**Discovering Java's runtime type information**

Runtime type information (RTTI) refers to the correct identification of the type of your objects at run time. When you write code, it is generally desirable to do so in a way that takes advantage of OOP features like encapsulation and inheritance to make your program easily extensible, by for instance, as much as possible manipulating references to base classes and letting polymorphism work for you. Essentially, when you create a new object with a base class reference, the process necessarily involves an upcast to the base class from a more specialised type.There are many advantages to this style of programming not least of which includes the ability to simulate a more dynamic behaviour. Upcasting is a fairly straightforward process that the Java interpreter undertakes automatically, because there is very little risk associated with it. RTTI describes the reverse of this process when you wish to focus on a specific type for example to perform a particular operation. For instance, consider the following Game class hierarchy:-

package runtime.types.java;  
   
abstract class Game {  
     public int players;  
     public String toString() {  
         return "Game";  
     }  
     void play() {  
         System.out.println("playing " + this + "()");  
     }  
}  
class Charade extends Game {  
     public String toString() {  
         return "CharadeGame";  
     }  
}  
class Chess extends Game {  
     public String toString() {  
         return "Chess";  
     }  
}  
class CardGame extends Game {  
     public String toString() {  
         return "CardGame";  
     }  
}  
class BlackJack extends CardGame {  
     public void shuffle() {  
         System.out.println("Shuffle cards");  
     }  
     public String toString() {  
         return "BlackJack";  
     }  
}  
class OnlineGame extends Game {  
     public String toString() {  
         return "OnlineGame";  
     }  
}

This is a fairly accurate representation of the structure of libraries that exist within an OO paradigm, even though in reality some classes inherit from base classes in other packages.  
In the following code a List is configured to hold subtypes of the abstract Game class. Each type of Game is upcast to the List container which treats all elements as Game references having no memory of individual types. We can identify the benefit of RTTI in the playGame() method where we iterate through the list and the correct type is identified in the call to play() because polymorphism allows us to make the correct binding at runtime:-

package runtime.types.java;  
   
import java.util.Arrays;  
import java.util.List;  
   
public class Arcade {  
     static void playGame(List<Game> options) {  
         for(Game selection : options)  
         selection.play();  
     }  
     public static void main(String[] args) {  
         List<Game> games = Arrays.asList(new Chess(), new CardGame(), new BlackJack());  
         playGame(games);  
     }  
}  
/\* Output  
playing Chess()  
playing CardGame()  
playing BlackJack()  
\*//

The above example is the most basic form of RTTI and why polymorphism is understood to be a general goal in object oriented programming. There are other circumstances when you might need to identify a specific type, perhaps to enable some operation unique to a specific type, circumstances for which you can not rely on polymorphism by itself.  
Java allows you to discover runtime type information in one of two ways. The first approach assumes all type information is available at compile time which you can obtain by querying the object reference. For instance, suppose we wanted to ensure the cards are shuffled before playing a game ofBlackJack. We could use RTTI to identify this specific type in order to call the appropriate method:-

package runtime.types.java;  
   
import java.util.Arrays;  
import java.util.List;  
   
public class GettingType {  
     public static void playGame(List<Game> options) {  
         for(Game selection : options) {  
             Class<? extends Game> type = selection.getClass();  
             if(type.isAssignableFrom(BlackJack.class))  
                 ((BlackJack)selection).shuffle();  
             selection.play();  
         }  
     }  
     public static void main(String[] args) {  
         List<Game> games = Arrays.asList(new Chess(), new BlackJack(), new Charade());  
         playGame(games);  
     }  
}  
/\* Output  
playing Chess()  
Shuffle cards  
playing BlackJack()  
playing Charade()  
\*//

In this instance, we query the object reference for type information using the getClass() method, and then apply the isAssignableFrom() method to the resulting reference determine if this class is the same or a superclass of the BlackJack class. If it is, we cast the object to a BlackJack object and then call shuffle(). The logic of type as you will no doubt come to understand is sound and quite uncomplicated.

You can also use the instanceof keyword to check for the type of an object, for example prior to making a cast:-

package runtime.types.java;  
   
public class TypeTest {  
     public static void main(String[] args) {  
         Game aGame = new BlackJack();  
         if(!(aGame instanceof BlackJack))  
             System.out.println("Not an instance");  
         ((BlackJack)aGame).play();  
     }  
}  
/\* Output  
playing BlackJack()  
\*//

The other approach to discovering type information is to make use of a reflection mechanism by which means you can discover and use previously unavailable class information, exclusively at runtime. In any event, either approach requires a sound knowledge of how Java represents type information at runtime.

**The Class object**

Java uses instances of the class Class to represent type information at runtime. Type refers to the classes and interfaces in a running Java application which are automatically loaded by the Java Virtual Machine (JVM) using a special ClassLoader mechanism. In effect, there exists a Class object for every class that is a part of your program which consists of information about your class and is used for creating all the other objects of your class. You can get a reference to this Class object by calling the getClass() method as we did in a previous example. However in circumstances where this option might not be available, you can create a Class reference directly which can then be manipulated just like you do any other reference to conveniently discover all sorts of type information. You can return a Class reference by either calling the static forName() method or by using the class literal. Here is an example using both techniques;-

public class GettingTypeInfo {  
     static void printTypeInfo(Class<? extends Game> whichClass) {  
         System.out.println("Class name - " + whichClass.getName());  
         System.out.println("Simple name - " + whichClass.getSimpleName());  
         System.out.println("isInterface - " + whichClass.isInterface());  
         System.out.println("Package - " + whichClass.getPackage());  
         System.out.println("SuperClass " + whichClass.getSuperclass());  
     }  
     public static void main(String[] args) throws ClassNotFoundException,   
                                 IllegalAccessException, InstantiationException {  
         Class<? extends Game> aClass = Class.forName("runtime.types.java.BlackJack");  
         //class literal syntax  
         Class anotherClass = Chess.class;  
         printTypeInfo(aClass);  
         System.out.println("isInstance - " + aClass.isInstance(new BlackJack()));  
         System.out.println("------------------------------- ");  
         printTypeInfo(anotherClass);  
         System.out.println("isInstance - " + anotherClass.isInstance(new Game()));  
         System.out.println("------------------------------- ");  
         Object anObj = aClass.newInstance();  
         System.out.println("NewInstance " + anObj.toString());  
      }  
}  
/\* Output  
Class name - runtime.types.java.BlackJack  
Simple name - BlackJack  
isInterface - false  
Package - package runtime.types.java  
SuperClass class runtime.types.java.CardGame  
isInstance - true  
-------------------------------  
Class name - runtime.types.java.Chess  
Simple name - Chess  
isInterface - false  
Package - package runtime.types.java  
SuperClass class runtime.types.java.Game  
isInstance - false  
-------------------------------  
NewInstance BlackJack  
\*//

The first statement in the main() makes use of the static forName() method to obtain a valid reference to the BlackJack Class, by contrast, the second statement makes use of a class literal to achieve the same effect. Both techniques are equally valid, but have subtle implications related to the way the class object is initialised. Notice the use of generic syntax with Class types in particular in the call to the forName() method. Generics provide a way to constrain objects to subtypes of a particular type. Here the wildcard symbol symbol is used in combination with the extends keyword to constrain all references to subtypes of Game. Class references also work fine without the use of generics, however it is advisable to use generic syntax as you are likely to find out about mistakes much sooner  
The printTypeInfo() method displays output from a number of methods available to objects of the class Class providing access to useful type information. The isInstance() and newInstance() methods are also used in the main() to test for compatibility and create a new instance respectively.

As you become used to working with types, you will find them useful in circumstances where you might wish to be able to identify individual types perhaps for some specific purpose. For instance, suppose you wanted a way to keep a track of the number of individual Game types generated by your application. The first thing you might need is a way to link the initialisation of each Game object with it's corresponding Class object. One way to do this might be to use a variation of the *template* design pattern to create an abstract method that returns a list of types with which you might generate new instances of Game types:-

package runtime.types.java;  
   
import java.util.List;  
import java.util.Random;  
   
public abstract class InitGame {  
     static private Random generator = new Random();  
     public abstract List<Class<? extends Game>> getTypes();  
      public Game aRandomGame() {  
         int index = generator.nextInt(getTypes().size());  
         try {  
             return getTypes().get(index).newInstance();  
         } catch(InstantiationException ref) {  
             throw new RuntimeException(ref);  
         } catch(IllegalAccessException ref) {  
             throw new RuntimeException(ref);  
         }  
     }  
     public Game[] createGames(int capacity) {  
         Game[] arrayOfGames = new Game[capacity];  
         for(int i = 0; i < capacity; i++)  
             arrayOfGames[i] = aRandomGame();  
         return arrayOfGames;  
     }  
}

Essentially, this class uses a random number generator to index into a list of types in order to instantiate a new Game object. A second method uses this pseudo-random algorithm to return an array of games. As with all abstract classes, an inheriting class will need to implement the abstractgetTypes() method. The following class uses class literals to create a list of Class references:-

package runtime.types.java;  
   
import java.util.Arrays;  
import java.util.List;  
   
public class GameTypeCreator extends InitGame {  
     @SuppressWarnings("unchecked")  
     public static final List<Class<? extends Game>> gameTypes = Arrays.asList(Chess.class,  
                                               CardGame.class, BlackJack.class,  
                                                   Charade.class, OnlineGame.class);  
     public List<Class<? extends Game>> getTypes() {  
         return gameTypes;  
     }  
}

The final class in this sequence uses a Map to track the number individual type of Game objects created. Here's how:-

package runtime.types.java;  
   
import java.util.HashMap;  
import java.util.Iterator;  
import java.util.LinkedHashMap;  
import java.util.List;  
import java.util.Map;  
   
class Util {  
     static public Map<Class<? extends Game>, Integer> mapData(List<Class<? extends Game>> key, int value) {  
            Map<Class<? extends Game>, Integer> data = new HashMap<>();  
            Iterator<Class<? extends Game>> sequence = key.iterator();  
            while(sequence.hasNext())  
                   data.put(sequence.next(), value);  
            return data;  
     }  
}  
public class CountGame extends LinkedHashMap<Class<? extends Game>, Integer> {  
         public CountGame() {  
             super(Util.mapData(GameTypeCreator.gameTypes, 0));  
         }  
         public void count(Game aGame) {  
             for(Map.Entry<Class<? extends Game>, Integer> anEntry : entrySet())  
                 //Class.isInstance to dynamically test for types  
                 if(anEntry.getKey().isInstance(aGame))  
                     put(anEntry.getKey(), anEntry.getValue() + 1);  
         }  
         public String toString() {  
             StringBuilder aString = new StringBuilder("{");  
             for(Map.Entry<Class<? extends Game>, Integer> anEntry : entrySet()) {  
                 aString.append(anEntry.getKey().getSimpleName());  
                 aString.append("=");  
                 aString.append(anEntry.getValue());  
                 aString.append(", ");  
             }  
             aString.delete(aString.length()-2, aString.length());  
             aString.append("} ");  
             return aString.toString();  
         }  
     }  
     public static void main(String[] args) {  
         CountGame aGameCounter = new CountGame();  
         GameTypeCreator aGameTypeCreator = new GameTypeCreator();  
         for(Game aGame : aGameTypeCreator.createGames(15)) {  
             System.out.print(aGame.getClass().getSimpleName() + " ");  
             aGameCounter.count(aGame);  
         }  
         System.out.println();  
         System.out.println(aGameCounter);  
     }  
}  
/\* Output  
OnlineGame BlackJack Charade BlackJack BlackJack Chess CardGame Chess   
OnlineGame BlackJack BlackJack Chess BlackJack Charade Charade  
{Chess=3, Charade=3, BlackJack=6, OnlineGame=2, CardGame=7}  
\*//

In this example, a Map is preloaded using a mapData() utility that accepts a list of Class types restricted to subtypes of the Game class and an integer value, zero in this particular instance. In the count(), the isInstance() method is used to test for type and then increment the value entry by one. AMap can not contain duplicate keys and each key can only map to one value so it is the perfect tool for this kind of application. The toString()method is overridden to provide a formatted view of the Map. Adding a new type of Game is a simply a matter of making an update to the staticgameTypes list in the GameTypeCreator class.

This is a useful tool limited only by the fact that it can only be used to count types of Game because the Map has been preloaded with Game types. A more useful tool would be one that we could use to count any type:-

package runtime.types.java;  
   
import java.util.HashMap;  
import java.util.Map;  
   
public class CountAnyType extends HashMap<Class<?>, Integer> {  
     private Class<?> baseType;  
     public CountAnyType(Class<?> aType) {  
         this.baseType = aType;  
     }  
     public void checkType(Object arg) {  
         Class<?> aType = arg.getClass();  
         if(!baseType.isAssignableFrom(aType))  
             throw new RuntimeException("Wrong type! " + arg);  
         count(aType);  
     }  
     public void count(Class<?> aType) {  
         Integer sum = get(aType);  
         put(aType, sum == null ? 1 : sum + 1);  
         Class<?> superClass = aType.getSuperclass();  
         if(superClass != null && baseType.isAssignableFrom(superClass))  
             count(superClass);  
     }  
     public String toString() {  
         StringBuilder aString = new StringBuilder("{");  
         for(Map.Entry<Class<?>, Integer> anEntry : entrySet()) {  
             aString.append(anEntry.getKey().getSimpleName());  
             aString.append("=");  
             aString.append(anEntry.getValue());  
             aString.append(", ");  
         }  
         aString.delete(aString.length()-2, aString.length());  
         aString.append("} ");  
         return aString.toString();  
     }  
     public static void main(String[] args) {  
         CountAnyType aTypeCounter = new CountAnyType(Game.class);  
         GameTypeCreator aGameCreator = new GameTypeCreator();  
         for(Game aGame : aGameCreator.createGames(15)) {  
             System.out.print(aGame.getClass().getSimpleName() + " ");  
             aTypeCounter.checkType(aGame);  
         }  
         System.out.println();  
         System.out.println(aTypeCounter);  
     }  
}  
/\* Output  
Charade Charade Chess BlackJack BlackJack BlackJack CardGame BlackJack   
Charade CardGame OnlineGame Charade Charade Chess CardGame  
{Game=15, Charade=5, BlackJack=4, OnlineGame=1, Chess=2, CardGame=7}  
\*//

In this example, we use a wildcard <?> argument to specify the type of the class, effectively making it possible to use any type with our container. We have also introduced a checkType() method that allows us to verify that we are counting the correct type using the isAssignableFrom() method. The count() method in this example, first counts the specific type and then recursively counts the base type.

You would be correct to imagine that it would be possible to make a more sophisticated design in our approach to generating Game objects in circumstances where we were unconcerned about keeping track of types or for that matter if we had to generate Game objects with a much greater frequency. Here is an approach that makes use of the *Factory* design pattern by registering factories for the types to be created directly in the base class:-

package runtime.types.java;  
   
import java.util.ArrayList;  
import java.util.List;  
import java.util.Random;  
   
interface GameFactory<T> { T createGame(); }  
   
class Game {  
     static List<GameFactory<? extends Game>> factory = new ArrayList<>();  
     private static Random generator = new Random();  
     public String toString() {  
         return getClass().getSimpleName();  
     }  
     static {  
         factory.add(new Charade.Factory());  
         factory.add(new Chess.Factory());  
         factory.add(new Scrabble.Factory());  
         factory.add(new Monopoly.Factory());  
         factory.add(new BlackJack.Factory());  
         factory.add(new OnlineGame.Factory());  
         factory.add(new VideoGame.Factory());  
     }  
     public static Game autoCreate() {  
         int index = generator.nextInt(factory.size());  
         return factory.get(index).createGame();  
     }  
   
}  
class PartyGame extends Game {}  
class Charade extends PartyGame {  
     public static class Factory implements GameFactory<Charade> {  
         public Charade createGame() { return new Charade(); }  
     }  
}  
class BoardGame extends Game {}  
class Chess extends BoardGame {  
     public static class Factory implements GameFactory<Chess> {  
         public Chess createGame() { return new Chess(); }  
     }  
}  
class Scrabble extends BoardGame {  
     public static class Factory implements GameFactory<Scrabble> {  
         public Scrabble createGame() { return new Scrabble(); }  
     }  
}  
class Monopoly extends BoardGame {  
     public static class Factory implements GameFactory<Monopoly>  {  
         public Monopoly createGame() { return new Monopoly(); }  
     }  
}  
class CardGame extends Game {}  
class BlackJack extends CardGame {  
     public static class Factory implements GameFactory<BlackJack> {  
         public BlackJack createGame() { return new BlackJack(); }  
     }  
}  
class OnlineGame extends Game {  
     public static class Factory implements GameFactory<OnlineGame> {  
         public OnlineGame createGame() { return new OnlineGame(); }  
     }  
}  
class VideoGame extends Game {  
     public static class Factory implements GameFactory<VideoGame> {  
         public VideoGame createGame() { return new VideoGame(); }  
     }  
}  
public class GameFactories {  
     public static void main(String[] args) {  
         for(int i = 0; i < 10; i++)  
             System.out.println(Game.autoCreate());  
     }  
}  
/\* Output  
Charade  
OnlineGame  
Charade  
BlackJack  
OnlineGame  
Scrabble  
Scrabble  
Chess  
OnlineGame  
Chess  
\*//

The GameFactory interface uses generic syntax and consequently is able to return a different type based on it's implementation. In this way, adding a new type of Game does not require too much effort, except to register it's inner GameFactory class in the factory list in the base class . The genius of this design is in its simplicity and this is exactly what design patterns are able to do. They provide you with the ability to encapsulate change.

**The Reflection Mechanism**

The RTTI mechanisms discussed thus far have in common the advantage of being able to discover type information for objects whose types will have been available at compile time, in other words these classes must have been compiled as a part of your program. Java uses reflection to discover and use classes at runtime about which you had no prior information. This capability is useful in circumstances where your application might need to make use of a foreign utility which might be accessed over a network connection, on the internet or in some other similar fashion to create and execute objects on remote platforms in a technique known as Remote Method Invocation (RMI).

Java's reflection implementation consists of a number of classes within the java.lang.reflect library that can be used to represent the fields, methods and constructors after which they have been appropriately named. The Constructor class for instance provides access to a single class constructor, theField class includes getter() and setter() methods to read and modify Field objects and the Method class includes an invoke() method by which means you might invoke the underlying method associated with the Method object. These classes implement the Member interface which reflects identifying information about a single member. The class Class supports reflection by including a number of convenience methods by which means you might discover and use type information at runtime.

Reflection is also a useful tool for dynamically extracting information about a class, for example to reveal the entire interface including inherited and overridden methods. Here is a nifty little tool that uses reflection to automatically reveal information about a class at runtime. You can run this tool on any class to save you time from having to trawl through endless lines of documentation:-

package runtime.types.java;  
   
import java.lang.reflect.Constructor;  
import java.lang.reflect.Method;  
import java.util.regex.Pattern;  
   
public class InterfaceLookup {  
     private static Pattern packageNames = Pattern.compile("\\w+\\.");   
     public static void showInterface(Object ref) {  
         try {  
             Class<?> whichClass = Class.forName(ref.getClass().getName());  
             Constructor[] constructors = whichClass.getConstructors();  
             Method[] methods = whichClass.getMethods();  
             for(Constructor aConstructor : constructors)  
                 System.out.println(packageNames.matcher(aConstructor.toString())  
                                   .replaceAll(""));  
             for(Method aMethod : methods)  
                 System.out.println(packageNames.matcher(aMethod.toString())  
                                   .replaceAll(""));  
         } catch(ClassNotFoundException exceptionRef) {  
             System.out.println("Class does not exist!");  
         }  
     }  
     public static void main(String[] args) {  
         showInterface(new InterfaceLookup());  
     }  
}  
/\* Output  
public InterfaceLookup()  
public static void main(String[])  
public static void showInterface(Object)  
public final native void wait(long) throws InterruptedException  
public final void wait(long,int) throws InterruptedException  
public final void wait() throws InterruptedException  
public boolean equals(Object)  
public String toString()  
public native int hashCode()  
public final native Class getClass()  
public final native void notify()  
public final native void notifyAll()  
\*//

The example uses the forName() to return a reference to the current class and then uses convenience methods to return an array of constructors and methods which it can then iterate through and print out individually. The output is the result of using a regular expression to replace class name qualifiers. This is a quick and easy way to find out all you need to know about a class's interface while coding.

As part of it's reflection mechanism, Java also provides classes and interfaces by which means you might implement dynamic proxies which are modeled on the *Proxy* design pattern. Proxy is a pattern that allows you to provide a surrogate or placeholder for another object in order to control access to it. This is useful in circumstances where you wish to differ the full cost of an objects creation until you actually need it or if you wanted to provide for additional operations you do not wish to be a part of your main object. For instance, consider a drawing application that renders graphics on screen in a scenario where drawing an image is a costly operation you only want to undertake when necessary. You can implement an image proxy between the Image and the calling client so that you can control when and how you create an Image. The following code provides a better illustration:-

package runtime.types.java;  
   
interface Graphic { void draw(); }  
   
class Image implements Graphic {  
     public void draw() {  
         System.out.println("Hello!");  
     }  
}  
class ImageProxy implements Graphic {  
     private Graphic anImage;  
     public ImageProxy(Graphic args) {  
         this.anImage = args;  
     }  
     public void draw() {  
         System.out.println(""Calling Image.."");  
         anImage.draw();  
     }  
}  
public class DrawingApplication {  
     public static void drawImage(Graphic aGraphic){  
         aGraphic.draw();  
     }  
     public static void main(String[] args) {  
         drawImage(new ImageProxy(new Image()));  
     }  
}  
/\* Output  
"Calling Image.."  
Hello!  
\*//

The benefits of such a design may not be immediately obvious, however, it does provide us with additional options for instance, we can easily add code to the image proxy to keep track of calls to the Image or to measure the overhead of such calls.  
Java provides classes to implement a proxy instance that will handle method calls dynamically through the use of an InvocationHandler interface. We can re-implement our image proxy to act as an invocation handler like so:-

package runtime.types.java;  
   
import java.lang.reflect.InvocationHandler;  
import java.lang.reflect.Method;  
import java.lang.reflect.Proxy;  
   
class AnotherImageProxy implements InvocationHandler {  
     private Graphic anImage;  
     public AnotherImageProxy(Graphic args) {  
         this.anImage = args;  
     }  
     public Object invoke(Object proxy, Method aMethod, Object[] args) throws Throwable {  
         System.out.println(""Calling Image.."");  
         return aMethod.invoke(anImage, args);  
     }  
}  
public class AnotherDrawingApplication {  
     public static void drawImage(Graphic whichImage){  
         whichImage.draw();  
     }  
     public static void main(String[] args) {  
         Graphic proxy = (Graphic)Proxy.newProxyInstance(Graphic.class.getClassLoader(),  
                               new Class[]{Graphic.class}, new AnotherImageProxy  
                                (new Image()));  
         drawImage(proxy);  
     }  
}  
/\* Output  
"Calling Image.."  
Hello!  
\*//

In this example, we implement the <>code>invoke() method which is responsible for processing a method invocation on a proxy instance and returning the result, which in this case happens to be a Method object used to invoke the underlying method which it represents. In the main() a proxy instance is returned using the static newProxyInstance() method which accepts a class loader, an array of interfaces to be implemented and the implemented invocation handler, which is handed the Image object as a part of its constructor. In this case it's probably easiest to pass an existingClassLoader like the Graphic.class reference.

RTTI opens up a whole new world of programming possibilities as you come to understand how types provide you with backdoor access into pretty much any interface regardless of access permissions. For instance, consider the following simple interface in the Interfaces.java package:-

package Interfaces.java;  
   
public interface AnInterface {  
     void interfaceMethod();  
}

Suppose you wanted to hide the implementation of this interface to deter your customers from relying on it so that you might be free to change it at a later stage? You might think it is enough to restrict the implementation to package access thereby making it inaccessible to clients outside of the package like so:-

package runtime.types.java;  
   
class HiddenImplementation implements AnInterface {  
     private String aField = "Can't touch this";  
     public void interfaceMethod(){ System.out.println("public HiddenImplementation.interfaceMethod()"); }  
     public void aMethod(){ System.out.println("public HiddenImplementation.aMethod()"); }  
     void bMethod() { System.out.println("package HiddenImplementation.bMethod()"); }  
     private void cMethod() { System.out.println("private HiddenImplementation.cMethod()"); }  
     protected void dMethod() { System.out.println("protected HiddenImplementation.dMethod()"); }  
   
     public String toString(){  
         return aField;  
     }  
}  
public class PublicInterface {  
     public static AnInterface makeAnInterface() {  
         return new HiddenImplementation();  
     }  
}

The PublicInterface class produces an implementation of AnInterface and is the only visible part of this package and you would expect your implementation to be fairly safe? Not exactly, I'm afraid. With reflection, anyone who knows the name of your class members can easily get around this restriction like so:-

import java.lang.reflect.Field;  
import java.lang.reflect.Method;  
import Interfaces.java.AnInterface;  
import runtime.types.java.\*;  
   
public class HackImplementation {  
     static void hackInterface(Object anObject, String member) throws Exception {  
         if(member.equals("aField")) {  
             Field whichField = anObject.getClass().getDeclaredField(member);  
             System.out.println(anObject.toString());  
             whichField.setAccessible(true);  
             whichField.set(anObject, "Yes I can");  
             System.out.println(anObject.toString());  
             } else {  
                 Method whichMethod = anObject.getClass().getDeclaredMethod(member);  
                 whichMethod.setAccessible(true);  
                 whichMethod.invoke(anObject);  
            }  
     }  
     public static void main(String[] args) throws Exception {  
         AnInterface newInterface = PublicInterface.makeAnInterface();  
         newInterface.interfaceMethod();  
         //Compile error aField can not be resolved  
         //!newInterface.aField;  
         //Compile error method undefined for type AnInterface  
         /\* newInterface.aMethod();  
            newInterface.bMethod();  
            newInterface.cMethod();  
            newInterface.dMethod();  
         \*/  
         hackInterface(newInterface, "aField");  
         hackInterface(newInterface, "aMethod");  
         hackInterface(newInterface, "bMethod");  
         hackInterface(newInterface, "cMethod");  
         hackInterface(newInterface, "dMethod");  
     }  
}  
/\* Output  
public HiddenImplementation.interfaceMethod()  
Can't touch this  
Yes I can  
public HiddenImplementation.aMethod()  
package HiddenImplementation.bMethod()  
private HiddenImplementation.cMethod()  
protected HiddenImplementation.dMethod()  
\*//

The example demonstrates how reflection allows you to get around all sorts of restrictions to access and change any field even those marked privateand call just about any method no matter the level of the access specifier. With the setAccessible() method you can modify fields and invoke methods, including even those within inner and anonymous classes, are still accessible with reflection. Even if you try to avoid this by distributing only compiled code so that your members are inaccessible, that is still not entirely the case as Java's class file disassembler (javap) can still be used to reveal all your class members. For instance the -private flag can be used to indicate that all members should be displayed including private ones. Running the following command will reveal everything you wish to know about our hidden class.  
  
  
javap -private HiddenImplementation  
  
In reality, hardly anything is hidden from reflection and most are susceptible to change, with the exception only of fields marked final. On the flip side if you choose to misuse reflection in your program, committing all sorts of access violations, then you really will only have yourself to blame if a subsequent update to your library succeeds in breaking your code, which I am certain will be a very unpleasant experience  
Reflection gives you the keys to a new whole new world of dynamic programming in which having access to runtime type information allows for a very different style of programming with which it is easy to get carried away. It is important to remember to use RTTI and reflection in a way that is not detrimental to polymorphism as the most effective way to provide for extensibility in your programs.