

# INF 212

## ANALYSIS OF PROG LANGS


### *REFLECTION*

Instructors: Crista Lopes

Copyright © Instructors.

# Outline

- **History and Background**
  - The meta-circular interpreter
- Definitions
- The reflective tower
- Reflection as used in
  - PHP
  - Ruby
  - Java
- Concerns to keep in mind while using Reflection.
- Practical applications.



*“Follow effective action with quiet reflection. From the quiet reflection will come even more effective action.”*

*Peter F. Drucker*



# History

- It arose naturally in artificial intelligence, where it is intimately linked to the end goal itself.
- Reflection is viewed as the emergent property responsible, at least in part, for what is considered an “intelligent behaviour”.
- Reflection helps us master new skills, cope with incomplete knowledge, define terms, examine assumptions, review our experiences, plan, check for consistency, and recover from mistakes.
- Key strategy for meta-programming.

# History

- Languages like LISP had inherent reflective properties.
  - The powerful ‘quote’ mechanism in LISP, Scheme etc enabled code to be treated as data – primitive manifestations of reflection.
- Brian Cantwell Smith's work in the 80s
  - Formalized the concept of reflection
  - Developed two dialects of Lisp namely 2-Lisp and 3-Lisp
  - Became famous in the functional domain and therefore inspired much work there.
- By the end of the 90s- the need for structuring mechanisms was noticed
  - The object-oriented paradigm imposed on itself to take up this challenge.
  - However they were also influenced by the Lisp community

# Meta-circular interpreter

- A **self-interpreter**, or **metainterpreter**, is a programming language interpreter written in the language it interprets
- A **meta-circular interpreter** is a special case of a **self-interpreter** in which the existing facilities of the parent interpreter are directly applied to the language being interpreted, without any need for additional implementation
  - ▣ Primarily in homoiconic languages

# Homoiconicity

- primary representation of programs is also a data structure in a primitive type of the language itself
  - ▣ internal and external representations are essentially the same
- *homo* = the same  
*icon* = representation
- Examples: Lisp, Scheme, R, Mathematica
- Counter-examples: Java, C, Python...
  - ▣ Programs are strings, text

# Homoiconicity

- `(* (sin 1.1) (cos 2))`

`>> -0.37087312359709645`

- ``(* (sin 1.1) (cos 2))`

`>> `(* (sin 1.1) (cos 2))`

Literal. Means:  
“don’t interpret me!”

- `(eval `(* (sin 1.1) (cos 2)))`

- `>> -0.37087312359709645`



# What is Eval?

- Way back from McCarthy's paper on LISP
- To a first approximation,  
eval is the exposure of the interpreter itself to the programmer
- In homoiconic languages, eval takes an expression of the language and interprets it
- In non-homoiconic languages, eval takes a string, parses it, and interprets the resulting expression

# Where have you seen eval?

## □ JavaScript `<script type="text/javascript">`

```
eval("x=10;y=20;document.write(x*y)");  
document.write("<br />" + eval("2+2"));  
document.write("<br />" + eval(x+17));  
  
</script>
```

## □ Python

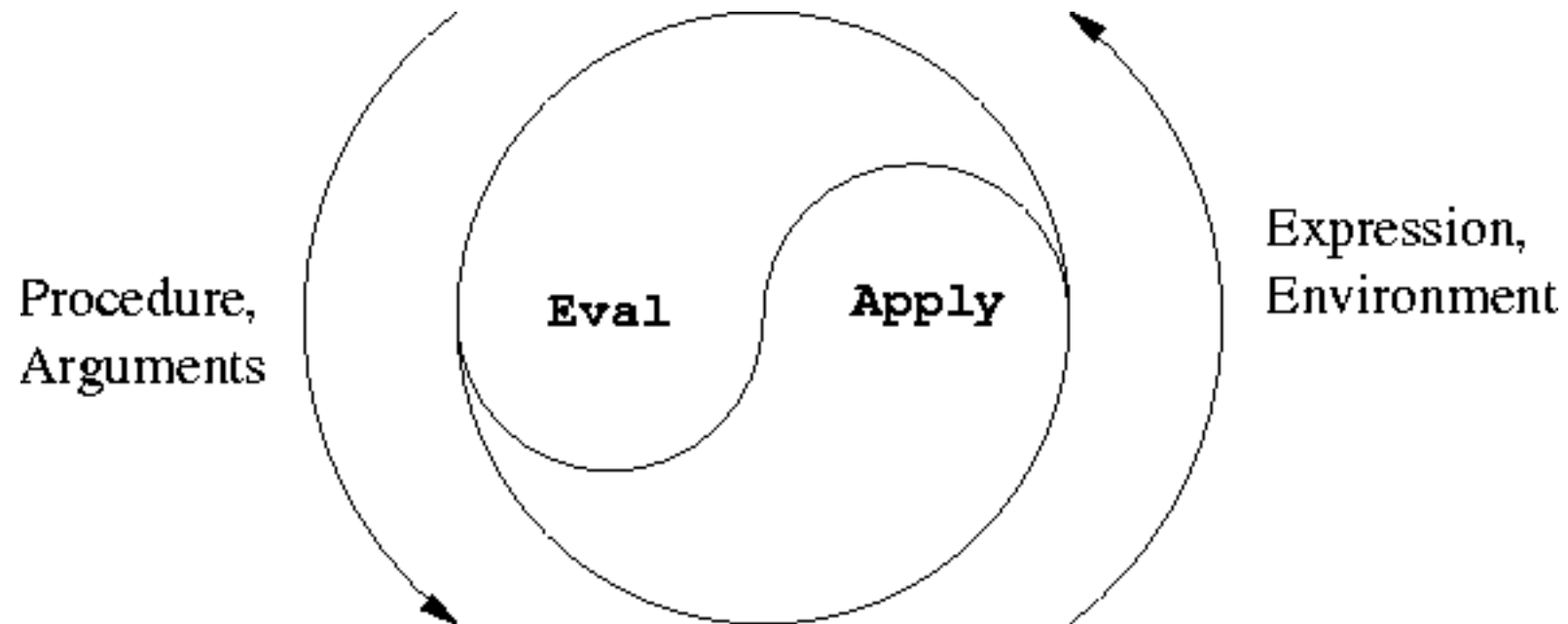
```
>>> x = 1  
>>> print eval('x+1')  
2
```

# Eval = Evil ?

---

- discuss

# Meta-circular interpreter



[source](#)

# Eval

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                          (lambda-body exp)
                          env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        (((application? exp)
         (apply (eval (operator exp) env)
                  (list-of-values (operands exp) env))))
        (else
         (error "Unknown expression type - EVAL" exp))))
```

# Apply

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
        (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type - APPLY" procedure))))
```

# Outline

- History and Background
  - The meta-circular interpreter
- **Definitions**
- The reflective tower
- Reflection as used in
  - PHP
  - Ruby
  - Java
- Concerns to keep in mind while using Reflection.
- Some practical applications.

# Definitions

- General definition of reflection by Brian Smith in the 80s

“An entity’s integral ability to represent, operate on, and otherwise deal with its self in the same way that it represents, operates on and deals with its primary subject matter.”

- In programming languages the incarnation of this definition is something like

“Reflection is the ability of a program to manipulate itself as data during execution.”



# Definitions

- **Reification** - mechanism for encoding execution state as data; providing such an encoding is called reification.
- Reification is the process by which a user program or any aspect of a programming language that was implicit in the translated program and the run-time system, are expressed in the language itself.
- This process makes the program available to the program, which can inspect all these aspects as ordinary data .
- Reification data is often said to be made a **first class object**.

# Reification - Examples

- Many programming languages encapsulate the details of memory allocation in the compiler and the run-time system, hidden from developers
  - In C#, reification is used to make parametric polymorphism implemented as generics as a first-class feature of the language.
- In the design of the C programming language, the memory address is reified and is available for direct manipulation by other language constructs.

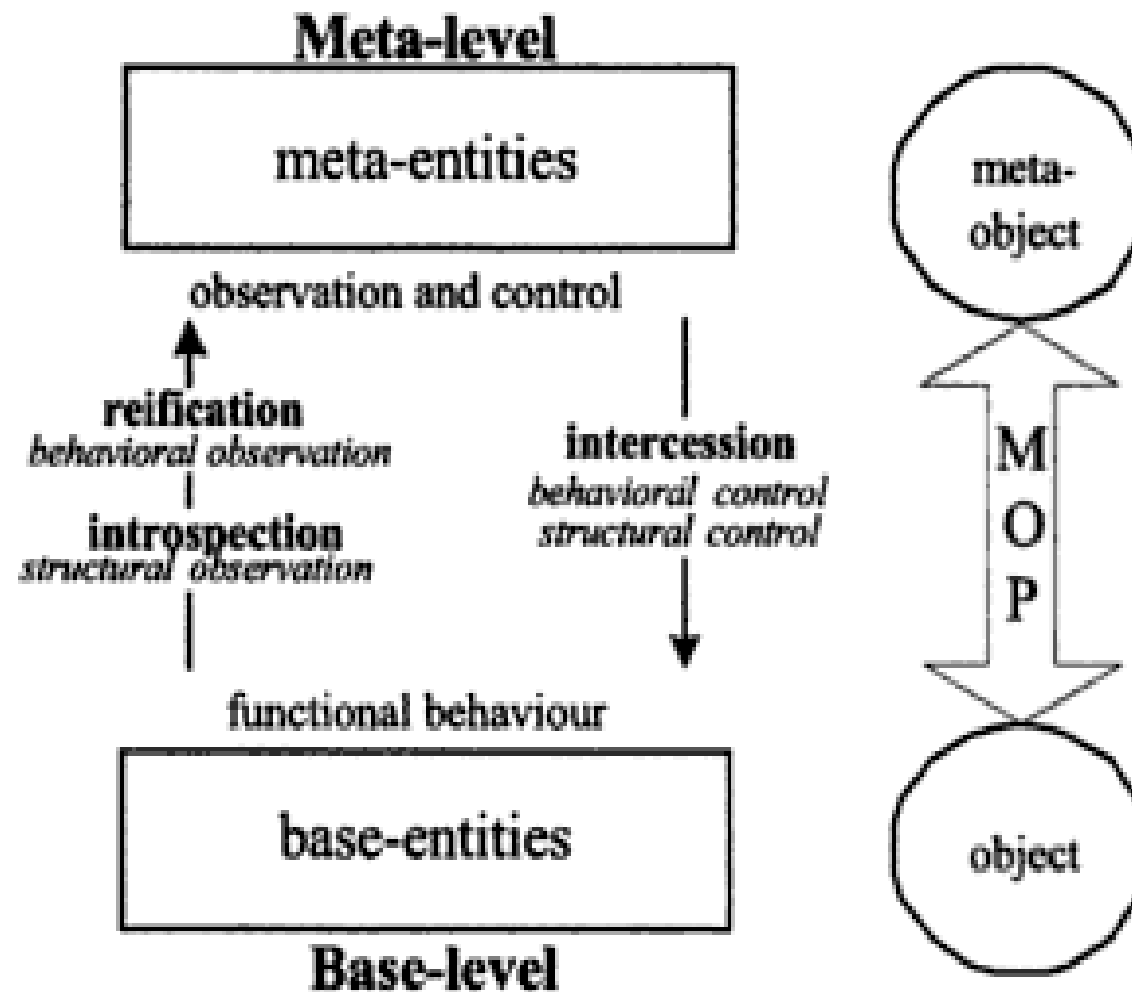
# Intercession

- Modify the characteristics of the elements of the program. Intercession allows you to change the properties of classes, variables, methods, functions, etc. at run-time

# Definitions

- **Structural Reflection** - is concerned with the ability of the language to provide a complete reification of both the program currently executed as well as a complete reification of its abstract data types.
- **Behavioral Reflection** - is concerned with the ability of the language to provide a complete reification of its own semantics and implementation (processor) as well as a complete reification of the data and implementation of the run-time system.
  - Behavioral Reflection has proven to be more difficult to implement than structural reflection.

# Definitions



# Outline

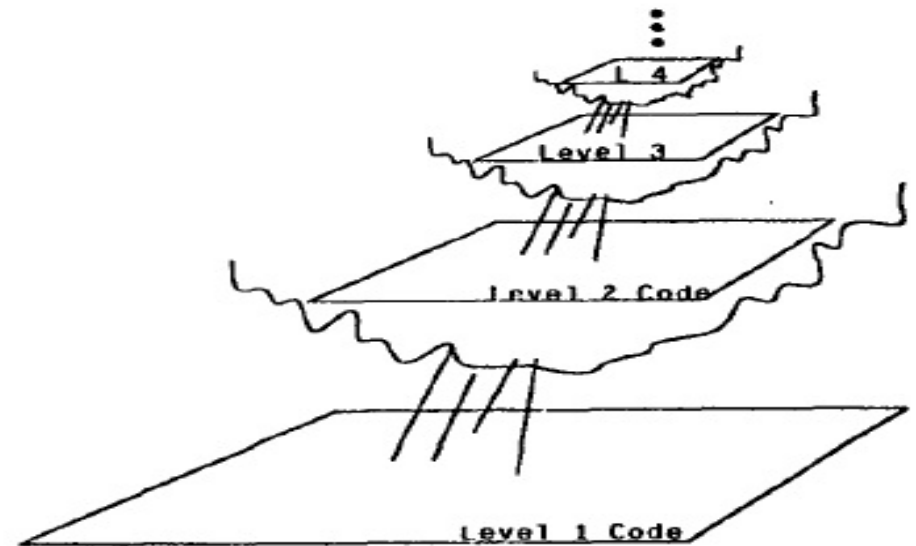
- History and Background
  - The meta-circular interpreter
- Definitions
- **The reflective tower**
- Reflection as used in
  - PHP
  - Ruby
  - Java
- Concerns to keep in mind while using Reflection.
- Some practical applications.

# The Reflective Tower

- **WHEN:** Brian Cantwell Smith's 1982 doctoral dissertation introduced the notion of computational reflection in programming languages.
- **WHY :** He felt that Lisp self referential properties were not adequate to reason about its operations and structures. He felt a truly reflective system should be able to
  - Provide an account of itself embedded within it.
  - Have a connection between that embedded account and the system it describes.
  - Have an appropriate vantage – not too far or not too close.

# The Reflective Tower

- **HOW:** His proposal to solve the above 3 problems and also to define an architecture for serial programming languages (procedural reflection) was an infinite tower of meta-circular interpreters connected together in a simple but critical way.
- To describe this new architecture he developed a new dialect of Lisp called 3-Lisp.



*Figure 15: The 3-LISP Reflective Tower*



# Reflection in 3-Lisp

- A reflective tower is constructed by stacking a virtually infinite number of meta-circular interpreters, each one executing the one under itself and the bottom one (level 1) executing the end-user program (level 0).
  - Although it is potentially infinite, just like well-defined recursions only a finite number of levels are required to run a program.
- Reflective computations are initiated by calling reflective procedures, procedures with three parameters, the body of which being executed one level up in the tower; upon invocation, a reflective procedure is passed a reification of the argument structure of its call, its current environment and its current continuation.

# Outline

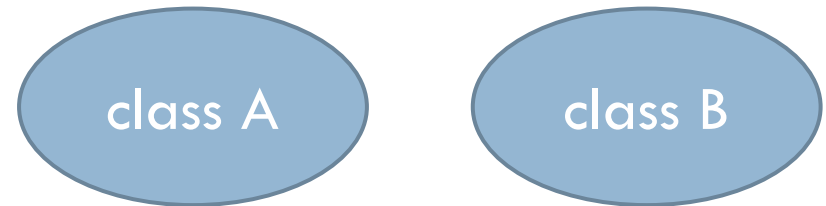
- History and Background
  - The meta-circular interpreter
- Definitions
- The reflective tower
- **Reflection as used in**
  - **PHP**
  - **Ruby**
  - **Java**
- Concerns to keep in mind while using Reflection.
- Some practical applications.

# Reflection in modern PLs

Program text:

```
class A {...}  
class B {...}  
Etc.
```

Runtime environment:



Etc.

# Web Programming - PHP Reflection

- **Reflection** is designed to reverse engineer various parts of PHP, including classes, functions, and extensions. By "reverse engineer" it means that it gives you all sorts of information that otherwise you would need to try to dig out yourself.
- There are **three primary uses** for reflection in PHP:
  - You have encoded scripts you need to interact with.
  - The PHP manual isn't wholly up to date and you are unable to, or you don't want to read the source code.
  - You're just curious how something works and would rather not read someone else's PHP .

# PHP - Reflection Example

```
<?php
    class myparent {
        public function foo($bar) {
            // do stuff
        }
    }
    class mychild extends myparent {
        public $val;
        private function bar(myparent &$baz) {
            // do stuff
        }
        public function __construct($val) {
            $this->val = $val;
        }
    }
    $child = new mychild('hello world');
    $child->foo('test');
?>
```

# PHP - Reflection Example

```
$childreflect = new ReflectionClass('mychild');  
echo "This class is abstract: ",  
    (int)$childreflect->isAbstract(), "\n";  
echo "This class is final: ", (int)$childreflect->isFinal(), "\n";  
echo "This class is actually an interface: ",  
    (int)$childreflect->isInterface(), "\n";  
echo "\$child is an object of this class: ",  
    (int)$childreflect->isInstance($child), "\n";  
$parentreflect = new ReflectionClass('myparent');  
echo "This class inherits from myparent: ",  
    (int)$childreflect->isSubclassOf($parentreflect), "\n";
```

The output of that is:

This class is abstract: 0

This class is final: 0

This class is actually an interface: 0

\$child is an object of this class: 1

This class inherits from myparent: 1

# Ruby Reflection

- Let's begin with an example : Assume that you want to create a class instance at runtime, and the name of this class depends on the parameter being passed to a function.
- One way to do this is to write conditional loops and create the object. But if there are too many classes then becomes a problem. Solution: Use **reflection**!
- In Ruby using reflection you can get the following information:
  1. What classes already exist
  2. Information on the methods in those classes
  3. Inheritance etc.

# Reflection in Ruby

- ❑ `ObjectSpace` allows us to obtain the reflective information.
- ❑ `ObjectSpace.each_object { |x| puts x }` gives us all the living, non-immediate objects in the process.
- ❑ `ObjectSpace.each_object(Class) { |x| puts x }` gives us all the classes in the ruby process.
- ❑ Now the problem becomes easier - Iterate over all the classes, and if the name matches then create an object of that class, and execute the required functions.



# Reflection in Ruby

```
class ClassFromString
```

```
  @@counter = 0
```

```
  def initialize
```

```
    @@counter += 1
```

```
  end
```

```
  def getCounterValue
```

```
    puts @@counter
```

```
  end
```

```
end
```

```
def createClassFromString(classname)
```

```
  ObjectSpace.each_object(Class) do |x|
```

```
    if x.name == classname
```

```
      object = x.new
```

```
      object.getCounterValue
```

```
    end
```

```
  end
```

```
end
```

```
createClassFromString("ClassFromString")
```

The above code illustrates the example in code. You can even use superclass method to get the parent name, and construct the entire hierarchy. *Exercise: Find methods in same way.*

# Reflection in Java

- **java.lang.Class.**

- **Class.forName()**

- **Class c = Class.forName("edu.uci.inf212.Example");**

# Reflection in Java

**Object.getClass()**

**Class c = "foo".getClass();**

**The.class Syntax()**

**Class c = boolean.class;**

# Reflection in Java

- **Reflection** is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine. This is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language. Reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible.
- **Extensibility Features:** An application may make use of external, user-defined classes by creating instances of objects using their fully-qualified names.

# Outline

- History and Background
  - The meta-circular interpreter
- Definitions
- The reflective tower
- Reflection as used in
  - PHP
  - Ruby
  - Java
- **Concerns to keep in mind while using Reflection.**
- Some practical applications.

# Performance Penalty

**Reflection** is powerful, but should not be used indiscriminately. If it is possible to perform an operation without using reflection, then it is preferable to avoid using it. The following concerns should be kept in mind when accessing code via reflection.

**Performance Overhead** : Because reflection involves types that are dynamically resolved, certain Java virtual machine optimizations can not be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.

# Security

- **Security Restrictions** : Reflection requires a runtime permission which may not be present when running under a security manager. This is an important consideration for code which has to run in a restricted security context, such as in an Applet.
- **Exposure of Internals** : Since reflection allows code to perform operations that would be illegal in non-reflective code, such as accessing private fields and methods, the use of reflection can result in unexpected side-effects, which may render code dysfunctional and may destroy portability. Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

# Outline

- History and Background
  - The meta-circular interpreter
- Definitions
- The reflective tower
- Reflection as used in
  - PHP
  - Ruby
  - Java
- Concerns to keep in mind while using Reflection.
- **Practical applications.**



# Practical Applications of Reflection

**Proxies:** e.g. a JDK Proxy of a large interface (20+ methods) to wrap (i.e. delegate to) a specific implementation. A couple of methods were overridden using an `InvocationHandler`, the rest of the methods were invoked via reflection.

**Plugins:** load specific classes at run-time.

**Class Browsers and Visual Development Environments:** A class browser has to be able to enumerate members of classes. Visual development environments can benefit from making use of type information available in reflection to aid code development.

**Debuggers and Test Tools:** Debuggers need to be able to examine private members on classes. Test harnesses can make use of reflection to systematically call a discoverable set APIs defined on a class, to insure a high level of code coverage in a test suite.



# Practical Use of Reflection

Database access

# Code and Databases: a difficult marriage

```
import java.sql.*;

class UpdateLogic {
    public static void main(String args[]) {
        Connection connection = null;
        try {
            Class.forName("imaginary.sql.iMysqlDriver");
            String url = "jdbc:mysql://athens.imaginary.com:4333/db_test";
            Statement s;
            con = DriverManager.getConnection(url, "borg", "");
            con.setAutoCommit(false);        // make sure auto commit is off!
            s = con.createStatement();        // create the first statement
            s.executeUpdate("INSERT INTO t_test (test_id, test_val) " +
                           "VALUES(" + args[0] + ", '" + args[1] + "')");
            s.close();                        // close the first statement
            s = con.createStatement();        // create the second statement
            s.executeUpdate("INSERT into t_test_desc (test_id, test_desc) " +
                           "VALUES(" + args[0] +
                           ", `This describes the test.`)");
            con.commit();                    // commit the two statements
            System.out.println("Insert succeeded.");
            s.close();                        // close the second statement
        }
        catch( SQLException e ) {
            if( con != null ) {...

```

# Problems

- SQL statements as literals
  - ▣ No syntax checking, errors occur at runtime
- Object model (in memory) vs. Relational model (on disk)
  - ▣ Constantly having to parse/unparse data
- Duplication of knowledge of tables
  - ▣ Column names

# Reflection to the rescue

```
public class MySQLGenericTableHandler<T>:MySQLFramework where T:class,new()  
    public MySQLGenericTableHandler(string connectionString,  
        string realm, string storeName) : base(connectionString){  
        m_connectionString = connectionString;  
  
        Type t = typeof(T);  
        FieldInfo[] fields = t.GetFields(BindingFlags.Public |  
                                           BindingFlags.Instance |  
                                           BindingFlags.DeclaredOnly);  
  
        if (fields.Length == 0)  
            return;  
        foreach (FieldInfo f in fields)  
        {  
            if (f.Name != "Data")  
                m_Fields[f.Name] = f;  
            else  
                m_DataField = f;  
        }  
    }  
}
```

# One Table

```
class MySqlGroupsGroupsHandler : MySQLGenericTableHandler<GroupData>
{
    public MySqlGroupsGroupsHandler(string connectionString,
                                     string realm,
                                     string store)
        : base(connectionString, realm, store)
    {
    }
}

public class GroupData
{
    public UUID GroupID;
    public Dictionary<string, string> Data;
}
```

# Reflection to the rescue

In MySQLGenericTableHandler<T>

```
public virtual T[] Get(string[] fields, string[] keys)
{
    if (fields.Length != keys.Length)
        return new T[0];
    List<string> terms = new List<string>();

    using (MySqlCommand cmd = new MySqlCommand())
    {
        for (int i = 0 ; i < fields.Length ; i++)
        {
            cmd.Parameters.AddWithValue(fields[i], keys[i]);
            terms.Add("`" + fields[i] + "` = ?" + fields[i]);
        }
        string where = String.Join(" and ", terms.ToArray());
        string query = String.Format("select * from {0} where {1}",
                                     m_Realm, where);

        cmd.CommandText = query;
        return DoQuery(cmd);
    }
}
```

Generic Get

# Reflection to the rescue

```
protected T[] DoQuery(MySqlCommand cmd) {
    List<T> result = new List<T>();
    using (MySqlConnection dbcon = new MySqlConnection(m_connectionString))
    {
        dbcon.Open();
        cmd.Connection = dbcon;

        using (IDataReader reader = cmd.ExecuteReader())
        {
            if (reader == null)
                return new T[0];

            CheckColumnNames(reader);

            while (reader.Read())
            {
                T row = new T();
```

Continued...



# Reflection to the rescue

```
foreach (string name in m_Fields.Keys)
{
    if (reader[name] is DBNull)
        continue;
    if (m_Fields[name].FieldType == typeof(bool))
    {
        int v = Convert.ToInt32(reader[name]);
        m_Fields[name].SetValue(row, v != 0 ? true : false);
    }
    else if (m_Fields[name].FieldType == typeof(UUID))
        m_Fields[name].SetValue(row, DBGuid.FromDB(reader[name]));
    else if (m_Fields[name].FieldType == typeof(int))
    {
        int v = Convert.ToInt32(reader[name]);
        m_Fields[name].SetValue(row, v);
    }
    else
        m_Fields[name].SetValue(row, reader[name]);
}
```

Continued...

# Reflection to the rescue

```
        if (m_DataField != null)
        {
            Dictionary<string, string> data =
                new Dictionary<string, string>();
            foreach (string col in m_ColumnNames)
            {
                data[col] = reader[col].ToString();
                if (data[col] == null)
                    data[col] = String.Empty;
            }
            m_DataField.SetValue(row, data);
        }
        result.Add(row);
    }
}
return result.ToArray();
}
```