REFLECTION
INTROSPECTION

• We need to begin with a more basic concept called **type introspection**
  
• The ability of a program to examine the type and properties of an object at runtime
  
• A few programming languages have this ability to “introspect”
  
  • Ruby has `kind_of?` and `instance_of?`
  
  • C++ uses `typeid` and `dynamic_cast`

  • Java uses `instanceof` and `java.lang.Class`

```java
if (obj instanceof Person) {
    Person p = (Person)obj;
    p.walk();
}

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

CSE 219, Stony Brook University
Reflection goes one step beyond introspection

Gives the program the ability to
- manipulate the values, meta-data, properties and methods of an object at runtime!
- examine, introspect, and modify its own structure and behavior at runtime!

This is an extremely powerful tool
- can enable applications to perform operations which would otherwise be impossible
- should only be used when you are confident of your grasp on the fundamentals of the language
REFLECTION

• Perform operations which would otherwise be impossible. **Like what?**
  • Call members at runtime that were not even known to exist at compile time!

• Isn’t that just polymorphism?
  • No, polymorphism uses inheritance and knows the overridden method signatures
  • Reflection doesn’t rely on any of that
  • And still manages to ask a class what members it has
  • … and then call the methods, use the fields, etc.
A SIMPLE EXAMPLE OF REFLECTION IN JAVA

• In Java, Class (i.e., java.lang.Class) is a class!
  • No public constructor
  • Instead, Class objects are constructed automatically by the JVM as classes are loaded
  • This gives rise to a “class literal”, which is a special kind of information used by Java to store all the information about the classes available at run time

```java
Method[] methods = MyObject.class.getMethods();
for (Method method : methods) {
    System.out.println("method = " + method.getName());
}
```
USES OF REFLECTION

Extensibility Features
• Dynamically use classes not known at compile time (e.g., plugins and add-ons)

Class Browsers and Visual Development Environments
• A class browser needs to be able to enumerate the members of classes
• Visual development environments can benefit from making use of type information available in reflection to aid the developer in writing correct code
  • e.g., a visual debugger

Debuggers and Test Tools
• examine private members on classes
• watch member values change
DRAWBACKS OF REFLECTION

Performance Overhead
• reflection involves types that are dynamically resolved
  • certain JVM optimizations can not be performed
  • therefore, reflective operations have slower performance than their non-reflective counterparts

Security Restrictions
• requires a runtime permission that may not be present under a security manager
  • important consideration for code which has to run in a restricted security context, such as in an Applet

Exposure of Internals
• allows code to accessing private fields and methods
• may destroy portability and design abstractions
MANIPULATING CLASSES

• Every object is either a reference or primitive type
  • Reference types all inherit from java.lang.Object
  • Classes, enums, arrays, and interfaces are all reference types
  • boolean, byte, short, int, long, char, float, and double are primitive types

• For every type of object, the JVM instantiates an immutable instance of java.lang.Class
  • provides methods to examine the runtime properties of the object including its members and type information
  • Class also provides the ability to create new classes and objects
  • it is the entry point for all of the Reflection APIs

• We will look at
  • Retrieving Class Objects
  • Examining Class Modifiers and Types
  • Discovering Class Members
• The entry point for all reflection operations is java.lang.Class
• There are many ways to get to a Class
  • Object.getClass()
  • .class
  • Class.forName()
  • TYPE field for primitive wrappers like Integer, Double, etc.
RETRIEVING CLASS OBJECTS

• If an instance of an object is available

  • Class c = "foo".getClass(); // String
  • Class c = System.console().getClass(); // java.io.Console
  • enum E { A, B }
    Class c = A.getClass(); // enumeration type E
  • byte[] bytes = new byte[1024];
    Class c = bytes.getClass(); // array with component type ‘byte’

Set<String> s = new HashSet<String>(); // In this case, Set is an interface to an object of type HashSet
Class c = s.getClass();
• The value returned by getClass() is the class corresponding to HashSet
RETRIEVING CLASS OBJECTS

• When instance is not available

    boolean b;
    Class c = b.getClass(); // compile-time error
    Class c = boolean.class; // correct

• If the fully-qualified name of a class is available

    Class c = Class.forName("full.path.MyObject");

• Cannot be used for primitives

• Primitive wrappers

    Class c = Double.TYPE;
A class may have modifiers

- access
  - public, protected, private
- requiring override
  - abstract
- restricting to one instance
  - static
- prohibiting further modification
  - final
• Not all modifiers are allowed on all classes
  • e.g., an interface cannot be final and an enum cannot be abstract
  • `java.lang.reflect.Modifier` contains declarations for all possible modifiers
  • it also contains methods which may be used to decode the set of modifiers returned by `Class.getModifiers()`

• Now let’s look at an example showing how to obtain the declaration components of a class
  • modifiers
  • generic type parameters
  • implemented interfaces
  • the inheritance path
import java.lang.annotation.Annotation;
import java.lang.reflect.Modifier;
import java.lang.reflect.Type;
import java.lang.reflect.TypeVariable;
import java.util.Arrays;
import java.util.ArrayList;
import java.util.List;
import static java.lang.System.out;

public class ClassDeclarationSpy {
    public static void main(String... args) {
        try {
            Class<?> c = Class.forName(args[0]);
            out.format("Class:%n  %s%n%n", c.getCanonicalName());
            out.format("Modifiers:%n  %s%n%n", Modifier.toString(c.getModifiers()));
            out.format("Type Parameters:%n");
            TypeVariable[] tv = c.getTypeParameters();
        }
    }
}
if (tv.length != 0) {
    out.format(" ");
    for (TypeVariable t : tv)
        out.format("%s ", t.getName());
    out.format("%n
");
} else {
    out.format(" -- No Type Parameters --%n
");
}

out.format("Implemented Interfaces:%n");
Type[] intfs = c.getGenericInterfaces();
if (intfs.length != 0) {
    for (Type intf : intfs)
        out.format(" %s%n", intf.toString());
    out.format("%n");
} else
    out.format(" -- No Implemented Interfaces --%n%n");

out.format("Inheritance Path:%n");
List<Class> l = new ArrayList<Class>();
printAncestor(c, l);
if (l.size() != 0) {
    for (Class<?> cl : l)
        out.format("  %s%n", cl.getCanonicalName());
    out.format("%n");
} else {
    out.format("  -- No Super Classes --%n
");
}

out.format("Annotations:%n");
Annotation[] ann = c.getAnnotations();
if (ann.length != 0) {
    for (Annotation a : ann)
        out.format("  %s%n", a.toString());
    out.format("%n");
} else {
    out.format("  -- No Annotations --%n
");
}

// production code should handle this exception more gracefully
} catch (ClassNotFoundException x) { x.printStackTrace(); }
private static void printAncestor(Class<?> c, List<Class> l) {
    Class<?> ancestor = c.getSuperclass();
    if (ancestor != null) {
        l.add(ancestor);
        printAncestor(ancestor, l);
    }
}
$ java ClassDeclarationSpy java.util.concurrent.ConcurrentNavigableMap
Class:
    java.util.concurrent.ConcurrentNavigableMap

Modifiers:
    public abstract interface
Type Parameters:
    K V
Implemented Interfaces:
    java.util.concurrent.ConcurrentMap<K, V>
    java.util.NavigableMap<K, V>

Inheritance Path:
    -- No Super Classes --
Annotations:
    -- No Annotations --