Typesafe Enum 1.1 Component Specification

1. Design

This component encapsulates the well-known approach to mimicking C/C++ enum types in Java (see http://developer.java.sun.com/developer/Books/shiftintojava/page1.html#replaceenums). It provides a single abstract class, Enum, which can be extended to easily create correct enum-like types – Enum takes care of several details like compareTo implementing and also handling serialization correctly.

This component's enumeration types are superior to C/C++ enums since they are full typesafe classes and not integers, which also mean that enumeration values can have whatever methods and properties are needed.

Another goal of this component is to still offer performance comparable to int-based enum types; internally many common methods are implemented in terms of a unique int ordinal value, which allows high performance.

A final goal of this component is to mimic some properties of C/C++ enums, in particular, a notion of ordering; enumeration values produced using this component sort/compare in the same order that they are defined, as in C/C++.

This component also provides some utility methods for the user convenience, like getting enum value by enum's string representation, or getting enum value by ordinal value.

Updated in version 1.1

1. A new constructor is added to the Enum class to accept a Class instance, which is used as the type identifier for the enum value. This could allow subclasses to use internal sub-classing to differentiate behavior based on enumeration type while maintaining proper ordinal counts. To understand it more clearly, you could see demo and java.util.lang.Enum#getDeclaringClass(), which is introduced in java 1.5.
2. Another feature introduced in this version is a new utility method. This method could be used to get enum value by its name.

Backwards compatibility

The goal in requirement specification is to support 100% backwards compatibility. But there are still some places which can't be guaranteed for 100% compatibility.

1. A new private [declaringClass:Class] is added because of the new constructor. This makes the old serialized data can't be used by new version. Even if we try to add a custom readObject method to support reading old data, the serialization UID still prevents this process.
2. Because of the newly added [declaringClass] field, we will use [declaringClass] instead of [this.getClass()] in many methods, like readResolve()and compareTo() methods.

1.1 Design Patterns

Typesafe Enum Pattern -- The component itself focuses solely on supporting correct implementation of the Typesafe Enum pattern in Java.

1.2 Industry Standards

Java Serialization -- This component is designed to work correctly with the Java serialization mechanism.

1.3 Required Algorithms

1.3.1 Assign ordinal value for enum instances

Ordinals for instances of each different Enum subclass need to be assigned in sequence starting from 0. This can be implemented by just maintaining a counter that is incremented as each instance is instantiated, but a separate counter must be maintained for each different Enum subclass. In this design, a list will be used to hold all the instances for each Enum type, and the list will be mapped to the Enum type in a static field, named as [enumsByClass].

This algorithm is utilized in Enum's constructor. Here is a sample implementation.

```
// ensure the list for specifc Enum type exists
synchronized(enumsByClass) {
if (!enumsByClass.containsKey(declaringClass)) {
    enumsByClass.put(declaringClass, new ArrayList());
  }
}

// add [this], and assign ordinal value
List allEnums = (List)enumsByClass.get(declaringClass);
this.ordinal = allEnums.size();
allEnums.add(this);
```

1.3.2 Get enum by string value

Enum instances usually can be identified by their toString() methods. In this design, a utility method is provided to retrieve enum instance by string value. A static field, [stringSearchByClass:Map], is defined as a cache to hold the result. It maps an enum type to a map, which maps a string to an enum instance.

NOTE: (1) Developer can remove this variable if he/she thinks cache strategy is a redundant. (2) As thread-safe is a big issue with the introduction of cache, synchronization strategy is also shown in this algorithm.

```
// determine whether given enum type (Class) exits.
ArrayList allEnums = (ArrayList)enumsByClass.get(enumClass);
if (allEnums == null) {
    return null;
}

// retrieve the cache for the specific enum Class.
Map searchTable = (Map)stringSearchByClass.get(enumClass);


//If not found - add safely to list
```

```
    if (searchTable == null) {
      synchronized(stringSearchByClass) {
        if (stringSearchByClass.containsKey(enumClass)) {
          //probably somebody has added just already in another thread
          searchTable = (Map)stringSearchByClass.get(enumClass);
        } else {
          // Add new
          searchTable = new Hashtable(allEnums.size());
          stringSearchByClass.put(enumClass, searchTable);
        }
      }
    }

    // In case if our Map was able to answer - return value ASAP
    // without using additional synchronization
    Enum ret = (Enum)searchTable.get(stringValue);
    if (ret != null) {
      return ret;
    }

    // If not - verify if our cache is up to date and try again
    // Worst case is that we will search twice for an item not available
:o(
    synchronized(searchTable) {
      if (searchTable.size() != allEnums.size()) {
        for (int i = 0; i < allEnums.size(); i++) {
            Enum elem = (Enum)allEnums.get(i);
            String key = elem.toString();

            // cache the value
            if (key != null) {
                searchTable.put(key, elem);
            }
        }
      }
    }
    return (Enum)searchTable.get(stringValue);
```

1.3.3 Get enum instance name by reflection

Sometimes, the toString() method may not return the same string as the enum instance name, especially under TopCoder coding convention. For example, a enum representing low priority could have name of "LOW_PRIORITY", and at the same time has a toString() method returning "Low Priority". So getting instance name is a useful function. In another aspect, this algorithm will be used as part of get enum value by its name.

NOTE, this algorithm requires that the enum instance is defined as a public static field of its [declaringClass], otherwise, this algorithm will return null. This pre-assumption can be ensured in most usage scenario of type-safe enum class.

A cache mechanism can be used in this algorithm. This cache can be a map, [enumNameByClass]. The key of map is enum Class type, and value is also a map, which maps enum instance to enum name.

```
           final List allEnums = (List) enumsByClass.get(declaringClass);
        // the returned allEnums will never be null
        Map instanceToName = (Map) enumNamesByClass.get(declaringClass);
        // If not found - add safely to map
        if (instanceToName == null) {
            synchronized (enumNamesByClass) {
                if (enumNamesByClass.containsKey(declaringClass)) {
                    // probably somebody has added just already in
another thread
                    instanceToName = (Map)
enumNamesByClass.get(declaringClass);
                } else {
                    instanceToName = new Hashtable(allEnums.size());
                    enumNamesByClass.put(declaringClass, instanceToName);
                }
            }
        }
        // In case if our Map was able to answer - return value ASAP
without using additional synchronization
        final String ret = (String) instanceToName.get(this);
        if (ret != null) {
            return ret;
        }
        // If not - verify if our cache is up to date and try again
        // Worth case is that we will search twice for an item not
available :o(
        synchronized (instanceToName) {
            if (instanceToName.size() != allEnums.size()) {
                // iterate through every fields in declaringClass
                Field[] fields = declaringClass.getFields();
                for (int i = 0; i < fields.length; ++i) {
                    try {
                        // if the field value is an Enum of this
declaring class, add to the cache
                        Object value = fields[i].get(null);
                        if (value instanceof Enum && ((Enum)
value).getDeclaringClass() == declaringClass) {
                            instanceToName.put(value,
fields[i].getName());
                        }
                    } catch (NullPointerException e) {
                        // silently ignore, this can only occur when the
field is not a STATIC field
                    }
                }
            }
        }

        // return the enum name from cache
        return (String) instanceToName.get(this);
```

1.3.4 Get enum by enum name

There are two ways to do it. And they use the same variable to act as a cache of the algorithm. This variable is the static field, [nameSearchByClass]. It maps a declaring class to a map value, which maps an enum name to an enum instance. NOTE, this algorithm requires that the expected enum instance is defined as a public static field of its [declaringClass], otherwise, this algorithm will return null.

First algorithm works almost the same as getting enum by string value (1.3.2), except two minor places.

(1) The cache used by this algorithm is [nameSearchByClass] but not [stringSearchByClass].

(2) Statement [String key = elem.toString();] should be replaced by [String key = elem.getEnumName();]

Second algorithm works almost the same as getting enum instance name by refection (1.3.3), except some minor places.

(1) The cache used by this algorithm should be [nameSearchByClass] instead of [enumNameByClass].

(2) The values of [nameSearchByClass] map should be a [nameToInstance] map instead of [instanctToName] map in [enumNameByClass].

1.4 Component Class Overview

Enum:

This class provides as much support as is possible in Java for correct implementation of the "typesafe enum" pattern. With some care, Java programs can make use of C-style enums, with all the benefits of type safety, but with even more functionality. The way to accomplish this is well-understood; this class encapsulates as much of these best practices as possible, so that applications create enumerated types with minimal work by subclassing Enum.

Version 1.1 adds supports for internal sub-classing of enum classes, and also utility methods related to enum constant name.

This class is thread-safe.

1.5 Component Exception Definitions

NONE

1.6 Thread Safety

Thread-safe is required by Type Safe Enum. Enum class doesn't hold any mutable variable member. But there are several class-members (static fields) shared between all Enum instances. To achieve thread-safe, synchronization must be used carefully for the static fields. Special handling for thread safety is mentioned in method doc, and 1.3.2 is a good example.

And of course, sub-class should also be immutable to maintain thread safety.

2. Environment Requirements

2.1 Environment

- Development language: java 1.3, java 1.4, and java 5.0.
- Compile/test target: java 1.3, java 1.4, and java 5.0.

2.2 TopCoder Software Components

- NONE

2.3 Third Party Components

- NONE.

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.collection.typesafeenum

3.2 Configuration Parameters

NONE

3.3 Dependencies Configuration

NONE

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

Applications first create an enumeration type by sub-classing Enum. The only real requirement is to make constructors private and to provide one or public static final values representing the enumeration values, although it is of course strongly recommended that the class instances are immutable as well.

4.3 Demo

First example is a basic usage scenario.

```
public class MyBool extends Enum {
  public static final MyBool FALSE = new MyBool();
  public static final MyBool TRUE = new MyBool();
  private MyBool() {}
}
```

Then the enumeration values can be accessed, compared, added to collections, serialized, etc:

```
        List list = new ArrayList();
        list.add(MyBool.TRUE);
        list.add(MyBool.FALSE);

        boolean notsame = MyBool.TRUE.equals(MyBool.FALSE);
        assertFalse("notsame must be false", notsame);
        int negative = MyBool.TRUE.compareTo(MyBool.FALSE);
        assertTrue("negative must be negative", negative < 0);
```

We can also get enum constant name (introduced in version 1.1):

```
        MyBool mybool = MyBool.FALSE;
        String enumConstantName = mybool.getEnumName();
        assertTrue("enum constant name must be 'FALSE'",
enumConstantName.equals("FALSE"));
```

But the enumeration type can be more complex; in particular it can have JavaBean properties or any method at all. In particular, a good toString() method is useful.

The following example also shows the usage of enum constants with constant-specifc class bodies, which is supported by version 1.1.

```
public abstract class TypeEnum extends Enum {

    /**
     * NUMBER enum for TypeEnum. Introduced in version 1.1, enum constant
with constant-specific class body.
     */
    public static final TypeEnum NUMBER = new TypeEnum("Number") {
        public boolean checkValue(Object val) {
            return val instanceof Number;
        }
    };

    /**
     * STRING enum for TypeEnum. Introduced in version 1.1, enum constant
with constant-specific class body.
     */
    public static final TypeEnum STRING = new TypeEnum("String") {
        public boolean checkValue(Object val) {
            return val instanceof String;
        }
    };

    /**
     * Represents a custom property for enum.
     */
    private final String name;

    /**
     * Creates a new <code>TypeEnum</code> enum.
     *
     * @param name
     *            the value for custom property of this enum
     */
    private TypeEnum(String name) {
        // Introduced in version 1.1, this is required to support
internal subclassing.
        super(TypeEnum.class);
        if (name == null) {
            throw new IllegalArgumentException("name shouldn't be
null.");
        }
        this.name = name;
    }

    /**
     * Checks the value.
     *
     * @param val
     *            the value to check
     * @return whether the value is passed the check
     */
    public abstract boolean checkValue(Object val);

    /**
     * Gets the custom property for enum.
     *
     * @return the custom property for enum
     */
    public String getName() {
        return this.name;
    }

    /**
     * Gets the string representation of the suit.
     *
     * @return the string representation of the suit
     */
    public String toString() {
        return getName();
    }
}
```

Enumeration classes can be looked up by ordinal value (for a given class, the first defined enumeration value has ordinal 0 and each successive one is assigned the next higher ordinal value):

```
        TypeEnum numberType = (TypeEnum)
Enum.getEnumByOrdinal(TypeEnum.NUMBER.getOrdinal(), TypeEnum.class);
        assertTrue("The TypeEnum is TypeEnum.NUMBER", numberType ==
TypeEnum.NUMBER);
```

Or even by String representation:

```
        TypeEnum stringType = (TypeEnum)
Enum.getEnumByStringValue("String", TypeEnum.class);
        assertTrue("The stringType is TypeEnum.STRING", stringType ==
TypeEnum.STRING);
```

Or even by enum constant name (introduced in 1.1):

```
        stringType = (TypeEnum) Enum.getEnumByName("STRING",
TypeEnum.class);
        assertTrue("The stringType is TypeEnum.STRING", stringType ==
TypeEnum.STRING);
```

Or all enumeration values can be retrieved as a sorted List:

```
        List typeList = Enum.getEnumList(TypeEnum.class);
        assertTrue("The typeList is [TypeEnum.NUMBER, TypeEnum.STRING]",
typeList.size() == 2);
        assertTrue("The typeList is [TypeEnum.NUMBER, TypeEnum.STRING]",
typeList.get(TypeEnum.NUMBER.getOrdinal()) == TypeEnum.NUMBER);
        assertTrue("The typeList is [TypeEnum.NUMBER, TypeEnum.STRING]",
typeList.get(TypeEnum.STRING.getOrdinal()) == TypeEnum.STRING);
```

5. Future Enhancements

Automatically determine the declaring class of enum.

Enhance the serialization mechanism. We can learn from Java 5.0 enum serialization mechanism.