1. **Reflection API**

Most of the stuff ’stolen’ from The represents, or reflects, the classes, interfaces, and objects in the current Java Virtual Machine.

Typical use:
- debuggers,
- class browsers,
- GUI builders.

With the reflection API you can:
- Determine the class of an object.
- Get information about a class’s modifiers, fields, methods, constructors, and superclasses.
- Find out what constants and method declarations belong to an interface.
- Create an instance of a class whose name is not known until runtime.
- Get and set the value of an object’s field, even if the field name is unknown to your program until runtime.
- Invoke a method on an object, even if the method is not known until runtime.
- Create a new array, whose size and component type are not known until runtime, and then modify the array’s component
1.1. Intro example

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

class Intro {
    static void printName(Object o) {
        Class c = o.getClass();
        String s = c.getName();
        System.out.println(s);
    }

    public static void main(String[] args) {
        String aS = new String();
        printName(aS);
    }
}
```

Result:

% javac Intro.java
% java Intro
java.lang.String
1.2. Discovering Class Modifiers

See

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

public final class MyModifier {

    void printName(Object o) {
        Class c = o.getClass();
        String s = c.getName();
        System.out.println(s);
    }

    public void printModifiers(Object o) {
        Class c = o.getClass();
        int m = c.getModifiers();

        if (Modifier.isPrivate(m))
            System.out.println("private");
        if (Modifier.isPublic(m))
            System.out.println("public");
        if (Modifier.isAbstract(m))
            System.out.println("abstract");
        if (Modifier.isFinal(m))
            System.out.println("final");
    }

    public static void main(String[] args) {
        MyModifier aM = new MyModifier();
        aM.printName(aM);
        aM.printModifiers(aM);
    }
}
```

Result:

```
% java MyModifier
MyModifier
public
final
```
1.3. Identifying Class Fields

Application such as a class browser, might want to find out what fields belong to a particular class. This can be identified by invoking the getFields method on a Class object. The getFields method returns an array of Field objects containing one object per accessible public field.

A public field is accessible if it is a member of either:

- this class
- a superclass of this class
- an interface implemented by this class
- an interface extended from an interface implemented by this class

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

class What {
    public int publicVar;;
    private int privateVar;;
    static int staticVar;;

    static void printFieldNames(Object o) {
        Class c = o.getClass();
        Field[] publicFields = c.getFields();
        for (int i = 0; i < publicFields.length; i++) {
            String fieldName = publicFields[i].getName();
            Class typeClass = publicFields[i].getType();
            String fieldType = typeClass.getName();
            System.out.println("Name: "+fieldName +
            ", Type: " + fieldType);
        }
    }

    public static void main(String[] args) {
        String aS = new String();
        Thread aT = new Thread();
        What aW = new What();
        System.out.println("String: ");
        printFieldNames(aS);
        System.out.println("Thread: ");
        printFieldNames(aT);
        System.out.println("What: ");
        printFieldNames(aW);
    }
}
```

Result:

```bash
% java What
string:
    Name: serialVersionUID, Type: long
    Name: CASE_INSENSITIVE_ORDER, Type: java.util.Comparator
Thread:
    Name: MIN_PRIORITY, Type: int
```
Name: NORM_PRIORITY, Type: int
Name: MAX_PRIORITY, Type: int

What:
Name: publicVar, Type: int
1.4. Getting Values

A development tool such as a debugger, must be able to obtain field values. This is a three-step process:

1. Create a Class object.
2. Create a Field object by invoking getField on the Class object.
3. Invoke one of the get methods on the Field object

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

class Get {
    public int publicVar = 42;
    private int privateVar;
    static int staticVar;

    static void getValue(Object o) {
        Class c = o.getClass();
        Integer value;
        try {
            Field publicVarField = c.getField("publicVar");
            value = (Integer) publicVarField.get(o);
            System.out.println("value: " + value);
        } catch (NoSuchFieldException e) {
            System.out.println(e);
        } catch (SecurityException e) {
            System.out.println(e);
        } catch (IllegalAccessException e) {
            System.out.println(e);
        }
    }

    public static void main(String[] args) {
        Get aG = new Get();
        System.out.println("Get: ");
        getValue(aG);
    }
}
```

% java Get
Get:
value: 42
1.5. Setting Values

Some debuggers allow users to change field values during a debugging session. A tool that has this capability must call one of the Field class’s set methods.

1. To modify the value of a field, perform the following steps: Create a Class object. For more information, see the section Retrieving Class Objects.
2. Create a Field object by invoking getField on the Class object.
3. Class Fields shows you how. Invoke the appropriate set method on the Field object.

The Field class provides several set methods. Specialized methods, such as setBoolean and setInt, are for modifying primitive types. If the field you want to change is an object invoke the set method. It is possible to set to modify a primitive type, but the appropriate wrapper object for the value parameter must be used.

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

class Set {
    public int publicVar = 42;
    private int privateVar;
    static int staticVar;

    static void setValue(Object o) {
        Class c = o.getClass();
        Integer value;
        try {
            Field publicVarField = c.getField("publicVar");
            publicVarField.set(o, new Integer(24));
        } catch (NoSuchFieldException e) {
            System.out.println(e);
        } catch (SecurityException e) {
            System.out.println(e);
        } catch (IllegalAccessException e) {
            System.out.println(e);
        }
    }

    public static void main(String[] args) {
        Set aS = new Set();
        System.out.println("before: aS.publicVar = "+ aS.publicVar);
        setValue(aS);
        System.out.println("after: aS.publicVar = "+ aS.publicVar);
    }
}
```

% java Set
before: aS.publicVar = 42
after: aS.publicVar = 24
1.6. Obtaining Method Information
To find out what public methods belong to a class, invoke the method named getMethods. The array
returned by getMethods contains Method objects. This can be used to uncover a method’s name, return
type, parameter types, set of modifiers, and set of throwable exceptions. All of this information would
be useful to write a class browser or a debugger. A method can be called with Method.invoke.

1. It retrieves an array of Method objects from the Class object by calling getMethods.
2. For every element in the Method array, the program:
   a. retrieves the method name by calling getName
   b. gets the return type by invoking getReturnType
   c. creates an array of Class objects by invoking getParameterTypes
3. The array of Class objects created in the preceding step represents the parameters of the method. To
   retrieve the class name for every one of these parameters, the program invokes getName against each
   Class object in the array.

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

class Show {
    public int publicVar = 42;
    private int privateVar;
    static int staticVar;

    public static int HPclassM(String aString) {
        return 1;
    }

    public int HPobjectM(int i, Boolean aBoolean) {
        return 1;
    }

    void showMethods(Object o) {
        Class c = o.getClass();
        Method[] theMethods = c.getMethods();
        for (int i = 0; i < theMethods.length; i++) {
            String methodString = theMethods[i].getName();
            System.out.println("Name: "+methodString);
            String returnTypeString = theMethods[i].getReturnType().getName();
            System.out.println(" Return Type: " + returnTypeString);
            Class[] parameterTypes = theMethods[i].getParameterTypes();
            System.out.print(" Parameter Types:");
            for (int k = 0; k < parameterTypes.length; k++) {
                String parameterString = parameterTypes[k].getName();
                System.out.print(" " + parameterString);
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        Show aS = new Show();
        System.out.println("Show: ");
        aS.showMethods(aS);
    }
}
```
% java Show
Show:
Name: HPclassM
  Return Type: int
  Parameter Types: java.lang.String
Name: main
  Return Type: void
  Parameter Types: [Ljava.lang.String;
...
1.7. Invoking Methods

A debugger should allow a user to select and then invoke methods during a debugging session. Since it is not know at compile time which methods the user will invoke, the method name cannot be hardcoded in the source code.

1. Create a Class object that corresponds to the object whose method you want to invoke. See the section Retrieving Class Objects for more information.

2. Create a Method object by invoking getMethod on the Class object. The getMethod method has two arguments: a String containing the method name, and an array of Class objects. Each element in the array corresponds to a parameter of the method you want to invoke. For more information on retrieving Method objects, see the section Obtaining Method Information

3. Invoke the method by calling invoke. The invoke method has two arguments: an array of argument values to be passed to the invoked method, and an object whose class declares or inherits the method.

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

class Invoke {
    public int publicVar = 42;
    private int privateVar;
    static int staticVar;

    public String objectM(Boolean aBoolean, Integer i) {
        System.out.println("objectM: i = " + i);
        System.out.println("objectM: aBoolean = " + aBoolean);
        return "Alles Verloren, alles Geboren ...";
    }

    void invokeMethod(Object o) {
        Class c = Invoke.class;
        Class[] parameterTypes = new Class[] {Boolean.class, Integer.class};
        Method objectM;
        Object[] arguments = new Object[] {new Boolean(true), new Integer(2)};
        try {
            objectM = c.getMethod("objectM", parameterTypes);
            System.out.println((String) objectM.invoke((Invoke)o, arguments));
        } catch (NoSuchMethodException e) {
            System.out.println(e);
        } catch (IllegalAccessException e) {
            System.out.println(e);
        } catch (InvocationTargetException e) {
            System.out.println(e);
        }
    }
}

class main {
    public static void main(String[] args) {
        Invoke as = new Invoke();
        System.out.println("Invoke: ");
    }
```
41 aS.invokeMethod(aS);
42 }
43 }
44 }

% java Invoke
Invoke:
objectM: i = 2
objectM: aBoolean = true
Alles Verloren, alles Geboren ...