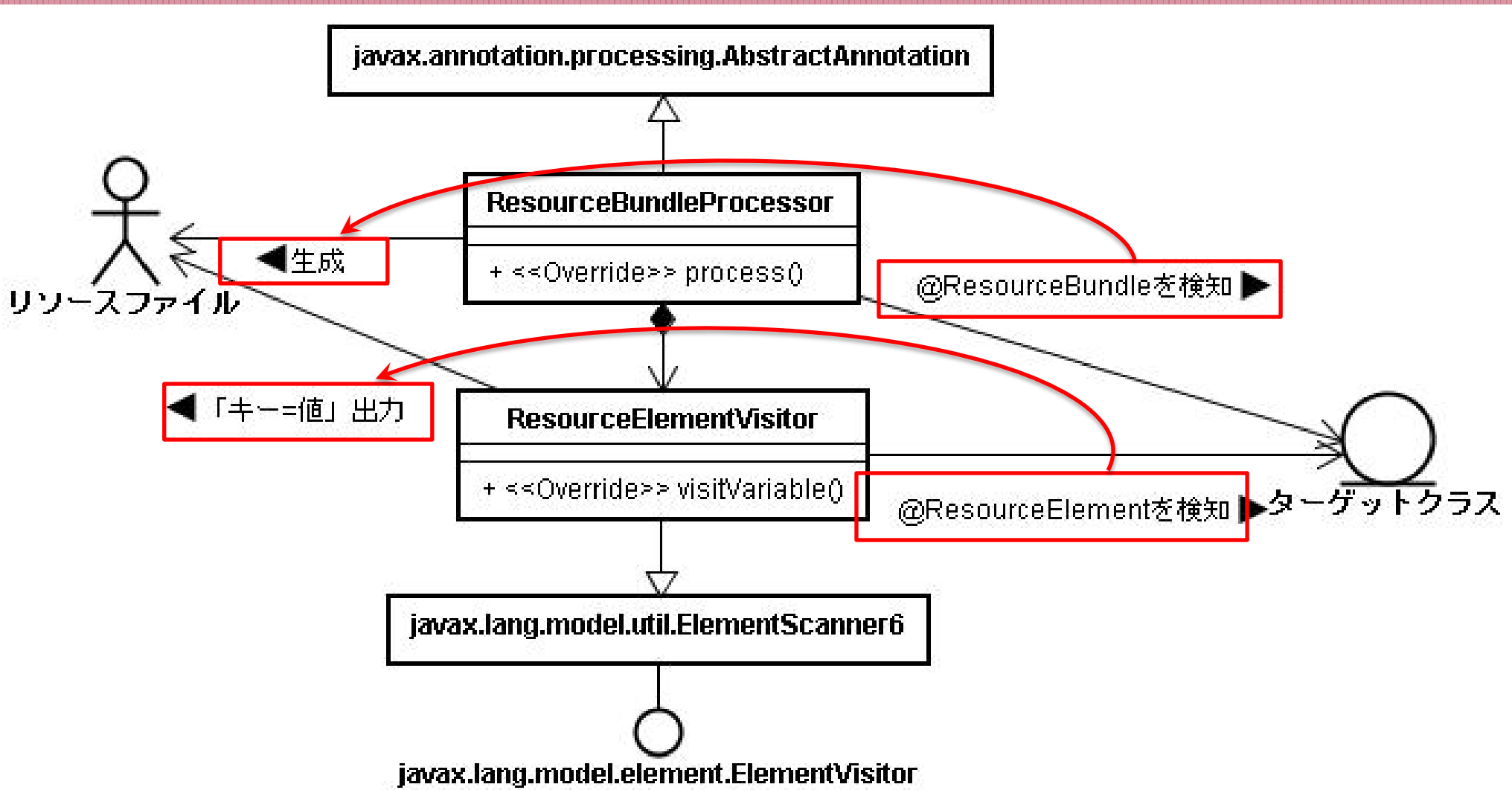
```
10 /**
11  * リソースファイルとバンドリングし、リソースファイルを読込、書出、生成する為のアノテーション.<br>
12  * @author honda
13  * @see jp.co.esplannning.annotation.processor.ResourceBundleAnnotationProcessor
14  */
15 @Retention(RetentionPolicy.RUNTIME)
16 @Target(value={ElementType.TYPE})
17 @Documented
18 public @interface ResourceBundle {
19
20     /** パッケージ名 */
21     String pkg() default "";
22
23     /** ファイル名 */
24     String file() default "";
25 }
```

Annotationとリソース
-@ResourceBundle ～ 定義(2)-

@ResourceElement定義

```java
/**
 * リソースファイルとバンドリングし、リソースファイルを読込、書出、生成する為のアノテーション.<br>
 * @author honda
 * @see jp.co.esplanning.annotation.ResourceBundle
 * @see jp.co.esplanning.annotation.processor.ResourceBundleAnnotationProcessor
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(value={ElementType.FIELD})
@Documented
public @interface ResourceElement {

    /** リソースのキー */
    String key();

    /** リソースの値 */
    String value();
}
```

Annotationとリソース
-@ResourceBundle ～ ResourceBundleProcessor-



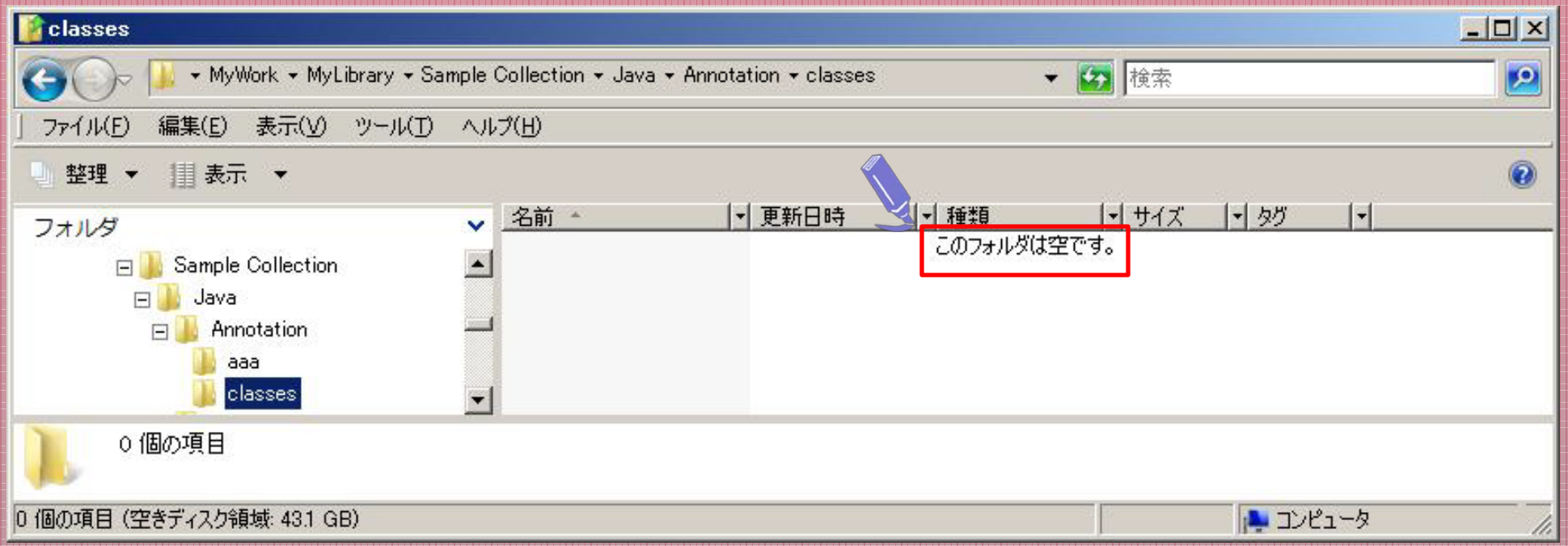
```java
/**
 * リソースファイルとマッピングし、リソースファイルを生成するプロセッサ<br>
 */
@SupportedSourceVersion(SourceVersion.RELEASE_6)
@SupportedAnnotationTypes({"jp.co.esplanning.annotation.ResourceBundle"})
public class ResourceBundleProcessor extends AbstractProcessor {

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {

        for (TypeElement annotation : annotations) {

            Set<? extends Element> elements = roundEnv.getElementsAnnotatedWith( annotation );

            for (Element element : elements) {

                // ResourceBundleアノテーションを取得する
                ResourceBundle rb = element.getAnnotation( ResourceBundle.class );

                Filer filer = super.processingEnv.getFiler();

                // リソースファイルを作成
                try {

                    FileObject file = filer.createResource( StandardLocation.CLASS_OUTPUT,
                            rb.pkg(), rb.file(), element );

                    Properties properties = new Properties();

                    ResourceElementVisitor visitor = new ResourceElementVisitor( properties );
                    visitor.scan( element );

                    properties.store( file.openOutputStream(), null );
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

        return true;
    }
```

```java
    /**
     * 「@ResourceElement」を解析するスキャナクラス
     * @author honda
     */
    private class ResourceElementVisitor extends ElementScanner6<Void, Void> {

        Properties _properties;

        public ResourceElementVisitor(Properties properties) {

            _properties = properties;
        }

        @Override
        public Void visitVariable(VariableElement e, Void p) {

            ResourceElement anno = e.getAnnotation( ResourceElement.class );
            if (anno != null) {

                _properties.put( anno.key(), anno.value() );
            }

            return super.visitVariable( e, p );
        }
    }
}
```

Annotationとリソース
-@ResourceBundle ～ Sampleを作成-

```
1  package sample;
2
3  import javax.swing.*;
4  import jp.co.esplanning.annotation.designpattern.*;
5  import jp.co.esplanning.annotation.*;
6
7  @ResourceBundle(pkg="sample", file="Sample.properties")
8  public class Sample extends JFrame {
9
10     public static void main(String[] argv) {
11     }
12
13     @ResourceElement(key="test_value", value="testです")
14     private String _value;
15 }
```

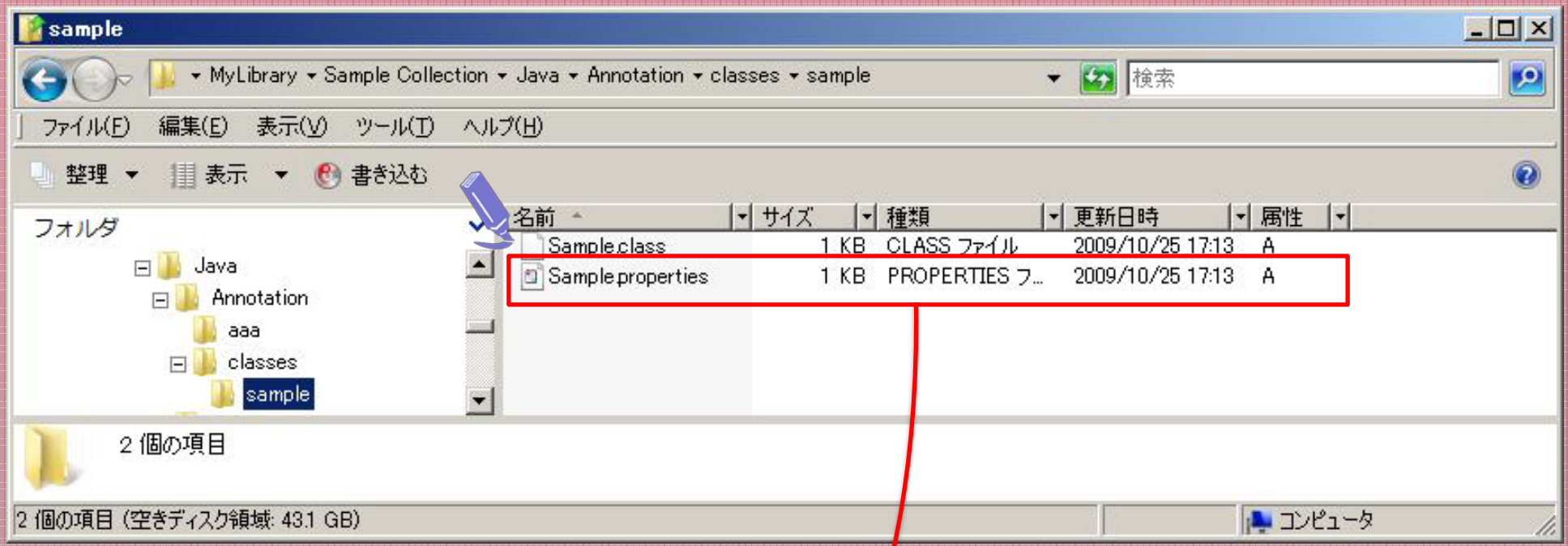
Annotationとリソース
-@ResourceBundle ～ 実行(1)-




```
C:¥MyWork¥MyLibrary¥Sample Collection¥Java¥Annotation>javac -cp EsAnnotation.jar -d classes Sample.java

C:¥MyWork¥MyLibrary¥Sample Collection¥Java¥Annotation>
```

javacを実行すると・・・

Annotationとリソース
-@ResourceBundle ～ 実行(2)-

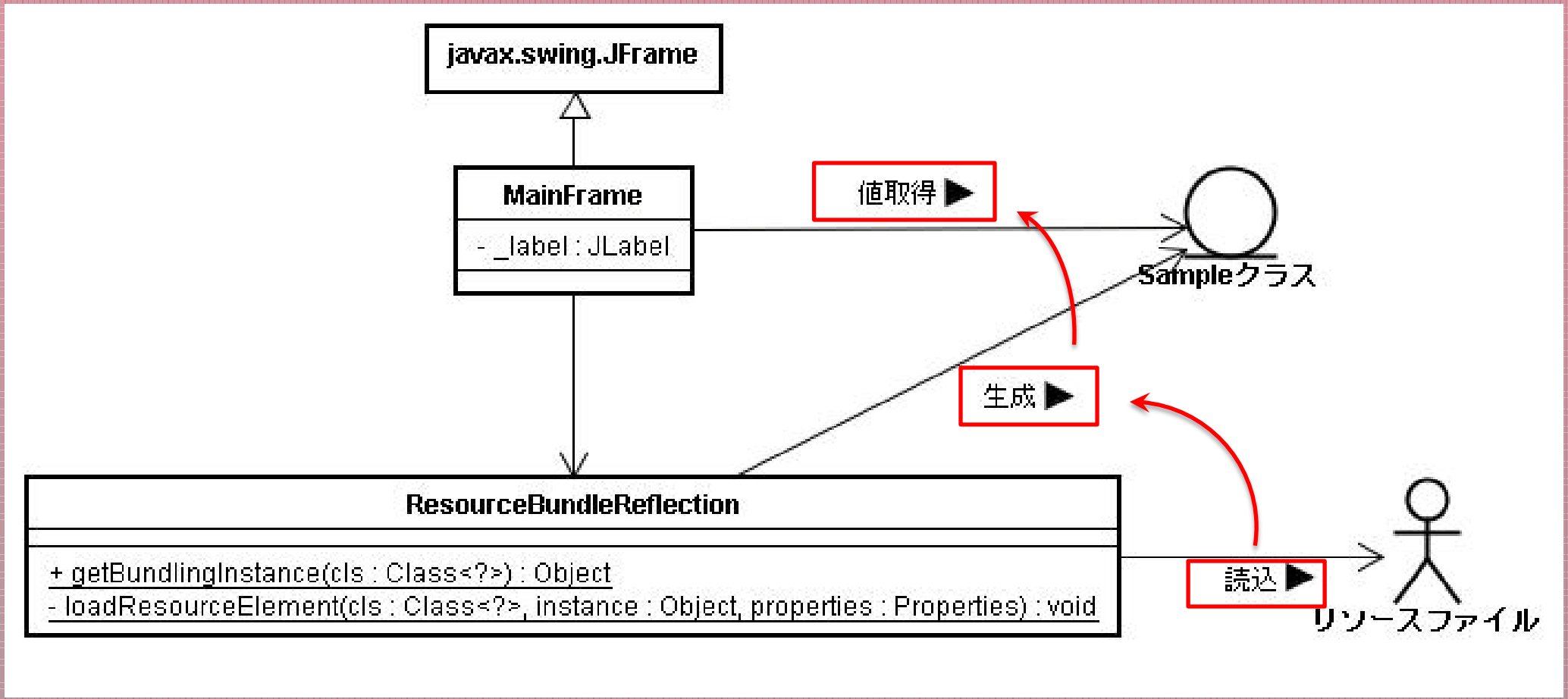


Sample.properties

```
1 #Sun Oct 25 17:13:48 JST 2009
2 test value=test\u3067\u3059
```

```
1 #Tue Oct 27 14:44:28 JST 2009
2 test value=testです
```

Annotationとリソース
-@ResourceBundle ～ リフレクション-



```java
/**
 * バントリングクラスのインスタンスの取得
 * @return インスタンス
 * @throws NoSuchMethodException
 * @throws SecurityException
 * @throws InvocationTargetException
 * @throws IllegalAccessException
 * @throws InstantiationException
 * @throws IllegalArgumentException
 * @throws IOException
 */
public static Object getBundlingInstance(Class<?> cls) throws SecurityException,
        NoSuchMethodException, IllegalArgumentException,
        InstantiationException, IllegalAccessException,
        InvocationTargetException, IOException {

    // 「@ResourceBundle」アノテーションの取得
    ResourceBundle rb = cls.getAnnotation( ResourceBundle.class );

    if (rb == null) {

        throw new IllegalArgumentException( "@ResourceBundleが定義されていません" );
    }

    // インスタンスの生成
    Object instance = cls.getConstructor( (Class[])null ).newInstance( (Object[])null );

    //リソースファイル名の取得
    String strResourceName = ResourceBundleReflection.getResourceName( cls, rb );

    // InputStreamを取得して、Propertiesを生成する
    Properties properties = new Properties();
    properties.load( cls.getResourceAsStream( strResourceName ) );

    // インスタンスにプロパティの値をロード
    ResourceBundleReflection.loadResourceElement( cls, instance, properties );

    return instance;
}
```

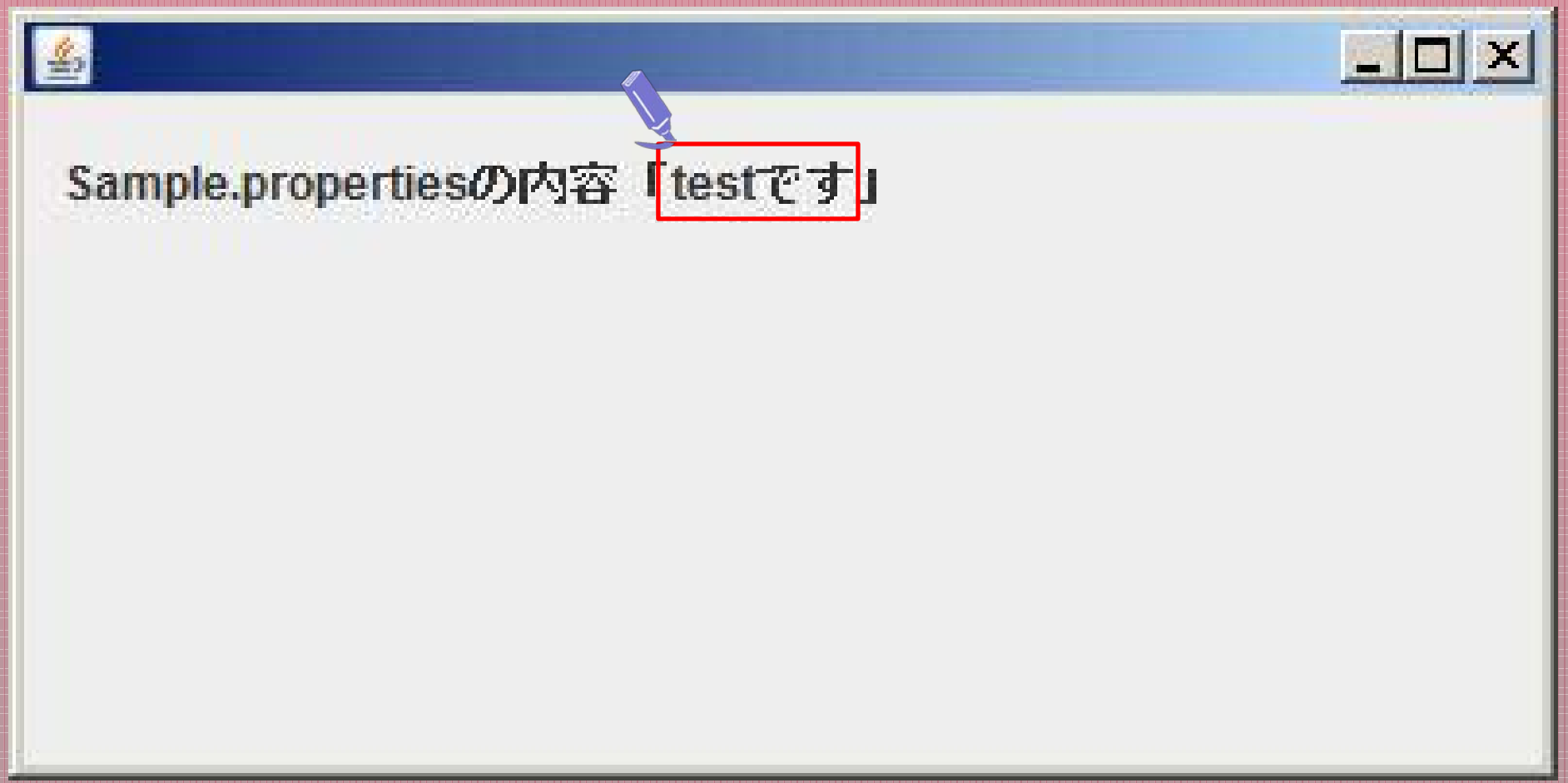
```java
    /**
     * 「@ResourceElement」をもとに指定のプロパティファイルから、インスタンスへ値を読み込む
     * @param instance
     * @param properties
     * @throws IllegalAccessException
     * @throws IllegalArgumentException
     */
    private static void loadResourceElement(Class<?> cls, Object instance, Properties properties)
            throws IllegalArgumentException, IllegalAccessException {

        Field[] fields = cls.getDeclaredFields();

        for (Field field : fields) {

            ResourceElement element = field.getAnnotation( ResourceElement.class );
            if (element != null) {

                field.setAccessible( true );
                field.set( instance, properties.getProperty( element.key(), element.value() ) );
                field.setAccessible( false );
            }
        }
    }
```

Annotationとリソース
-@ResourceBundle ～ Sampleの作成-

MainFrame抜粋

```
26    private void init() {
27
28        JLabel label = new JLabel();
29        label.setBounds( 10, 10, 300, 25 );
30        this.getContentPane().add( label );
31
32        try {
33
34            Sample obj = (Sample)ResourceBundleReflection.getBundlingInstance( Sample.class );
35            label.setText( "Sample.propertiesの内容「" + obj.getValue() + "」" );
36        } catch (Exception ex) {
37
38            ex.printStackTrace();
39        }
40    }
```

Annotationとリソース
-@ResourceBundle ～ Sample実行-



Annotation マーキング

@Signatureとプラグイン

Annotation マーキング
-@Signature ～ 概要-

- Jarファイル中の@Signatureを検出する
- @Signatureの内容を表示し承認を求める
- インスタンスを生成し、プラグインする

Annotation マーキング
-@Signature ～ 定義-

@Signature定義

```
10 /**
11  * 署名の為のアノテーション.<br>
12  * @author honda
13  */
14 @Retention(RetentionPolicy.RUNTIME)
15 @Target(value={ElementType.TYPE})
16 @Documented
17 @Inherited
18 public @interface Signature {
19
20     String value();
21 }
```

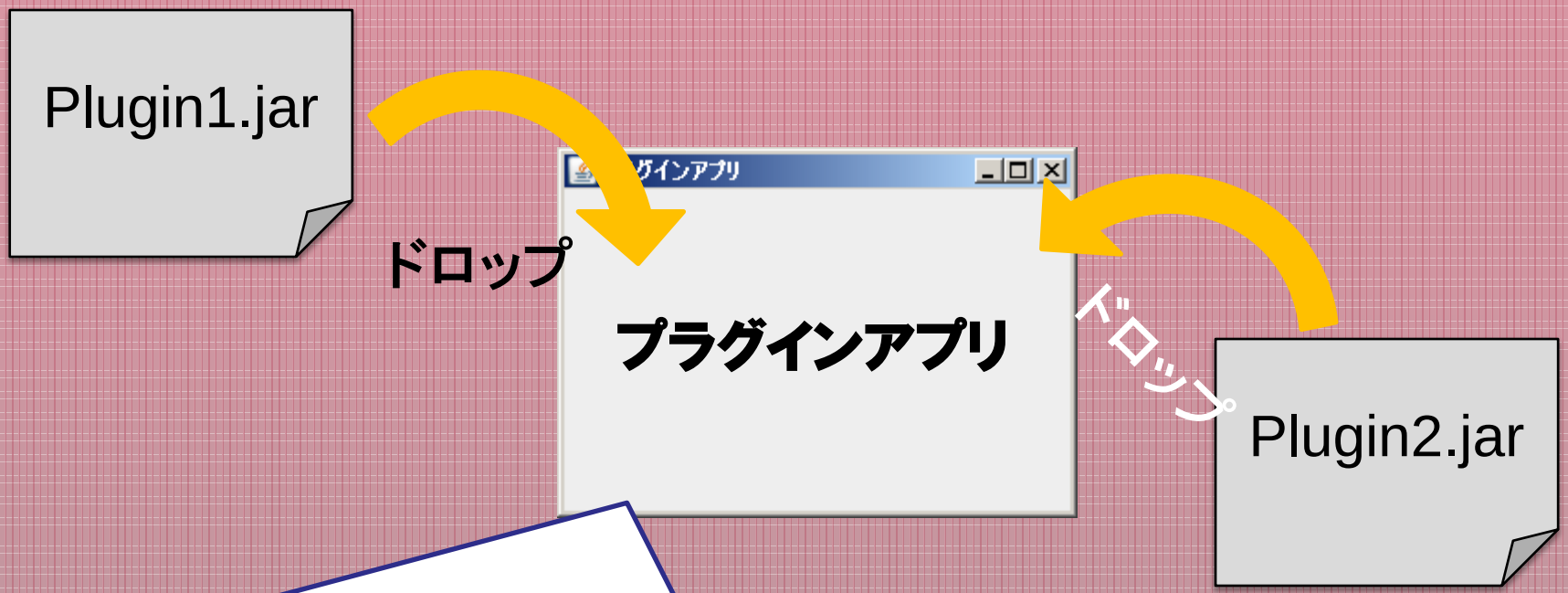
Annotation マーキング
-@Signature ～ プラグインクラス①-

```
1  package jp.co.esplanning.samplecollectiono.plugin;
2
3  import javax.swing.Icon;
10
11 /**
12  * プラグインサンプルです.
13  */
14 @Signature("2009 by es-planning Co,Ltd.")
15 @ResourceBundle
16 public class Plugin1 implements Iconable {
17
18     @ImageElement( "logo.gif" )
19     private Icon _image;
20
21     public Plugin1() {
22
23         try {
24             ResourceBundleReflection.applyIcon( this );
25         } catch (Exception e) {
26             e.printStackTrace();
27         }
28     }
29
30     @Override public String toString() {
31         return "Plugin1";
32     }
33
34     @Override public Icon getIcon() {
35         return _image;
36     }
37 }
```

Annotation マーキング
-@Signature ～ プラグインクラス②-

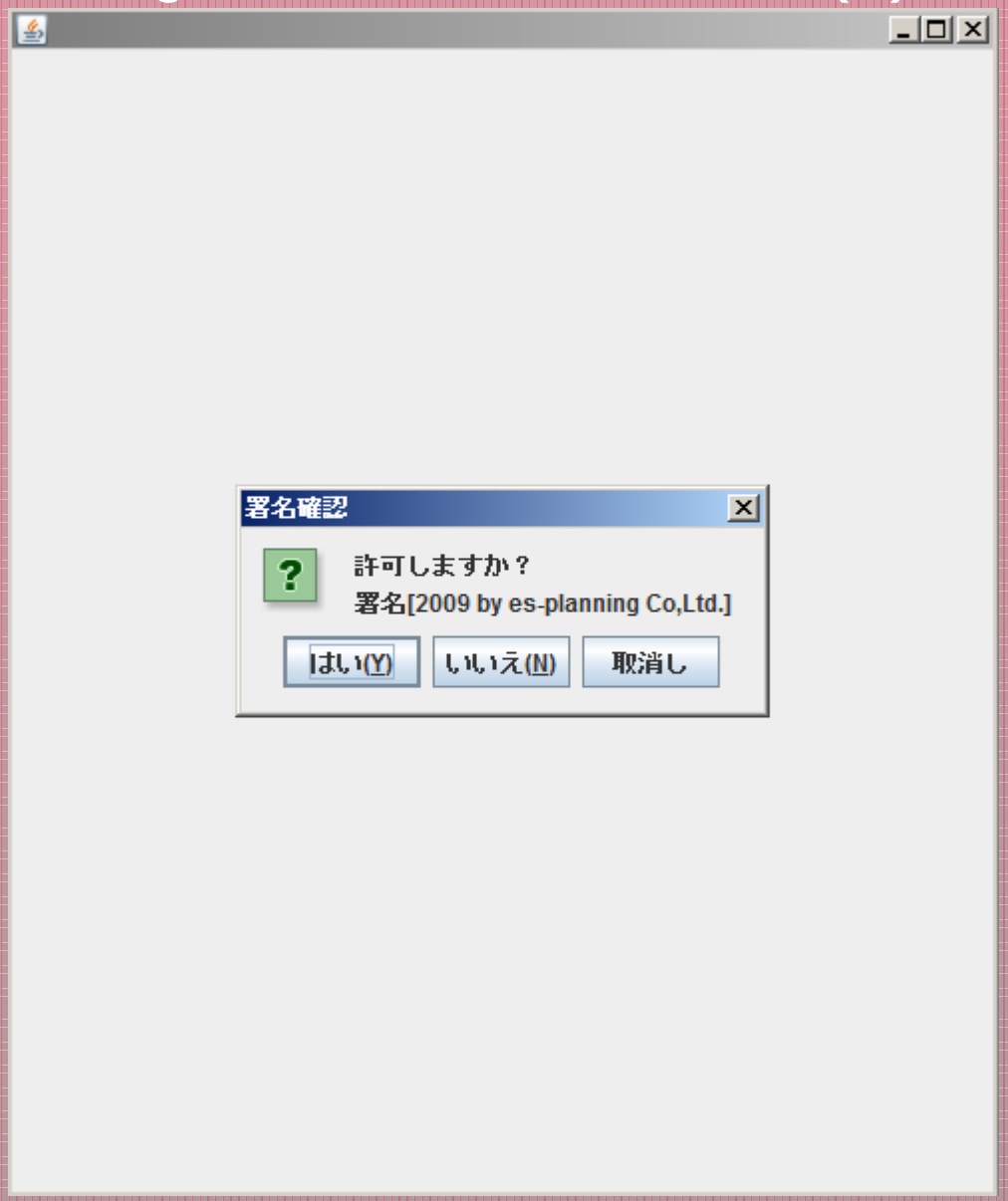
```java
package hoho;

import javax.swing.Icon;

/**
 * プラグインサンプルです.
 */
@Signature( "2009 by honda" )
@ResourceBundle
public class Test implements Iconable {

    @ImageElement( "Maximum.gif" )
    private Icon _image;

    public Test() {

        try {
            ResourceBundleReflection.applyIcon( this );
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    @Override public Icon getIcon() {
        return _image;
    }

    @Override public String toString() {
        return "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
    }
}
```

Annotation マーキング
-@Signature ～ プラグイン概要-



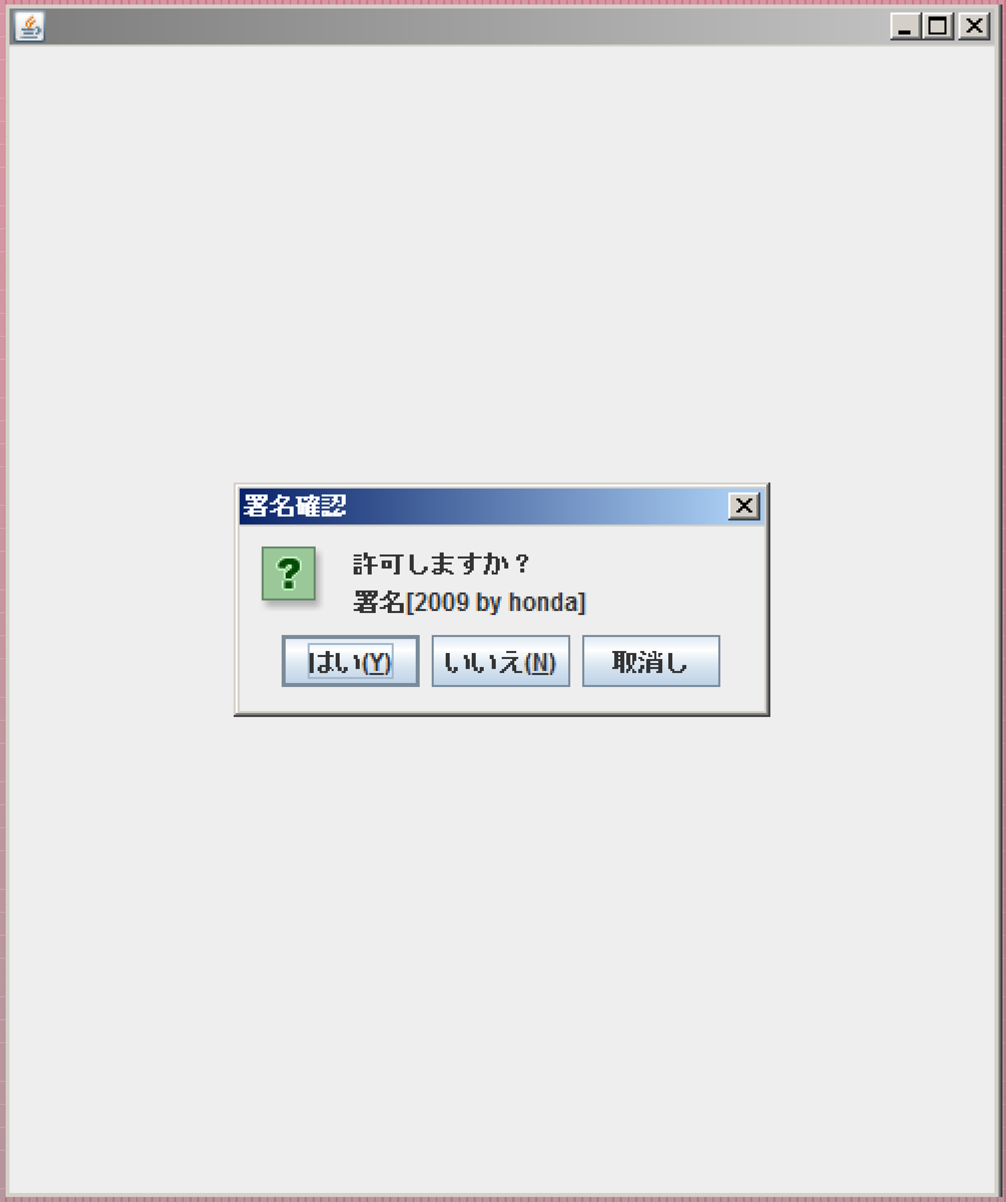
- ドロップされたjarから、@Signatureマークのあるクラスを検出する
- @Signatureのvalue要素から値を取得し、承認確認ダイアログを表示する
- アイコンと文字列を取得・表示する

Annotation マーキング
-@Signature ～ 実行結果(1)-



Annotation マーキング

-@Signature ～ 実行結果(2)-



Annotation まとめ

Annotationのまとめ

- ビルド時に固有のチェック機能を付けれる
- ビルド時にファイルの自動生成が出来る
- リフレクションを使用してデータの設定や取得が出来る
- リフレクションを使用してインスタンス生成前にクラスのAnnotation情報を入手出来る

さいごに

現在厳しい経済状況ではありますが、今回ご紹介した内容が、皆様の現場に役立ち、少しでも発展の手段となって頂ければ幸いです。

長い間ご静聴ありがとうございました。