Reflection API

What is Reflection?

In computer science, **reflection** is the ability of a computer program to examine and modify the structure and behavior (specifically the values, metadata, properties and functions) of the program at runtime.

From Wikipedia, the free encyclopedia

Reflection is the ability of a running program to examine itself and its software environment, and to change what it does depending on what it finds. To perform this self-examination, a program needs to have a representation of itself. This information we call **metadata**. In an object-oriented world, metadata is organized into objects, called **metaobjects**. The runtime self-examination of the metaobjects is called **introspection**.

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.

Extensibility Features - An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names (eg. frameworks).

Class Browsers and Visual Developmenat 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.

Uses of Reflection

1. **JUnit** – uses reflection to parse @Test annotation to get the test methods and then invoke it.
2. **Spring** – dependency injection.
3. **Eclipse** auto completion of method names

Drawbacks of 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 an 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.

Classes

Introduction

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. There is a fixed set of primitive types: boolean, byte, short, int, long, char, float, and double. Examples of reference types include `java.lang.String`, all of the wrapper classes for primitive types such as `java.lang.Double`, the interface `java.io.Serializable`, and the enum `javax.swing.SortOrder`.

For every type of object, the Java virtual machine instantiates an immutable instance of `java.lang.Class` which 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. Most importantly, it is the entry point for all of the Reflection APIs.

Retrieving Class Object

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();
System.out.println(c);
```

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();
System.out.println(c);
```

Returns the `Class` for `String`

```
class java.lang.String
```

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();
System.out.println(c);
```

```
class java.lang.String
```

```
Class d = System.console().getClass();
System.out.println(d);
```

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();
System.out.println(c);
```

```
class java.lang.String
```

```
Class d = System.console().getClass();
System.out.println(d);
```

```
class java.io.Console
```

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`.

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();
System.out.println(c);
```

```
class java.lang.String
```

```
Class d = System.console().getClass();
System.out.println(d);
```

```
class java.io.Console
```

```
enum E {A, B}
Class e = E.A.getClass();
System.out.println(e);
```

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();
System.out.println(c);
```

```
class java.lang.String
```

```
Class d = System.console().getClass();
System.out.println(d);
```

```
class java.io.Console
```

```
enum E {A, B}
Class e = E.A.getClass();
System.out.println(e);
```

```
class E
```

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

Object.getClass()

```
byte[] bytes = new byte[1024];
Class f = bytes.getClass();
System.out.println(f);
```

```
class [B
```

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 g = s.getClass();
System.out.println(g);
```

```
class java.util.HashSet
```

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 d = boolean.class;  // correct
```

```
boolean
```

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.

The `.class` Syntax

```
Class c = java.io.PrintStream.class;
System.out.println(c);
```

```
class java.io.PrintStream
```

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

```
Class c = int[][][].class;
System.out.println(c);
```

```
class [[[I
```

The `.class` syntax may be used to retrieve a `Class` corresponding to a multidimensional 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");
System.out.println(c);
```

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

Class.forName()

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

Output:

```
class [D
class [[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;
System.out.println(c);
```

```
double
```

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`.

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;
System.out.println(c);
```

```
double
```

```
Class c = Void.TYPE;
System.out.println(c);
```

```
void
```

`Void.TYPE` is identical to `void.class`.

Methods that Return Classes

`Class.getSuperclass()`

Returns the superclass for the given class.

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

The superclass of `javax.swing.JButton` is `javax.swing.AbstractButton`.

Methods that Return Classes

`Class.getSuperclass()`

Returns the superclass for the given class.

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

The superclass 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 three member classes `Character.Subset`, `Character.UnicodeBlock` and `Character.UnicodeScript`.

Methods that Return Classes

`Class.getDeclaredClasses()`

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

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

`Character` contains three member classes `Character.Subset`, `Character.UnicodeBlock` and `Character.UnicodeScript` and one private class `Character.CharacterCache`.

Methods that Return Classes

`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.

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

The field `out` is declared in `System`.

Examining Class Modifiers and Types

A class may be declared with one or more modifiers which affect its runtime behavior:

- Access modifiers: `public`, `protected`, and `private`
- Modifier requiring override: `abstract`
- Modifier restricting to one instance: `static`
- Modifier prohibiting value modification: `final`

Examining Class Modifiers and Types

Not all modifiers are allowed on all classes, for example 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()`.

The next example shows how to obtain the declaration components of a class including the modifiers, generic type parameters, implemented interfaces, and the inheritance path. Since `Class` implements the `java.lang.reflect.AnnotatedElement` interface it is also possible to query the runtime annotations.

Examining Class Modifiers and Types

```
import java.lang.annotation.Annotation;
import java.lang.reflect.Modifier;
import java.lang.reflect.Type;
import java.lang.reflect.TypeVariable;
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()));
```

Examining Class Modifiers and Types

```
$ java ClassDeclarationSpy java.util.concurrent.ConcurrentNavigableMap

Class:
  java.util.concurrent.ConcurrentNavigableMap

Modifiers:
  public abstract interface
```

```
import java.lang.annotation.Annotation;
import
import
import
import
import
import
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()));
```

Examining Class Modifiers and Types

```
$ java ClassDeclarationSpy java.util.concurrent.ConcurrentNavigableMap

Class:
  java.util.concurrent.ConcurrentNavigableMap

Modifiers:
```

This is the declaration for `java.util.concurrent.ConcurrentNavigableMap` in the code:

```
public interface ConcurrentNavigableMap<K,V>
    extends ConcurrentMap<K,V>, NavigableMap<K,V>
```

Note that since this is an interface, it is implicitly `abstract`. The compiler adds this modifier for every interface.

```
out.format(
                                    Modifier.toString(c.getModifiers()));
```

Examining Class Modifiers and Types

```
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%n");
} else {
        out.format("  -- No Type Parameters --%n%n");
}
```

Examining Class Modifiers and Types

This is the declaration for java.util.concurrent.ConcurrentNavigableMap in the code:

```
public interface ConcurrentNavigableMap<K,V>
    extends ConcurrentMap<K,V>, NavigableMap<K,V>
```

```
out.format("Type Parameters:%n");
TypeVariable[] tv = c.getTypeParameters();
if (tv.length != 0) {
        out.format("  ");
        for (TypeVariable t : tv)
                out.f
        out.for
} else {
        out.form
}
```

```
Type Parameters:
  K V
```

This declaration contains two generic type parameters, K and V. The example code simply prints the names of these parameters, but is it possible to retrieve additional information about them using methods in java.lang.reflect.TypeVariable.

```
%n");
```

Examining Class Modifiers and Types

```
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");
}
```

Examining Class Modifiers and Types

This is the declaration for java.util.concurrent.ConcurrentNavigableMap in the code:

```
public interface ConcurrentNavigableMap<K,V>
    extends ConcurrentMap<K,V>, NavigableMap<K,V>
```

```
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");

                                -- No Implemented Interfaces --%n%n");
```

```
Implemented Interfaces:
  java.util.concurrent.ConcurrentMap<K, V>
  java.util.NavigableMap<K, V>
```

Examining Class Modifiers and Types

```
                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 SuperClasses --%n%n");
                }
```

Examining Class Modifiers and Types

This is the declaration for java.util.concurrent.ConcurrentNavigableMap in the code:

```
public interface ConcurrentNavigableMap<K,V>
    extends ConcurrentMap<K,V>, NavigableMap<K,V>
```

```
List<Class> l = new ArrayList<Class>();
printAncestor(c, l);
if (l.size() != 0) {
        for (Class<?> cl : l)
                out.format("  %s%n",

        out.format("%n");
} else {
        out.format("  -- No SuperClasses --%n%n");
}
```

```
Inheritance Path:
  -- No SuperClasses --
```

Examining Class Modifiers and Types

```
            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%n");
            }
            } catch (ClassNotFoundException x) {
            x.printStackTrace();
            }
}
```

Examining Class Modifiers and Types

```
Annotations:
  -- No Annotations --
```

```
                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%n");
                }
                } catch (ClassNotFoundException x) {
                x.printStackTrace();
                }
        }
```

Examining Class Modifiers and Types

```
        private static void printAncestor(Class<?> c, List<Class> l) {
                Class<?> ancestor = c.getSuperclass();
                if (ancestor != null) {
                l.add(ancestor);
                printAncestor(ancestor, l);
                }
        }
}
```

Examining Class Modifiers and Types

```
$ java ClassDeclarationSpy "[Ljava.lang.String;"
Class:
  java.lang.String[]
Modifiers:
  public abstract final
Type Parameters:
  -- No Type Parameters --
Implemented Interfaces:
  interface java.lang.Cloneable
  interface java.io.Serializable
Inheritance Path:
  java.lang.Object
Annotations:
  -- No Annotations --
```

Since arrays are runtime objects, all of the type information is defined by the Java virtual machine. In particular, arrays implement `Cloneable` and `java.io.Serializable` and their direct superclass is always `Object`.

Examining Class Modifiers and Types

```
$ java ClassDeclarationSpy java.io.InterruptedIOException
Class:
  java.io.InterruptedIOException
Modifiers:
  public
Type Parameters:
  -- No Type Parameters --
Implemented Interfaces:
  -- No Implemented Interfaces --
Inheritance Path:
  java.io.IOException
  java.lang.Exception
  java.lang.Throwable
  java.lang.Object
Annotations:
  -- No Annotations --
```

From the inheritance path, it may be deduced that `java.io.InterruptedIOException` is a checked exception because `RuntimeException` is not present.

Examining Class Modifiers and Types

```
$ java ClassDeclarationSpy java.security.Identity
Class:
  java.security.Identity
Modifiers:
  public abstract
Type Parameters:
  -- No Type Parameters --
Implemented Interfaces:
  interface java.security.Principal
  interface java.io.Serializable
Inheritance Path:
  java.lang.Object
Annotations:
  @java.lang.Deprecated()
```

This output shows that `java.security.Identity`, a deprecated API, possesses the annotation `java.lang.Deprecated`. This may be used by reflective code to detect deprecated APIs.

Examining Class Modifiers and Types

Note: Not all annotations are available via reflection. Only those which have a `java.lang.annotation.RetentionPolicy` of `RUNTIME` are accessible. Of the three annotations pre-defined in the language `@Deprecated`, `@Override`, and `@SuppressWarnings` only `@Deprecated` is available at runtime.

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	yes
`getConstructor()`	no	N/A	no
`getDeclaredConstructors()`	yes	N/A	yes
`getConstructors()`	yes	N/A	no

Discovering Class Members

```
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 --"));
```

Discovering Class Members

```
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;
```

Discovering Class Members

```
        case ALL:
          printMembers(c.getConstructors(), "Constructors");
          printMembers(c.getFields(), "Fields");
          printMembers(c.getMethods(), "Methods");
          printClasses(c);
          break;
        default:
          assert false;
        }
      }
    } catch (ClassNotFoundException x) {
      x.printStackTrace();
    }
}
```

Discovering Class Members

```
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");
  }
```

Discovering Class Members

```
private static void printMembers(Member[] mbrs, String s) {
  out.format("%s:%n", s);
  for (Member mbr : mbrs) {
    i
    e
  }
  if
    out.format(
    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()`.

Discovering Class Members

```
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");
  }
}
```

Discovering Class Members

```
$ 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 superclass `RuntimeException` and other superclasses are not found.

Discovering Class Members

```
$ 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()`.

Members

Fields

A *field* is a class, interface, or enum with an associated value. Methods in the `java.lang.reflect.Field` class can retrieve information about the field, such as its name, type, modifiers, and annotations. There are also methods which enable dynamic access and modification of the value of the field. These tasks are covered in the following sections:

Obtaining Field Types describes how to get the declared and generic types of a field

Retrieving and Parsing Field Modifiers shows how to get portions of the field declaration such as public or transient

Getting and Setting Field Values illustrates how to access field values

Fields

When writing an application such as a class browser, it might be useful to find out which fields belong to a particular class. A class's fields are identified by invoking `Class.getFields()`. 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

A field may be a class (instance) field, such as `java.io.Reader.lock` , a static field, such as `java.lang.Integer.MAX_VALUE`, or an enum constant, such as `java.lang.Thread.State.WAITING`.

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`.

Obtaining Field Types

```
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());
                } catch (ClassNotFoundException x) {
                x.printStackTrace();
                }
        }
}
```

Obtaining Field Types

```
$ 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
```

Obtaining Field Types

```
$ 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`.

Obtaining Field Types

```
$ 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 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`.

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

Retrieving and Parsing Field Modifiers

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 next 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.

Retrieving and Parsing Field Modifiers

```
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]);
      }
```

Retrieving and Parsing Field Modifiers

```
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();
  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");
}} catch (ClassNotFoundException x) {
x.printStackTrace();}}
```

Retrieving and Parsing Field Modifiers

```
  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;
  }
}
```

Retrieving and Parsing Field Modifiers

```
$ 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 ]
```

In the next example 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()`.

Retrieving and Parsing Field Modifiers

```
$ 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 ]
```

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.

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.

Getting and Setting Field Values

```
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";
```

Getting and Setting Field Values

```
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));
```

Getting and Setting Field Values

```
$ 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.

Getting and Setting Field Values

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

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.

Note: Setting a field's value via reflecti…e overhead because various operations must occu…ss permissions. From the runtime's point of view, the effects a…the operation is as atomic as if the value was changed in the class code directly.

Methods

A *method* contains executable code which may be invoked. Methods are inherited and in non-reflective code behaviors such as overloading, overriding, and hiding are enforced by the compiler. In contrast, reflective code makes it possible for method selection to be restricted to a specific class without considering its superclasses. Superclass methods may be accessed but it is possible to determine their declaring class; this is impossible to discover programmatically without reflection and is the source of many subtle bugs.

Methods

The `java.lang.reflect.Method` class provides APIs to access information about a method's modifiers, return type, parameters, annotations, and thrown exceptions. It also be used to invoke methods. These topics are covered by the following sections:

Obtaining Method Type Information shows how to enumerate methods declared in a class and obtains type information

Obtaining Names of Method Parameters shows how to retrieve names and other information of a method or constructor's parameters

Retrieving and Parsing Method Modifiers describes how to access and decode modifiers and other information associated with the method

Invoking Methods illustrates how to execute a method and obtain its return value

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 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.

Obtaining Method Type Information

```
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";
  <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());
```

Obtaining Method Type Information

```
      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]);}}
    } catch (ClassNotFoundException x) {
      x.printStackTrace();}}
}
```

Obtaining Method Type Information

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
```

Obtaining Method Type Information

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

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

Here thod
with

```
$ 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
```

Obtaining Method Type Information

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.

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()`.

Obtaining Method Type Information

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`.

Obtaining Method Type Information

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;
```

Obtaining Method Type Information

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.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;
```

Obtaining Method Type Information

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 displayed.

Note: `Method.getGenericExceptionTypes()` exists because it is actually possible to declare a method with a generic exception type. However this is rarely used since it is not possible to catch a generic exception type.

Obtaining Names of Method Parameters

You can obtain the names of the formal parameters of any method or constructor with the method `java.lang.reflect.Executable.getParameters`. (The classes `Method` and `Constructor` extend the class `Executable` and therefore inherit the method `Executable.getParameters`.) However, `.class` files do not store formal parameter names by default. This is because many tools that produce and consume class files may not expect the larger static and dynamic footprint of `.class` files that contain parameter names. In particular, these tools would have to handle larger `.class` files, and the Java Virtual Machine (JVM) would use more memory. In addition, some parameter names, such as secret or password, may expose information about security-sensitive methods.

To store formal parameter names in a particular `.class` file, and thus enable the Reflection API to retrieve formal parameter names, compile the source file with the `-parameters` option to the `javac` compiler.

Obtaining Names of Method Parameters

The example illustrates how to retrieve the names of the formal parameters of all constructors and methods of a given class. The example also prints other information about each parameter.

The following command prints the formal parameter names of the constructors and methods of the class `ExampleMethods`.

Obtaining Names of Method Parameters

```
public static void printClassConstructors(Class c) {
  Constructor[] allConstructors = c.getConstructors();
  out.format(fmt, "Number of constructors", allConstructors.length);
  for (Constructor currentConstructor : allConstructors) {
    printConstructor(currentConstructor);
  }
  Constructor[] allDeclConst = c.getDeclaredConstructors();
  out.format(fmt, "Number of declared constructors", allDeclConst.length);
  for (Constructor currentDeclConst : allDeclConst) {
    printConstructor(currentDeclConst);
  }
}
```

Obtaining Names of Method Parameters

```
public static void printClassMethods(Class c) {
  Method[] allMethods = c.getDeclaredMethods();
  out.format(fmt, "Number of methods", allMethods.length);
  for (Method m : allMethods) {
    printMethod(m);
  }
}
```

Obtaining Names of Method Parameters

```
public static void printConstructor(Constructor c) {
  out.format("%s%n", c.toGenericString());
  Parameter[] params = c.getParameters();
  out.format(fmt, "Number of parameters", params.length);
  for (int i = 0; i < params.length; i++) {
    printParameter(params[i]);}}

public static void printMethod(Method m) {
  out.format("%s%n", m.toGenericString());
  out.format(fmt, "Return type", m.getReturnType());
  out.format(fmt, "Generic return type", m.getGenericReturnType());
  Parameter[] params = m.getParameters();
  for (int i = 0; i < params.length; i++) {
  printParameter(params[i]);}}
```

Obtaining Names of Method Parameters

```
public static void printParameter(Parameter p) {
  out.format(fmt, "Parameter class", p.getType());
  out.format(fmt, "Parameter name", p.getName());
  out.format(fmt, "Modifiers", p.getModifiers());
  out.format(fmt, "Is implicit?", p.isImplicit());
  out.format(fmt, "Is name present?", p.isNamePresent());
  out.format(fmt, "Is synthetic?", p.isSynthetic());
}
public static void main(String... args) {
  try {
    printClassConstructors(Class.forName(args[0]));
    printClassMethods(Class.forName(args[0]));
  } catch (ClassNotFoundException x) {
    x.printStackTrace();
  }
}
```

Obtaining Names of Method Parameters

```
Method #1
public boolean ExampleMethods.simpleMethod(java.lang.String,int)
              Return type: boolean
      Generic return type: boolean
          Parameter class: class java.lang.String
           Parameter name: stringParam
                Modifiers: 0
             Is implicit?: false
          Is name present?: true
             Is synthetic?: false
           Parameter class: int
            Parameter name: intParam
                 Modifiers: 0
              Is implicit?: false
          Is name present?: true
             Is synthetic?: false
```

Obtaining Names of Method Parameters

The example uses the following methods from the `Parameter` class:

`getType`: Returns a `Class` object that identifies the declared type for the parameter.

`getName`: Returns the name of the parameter. If the parameter's name is present, then this method returns the name provided by the `.class` file. Otherwise, this method synthesizes a name of the form `argN`, where *`N`* is the index of the parameter in the descriptor of the method that declares the parameter.

`getModifiers`: Returns an integer that represents various characteristics that the formal parameter possesses. This value is the sum of the following values, if applicable to the formal parameter:

getModifiers()

Value	Value (in hex)	Description
16	0x0010	The formal parameter is declared `final`
4096	0x1000	The formal parameter is `synthetic`. Alternatively, you can invoke the method `isSynthetic`.
32768	0x8000	The parameter is implicitly declared in source code. Alternatively, you can invoke the method `isImplicit`

Obtaining Names of Method Parameters

`isImplicit`: Returns `true` if this parameter is implicitly declared in source code.

`isNamePresent`: Returns `true` if the parameter has a name according to the `.class` file.

`isSynthetic`: Returns `true` if this parameter is neither implicitly nor explicitly declared in source code.

Implicit and Synthetic Parameters

Certain constructs are implicitly declared in the source code if they have not been written explicitly. For example, the `ExampleMethods` example does not contain a constructor. A default constructor is implicitly declared for it. The example prints information about the implicitly declared constructor of ExampleMethods:

```
Number of declared constructors: 1
public ExampleMethods()
```

Implicit and Synthetic Parameters

Consider the following excerpt:

```
public class MethodParameterExamples {
    public class InnerClass { }}
```

The class `InnerClass` is a non-static nested class or inner class. A constructor for inner classes is also implicitly declared. However, this constructor will contain a parameter. When the Java compiler compiles `InnerClass`, it creates a `.class` file that represents code similar to the following:

```
public class MethodParameterExamples {
    public class InnerClass {
        final MethodParameterExamples parent;
        InnerClass(final MethodParameterExamples this$0) {
            parent = this$0;
 }}}
```

Implicit and Synthetic Parameters

The InnerClass constructor contains a parameter whose type is the class that encloses `InnerClass`, which is `MethodParameterExamples`. Consequently, the example `MethodParameterExamples` prints the following:

```
public MethodParameterExamples$InnerClass(MethodParameterExamples)
         Parameter class: class MethodParameterExamples
          Parameter name: this$0
               Modifiers: 32784
            Is implicit?: true
          Is name present?: true
            Is synthetic?: false
```

Because the constructor of the class `InnerClass` is implicitly declared, its parameter is implicit as well.

Implicit and Synthetic Parameters

Note:

The Java compiler creates a formal parameter for the constructor of an inner class to enable the compiler to pass a reference (representing the immediately enclosing instance) from the creation expression to the member class's constructor.

The value 32784 means that the parameter of the InnerClass constructor is both final (16) and implicit (32768).

The Java programming language allows variable names with dollar signs ($); however, by convention, dollar signs are not used in variable names.

Implicit and Synthetic Parameters

Constructs emitted by a Java compiler are marked as *synthetic* if they do not correspond to a construct declared explicitly or implicitly in source code, unless they are class initialization methods. Synthetic constructs are artifacts generated by compilers that vary among different implementations. Consider the following excerpt:

```
public class MethodParameterExamples {
    enum Colors {
        RED, WHITE;
    }
}
```

When the Java compiler encounters an enum construct, it creates several methods that are compatible with the .class file structure and provide the expected functionality of the enum construct. For example, the Java compiler would create a .class file for the enum construct Colors that represents code similar to the following:

Implicit and Synthetic Parameters

```
final class Colors extends java.lang.Enum<Colors> {
    public final static Colors RED = new Colors("RED", 0);
    public final static Colors BLUE = new Colors("WHITE", 1);
    private final static values = new Colors[]{ RED, BLUE };

    private Colors(String name, int ordinal) {
        super(name, ordinal);
    }

    public static Colors[] values(){
        return values;
    }

    public static Colors valueOf(String name){
        return (Colors)java.lang.Enum.valueOf(Colors.class, name);
    }
}
```

Implicit and Synthetic Parameters

The Java compiler creates three constructors and methods for this enum construct: Colors(String name, int ordinal), Colors[] values(), and Colors valueOf(String name). The methods values and valueOf are implicitly declared. Consequently, their formal parameter names are implicitly declared as well.

The enum constructor Colors(String name, int ordinal) is a default constructor and it is implicitly declared. However, the formal parameters of this constructor (name and ordinal) are *not* implicitly declared. Because these formal parameters are neither explicitly or implicitly declared, they are synthetic. (The formal parameters for the default constructor of an enum construct are not implicitly declared because different compilers need not agree on the form of this constructor; another Java compiler might specify different formal parameters for it. When compilers compile expressions that use enum constants, they rely only on the public static fields of the enum construct, which are implicitly declared, and not on their constructors or how these constants are initialized.)

Implicit and Synthetic Parameters

```
enum Colors:
Number of constructors: 0
Number of declared constructors: 1
Declared constructor #1
private MethodParameterExamples$Colors()
         Parameter class: class java.lang.String
          Parameter name: $enum$name
               Modifiers: 4096
            Is implicit?: false
          Is name present?: true
             Is synthetic?: true
           Parameter class: int
            Parameter name: $enum$ordinal
                 Modifiers: 4096
              Is implicit?: false
          Is name present?: true
             Is synthetic?: true
```

Implicit and Synthetic Parameters

```
Number of methods: 2
Method #1
public static MethodParameterExamples$Colors[]
    MethodParameterExamples$Colors.values()
             Return type: class [LMethodParameterExamples$Colors;
     Generic return type: class [LMethodParameterExamples$Colors;
Method #2
public static MethodParameterExamples$Colors
    MethodParameterExamples$Colors.valueOf(java.lang.String)
             Return type: class MethodParameterExamples$Colors
     Generic return type: class MethodParameterExamples$Colors
         Parameter class: class java.lang.String
          Parameter name: name
               Modifiers: 32768
            Is implicit?: true
         Is name present?: true
            Is synthetic?: false
```

THE END