Overview

- Static typing
- Run-Time Type Information in C++
- Reflection in Java
- Dynamic Proxies in Java

- Syntax, uses and misuses for each

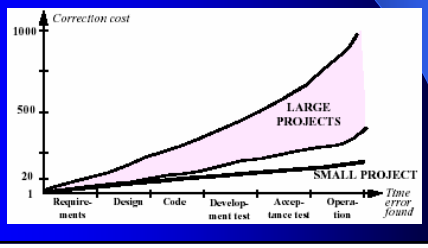
RTTI and Reflection

David Talby

Advantages of Static Typing

- Reliability

  Relative cost of error correction, from Boehm, "Software Engineering Economics", 1981:



Typing

- Static Typing, or Strong Typing:
  - The type of every object and expression must be specified and checked at compile-time
  - Or: All type violations are caught at compile-time
- Dynamic Typing or Weak Typing:
  - Type violations are only detected at run-time
- What is a type violation?
  - Given *C x; x.foo(arg);* C has no method/operator *foo*
  - Or, *arg* is not an acceptable argument for it

Run-Time Type Information

David Talby

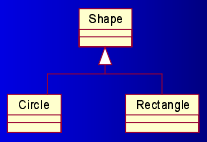
Advantages II

- Readability
  - Part of an interface, like Design by Contract
- Efficiency
  - Constant-time dynamic binding
  - Static binding and inlining
  - Optimized machine-code instructions

- Static Typing ≠ Static Binding
  - Object-oriented language usually use static typing with dynamic binding

Polymorphic Collections

- A *list<Shape*>* is polymorphic
- How to count rectangles, or find max radius?



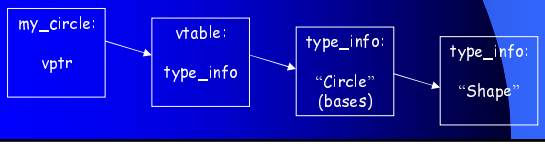
- The language requires a mechanism to test whether a given object is of a given type

Run-Time Type Information

- A mechanism for safely bypassing typing
- Controversial: can be easily abused
- Capabilities (C++):
  - Testing if an object is of a given type
  - Testing if two objects are of the same type
  - Retrieving the name of a type
  - Imposing a full order on types
- C++: *dynamic_cast* and *typeid* operators
- Java: casting and *instanceof* operator

dynamic_cast

- Source type must be polymorphic
  - But not the target type
  - Dynamic casts aren't useful for static types
- This enables an efficient implementation: Hold a pointer to *type_info* in the v-table



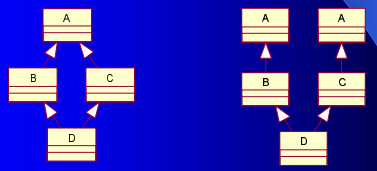
Polymorphic Collections II

```
double max_radius = 0.0;
for (iterator<Shape*> it = list->begin(); it != list->end(); it++)
  if (Circle* c = dynamic_cast<Circle*>(*it))
    max_radius = max(max_radius, c->radius());
```

- In Java: combine *instanceof* and C-style cast
- Cast will succeed for descendants of *Circle*
  - Obeys Liskov Substitution Principle

dynamic_cast III

- Fails if the source object has more than one unique target base class



Virtual inheritance: each D has one A

Ordinary inheritance: each D has two A's

dynamic_cast II

- Can be used for pointers and references
  - If cast to a pointer failed, return 0
  - If cast to a referece failed, throw *bad_cast*
- In Java
  - The *instanceof* always returns a boolean
  - All casts are dynamic (checked): *ClassCastException*
- What about *static_cast*?
  - Equivalent to deprecated C-style casts: *(Circle*)s*
  - More efficient: doesn't examine source object
  - Required to cast from *void**
  - Best avoided - *dynamic_cast* is safer

More Uses for RTTI

- Receiving an object of uncertain content

```
Object o = myObjectStream.readObject();
if (o instanceof Circle) c = (Circle)o;
```

- Inability to alter ancestor classes
  - Let *Lineman*, *Boss*, *BigBoss* inherit *Worker*
  - Everyone gets a bonus except the *Lineman*
  - Best solution: *virtual bonus()* in *Worker*
  - If *Worker* can't be changed (no source, many dependencies) – RTTI is the only solution

The typeid operator

- Returns the *type_info** of an object
- type_info supports comparison
  - Several type_info* may exist in memory for the same type – required for DLLs
  - Compare its objects, not pointers to objects
- type_info has a before() order function
  - Unrelated to inheritance relationships
- type_info has a char* name() function
  - Useful for printing debug messages
  - Useful for keying more per-type data

Misuses of RTTI II

- Using over-generic base classes

```
class List {
  void put(Object o) { ... }
  Object get(int index) { ... }
```

  Instead of:

```
list<Shape*>
```

- This poses several problems
  - Forces casting in source code
  - Less efficient
  - Reduces compiler type-checking
- This is why Java will have generics

Misuses of RTTI

- It's easy to use RTTI to violate basics
  - Single Choice Principle
  - Open-Closed Principle
- Virtual functions are required instead of:

```
void rotate(Shape* s) {
  if (typeid(s) == typeid(Circle))
    // rotate circle algorithm
  else if (typeid(s) == typeid(Rectangle))
    // rotate rectangle algorithm
  else ...
```

RTTI & Casting Guidelines

- There are very few correct uses
  - Polymorphic collections
  - Validation of an received object
  - Compromise, when a class can't be changed
- Usually, casting means a design error
  - Excluding casting between primitive types
  - Excluding the current Java collections
- Static typing also tests your design

Misuses of RTTI III

- Casting instead of using adapters

```
interface Storable { int objectId(); }
interface Book {
  String getName() { ... }
  String getAuthor() { ... }
}
class BookImpl implements Book, Storable { ... }
```

- "Clients should only work with interfaces"
  - Only some clients should know about *objectId()*
- Accessing *objectId()* requires casting
- Violates Liskov Substitution Principle

What is Reflection?

- Library and runtime support for:
  - Creating class instances and arrays
  - Access and modify fields of objects, classes and elements of arrays
  - Invoke methods on objects and classes
- Java is the most widely used example
  - The *java.lang.reflect* package
  - In *java.lang*: classes *Class* and *Object*
- All under the Java security model

Reflection

David Talby

Reflection API – Class II

- class Class also provides:

```
Field getField(String name); // and 'declared' version
Field[] getFields(); // and 'declared' version
Class getDeclaringClass(); // for inner classes
Class[] getInterfaces();
Method getMethod(String name, Class[] params);
Method[] getMethods(); // and 'declared' version
String getName();
String getPackage();
int getModifiers();
```

Reflection API – Class

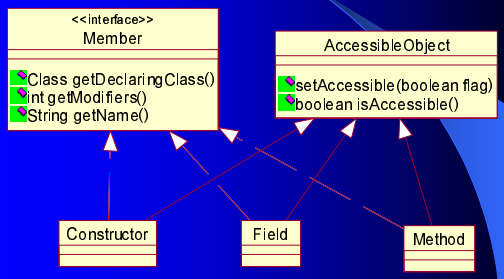
- class *Object* has a *getClass()* method:

```
System.out.println(obj.getClass().getName());
```

- class Class provides:

```
static Class forName(String className);
Class[] getClasses();                  // all inner classes
Class[] getDeclaredClasses();  // excludes inherited
ClassLoader getClassLoader();
Constructor getConstructor(Class[] parameters);
Constructor[] getConstructors();
Constructor[] getDeclaredConstructors();
```

Reflection API - Members



Reflection API – Class III

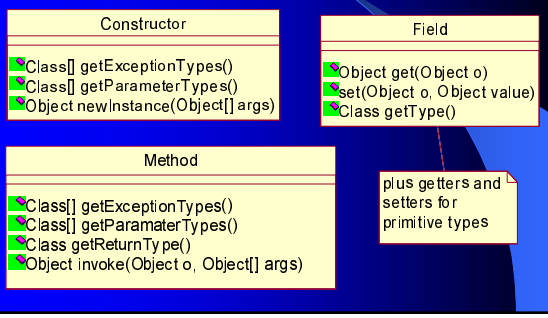
- class Class even provides:

```
Class getSuperClass();
boolean isArray();
boolean isAssignableFrom(Class c);
boolean isInstance(Class c);
boolean isInterface();
boolean isPrimitive();
Object[] getSigners();  // and other security data
String toString();
Object newInstance();  // uses default constructor
```

Reflection API - Others

- class Array
  - Getters and setters by index, *getLength()*
  - *newInstance()* of single- or multi-dimensional
- class Modifier
  - Check return values of *getModifiers()*
- class ReflectPermission
  - Beyond normal access/modify/create checks
  - Currently only supports *SuppressAccessChecks*
- Several exception types (ignored here)

Reflection API – Members II



Object Inspection

- Printing an object fields' names and values

```
Field[] fa = obj.getClass().getFields();
for (int i=0; i < fa.length; i++)
    print(fa[i].getName(); fa[i].get(obj).toString());
```

- Changing a field's value given its name

```
obj.getClass().getField(theFieldName).set(obj,newValue);
```

- Printing an inheritance tree, given a class name

```
Class c = Class.forName(theClassName);
while (c != null) {
    System.out.println(c.getName());
    c = c.getSuperClass();
}
```

What is it good for?

- Development Tools
  - Inspectors of JavaBeans
  - Debuggers, class browsers
- Runtime services
  - Object Serialization
  - Object-relational database mapping
- Frameworks
  - Hooks through a class naming convention
  - Plug-ins and add-ins

Serialization II

- Non-primitive reference objects must be recursively written as XML elements
  - Use *Class.isPrimitive()* to test each field type
- Fields marked as *transient* aren't written
  - Use *Modifier.isTransient(Field.getModifiers())*
- Objects must be rebuilt from XML data
  - Use *Class.forName()* to find object's class, *Class.getField()* to find fields, and *Field.set()*
- Java serialization is implemented this way
  - But writes to a more efficient binary format

Serialization

- Writing an object to XML:

```
Class c = obj.getClass();
StringBuffer output = new StringBuffer();
output.append("<" + c.getName() + ">");
Field[] fa = c.getFields();
for (int i=0; i < fa.length; i++) {
    output.append("<"+fa[i].getName()+">");
    output.append(fa[i].get(obj).toString());
    output.append("</"+fa[i].getName()+">");
}
```

Plug-Ins II

- The weapons array is a list of prototypes
  - Alternative: Hold Class[] array
- Multiple interfaces are easy to support
  - Use *Class.getInterfaces()* on downloaded files
- All *Weapon* code is type-safe
  - And secure, if there's a security policy
- There are better plug-in implementations
  - See class *ClassLoader*
  - Classes can be stored and used from the net

Plug-Ins

- Your new game enables downloading new weapons from the web
- Define an interface for *Weapon*
- Download *.class files of new stuff into:
  - A directory called c:\MyGame\weapons
  - Or a file called c:\MyGame\weapons.jar
  - where c:\MyGame is the home class path
- When the program starts:

```
String[] files = findFileNames(pluginDir + "\*.class");
Weapon[] weapons = new Weapon[files.length];
for (int i=0; i < weapons.length; i++)
    weapons [i] = (Weapon)Class.forName(
                    "weapons."+files[i]).newInstance();
```

Dynamic Proxies

David Talby

Reflection Guidelines

- ☺ Reflection is a new reuse mechanism
- ☹ It's a very expensive one
- Use it when field and class names as strings were necessary anyway
  - Class browsers and debuggers, serialization to files or databases, plug-in class names
- Use it to write very generic frameworks
  - Plug-ins and hooks to be written by others
- Don't use it just to show off…

Invocation Handlers

- Start by defining the handler:
  - interface java.lang.reflect.InvocationHandler
  - With a single method:

```
Object invoke(       // return value of call
    Object proxy,    // call's target
    Method method,   // the method called
    Object[] args)   // method's arguments
```

- The "real" call made: *proxy.method(args)*
  - Simplest invoke(): *method.invoke(proxy,args)*

Dynamic Proxies

- Support for creating classes at runtime
  - Each such class implements interface(s)
  - Every method call to the class will be delegated to a handler, using reflection
  - The created class is a proxy for its handler
- Applications
  - Aspect-Oriented Programming: standard error handling, log & debug for all objects
  - Creating dynamic event handlers

Creating a Proxy Instance

- A proxy class has one constructor which takes one argument – the invocation handler
- Given a proxy class, find and invoke this constructor:

```
Foo foo = (Foo)proxyClass.
  getConstructor(new Class[] { InvocationHandler.class }).
  newInstance(new Object[] { new MyInvocationHandler() });
```

- Class Proxy provides a shortcut:

```
Foo f = (Foo) Proxy.newProxyInstance(
                Foo.class.getClassLoader(),
                new Class[] { Foo.class },
                new MyInvocationHandler());
```

Creating a Proxy Class

- Define the proxy interface:

```
interface Foo { Object bar(Object obj); }
```

- Use java.lang.reflect.Proxy static methods to create the proxy class:

```
Class proxyClass = Proxy.getProxyClass(
  Foo.class.getClassLoader(), new Class[] { Foo.class });
```

- First argument – the new class's class loader
- 2nd argument – list of implemented interfaces
- The expression *C.class* for a class C is the static version of *C_obj.getClass()*

A Debugging Example

- We'll write an extremely generic class, that can wrap any object and print a debug message before and after every method call to it
- Instead of a public constructor, it will have a static factory method to encapsulate the proxy instance creation
- It will use *InvocationTargetException* to be exception-neutral to the debugged object

A Few More Details

- We ignored a bunch of exceptions
  - *IllegalArgumentException* if proxy class can't exist
  - *UndeclaredThrowableException* if the handler throws an exception the interface didn't declare
  - *ClassCastException* if return value type is wrong
  - *InvocationTargetException* wraps checked exceptions
- A proxy class's name is undefined
  - But begins with Proxy$
- Primitive types are wrapped by *Integer*, *Boolean*, and so on for argument and return values
- The syntax is very unreadable!
  - Right, but it can be encapsulated inside the handler

A Debugging Example III

- The invoke() method:

```
public Object invoke(Object proxy, Method m,
                     Object[] args) throws Throwable {
  Object result;
  try {
      System.out.println("before method " + m.getName());
      result = m.invoke(obj, args);
  } catch (InvocationTargetException e) {
      throw e.getTargetException();
  } catch (Exception e) {
      throw new RuntimeException("unexpected:" + e.getMessage());
  } finally {
      System.out.println("after method " + m.getName()); }
  return result; }
```

A Debugging Example II

- The class's definition and construction:

```
public class DebugProxy
implements java.lang.reflect.InvocationHandler {
  private Object obj;
  public static Object newInstance(Object obj) {
    return java.lang.reflect.Proxy.newProxyInstance(
          obj.getClass().getClassLoader(),
          obj.getClass().getInterfaces(),
          new DebugProxy(obj)); }
  private DebugProxy(Object obj) { this.obj = obj; }
```

Dynamic Proxies: Summary

- Applications similar to above example:
  - Log every exception to a file and rethrow it
  - Apply an additional security policy
- Other kinds of applications exist as well
  - Dynamic event listeners in Swing
  - In general, being an observer to many different objects or interfaces at once
- It's a very new feature – from JDK 1.3
  - There may be other future applications

A Debugging Example IV

- Now that the handler is written, it's very simple to use. Just define an interface:

```
interface Foo { Object bar(Object o); }
class FooImpl implements Foo { ... }
```

- And wrap it with a DebugProxy:

```
Foo foo = (Foo)DebugProxy.newInstance(new FooImpl());
```

- This is not much different than using any proxy or decorator
- Just much, much slower