Reflexió en Java

Què és la reflexió?

La **reflexió** permet al programador inspeccionar i manipular classes i interfaces, així com els seus mètodes i les seves propietats, en temps d'execució, i sense necessitat de conèixer en temps de compilació els noms dels identificadors.

Molts *frameworks* d'alt nivell (com Hibernate o Spring) utilitzen molt la reflexió per permetre l'ús de classes simples POJO (*Plain Old Java Object*) en comptes d'obligar al programador a implementar complexes jerarquies d'herència.

Bona part de la informació que segueix s'ha obtingut de http://java.sun.com

La classe Object

La classe Object és la classe arrel de tota la jerarquia de classes de Java i proporciona mètodes d'utilitat general per a tots els objectes. Tota classe que no hereti d'una altra amb extends és filla directa de Object.

La taula següent recull els mètodes proporcionats per la classe Object.

Class Object: Method Summary	
protected Object	**clone**() Creates and returns a copy of this object. The class `Object` does not itself implement the interface `Cloneable`, so calling the `clone` method on an object whose class is `Object` will result in throwing an exception at run time. The method `clone` for class `Object` performs a specific cloning operation. First, if the class of this object does not implement the interface `Cloneable`, then a `CloneNotSupportedException` is thrown. Note that all arrays are considered to implement the interface `Cloneable`. Otherwise, this method creates a new instance of the class of this object and initializes all its fields with exactly the contents of the corresponding fields of this object, as if by assignment; the contents of the fields are not themselves cloned. Thus, this method performs a "shallow copy" of this object, not a "deep copy" operation.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this one. returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).
protected void	**finalize**() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<? extends Object>	**getClass**() Returns the runtime class of an object.
int	**hashCode**() Returns a hash code value for the object.
void	**notify**() Wakes up a single thread that is waiting on this object's monitor.
void	**notifyAll**() Wakes up all threads that are waiting on this object's monitor.
String	**toString**() Returns a string representation of the object: getClass().getName() + '@' + Integer.toHexString(hashCode())

	It is recommended that all subclasses override this method.
void	**wait**() Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	**wait**(long timeout) Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	**wait**(long timeout, int nanos) Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

Aquests mètodes es poden redefinir per concretar a la classe específica quin és el comportament desitjat.

En concret, és molt convenient redefinir els mètodes toString(), equals() i, en alguns casos, clone().

El mètode getClass() retorna un objecte contenint la informació sobre la classe a la qual pertany l'objecte.

La classe Class

La taula següent reuneix un bon nombre de mètodes subministrats per la classe Class.

Class: Method summary	
<U> Class <? extends U>	**asSubclass**(Class<U>clazz) Casts this Class object to represent a subclass of the class represented by the specified class object.
T	**cast**(Object obj) Casts an object to the class or interface represented by this Class object.
Static Class<?>	**forName**(String className) Returns the Class object associated with the class or interface with the given string name.
Static Class<?>	**forName**(String name, boolean initialize, ClassLoader loader) Returns the Class object associated with the class or interface with the given string name, using the given class loader.
String	**getCanonicalName**() Returns the canonical name of the the underlying class as defined by the Java Language Specification.
Class[]	**getClasses**() Returns an array containing Class objects representing all the public classes and interfaces that are members of the class represented by this Class object.
ClassLoader	**getClassLoader**() Returns the class loader for the class.
Class<?>	**getComponentType**() Returns the Class representing the component type of an array.
Constructor<T>	**getConstructor**(Class... parameterTypes) Returns a Constructor object that reflects the specified public constructor of the class represented by this Class object.
Constructor[]	**getConstructors**() Returns an array containing Constructor objects reflecting all the public constructors of the class represented by this Class object.

Class: Method summary	
Class[]	**getDeclaredClasses**() Returns an array of Class objects reflecting all the classes and interfaces declared as members of the class represented by this Class object.
Constructor<T>	**getDeclaredConstructor**(Class... parameterTypes) Returns a Constructor object that reflects the specified constructor of the class or interface represented by this Class object.
Constructor[]	**getDeclaredConstructors**() Returns an array of Constructor objects reflecting all the constructors declared by the class represented by this Class object.
Field	**getDeclaredField**(String name) Returns a Field object that reflects the specified declared field of the class or interface represented by this Class object.
Field[]	**getDeclaredFields**() Returns an array of Field objects reflecting all the fields declared by the class or interface represented by this Class object.
Method	**getDeclaredMethod**(String name, Class... parameterTypes) Returns a Method object that reflects the specified declared method of the class or interface represented by this Class object.
Method[]	**getDeclaredMethods**() Returns an array of Method objects reflecting all the methods declared by the class or interface represented by this Class object.
Class<?>	**getDeclaringClass**() If the class or interface represented by this Class object is a member of another class, returns the Class object representing the class in which it was declared.
Class<?>	**getEnclosingClass**() Returns the immediately enclosing class of the underlying class.
Constructor<?>	**getEnclosingConstructor**() If this Class object represents a local or anonymous class within a constructor, returns a Constructor object representing the immediately enclosing constructor of the underlying class.
Method	**getEnclosingMethod**() If this Class object represents a local or anonymous class within a method, returns a Method object representing the immediately enclosing method of the underlying class.
T[]	**getEnumConstants**() Returns the elements of this enum class or null if this Class object does not represent an enum type.
Field	**getField**(String name) Returns a Field object that reflects the specified public member field of the class or interface represented by this Class object.
Field[]	**getFields**() Returns an array containing Field objects reflecting all the accessible public fields of the class or interface represented by this Class object.
Type[]	**getGenericInterfaces**() Returns the Types representing the interfaces directly implemented by the class or interface represented by this object.
Type	**getGenericSuperclass**() Returns the Type representing the direct superclass of the entity (class, interface, primitive type or void) represented by this Class.
Class[]	**getInterfaces**() Determines the interfaces implemented by the class or interface represented by this object.
Method	**getMethod**(String name, Class... parameterTypes)

Class: Method summary	
	Returns a Method object that reflects the specified public member method of the class or interface represented by this Class object.
Method[]	**getMethods**() Returns an array containing Method objects reflecting all the public *member* methods of the class or interface represented by this Class object, including those declared by the class or interface and those inherited from superclasses and superinterfaces.
int	**getModifiers**() Returns the Java language modifiers for this class or interface, encoded in an integer.
String	**getName**() Returns the name of the entity (class, interface, array class, primitive type, or void) represented by this Class object, as a String.
Package	**getPackage**() Gets the package for this class.
ProtectionDomain	**getProtectionDomain**() Returns the ProtectionDomain of this class.
URL	**getResource**(String name) Finds a resource with a given name.
InputStream	**getResourceAsStream**(String name) Finds a resource with a given name.
Object[]	**getSigners**() Gets the signers of this class.
String	**getSimpleName**() Returns the simple name of the underlying class as given in the source code.
Class <? super T>	**getSuperclass**() Returns the Class representing the superclass of the entity (class, interface, primitive type or void) represented by this Class.
TypeVariable <Class<T>>[]	**getTypeParameters**() Returns an array of TypeVariable objects that represent the type variables declared by the generic declaration represented by this GenericDeclaration object, in declaration order.
boolean	**isAnonymousClass**() Returns true if and only if the underlying class is an anonymous class.
boolean	**isArray**() Determines if this Class object represents an array class.
boolean	**isAssignableFrom**(Class<?> cls) Determines if the class or interface represented by this Class object is either the same as, or is a superclass or superinterface of, the class or interface represented by the specified Class parameter.
boolean	**isEnum**() Returns true if and only if this class was declared as an enum in the source code.
boolean	**isInstance**(Object obj) Determines if the specified Object is assignment-compatible with the object represented by this Class.
boolean	**isInterface**() Determines if the specified Class object represents an interface type.
boolean	**isLocalClass**() Returns true if and only if the underlying class is a local class.
boolean	**isMemberClass**() Returns true if and only if the underlying class is a member class.
boolean	**isPrimitive**()

Class: Method summary	
	Determines if the specified Class object represents a primitive type.
boolean	**isSynthetic**() Returns true if this class is a synthetic class; returns false otherwise.
T	**newInstance**() Creates a new instance of the class represented by this Class object.
String	**toString**() Converts the object to a string.

Retrieving Class Objects

The entry point for all reflection operations is java.lang.Class. With the exception of java.lang.reflect.ReflectPermission, none of the classes in java.lang.reflect have public constructors. To get to these classes, it is necessary to invoke appropriate methods on Class. There are several ways to get a Class depending on whether the code has access to an object, the name of class, a type, or an existing Class.

Object.getClass()

If an instance of an object is available, then the simplest way to get its Class is to invoke Object.getClass(). Of course, this only works for reference types which all inherit from Object. Some examples follow.

```
Class c = "foo".getClass();
```

Returns the Class for String

```
Class c = System.console().getClass();
```

There is a unique console associated with the virtual machine which is returned by the static method System.console(). The value returned by getClass() is the Class corresponding to java.io.Console.

```
enum E { A, B }
Class c = A.getClass();
```

A is is an instance of the enum E; thus getClass() returns the Class corresponding to the enumeration type E.

```
byte[] bytes = new byte[1024];
Class c = bytes.getClass();
```

Since arrays are Objects, it is also possible to invoke getClass() on an instance of an array. The returned Class corresponds to an array with component type byte.

```
import java.util.HashSet;
import java.util.Set;
Set<String> s = new HashSet<String>();
Class c = s.getClass();
```

In this case, java.util.Set is an interface to an object of type java.util.HashSet. The value returned by getClass() is the class corresponding to java.util.HashSet.

The .class Syntax

If the type is available but there is no instance then it is possible to obtain a Class by appending ".class" to the name of the type. This is also the easiest way to obtain the Class for a primitive type.

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

Note that the statement boolean.getClass() would produce a compile-time error because a boolean is a primitive type and cannot be dereferenced. The .class syntax returns the Class corresponding to the type boolean.

```
Class c = java.io.PrintStream.class;
```

The variable c will be the Class corresponding to the type java.io.PrintStream.

```
Class c = int[][][].class;
```

The .class syntax may be used to retrieve a Class corresponding to a multi-dimensional array of a given type.

Class.forName()

If the fully-qualified name of a class is available, it is possible to get the corresponding Class using the static method Class.forName(). This cannot be used for primitive types. The syntax for names of array classes is described by Class.getName(). This syntax is applicable to references and primitive types.

```
Class c = Class.forName("com.duke.MyLocaleServiceProvider");
```

This statement will create a class from the given fully-qualified name.

```
Class cDoubleArray = Class.forName("[D");
Class cStringArray = Class.forName("[[Ljava.lang.String;");
```

The variable cDoubleArray will contain the Class corresponding to an array of primitive type double (i.e. the same as double[].class). The cStringArray variable will contain the Class corresponding to a two-dimensional array of String (i.e. identical to String[][].class).

TYPE Field for Primitive Type Wrappers

The .class syntax is a more convenient and the preferred way to obtain the Class for a primitive type; however there is another way to acquire the Class. Each of the primitive types and void has a wrapper class in java.lang that is used for boxing of primitive types to reference types. Each wrapper class contains a field named TYPE which is equal to the Class for the primitive type being wrapped.

```
Class c = Double.TYPE;
```

There is a class java.lang.Double which is used to wrap the primitive type double whenever an Object is required. The value of Double.TYPE is identical to that of double.class.

```
Class c = Void.TYPE;
```

Void.TYPE is identical to void.class.

Methods that Return Classes

There are several Reflection APIs which return classes but these may only be accessed if a Class has already been obtained either directly or indirectly.

Class.getSuperclass()

Returns the super class for the given class.

```
Class c = javax.swing.JButton.class.getSuperclass();
```

The super class of javax.swing.JButton is javax.swing.AbstractButton.

Class.getClasses()

Returns all the public classes, interfaces, and enums that are members of the class including inherited members.

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

Character contains two member classes Character.Subset and Character.UnicodeBlock.

Class.getDeclaredClasses()

Returns all of the classes interfaces, and enums that are explicitly declared in this class.

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

Character contains two public member classes Character.Subset and Character.UnicodeBlock and one private class Character.CharacterCache.

Class.getDeclaringClass()
java.lang.reflect.Field.getDeclaringClass()
java.lang.reflect.Method.getDeclaringClass()
java.lang.reflect.Constructor.getDeclaringClass()

Returns the Class in which these members were declared. Anonymous Class Declarations will not have a declaring class but will have an enclosing class.

```
import java.lang.reflect.Field;
Field f = System.class.getField("out");
Class c = f.getDeclaringClass();
```

The field out is declared in System.

```
   public class MyClass {
      static Object o = new Object() {
         public void m() {}
      };
      static Class<c> = o.getClass().getEnclosingClass();
   }
```

The declaring class of the anonymous class defined by o is null.

Class.getEnclosingClass()

Returns the immediately enclosing class of the class.

```
Class c = Thread.State.class().getEnclosingClass();
```

The enclosing class of the enum Thread.State is Thread.

```
public class MyClass {
   static Object o = new Object() {
      public void m() {}
   };
   static Class<c> = o.getClass().getEnclosingClass();
}
```

The anonymous class defined by o is enclosed by MyClass.

Discovering Class Members

There are two categories of methods provided in Class for accessing fields, methods, and constructors: methods which enumerate these members and methods which search for particular members. Also there are distinct methods for accessing members declared directly on the class versus methods which search the superinterfaces and superclasses for inherited members. The following tables provide a summary of all the member-locating methods and their characteristics.

Class Methods for Locating Fields

Class API	**List of members?**	**Inherited members?**	**Private members?**
getDeclaredField()	no	no	yes
getField()	no	yes	no
getDeclaredFields()	yes	no	yes
getFields()	yes	yes	no

Class Methods for Locating Methods

Class API	**List of members?**	**Inherited members?**	**Private members?**
getDeclaredMethod()	no	no	yes
getMethod()	no	yes	no
getDeclaredMethods()	yes	no	yes
getMethods()	yes	yes	no

Class Methods for Locating Constructors

Class API	**List of members?**	**Inherited members?**	**Private members?**
getDeclaredConstructor()	no	N/A[1]	yes
getConstructor()	no	N/A[1]	no
getDeclaredConstructors()	yes	N/A[1]	yes
getConstructors()	yes	N/A[1]	no

[1] Constructors are not inherited.

Given a class name and an indication of which members are of interest, the ClassSpy example uses the get*s() methods to determine the list of all public elements, including any which are inherited.

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Member;
import static java.lang.System.out;

enum ClassMember { CONSTRUCTOR, FIELD, METHOD, CLASS, ALL }

public class ClassSpy {
    public static void main(String... args) {
      try {
          Class<?> c = Class.forName(args[0]);
          out.format("Class:%n  %s%n%n", c.getCanonicalName());

          Package p = c.getPackage();
          out.format("Package:%n  %s%n%n",
                   (p != null ? p.getName() : "-- No Package --"));

          for (int i = 1; i < args.length; i++) {
            switch (ClassMember.valueOf(args[i])) {
            case CONSTRUCTOR:
                printMembers(c.getConstructors(), "Constructor");
                break;
            case FIELD:
                printMembers(c.getFields(), "Fields");
                break;
            case METHOD:
                printMembers(c.getMethods(), "Methods");
                break;
            case CLASS:
                printClasses(c);
                break;
            case ALL:
                printMembers(c.getConstructors(), "Constuctors");
                printMembers(c.getFields(), "Fields");
                printMembers(c.getMethods(), "Methods");
                printClasses(c);
                break;
            default:
                assert false;
            }
          }

        // production code should handle these exceptions more gracefully
      } catch (ClassNotFoundException x) {
          x.printStackTrace();
      }
    }

    private static void printMembers(Member[] mbrs, String s) {
      out.format("%s:%n", s);
      for (Member mbr : mbrs) {
          if (mbr instanceof Field)
            out.format("  %s%n", ((Field)mbr).toGenericString());
          else if (mbr instanceof Constructor)
            out.format("  %s%n", ((Constructor)mbr).toGenericString());
          else if (mbr instanceof Method)
            out.format("  %s%n", ((Method)mbr).toGenericString());
      }
      if (mbrs.length == 0)
          out.format("  -- No %s --%n", s);
      out.format("%n");
    }

    private static void printClasses(Class<?> c) {
      out.format("Classes:%n");
      Class<?>[] clss = c.getClasses();
      for (Class<?> cls : clss)
          out.format("  %s%n", cls.getCanonicalName());
      if (clss.length == 0)
          out.format("  -- No member interfaces, classes, or enums --%n");
      out.format("%n");
    }
  }
```

This example is relatively compact; however the printMembers() method is slightly awkward due to the fact that the java.lang.reflect.Member interface has existed since the earliest implementations of reflection and it could not be modified to include the more useful getGenericString() method when generics were introduced. The only alternatives are to test and cast as shown, replace this method with printConstructors(), printFields(), and printMethods(), or to be satisfied with the relatively spare results of Member.getName().

Samples of the output and their interpretation follows.

```
$ java ClassSpy java.lang.ClassCastException CONSTRUCTOR
Class:
  java.lang.ClassCastException

Package:
  java.lang

Constructor:
  public java.lang.ClassCastException()
  public java.lang.ClassCastException(java.lang.String)
```

Since constructors are not inherited, the exception chaining mechanism constructors (those with a Throwable parameter) which are defined in the immediate super class RuntimeException and other super classes are not found.

```
$ java ClassSpy java.nio.channels.ReadableByteChannel METHOD
Class:
  java.nio.channels.ReadableByteChannel

Package:
  java.nio.channels

Methods:
  public abstract int java.nio.channels.ReadableByteChannel.read
    (java.nio.ByteBuffer) throws java.io.IOException
  public abstract void java.nio.channels.Channel.close() throws
    java.io.IOException
  public abstract boolean java.nio.channels.Channel.isOpen()
```

The interface java.nio.channels.ReadableByteChannel defines read(). The remaining methods are inherited from a super interface. This code could easily be modified to list only those methods that are actually declared in the class by replacing get*s() with getDeclared*s().

```
$ java ClassSpy ClassMember FIELD METHOD
Class:
  ClassMember

Package:
  -- No Package --

Fields:
  public static final ClassMember ClassMember.CONSTRUCTOR
  public static final ClassMember ClassMember.FIELD
  public static final ClassMember ClassMember.METHOD
  public static final ClassMember ClassMember.CLASS
  public static final ClassMember ClassMember.ALL

Methods:
  public static ClassMember ClassMember.valueOf(java.lang.String)
  public static ClassMember[] ClassMember.values()
  public final int java.lang.Enum.hashCode()
  public final int java.lang.Enum.compareTo(E)
  public int java.lang.Enum.compareTo(java.lang.Object)
  public final java.lang.String java.lang.Enum.name()
  public final boolean java.lang.Enum.equals(java.lang.Object)
  public java.lang.String java.lang.Enum.toString()
  public static <T> T java.lang.Enum.valueOf
    (java.lang.Class<T>,java.lang.String)
  public final java.lang.Class<E> java.lang.Enum.getDeclaringClass()
  public final int java.lang.Enum.ordinal()
  public final native java.lang.Class<?> java.lang.Object.getClass()
  public final native void java.lang.Object.wait(long) throws
    java.lang.InterruptedException
  public final void java.lang.Object.wait(long,int) throws
    java.lang.InterruptedException
  public final void java.lang.Object.wait() hrows java.lang.InterruptedException
  public final native void java.lang.Object.notify()
  public final native void java.lang.Object.notifyAll()
```

In the fields portion of these results, enum constants are listed. While these are technically fields, it might be useful to distinguish them from other fields. This example could be modified to use java.lang.reflect.Field.isEnumConstant() for this purpose. The EnumSpy example in a later section of this trail, Examining Enums, contains a possible implementation.

In the methods section of the output, observe that the method name includes the name of the declaring class. Thus, the toString() method is implemented by Enum, not inherited from Object. The code could be amended to make this more obvious by using Field.getDeclaringClass(). The following fragment illustrates part of a potential solution.

```
if (mbr instanceof Field) {
    Field f = (Field)mbr;
    out.format("  %s%n", f.toGenericString());
    out.format("  -- declared in: %s%n", f.getDeclaringClass());
}
```

Obtaining Field Types

A field may be either of primitive or reference type. There are eight primitive types: boolean, byte, short, int, long, char, float, and double. A reference type is anything that is a direct or indirect subclass of java.lang.Object including interfaces, arrays, and enumerated types.

The FieldSpy example prints the field's type and generic type given a fully-qualified binary class name and field name.

```
import java.lang.reflect.Field;
import java.util.List;

public class FieldSpy<T> {
   public boolean[][] b = {{ false, false }, { true, true } };
   public String name  = "Alice";
   public List<Integer> list;
   public T val;

   public static void main(String... args) {
     try {
        Class<?> c = Class.forName(args[0]);
        Field f = c.getField(args[1]);
        System.out.format("Type: %s%n", f.getType());
        System.out.format("GenericType: %s%n", f.getGenericType());

      // production code should handle these exceptions more gracefully
     } catch (ClassNotFoundException x) {
        x.printStackTrace();
     } catch (NoSuchFieldException x) {
        x.printStackTrace();
     }
   }
}
```

Sample output to retrieve the type of the three public fields in this class (b, name, and the parameterized type list), follows. User input is in italics.

```
$ java FieldSpy FieldSpy b
Type: class [[Z
GenericType: class [[Z
$ java FieldSpy FieldSpy name
Type: class java.lang.String
GenericType: class java.lang.String
$ java FieldSpy FieldSpy list
Type: interface java.util.List
GenericType: java.util.List<java.lang.Integer>
$ java FieldSpy FieldSpy val
Type: class java.lang.Object
GenericType: T
```

The type for the field b is two-dimensional array of boolean. The syntax for the type name is described in Class.getName().

The type for the field val is reported as java.lang.Object because generics are implemented via type erasure which removes all information regarding generic types during compilation. Thus T is replaced by the upper bound of the type variable, in this case, java.lang.Object.

Field.getGenericType() will consult the Signature Attribute in the class file if it's present. If the attribute isn't available, it falls back on Field.getType() which was not changed by the introduction of generics. The other methods in reflection with name getGenericFoo for some value of Foo are implemented similarly.

Retrieving and Parsing Field Modifiers

There are several modifiers that may be part of a field declaration:

- Access modifiers: public, protected, and private
- Field-specific modifiers governing runtime behavior: transient and volatile
- Modifier restricting to one instance: static
- Modifier prohibiting value modification: final
- Annotations

The method Field.getModifiers() can be used to return the integer representing the set of declared modifiers for the field. The bits representing the modifiers in this integer are defined in java.lang.reflect.Modifier.

The FieldModifierSpy example illustrates how to search for fields with a given modifier. It also determines whether the located field is synthetic (compiler-generated) or is an enum constant by invoking Field.isSynthetic() and Field.isEnumCostant() respectively.

```
import java.lang.reflect.Field;
import java.lang.reflect.Modifier;
import static java.lang.System.out;

enum Spy { BLACK , WHITE }

public class FieldModifierSpy {
    volatile int share;
    int instance;
    class Inner {}

    public static void main(String... args) {
      try {
          Class<?> c = Class.forName(args[0]);
          int searchMods = 0x0;
          for (int i = 1; i < args.length; i++) {
            searchMods |= modifierFromString(args[i]);
          }

          Field[] flds = c.getDeclaredFields();
          out.format("Fields in Class '%s' containing modifiers:  %s%n",
                   c.getName(),
                   Modifier.toString(searchMods));
          boolean found = false;
          for (Field f : flds) {
            int foundMods = f.getModifiers();
            // Require all of the requested modifiers to be present
            if ((foundMods & searchMods) == searchMods) {
                out.format("%-8s [ synthetic=%-5b enum_constant=%-5b ]%n",
                          f.getName(), f.isSynthetic(),
                          f.isEnumConstant());
                found = true;
            }
          }

          if (!found) {
            out.format("No matching fields%n");
          }
       // production code should handle this exception more gracefully
      } catch (ClassNotFoundException x) {
          x.printStackTrace();
      }
    }

    private static int modifierFromString(String s) {
      int m = 0x0;
      if ("public".equals(s))           m |= Modifier.PUBLIC;
      else if ("protected".equals(s))   m |= Modifier.PROTECTED;
      else if ("private".equals(s))     m |= Modifier.PRIVATE;
      else if ("static".equals(s))      m |= Modifier.STATIC;
      else if ("final".equals(s))       m |= Modifier.FINAL;
      else if ("transient".equals(s))   m |= Modifier.TRANSIENT;
      else if ("volatile".equals(s))    m |= Modifier.VOLATILE;
      return m;
    }
}
```

Sample output follows:

```
$ java FieldModifierSpy FieldModifierSpy volatile
Fields in Class 'FieldModifierSpy' containing modifiers:
volatile share
[ synthetic=false enum_constant=false ]
$ java FieldModifierSpy Spy public
Fields in Class 'Spy' containing modifiers:
public BLACK
[ synthetic=false enum_constant=true ]
WHITE
[ synthetic=false enum_constant=true ]
$ java FieldModifierSpy FieldModifierSpy\$Inner final
Fields in Class 'FieldModifierSpy$Inner' containing modifiers:
final this$0
[ synthetic=true  enum_constant=false ]
$ java FieldModifierSpy Spy private static final
Fields in Class 'Spy' containing modifiers:
private static final $VALUES
[ synthetic=true  enum_constant=false ]
```

Notice that some fields are reported even though they are not declared in the original code. This is because the compiler will generate some synthetic fields which are needed during runtime. To test whether a field is synthetic, the example invokes Field.isSynthetic(). The set of synthetic fields is compiler-dependent; however commonly used fields include this$0 for inner classes (i.e. nested classes that are not static member classes) to reference the outermost enclosing class and $VALUES used by enums to implement the implicitly defined static method values(). The names of synthetic class members are not specified and may not be the same in all compiler implementations or releases. These and other synthetic fields will be included in the array returned by Class.getDeclaredFields() but not identified by Class.getField() since synthetic members are not typically public.

Because Field implements the interface java.lang.reflect.AnnotatedElement, it is possible to retrieve any runtime annotation with java.lang.annotation.RetentionPolicy.RUNTIME. For an example of obtaining annotations see the section Examining Class Modifiers and Types.

Getting and Setting Field Values

Given an instance of a class, it is possible to use reflection to set the values of fields in that class. This is typically done only in special circumstances when setting the values in the usual way is not possible. Because such access usually violates the design intentions of the class, it should be used with the utmost discretion.

The Book class illustrates how to set the values for long, array, and enum field types. Methods for getting and setting other primitive types are described in Field.

```
import java.lang.reflect.Field;
import java.util.Arrays;
import static java.lang.System.out;

enum Tweedle { DEE, DUM }

public class Book {
   public long chapters = 0;
   public String[] characters = { "Alice", "White Rabbit" };
   public Tweedle twin = Tweedle.DEE;

   public static void main(String... args) {
     Book book = new Book();
     String fmt = "%6S:  %-12s = %s%n";

     try {
        Class<?> c = book.getClass();

        Field chap = c.getDeclaredField("chapters");
        out.format(fmt, "before", "chapters", book.chapters);
        chap.setLong(book, 12);
        out.format(fmt, "after", "chapters", chap.getLong(book));

        Field chars = c.getDeclaredField("characters");
        out.format(fmt, "before", "characters",
               Arrays.asList(book.characters));
        String[] newChars = { "Queen", "King" };
        chars.set(book, newChars);
        out.format(fmt, "after", "characters",
               Arrays.asList(book.characters));

        Field t = c.getDeclaredField("twin");
        out.format(fmt, "before", "twin", book.twin);
        t.set(book, Tweedle.DUM);
        out.format(fmt, "after", "twin", t.get(book));

       // production code should handle these exceptions more gracefully
     } catch (NoSuchFieldException x) {
        x.printStackTrace();
     } catch (IllegalAccessException x) {
        x.printStackTrace();
     }
   }
}
```

This is the corresponding output:

```
$ java Book
BEFORE:  chapters     = 0
 AFTER:  chapters     = 12
BEFORE:  characters   = [Alice, White Rabbit]
 AFTER:  characters   = [Queen, King]
BEFORE:  twin         = DEE
 AFTER:  twin         = DUM
```

Note: Setting a field's value via reflection has a certain amount of performance overhead because various operations must occur such as validating access permissions. From the runtime's point of view, the effects are the same, and the operation is as atomic as if the value was changed in the class code directly.

Use of reflection can cause some runtime optimizations to be lost. For example, the following code is highly likely be optimized by a Java virtual machine:

```
int x = 1;
x = 2;
x = 3;
```

Equivalent code using Field.set*() may not.

Obtaining Method Type Information

A method declaration includes the name, modifiers, parameters, return type, and list of throwable exceptions. The java.lang.reflect.Method class provides a way to obtain this information.

The MethodSpy example illustrates how to enumerate all of the declared methods in a given class and retrieve the return, parameter, and exception types for all the methods of the given name.

```
import java.lang.reflect.Method;
import java.lang.reflect.Type;
import static java.lang.System.out;

public class MethodSpy {
    private static final String  fmt = "%24s: %s%n";

    // for the morbidly curious
    <E extends RuntimeException> void genericThrow() throws E {}

    public static void main(String... args) {
      try {
          Class<?> c = Class.forName(args[0]);
          Method[] allMethods = c.getDeclaredMethods();
          for (Method m : allMethods) {
            if (!m.getName().equals(args[1])) {
                continue;
            }
            out.format("%s%n", m.toGenericString());

            out.format(fmt, "ReturnType", m.getReturnType());
            out.format(fmt, "GenericReturnType", m.getGenericReturnType());

            Class<?>[] pType  = m.getParameterTypes();
            Type[] gpType = m.getGenericParameterTypes();
            for (int i = 0; i < pType.length; i++) {
```

```
            out.format(fmt,"ParameterType", pType[i]);
            out.format(fmt,"GenericParameterType", gpType[i]);
         }

         Class<?>[] xType  = m.getExceptionTypes();
         Type[] gxType = m.getGenericExceptionTypes();
         for (int i = 0; i < xType.length; i++) {
            out.format(fmt,"ExceptionType", xType[i]);
            out.format(fmt,"GenericExceptionType", gxType[i]);
         }
       }

      // production code should handle these exceptions more gracefully
    } catch (ClassNotFoundException x) {
       x.printStackTrace();
    }
  }
}
```

Here is the output for Class.getConstructor() which is an example of a method with parameterized types and a variable number of parameters.

```
$ java MethodSpy java.lang.Class getConstructor
public java.lang.reflect.Constructor<T> java.lang.Class.getConstructor
  (java.lang.Class<?>[]) throws java.lang.NoSuchMethodException,
  java.lang.SecurityException
              ReturnType: class java.lang.reflect.Constructor
       GenericReturnType: java.lang.reflect.Constructor<T>
           ParameterType: class [Ljava.lang.Class;
    GenericParameterType: java.lang.Class<?>[]
           ExceptionType: class java.lang.NoSuchMethodException
    GenericExceptionType: class java.lang.NoSuchMethodException
           ExceptionType: class java.lang.SecurityException
    GenericExceptionType: class java.lang.SecurityException
```

This is the actual declaration of the method in source code:

```
public Constructor<T> getConstructor(Class<?>... parameterTypes)
```

First note that the return and parameter types are generic. Method.getGenericReturnType() will consult the Signature Attribute in the class file if it's present. If the attribute isn't available, it falls back on Method.getReturnType() which was not changed by the introduction of generics. The other methods with name getGenericFoo() for some value of Foo in reflection are implemented similarly.

Next, notice that the last (and only) parameter, parameterType, is of variable arity (has a variable number of parameters) of type java.lang.Class. It is represented as a single-dimension array of type java.lang.Class. This can be distinguished from a parameter that is explicitly an array of java.lang.Class by invoking Method.isVarArgs(). The syntax for the returned values of Method.get*Types() is described in Class.getName().

The following example illustrates a method with a generic return type.

```
$ java MethodSpy java.lang.Class cast
public T java.lang.Class.cast(java.lang.Object)
              ReturnType: class java.lang.Object
       GenericReturnType: T
           ParameterType: class java.lang.Object
    GenericParameterType: class java.lang.Object
```

The generic return type for the method Class.cast() is reported as java.lang.Object because generics are implemented via type erasure which removes all information regarding generic types during compilation. The erasure of T is defined by the declaration of Class:

```
public final class Class<T> implements ...
```

Thus T is replaced by the upper bound of the type variable, in this case, java.lang.Object.

The last example illustrates the output for a method with multiple overloads.

```
$ java MethodSpy java.io.PrintStream format
public java.io.PrintStream java.io.PrintStream.format
  (java.util.Locale,java.lang.String,java.lang.Object[])
             ReturnType: class java.io.PrintStream
      GenericReturnType: class java.io.PrintStream
          ParameterType: class java.util.Locale
   GenericParameterType: class java.util.Locale
          ParameterType: class java.lang.String
   GenericParameterType: class java.lang.String
          ParameterType: class [Ljava.lang.Object;
   GenericParameterType: class [Ljava.lang.Object;
public java.io.PrintStream java.io.PrintStream.format
  (java.lang.String,java.lang.Object[])
             ReturnType: class java.io.PrintStream
      GenericReturnType: class java.io.PrintStream
          ParameterType: class java.lang.String
   GenericParameterType: class java.lang.String
          ParameterType: class [Ljava.lang.Object;
   GenericParameterType: class [Ljava.lang.Object;
```

If multiple overloads of the same method name are discovered, they are all returned by Class.getDeclaredMethods(). Since format() has two overloads (with with a Locale and one without), both are shown by MethodSpy.

Retrieving and Parsing Method Modifiers

There a several modifiers that may be part of a method declaration:

- Access modifiers: public, protected, and private
- Modifier restricting to one instance: static
- Modifier prohibiting value modification: final
- Modifier requiring override: abstract
- Modifier preventing reentrancy: synchronized
- Modifier indicating implementation in another programming language: native
- Modifier forcing strict floating point behavior: strictfp
- Annotations

The MethodModifierSpy example lists the modifiers of a method with a given name. It also displays whether the method is synthetic (compiler-generated), of variable arity, or a bridge method (compiler-generated to support generic interfaces).

```
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import static java.lang.System.out;

public class MethodModifierSpy {

    private static int count;
    private static synchronized void inc() { count++; }
    private static synchronized int cnt() { return count; }

    public static void main(String... args) {
      try {
          Class<?> c = Class.forName(args[0]);
          Method[] allMethods = c.getDeclaredMethods();
          for (Method m : allMethods) {
            if (!m.getName().equals(args[1])) {
                continue;
            }
            out.format("%s%n", m.toGenericString());
            out.format("  Modifiers:  %s%n",
                       Modifier.toString(m.getModifiers()));
            out.format("  [ synthetic=%-5b var_args=%-5b bridge=%-5b ]%n",
                       m.isSynthetic(), m.isVarArgs(), m.isBridge());
            inc();
          }
          out.format("%d matching overload%s found%n", cnt(),
                     (cnt() == 1 ? "" : "s"));

        // production code should handle this exception more gracefully
      } catch (ClassNotFoundException x) {
          x.printStackTrace();
      }
    }
}
```

A few examples of the output MethodModifierSpy produces follow.

```
$ java MethodModifierSpy java.lang.Object wait
public final void java.lang.Object.wait() throws java.lang.InterruptedException
  Modifiers:  public final
  [ synthetic=false var_args=false bridge=false ]
public final void java.lang.Object.wait(long,int)
  throws java.lang.InterruptedException
  Modifiers:  public final
  [ synthetic=false var_args=false bridge=false ]
public final native void java.lang.Object.wait(long)
  throws java.lang.InterruptedException
  Modifiers:  public final native
  [ synthetic=false var_args=false bridge=false ]
3 matching overloads found
```

```
$ java MethodModifierSpy java.lang.StrictMath toRadians
public static double java.lang.StrictMath.toRadians(double)
  Modifiers:  public static strictfp
  [ synthetic=false var_args=false bridge=false ]
1 matching overload found
```

```
$ java MethodModifierSpy MethodModifierSpy inc
private synchronized void MethodModifierSpy.inc()
  Modifiers: private synchronized
  [ synthetic=false var_args=false bridge=false ]
1 matching overload found
```

```
$ java MethodModifierSpy java.lang.Class getConstructor
public java.lang.reflect.Constructor<T> java.lang.Class.getConstructor
  (java.lang.Class<T>[]) throws java.lang.NoSuchMethodException,
  java.lang.SecurityException
  Modifiers: public transient
  [ synthetic=false var_args=true bridge=false ]
1 matching overload found
```

```
$ java MethodModifierSpy java.lang.String compareTo
public int java.lang.String.compareTo(java.lang.String)
  Modifiers: public
  [ synthetic=false var_args=false bridge=false ]
public int java.lang.String.compareTo(java.lang.Object)
  Modifiers: public volatile
  [ synthetic=true  var_args=false bridge=true  ]
2 matching overloads found
```

Note that Method.isVarArgs() returns true for Class.getConstructor(). This indicates that the method declaration looks like this:

```
public Constructor<T> getConstructor(Class<?>... parameterTypes)
```

not like this:

```
public Constructor<T> getConstructor(Class<?> [] parameterTypes)
```

Notice that the output for String.compareTo() contains two methods. The method declared in String.java:

```
public int compareTo(String anotherString);
```

and a second synthetic or compiler-generated bridge method. This occurs because String implements the parameterized interface Comparable. During type erasure, the argument type of the inherited method Comparable.compareTo() is changed from java.lang.Object to java.lang.String. Since the parameter types for the compareTo methods in Comparable and String no longer match after erasure, overriding can not occur. In all other circumstances, this would produce a compile-time error because the interface is not implemented. The addition of the bridge method avoids this problem.

Method implements java.lang.reflect.AnnotatedElement. Thus any runtime annotations with java.lang.annotation.RetentionPolicy.RUNTIME may be retrieved. For an example of obtaining annotations see the section Examining Class Modifiers and Types.

Invoking Methods

Reflection provides a means for invoking methods on a class. Typically, this would only be necessary if it is not possible to cast an instance of the class to the desired type in non-reflective code. Methods are invoked with java.lang.reflect.Method.invoke(). The first argument is the object instance on which this particular method is to be invoked. (If the method is static, the first argument should be null.) Subsequent arguments are the method's parameters. If the underlying method throws an exception, it will be wrapped by an java.lang.reflect.InvocationTargetException. The method's original exception may be retrieved using the exception chaining mechanism's InvocationTargetException.getCause() method.

Finding and Invoking a Method with a Specific Declaration

Consider a test suite which uses reflection to invoke private test methods in a given class. The Deet example searches for public methods in a class which begin with the string "test", have a boolean return type, and a single Locale parameter. It then invokes each matching method.

```
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Type;
import java.util.Locale;
import static java.lang.System.out;
import static java.lang.System.err;

public class Deet<T> {
    private boolean testDeet(Locale l) {
      // getISO3Language() may throw a MissingResourceException
      out.format("Locale = %s, ISO Language Code = %s%n", l.getDisplayName(),
l.getISO3Language());
      return true;
    }

    private int testFoo(Locale l) { return 0; }
    private boolean testBar() { return true; }

    public static void main(String... args) {
      if (args.length != 4) {
          err.format("Usage: java Deet <classname> <langauge> <country>
<variant>%n");
          return;
      }

      try {
          Class<?> c = Class.forName(args[0]);
          Object t = c.newInstance();

          Method[] allMethods = c.getDeclaredMethods();
          for (Method m : allMethods) {
            String mname = m.getName();
            if (!mname.startsWith("test")
                || (m.getGenericReturnType() != boolean.class)) {
                continue;
            }
            Type[] pType = m.getGenericParameterTypes();
            if ((pType.length != 1)
                || Locale.class.isAssignableFrom(pType[0].getClass())) {
                continue;
            }

            out.format("invoking %s()%n", mname);
            try {
                m.setAccessible(true);
                Object o = m.invoke(t, new Locale(args[1], args[2], args[3]));
                out.format("%s() returned %b%n", mname, (Boolean) o);
            // Handle any exceptions thrown by method to be invoked.
            } catch (InvocationTargetException x) {
                Throwable cause = x.getCause();
                err.format("invocation of %s failed: %s%n",
                           mname, cause.getMessage());
            }
          }

        // production code should handle these exceptions more gracefully
      } catch (ClassNotFoundException x) {
          x.printStackTrace();
      } catch (InstantiationException x) {
          x.printStackTrace();
      } catch (IllegalAccessException x) {
          x.printStackTrace();
      }
    }
}
```

Deet invokes getDeclaredMethods() which will return all methods explicitly declared in the class. Also, Class.isAssignableFrom() is used to determine whether the parameters of the located method are compatible with the desired invocation. Technically the code could have tested whether the following statement is true since Locale is final:

```
Locale.class == pType[0].getClass()
```

However, Class.isAssignableFrom() is more general.

```
$ java Deet Deet ja JP JP
invoking testDeet()
Locale = Japanese (Japan,JP),
ISO Language Code = jpn
testDeet() returned true
```

```
$ java Deet Deet xx XX XX
invoking testDeet()
invocation of testDeet failed:
Couldn't find 3-letter language code for xx
```

First, note that only testDeet() meets the declaration restrictions enforced by the code. Next, when testDeet() is passed an invalid argument it throws an unchecked java.util.MissingResourceException. In reflection, there is no distinction in the handling of checked versus unchecked exceptions. They are all wrapped in an InvocationTargetException

Invoking Methods with a Variable Number of Arguments

Method.invoke() may be used to pass a variable number of arguments to a method. The key concept to understand is that methods of variable arity are implemented as if the variable arguments are packed in an array.

The InvokeMain example illustrates how to invoke the main() entry point in any class and pass a set of arguments determined at runtime.

```
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.Arrays;
```

```
public class InvokeMain {
   public static void main(String... args) {
    try {
       Class<?> c = Class.forName(args[0]);
       Class[] argTypes = new Class[] { String[].class };
       Method main = c.getDeclaredMethod("main", argTypes);
       String[] mainArgs = Arrays.copyOfRange(args, 1, args.length);
       System.out.format("invoking %s.main()%n", c.getName());
       main.invoke(null, (Object)mainArgs);

      // production code should handle these exceptions more gracefully
    } catch (ClassNotFoundException x) {
       x.printStackTrace();
    } catch (NoSuchMethodException x) {
       x.printStackTrace();
    } catch (IllegalAccessException x) {
       x.printStackTrace();
    } catch (InvocationTargetException x) {
       x.printStackTrace();
    }
   }
}
```

First, to find the main() method the code searches for a class with the name "main" with a single parameter that is an array of String Since main() is static, null is the first argument to Method.invoke(). The second argument is the array of arguments to be passed.

```
$ java InvokeMain Deet Deet ja JP JP
invoking Deet.main()
invoking testDeet()
Locale = Japanese (Japan,JP),
ISO Language Code = jpn
testDeet() returned true
```

Creating New Class Instances

There are two reflective methods for creating instances of classes: java.lang.reflect.Constructor.newInstance() and Class.newInstance(). The former is preferred and is thus used in these examples because:

- Class.newInstance() can only invoke the zero-argument constructor, while Constructor.newInstance() may invoke any constructor, regardless of the number of parameters.
- Class.newInstance() throws any exception thrown by the constructor, regardless of whether it is checked or unchecked. InvocationTargetException.
- Class.newInstance() requires that the constructor be visible; Constructor.newInstance() may invoke private constructors under certain circumstances.

Sometimes it may be desirable to retrieve internal state from an object which is only set after construction. Consider a scenario where it is necessary to obtain the internal character set used by java.io.Console. (The Console character set is stored in an private field and is not necessarily the same as the Java virtual machine default character set returned by java.nio.charset.Charset.defaultCharset()). The ConsoleCharset example shows how this might be achieved:

```
import java.io.Console;
import java.nio.charset.Charset;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import static java.lang.System.out;

public class ConsoleCharset {
  public static void main(String... args) {
   Constructor[] ctors = Console.class.getDeclaredConstructors();
   Constructor ctor = null;
   for (int i = 0; i < ctors.length; i++) {
      ctor = ctors[i];
      if (ctor.getGenericParameterTypes().length == 0)
        break;
   }

   try {
      ctor.setAccessible(true);
      Console c = (Console)ctor.newInstance();
      Field f = c.getClass().getDeclaredField("cs");
      f.setAccessible(true);
      out.format("Console charset         :  %s%n", f.get(c));
      out.format("Charset.defaultCharset():  %s%n",
             Charset.defaultCharset());

    // production code should handle these exceptions more gracefully
   } catch (InstantiationException x) {
      x.printStackTrace();
   } catch (InvocationTargetException x) {
      x.printStackTrace();
   } catch (IllegalAccessException x) {
      x.printStackTrace();
   } catch (NoSuchFieldException x) {
      x.printStackTrace();
   }
  }
}
```

Note:

Class.newInstance() will only succeed if the constructor is has zero arguments and is already accessible. Otherwise, it is necessary to use Constructor.newInstance() as in the above example.

Example output for a Unix system:

```
$ java ConsoleCharset
Console charset          :  ISO-8859-1
Charset.defaultCharset() :  ISO-8859-1
```

Example output for a Windows system:

```
C:\> java ConsoleCharset
Console charset          :  IBM437
Charset.defaultCharset() :  windows-1252
```

Another common application of Constructor.newInstance() is to invoke constructors which take arguments. The RestoreAliases example finds a specific single-argument constructor and invokes it:

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import static java.lang.System.out;

class EmailAliases {
   private Set<String> aliases;
   private EmailAliases(HashMap<String, String> h) {
    aliases = h.keySet();
   }

   public void printKeys() {
    out.format("Mail keys:%n");
    for (String k : aliases)
       out.format("  %s%n", k);
   }
}

public class RestoreAliases {
   private static Map<String, String> defaultAliases = new HashMap<String,
String>();
   static {
    defaultAliases.put("Duke", "duke@i-love-java");
    defaultAliases.put("Fang", "fang@evil-jealous-twin");
   }
   public static void main(String... args) {
    try {
       Constructor ctor =
EmailAliases.class.getDeclaredConstructor(HashMap.class);
       ctor.setAccessible(true);
       EmailAliases email = (EmailAliases)ctor.newInstance(defaultAliases);
       email.printKeys();
      // production code should handle these exceptions more gracefully
    } catch (InstantiationException x) {
       x.printStackTrace();
    } catch (IllegalAccessException x) {
       x.printStackTrace();
    } catch (InvocationTargetException x) {
       x.printStackTrace();
    } catch (NoSuchMethodException x) {
       x.printStackTrace();
    }
   }
}
```

This example uses Class.getDeclaredConstructor() to find the constructor with a single argument of type java.util.HashMap. Note that it is sufficient to pass HashMap.class since the parameter to any get*Constructor() method requires a class only for type purposes. Due to type erasure, the following expression evaluates to true:

```
HashMap.class == defaultAliases.getClass()
```

The example then creates a new instance of the class using this constructor with Constructor.newInstance().

```
$ java RestoreAliases
Mail keys:
  Duke
  Fang
```