Java Run Time Type Information (Reflection)

Reflection APIs are used in OOP languages to get information about objects at runtime, and possibly modify them. Not all OOP languages support it (for example, C++ has no reflection API). Reflection is a feature of statically typed languages like Java; dynamically typed languages like Python do not need them.

Determining type of an object

obj_ref instanceof *class_name*
obj_ref instanceof *interface_name*

```
If (account instanceof CheckingAccount){
	//process Checking Account
}
else {
	//process other account type
}
```

Accessing information about classes at Run Time

For every class and instance of a class a program uses, JVM creates an object of the type **Class**. This object contains information about the class, and is itself an instance of a class called **Class**.
You do not create **Class** objects - JVM does it automatically. No constuctors are available for the class **Class**. Instead, use one of the following methods:

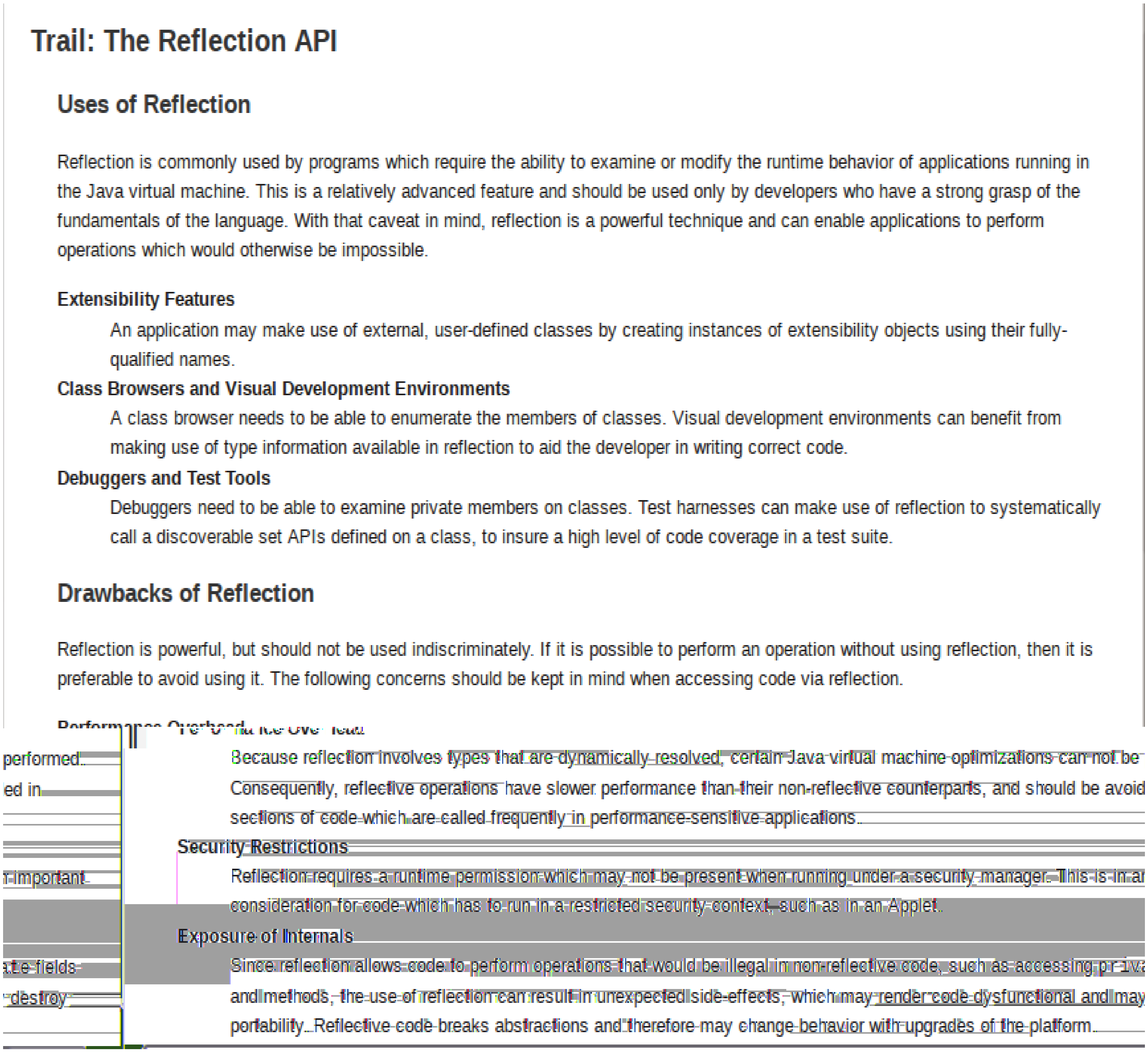
- Use Object.getClass() method for any object. getClass() is public and final.
  ```
  Class classacct = account.getClass();
  ```

- If you know the name of a class or interface, you can use a class variable that Java adds automatically for every class. It is called **class**, and it contains a reference to the corresponding **Class** object.
  ```
  Class classacct = BankAccount.class
  ```

- If you know the full name of the class or interface, use the forName method, passinng the fully qualified class name as a string argument to it:
  ```
  Class classacct = Class.forName("Fall2008.OOP.BankAccount")
  ```

Once you have the **Class** class for an object reference, you can send it messages to obtain meta information about that object.

From: *https://docs.oracle.com/javase/tutorial/reflect/*



Trail: The Reflection API

Uses of Reflection

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. With that caveat in mind, 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 extensibility objects using their fully-qualified names.

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.

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.

Drawbacks of Reflection

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.

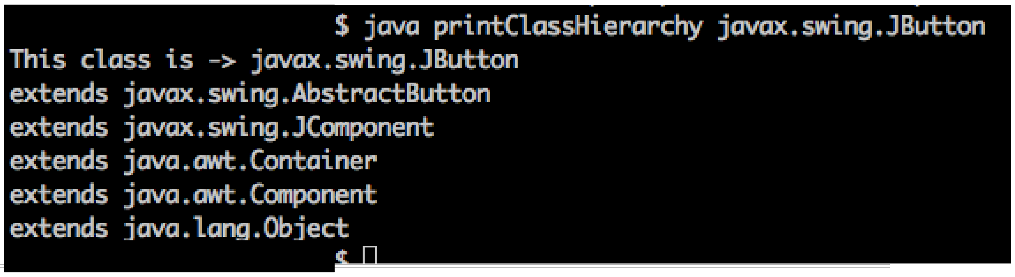
Security Restrictions
: Reflection requires a runtime permission which may not be present when running under a security manager. This is in a[illegible] important consideration for code which has to run in a restricted security context, such as in an Applet.

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.

Examples of using *Class* class

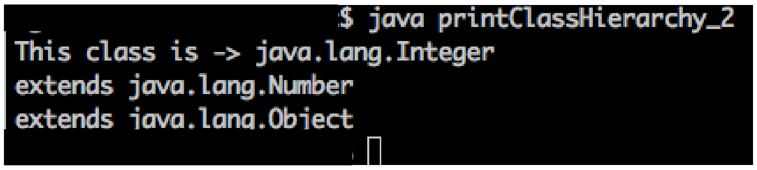
The examples below show three different ways of getting hold of the Class associated with a java class.

```
public class printClassHierarchy{
   public static void main(String[] args)throws Exception{
            Class c1=Class.forName(args[0]);
            System.out.println("This class is -> "+c1.getName());
            while ((c1 = c1.getSuperclass()) != null) {
            System.out.println("extends " + c1.getName());
            }
   }
}
```



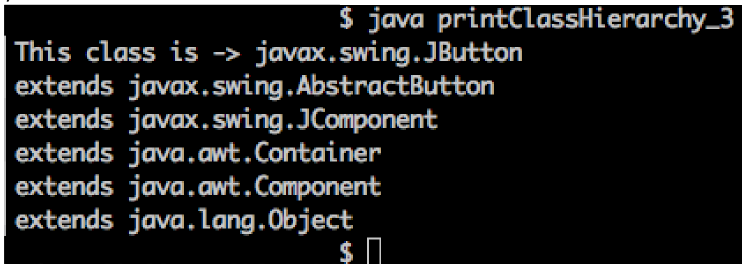
```
$ java printClassHierarchy javax.swing.JButton
This class is -> javax.swing.JButton
extends javax.swing.AbstractButton
extends javax.swing.JComponent
extends java.awt.Container
extends java.awt.Component
extends java.lang.Object
$
```

```
public class printClassHierarchy_2{
   public static void main(String[] args)throws Exception{
        Integer i1=new Integer(12);
        Class c1=i1.getClass();
        System.out.println("This class is -> "+c1.getName());
        while ((c1 = c1.getSuperclass()) != null) {
           System.out.println("extends " + c1.getName());
        }
   }
}
```

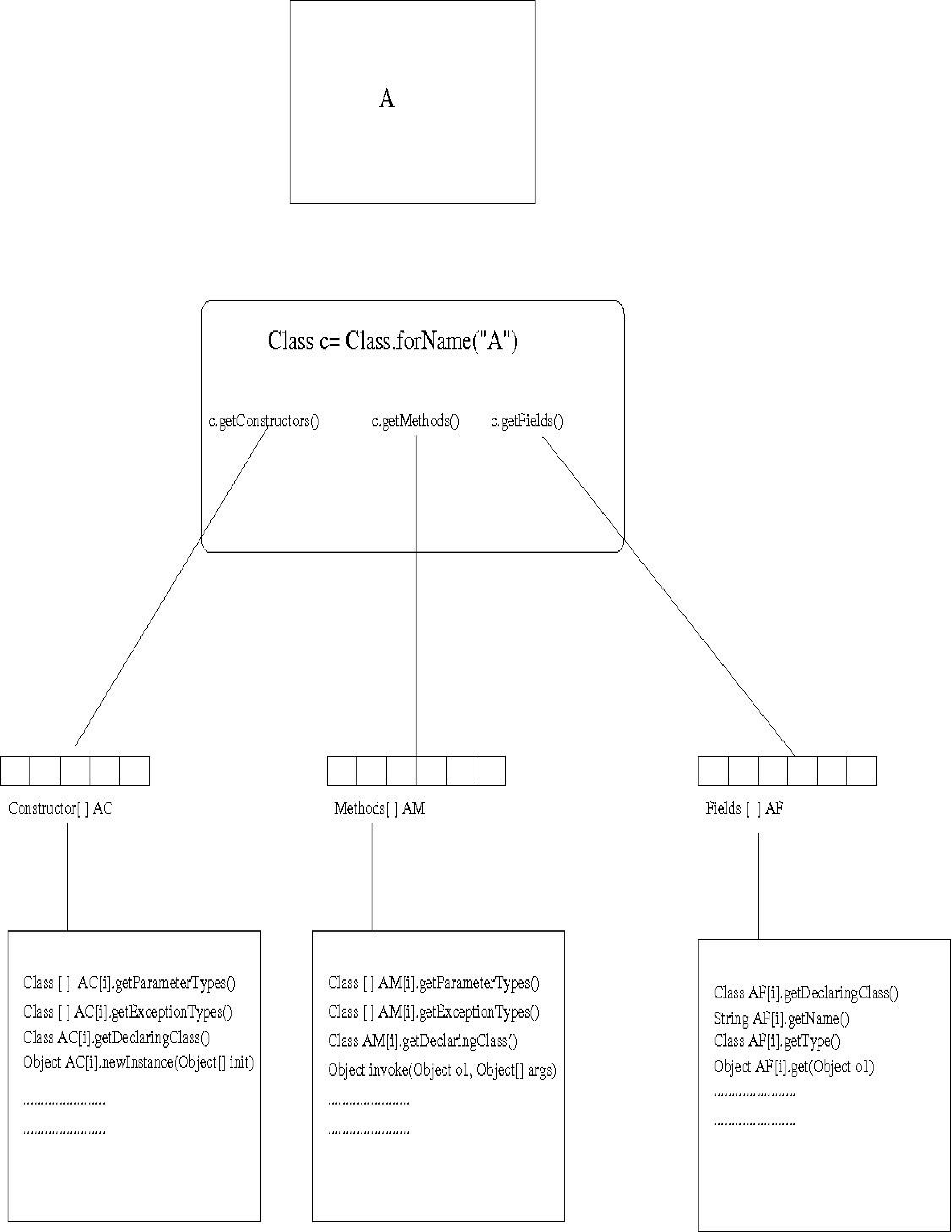


```
$ java printClassHierarchy_2
This class is -> java.lang.Integer
extends java.lang.Number
extends java.lang.Object
```

```
public class printClassHierarchy_3{
   public static void main(String[] args)throws Exception{
      Class c1=javax.swing.JButton.class;
      System.out.println("This class is -> "+c1.getName());
      while ((c1 = c1.getSuperclass()) != null) {
         System.out.println("extends " + c1.getName());
      }
   }
}
```



```
$ java printClassHierarchy_3
This class is -> javax.swing.JButton
extends javax.swing.AbstractButton
extends javax.swing.JComponent
extends java.awt.Container
extends java.awt.Component
extends java.lang.Object
$
```


A
Class c= Class.forName("A")
c.getConstructors()
c.getMethods()
c.getFields()
Constructor[ ] AC
Methods[ ] AM
Fields [ ] AF
Class [ ] AC[i].getParameterTypes()
Class [ ] AC[i].getExceptionTypes()
Class AC[i].getDeclaringClass()
Object AC[i].newInstance(Object[] init)
....
....
Class [ ] AM[i].getParameterTypes()
Class [ ] AM[i].getExceptionTypes()
Class AM[i].getDeclaringClass()
Object invoke(Object o1, Object[] args)
....
....
Class AF[i].getDeclaringClass()
String AF[i].getName()
Class AF[i].getType()
Object AF[i].get(Object o1)
....
....