CSCE 314
Programming Languages

Reflection

Dr. Hyunyoung Lee

Reflection and Metaprogramming

- Metaprogramming: Writing (meta)programs that represent and manipulate other programs
- Reflection: Writing (meta)programs that represent and manipulate themselves
- Reflection is metaprogramming

Uses of Reflection

- Configuring programs for specific deployment environment
- Writing debuggers, class browsers, GUI tools, ...
- Optimizing programs for specific inputs
- Analyzing running systems
- Extending running systems
- Of course, compilers, interpreters, program generators, even macros, are examples of metaprograms

Definitions of Reflection

- General:

  Reflection: 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.

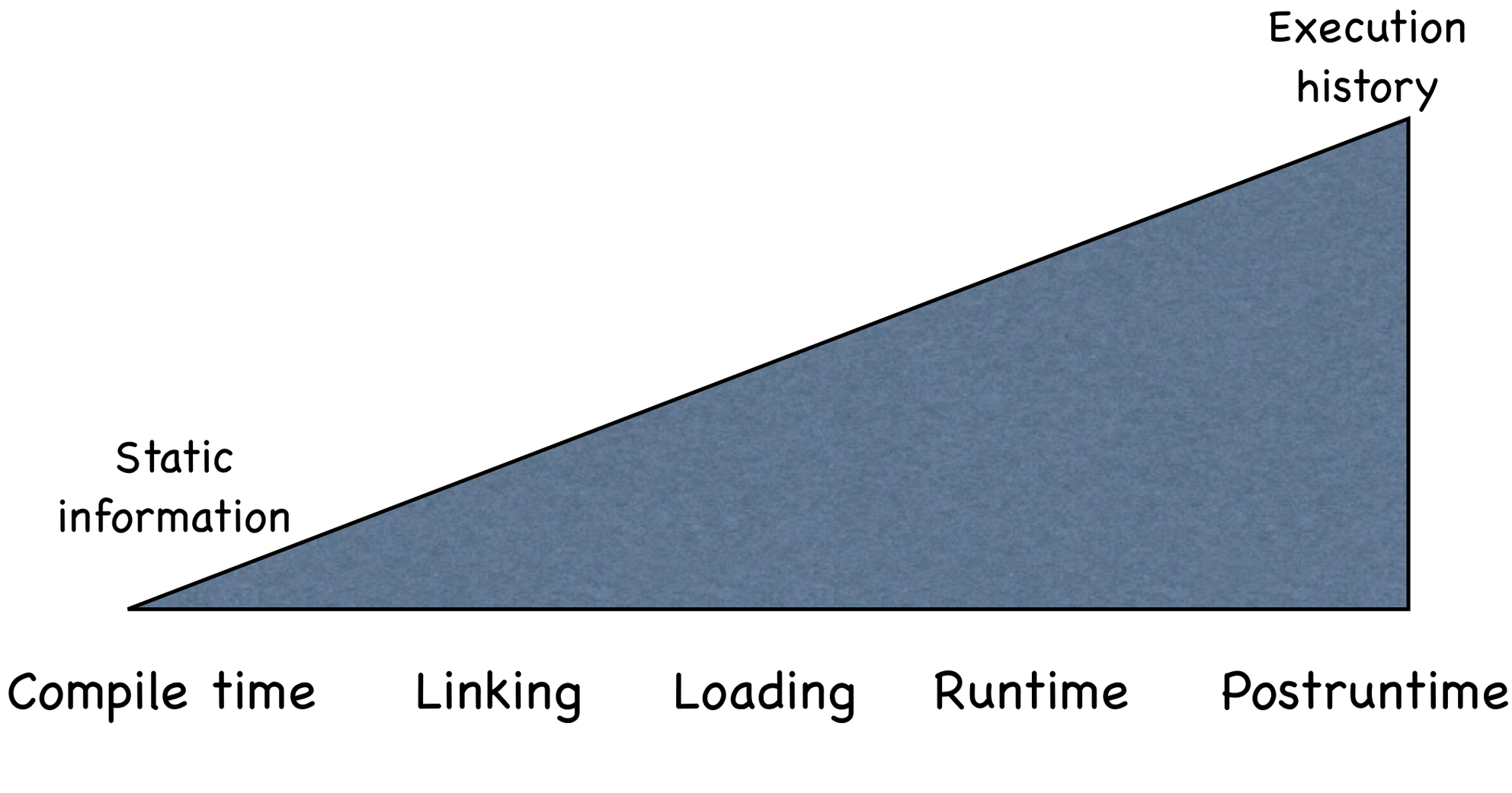
- For computer programs:

  Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession. Introspection is the ability of a program to observe and therefore reason about its own state. Intercession is the ability of a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called reification.

Language Support for Reflection

- Metaobjects: Objects that represent methods, execution stacks, processor
- Extreme: Dynamic languages such as Smalltalk and CLOS
  - Language definitions are largely just libraries
  - Smalltalk:
    - Classes are represented as objects, can be manipulated at run time
    - Class objects are objects as well, and thus can be manipulated, for example, to modify the properties of Smalltalk's object model
    - Regular Smalltalk code can access and modify metaobjects
- Intermediate: Java, C#
  - Java Reflection API: discover methods and attributes at runtime, create objects of classes whose names discovered only at run time
  - Mostly for introspection, no runtime modification of class's metaobjects
- Low: C++ Runtime type identification (RTTI), simple introspection

Why some tasks are delayed until runtime (and why reflection is useful)



The later in the application's life cycle, the more information is available for adapting the program

Java Reflection API

Java's reflection capabilities are offered as a library API, though of course supported by the language

java.lang.Class

java.lang.reflect

Capabilities:

- Knowing a name of a class, load a class file into memory at runtime
- examine methods, fields, constructors, annotations, etc.;
- invoke constructors or methods; and
- access fields.
- Create instances of interfaces dynamically (special API for creating "proxy" classes)

Large API, we touch just a few points

Obtaining a class object

The entry point to reflection operations: java.lang.Class

Various means to obtain an object of Class type

1. From instance (using Object.getClass()):

```
MyClass mc = new Myclass();
Class c = mc.getClass();
```

2. From type:

```
c = boolean.class;
```

3. By name:

```
Class cls = null;
try {
   cls = Class.forName("MyClass");
} catch (ClassNotFoundException e) {
   . . .
}
// use cls
```

Obtaining other classes from a Class object

- Class.getSuperclass()
- Class.getClasses()
  - returns all public class, interface, enum members as an array. Includes inherited members. Example:

```
Class<?>[] c = Character.class.getClasses();
```

- Class.getDeclaredClasses()
  - similar to getClasses() but includes non-public classes and does not recur to base classes
- Class.getEnclosingClass()
  - immediately enclosing class

Class Modifiers

Java's class modifiers

- public, protected, and private
- abstract
- static
- final
- strictfp
- Annotations

Example:

```
class public static MyClass { . . . }
  . . .
int mods = MyClass.class.getModifiers();
if (Modifier.isPublic(mods))
   System.out.println("public class");
```

Accessing Members

- Same set of methods for fields, methods, and constructors. Let X be one of Field, Method, or Constructor:

  X Class.getDeclaredX(String)
  X Class.getX(String)
  X[] Class.getDeclaredXs()
  X[] Class.getXs()

- "Declared" versions obtain private members too
- "non-Declared" versions obtain inherited members
- Method and constructor versions need the parameter types as parameters too:

  Method getDeclaredMethod(String name,
  Class<?> . . . parameterTypes)
  throws NoSuchMethodException, SecurityException

Example 1

```
Class c = Class.forName(className);
System.out.println(c + " {");
int mods;
Field fields[] = c.getDeclaredFields();
for (Field f : fields) {
   if (!Modifier.isPrivate(f.getModifiers()) &&
       !Modifier.isProtected(f.getModifiers()))
       System.out.println("\t" + f);
}
Constructor[] constructors = c.getConstructors();
for (Constructor con : constructors) { System.out.println("\t" + con); }
Method methods[] = c.getDeclaredMethods();
for (Method m : methods) {
   if (!Modifier.isPrivate(m.getModifiers())) { System.out.println("\t" + m);
}
}
System.out.println("}");
```

Example 2. Dynamic Method Invocation

```
// Let this be a method of object x
public void work(int i, String s) {
    System.out.printf("Called: i=%d, s=%s%n", i, s);
}
Class clX = x.getClass();
// To find a method, need array of matching Class types.
Class[] argTypes = { int.class, String.class };
// Find a Method object for the given method.
Method worker = clX.getMethod("work", argTypes);
// To invoke the method, need the invocation arguments, as an Object array.
Object[] theData = { 42, "Chocolate Chips" };
// The last step: invoke the method.
worker.invoke(x, theData);
```

Notes about Reflection

- Reflection can break (intentionally) abstraction boundaries set by type system. For example:
  - Reflection allows access to private fields and methods - The reflection API has means to set security policies to control this
  - It allows attempts to calls to methods that do not exist
- Reflection incurs a performance overhead, because more checking needs to occur at run-time