Concepts of Object-Oriented Programming

Peter Müller

Chair of Programming Methodology

Autumn Semester 2009

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Meeting the Requirements

Cooperating Program Parts with Well-Defined Interfaces

→ Objects (data + code)
→ Interfaces
→ Encapsulation

Classification and Specialization

→ Classification, subtyping
→ Polymorphism
→ Substitution principle

Highly Dynamic Execution Model

→ Active objects
→ Message passing

Correctness

→ Interfaces
→ Encapsulation
→ Simple, powerful concepts

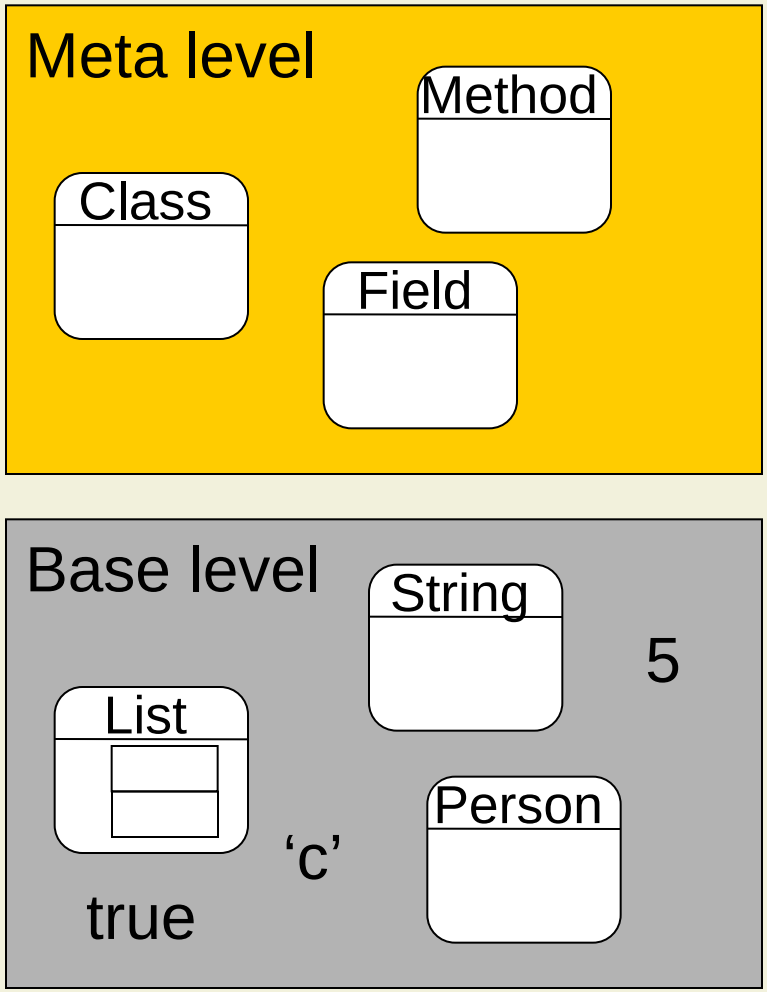
Repetition: Dynamic Type Checking

- **instanceof** can be used to avoid runtime errors
- **instanceof** makes type information available to programs

```
Object[ ] oa = new Object[ 10 ];
String s = “A String”;

oa[ 0 ] = s;

…
if ( oa[ 0 ] instanceof String )
  s = ( String ) oa[ 0 ];

s = s.concat( “Another String” );
```

Reflection

- A program can observe and modify its own structure and behavior
- Simplest form
  - Type information is available at run time
- Most elaborate
  - All compile-time information can be observed and modified



10. Reflection

Class Objects

```
class Class<T> … {
  static Class <?>     forName( String name )  throws …          {…}
  Method[ ]   getMethods( )                                      {…}
  Method[ ]   getDeclaredMethods( )                              {…}
  Method      getMethod( String name, Class<?>… parTypes )       {…}
  Class<? super T>  getSuperclass( )                             {…}
  boolean     isAssignableFrom( Class<?> cls )                   {…}
  T           newInstance( ) throws …                            {…}
  … }
```

Java

- The Class-object for a class can be obtained by the pre-defined class-field

```
Class StringClass = String.class;
```

Example: Simple Introspection

```
import java.lang.reflect.*;

public class FieldInspector {
  public static void main( String[ ] ss ) {
    Class cl = Class.forName( ss[ 0 ] );
    Field[ ] fields = cl.getFields( );

    for( int i = 0; i < fields.length; i++ ) {
      Field f = fields[ i ];
      Class type = f.getType( );
      String name = f.getName( );
      System.out.println( type.getName( ) + " “ + name + “;“ );
    }
  }
}
```

Java

Error handling omitted

Example: Universal toString-Method

```
public static String getString( Object o ) {
  if ( o == null ) return "null";
  Class toExamine = o.getClass( );
  String state = "[";
  Field[ ] fields = toExamine.getDeclaredFields( );
  for ( int fi = 0; fi < fields.length; fi++ ) {
    Field f = fields[ fi ];
    if ( !Modifier.isStatic( f.getModifiers( ) ) )
      state += f.getName( ) + "=" + f.get( o ) + ", ";
  }
  return state + "]";
}
```

Java

Obtain Class-object

Ignore static fields

Include field name

Include field value

Compile-time error: uncaught exception

Field.get

```
public Object get( Object obj ) throws
    IllegalArgumentException,
    IllegalAccessException
```

Java

- Safety checks have to be done at run-time
  - Type checking: does obj have that field?
  - Accessibility: is the client allowed to access that field?

Universal toString-Method (cont'd)

```
public static String getString( Object o ) {
  if ( o == null ) return "null";
  Class toExamine = o.getClass( );
  String state = "[";
  Field[ ] fields = toExamine.getDeclaredFields( );
  for ( int fi = 0; fi < fields.length; fi++ )
    try {
      Field f = fields[ fi ];
      if ( !Modifier.isStatic( f.getModifiers( ) ) )
        state += f.getName() + "=" + f.get( o ) + ", ";
    } catch ( Exception e ) { return "Exception"; }
  return state + "]";
}
```

Java

```
class Cell {
  private int value = 5;
  …
}
```

```
Cell c = new Cell( );
String s = getString( c );
System.out.println( s );
```

```
Exception
```

Universal toString-Method (cont'd)

```
public static String getString( Object o ) {
  if ( o == null ) return "null";
  Class toExamine = o.getClass( );
  String state = "[";
  Field[ ] fields = toExamine.getDeclaredFields( );
  for ( int fi = 0; fi < fields.length; fi++ )
    try {
      Field f = fields[ fi ];
      f.setAccessible( true );
      if ( !Modifier.isStatic( f.getModifiers( ) ) )
        state += f.getName() + "=" + f.get( o ) + ", ";
    } catch ( Exception e ) { return "Exception"; }
  return state + "]";
}
```

Java

Suppress Java's access checking

```
class Cell {
  private int value = 5;
  …
}
```

```
Cell c = new Cell( );
String s = getString( c );
System.out.println( s );
```

```
[value=5, ]
```

Example: Unit Testing

```
class Cell {
  int value;

  Cell( int v ) { value = v; }

  int get( ) { return value; }

  void set( int v ) { value = v; }

  void swap( Cell c ) {
    int tmp = value;
    value = c.value;
    c.value = tmp;
  }
}
```

```
class TestCell {
  void testSet( ) { … }

  void testSwap( ) {
    Cell c1 = new Cell( 5 );
    Cell c2 = new Cell( 7 );
    c1.swap( c2 );
    assert c1.get( ) == 7;
    assert c2.get( ) == 5;
  }
}
```

- Goal: Write generic test driver that executes tests

Unit Testing: Test Driver

```
public static void testDriver( String testClass ) {
    Class c = Class.forName( testClass );
    Object tc = c.newInstance( );
    Method[ ] methods = c.getDeclaredMethods( );

    for( int i = 0; i < methods.length; i++ ) {
      if( methods[ i ].getName( ).startsWith( "test“ ) &&
          methods[ i ].getParameterTypes( ).length == 0 )

        methods[ i ].invoke( tc );

    }
}
```

Java

Find all parameter-less methods whose name starts with “test”

Invoke the method

Error handling omitted

- Basic mechanism behind JUnit
  - Newer versions use annotation @Test instead of prefix

Unit Testing: Error Handling

```
public static void testDriver( String testClass ) {
  try {
    Class c = Class.forName( testClass );
    Object tc = c.newInstance( );
    Method[ ] methods = c.getDeclaredMethods( );

    for( int i = 0; i < methods.length; i++ ) {
      if( methods[ i ].getName( ).startsWith( "test“ ) &&
          methods[ i ].getParameterTypes( ).length == 0 )

        methods[ i ].invoke( tc );

    }
  } catch( Exception e ) { … }
}
```

Java

Class.newInstance

```
public T newInstance( ) throws
    InstantiationException,
    IllegalAccessException
```
Java

- Safety checks have to be done at run-time
  - Type checking:
    Does the Class-object represent a concrete class?
    Does the class have a parameter-less constructor?
  - Accessibility:
    Are the class accessible and the parameter-less constructor accessible?

Reminder: Double Invocation

```
class Shape {
  Shape intersect( Shape s )
  { return s.intersectShape( this ); }

  Shape intersectShape( Shape s )
  { // general code for all shapes }

  Shape intersectRectangle( Rectangle r )
  { return intersectShape( r ); }
}
```

- Additional dynamically-bound call for specialization based on dynamic type of explicit argument

```
class Rectangle extends Shape {
  Shape intersect( Shape s )
  { return s.intersectRectangle( this ); }

  Shape intersectRectangle( Rectangle r )
  { // efficient code for two rectangles }
}
```

Visitor Pattern

```
interface Visitor {
  void visitExpr( Expr e );
  void visitLiteral( Literal l );
  void visitVariable( Variable v );
}
```

```
class PrintVisitor implements Visitor {
  void visitExpr( Expr e ) { … }
  void visitLiteral( Literal l ) { … }
  void visitVariable( Variable v ) { … }
}
```

```
class EvalVisitor implements Visitor {
  …
}
```

```
class Expr {
  void accept( Visitor v )
  { v.visitExpr( this ); }
}
```

```
class Literal extends Expr {
  void accept( Visitor v )
  { v.visitLiteral( this ); }
}
```

```
class Variable extends Expr {
  void accept( Visitor v )
  { v.visitVariable( this ); }
}
```

Reflective Visitor

```
abstract class Visitor {
  void visit( Expr e ) {
    String name = "visit" + e.getClass( ).getName( );
    Method m = this.getClass( ).getMethod( name, e.getClass( ) );
    m.invoke( this, o );
  }
}
```

Java

Construct method name, e.g., visitLiteral

Find method visitX(X) in dynamic type of **this**

Invoke the method

```
class PrintVisitor extends Visitor {
  void visitExpr( Expr e ) { … }
  void visitLiteral( Literal l ) { … }
  void visitVariable( Variable v ) { … }
}
```

Error handling omitted

Reflective Visitor: Discussion

Pros

- Much simpler code
  - Second dynamic dispatch implemented via reflection
  - No accept-methods
- Flexible look-up mechanism
  - E.g., visit could look for most specific method

Cons

- Not statically safe
  - Missing method detected at run-time
- Slower
  - Many run-time checks involved

Java Generics

- Due to Java's erasure semantics, generic type information is not represented in the class file and at run time

```
try {
  LinkedList<String> list = new LinkedList<String>( );
  Class c = list.getClass( );
  Method add = c.getMethod( "add", String.class );
} catch( Exception e ) {
  System.out.println( "Method not found" );
}
```

Java

Method not found

Java Generics

- Due to Java's erasure semantics, generic type information is not represented in the class file and at run time

```
try {
  LinkedList<String> list = new LinkedList<String>( );
  Class c = list.getClass( );
  Method add = c.getMethod( "add", Object.class );
} catch( Exception e ) {
  System.out.println( "Method not found" );
}
```

Java

```
// no exception
```

10. Reflection

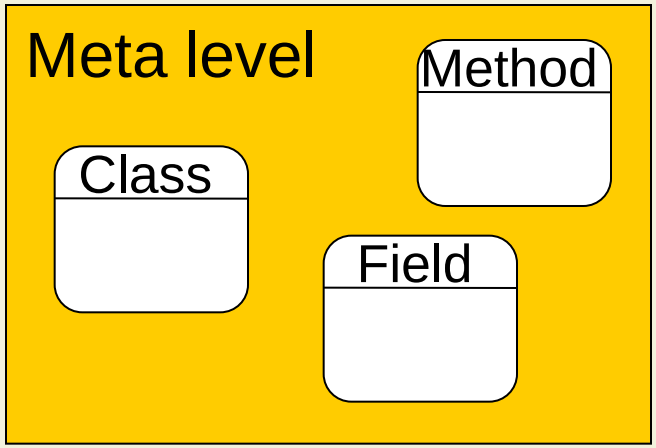
10.1 Introspection

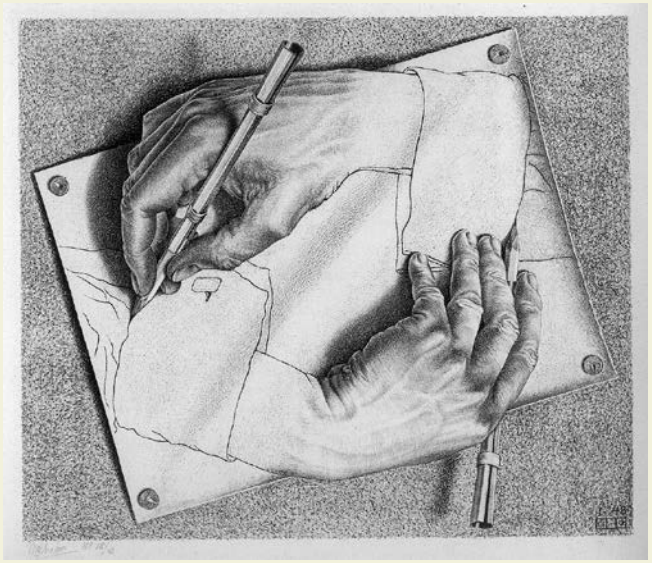
10.2 Reflective Code Generation

10.3 Dynamic Code Manipulation

Motivation

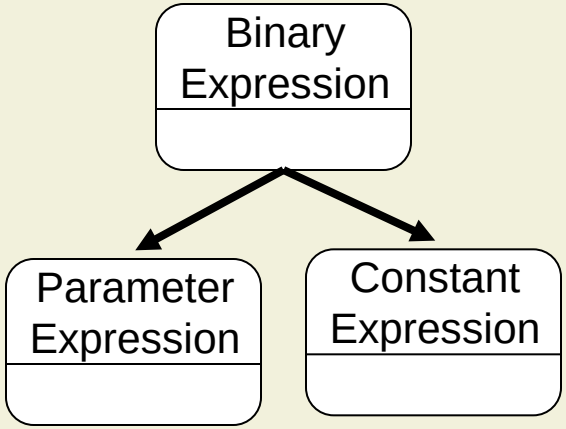
- If code is represented as data, we can as well allow programs to create code from data
- Generate code dynamically according to user input and execution environment
- Examples
  - Class loading in Java
  - Expression trees in C#





C# Expression Trees

- Expression trees represent the abstract syntax tree of C# expressions
  - Can be created like any other data structure
- Class Expression provides a Compile-method, which, compiles expression tree to executable code
  - Compilation happens at run time
- Main application: generation of SQL queries



Expression Trees: Example

```
Expression<Func<int, bool>> Build( string l, string op, int c ) {
  ParameterExpression lhs = Expression.Parameter( typeof( int ), l );
  ConstantExpression ce = Expression.Constant( c, typeof( int ) );
  BinaryExpression be = null;

  switch ( op ) {
    case "<": be = Expression.LessThan( lhs, ce ); break;
    case ">": be = Expression.GreaterThan( lhs, ce ); break;
    …
  }

  return Expression.Lambda<Func<int, bool>>
                                  ( be, new ParameterExpression[ ] { lhs } );
}
```

C#

AST for lambda-expression
l => l op c

Expression Trees: Example (cont'd)

```
class Filter {
  void Demo( string condition, int[ ] data ) {
    string op; int c;
    Parse( condition, out op, out c );

    Func<int, bool> filter = Build( "x", op, c ).Compile( );

    foreach ( int i in data ) {
      if ( filter( i ) )  Console.WriteLine( i );
    }
  }
  …
}
```

C#

Parse condition to determine operator and constant

Compile expression tree

Invoke compiled lambda-expression

10. Reflection

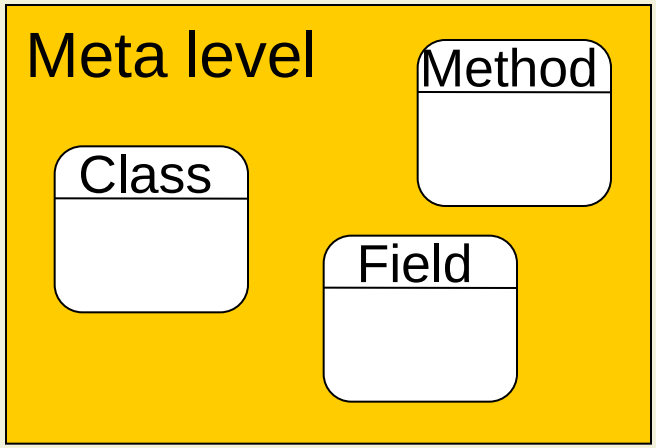
10.1 Introspection

10.2 Reflective Code Generation

10.3 Dynamic Code Manipulation

Motivation

- If code is represented as data, we can as well allow programs to modify the code
- Adapt program dynamically according to user input and execution environment
- Apply systematic changes to programs
  - Code instrumentation



Code Manipulation in Python

- Example: create a case-insensitive string class

```
class iStr( str ):
  def __init__( self, *args ):
    self._lowered = str.lower( self )

def _makeCI( name ):
  theMethod = getattr( str, name )
  def newCImethod( self, other, *args ):
    other = other.lower( )
    return theMethod( self._lowered, other, *args )
  setattr( iStr, name, newCImethod )
```

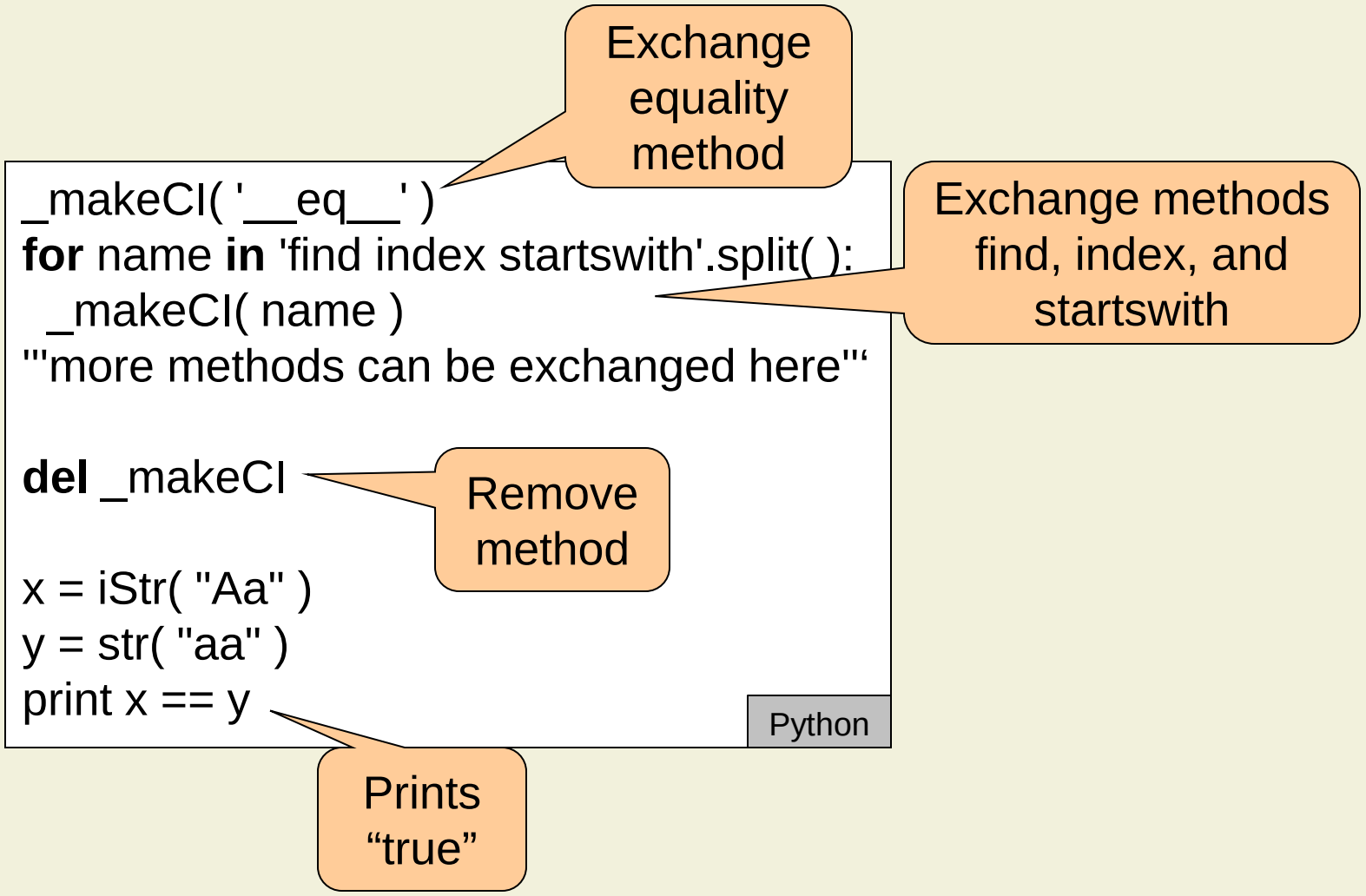
Python

Create a new string class that inherits from str

Method that wraps theMethod

Exchange method implementation in class iStr

Code Manipulation in Python (cont'd)



Reflection and Type Checking

Degree of Reflection	Type Checking
Introspection	Code can be type checked once, during compilation
Reflective code generation	Code can be type checked once, when it is created
Dynamic code manipulation	Typically requires dynamic type checking

Reflection: Summary

Applications

- Flexible architectures (plug-ins)
- Object serialization
- Persistence
- Design patterns
- Dynamic code generation

Drawbacks

- Not statically safe
- May compromise information hiding
- Code is difficult to understand and debug
- Performance