**Dynamic Proxy in Java**

The dynamic proxy facility, part of the java.lang.reflect package and added to the JDK in version 1.3, allows programs to create proxy objects, which can implement one or more known interfaces and dispatch calls to interface methods programmatically using reflection instead of using the built-in virtual method dispatch. This process allows implementations to "intercept" method calls and reroute them or add functionality dynamically.

Dynamic proxies provide an alternate, dynamic mechanism for implementing many common design patterns, including the Facade, Bridge, Interceptor, Decorator, Proxy (including remote and virtual proxies), and Adapter patterns. While all of these patterns can be easily implemented using ordinary classes instead of dynamic proxies, in many cases the dynamic proxy approach is more convenient and compact and can eliminate a lot of handwritten or generated classes.

**The Proxy pattern**

The Proxy pattern involves the creation of a "stub" or "surrogate" object, whose purpose is to accept requests and forward them to another object that actually does the work. The Proxy pattern is used by Remote Method Invocation (RMI) to make an object executing in another JVM appear like a local object; by Enterprise JavaBeans (EJB) to add remote invocation, security, and transaction demarcation; and by JAX-RPC Web services to make remote services appear as local objects. In each case, the behavior of a potentially remote object is defined by an interface, which by its nature admits multiple implementations. The caller cannot (for the most part) tell that they only hold a reference to a stub and not the real object because they both implement the same interface; the stub takes care of the work of finding the real object, marshalling the arguments, sending them to the real object, unmarshalling the return value, and returning it to the caller. Proxies can be used to provide remoting (as in RMI, EJB, and JAX-RPC), wrap objects with security policies (EJB), provide lazy loading for expensive objects (EJB Entity Beans), or add instrumentation such as logging.

At the heart of the dynamic proxy mechanism is the InvocationHandler interface, shown in Listing 1. The job of an invocation handler is to actually perform the requested method invocation on behalf of a dynamic proxy. The invocation handler is passed a Method object (from the java.lang.reflect package) and the list of arguments to be passed to the method; in the simplest case, it could simply call the reflective method Method.invoke() and return the result.

Also dynamic proxies can be used as AOP like Spring.

**Complete example on Dynamic Proxies**

interface **Calculator** {

public int add(int a,int b);

}

public class **CalculatorImpl** implements Calculator {

public int add(int a,int b) {

int c = a + b;

System.out.println("[TARGET METHOD CALL] Result is "+c);

return c;

}

}

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.util.Arrays;

public class LoggingValidationHandler implements InvocationHandler

{

private Object target;

public LoggingValidationHandler(Object target) {

this.target = target;

}

**public Object invoke(Object proxy, Method method, Object[] args)** throws Throwable {

//Prints the input

System.out.println("[BEFORE METHOD CALL] The method " + method.getName() + "() begins with "+ Arrays.toString(args));

//Negative check for input

Integer a = (Integer)args[0];

Integer b = (Integer)args[1];

if (a < 0 || b < 0) {

throw new IllegalArgumentException("Positive numbers only");

}

Object result = method.invoke(target, args);

//Prints the output

System.out.println("[AFTER METHOD CALL ] The method " + method.getName() + "() ends with " + result.toString());

return result;

}

}

import java.lang.reflect.Proxy;

public class ProxyTest {

public static void main(String[] args) {

//Plain business object (arithmeticCalculatorImpl)

Calculator plainCalculator = new CalculatorImpl();

//Proxied business object (proxiedCalculator).

Calculator proxiedCalculator =

(Calculator) Proxy.newProxyInstance(

plainCalculator.getClass().getClassLoader(),

plainCalculator.getClass().getInterfaces(),

new LoggingValidationHandler(plainCalculator));

System.out.print("Call to add method without using the proxy.");

plainCalculator.add(1,2);

System.out.println("-------------------------------------------------------");

System.out.println("Call to add method with +ve arguments using the proxy.");

proxiedCalculator.add(1,2);

System.out.println("-------------------------------------------------------");

System.out.println("Call to add method with -ve arguments using the proxy.");

proxiedCalculator.add(-1,2);

}

}

**Event Dispatch Thread**

Swing event handling code runs on a special thread known as the event dispatch thread. Most code that invokes Swing methods also runs on this thread. This is necessary because most Swing object methods are not "thread safe": invoking them from multiple threads risks thread interference or memory consistency errors.

The EventDispatching thread is a special thread that is managed by the AWT. Basically it is a thread that runs in an infinite loop processing event. The java.awt.EventQueue.invokeLater method is a special way to provide some code that will run on the event queue. Writing a ui framework that is safe in a multithreading environment is very difficult so the AWT authors decided that they would only allow operations on GUI objects to occur on a single special thread. All event handlers will execute on this thread and all code that modifies the gui should also operate on this thread.

Now the AWT does not usually check that you are not issues gui commands from another thread (The WPF framework for C# does do this). so it is possible to write a lot of code and be pretty much agnostic to this and not run into any problems. But this can lead to undefined behavior so the best thing to do is to always ensure that gui code runs on the event dispatcher thread. invokeLater provides a mechanism to do this.

So a classic example is that you need to run a long running operation like downloading a file. So you launch a thread to perform this action then when it is completed you will use invokeLater to update the UI. If you didn't use invokeLater and instead you just updated the ui directly you might have a race condition and undefined behavior could occur.

The event dispatching thread (EDT) is a background thread used in Java to process events from the Abstract Window Toolkit (AWT) graphical user interface event queue. These events are primarily update events that cause user interface components to redraw themselves, or input events from input devices such as the mouse or keyboard. The AWT uses a single-threaded painting model in which all screen updates must be performed from a single thread. The event dispatching thread is the only valid thread to update the visual state of visible user interface components. Updating visible components from other threads is the source of many common bugs in Java programs that use Swing.

The event dispatching thread (EDT) is a background thread used in Java to process events from the Abstract Window Toolkit (AWT) graphical user interface event queue. These events are primarily update events that cause user interface components to redraw themselves, or input events from input devices such as the mouse or keyboard. The AWT uses a single-threaded painting model in which all screen updates must be performed from a single thread. The event dispatching thread is the only valid thread to update the visual state of visible user interface components.

**Difference between newFixedThreadPool and newCachedThreadPool**

**newFixedThreadPool**

Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue. At any point, at most nThreads threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks. The threads in the pool will exist until it is explicitly shutdown.

**newCachedThreadPool**

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. These pools will typically improve the performance of programs that execute many short-lived asynchronous tasks. Calls to execute will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool. Threads that have not been used for sixty seconds are terminated and removed from the cache. Thus, a pool that remains idle for long enough will not consume any resources. Note that pools with similar properties but different details (for example, timeout parameters) may be created using ThreadPoolExecutor constructors.

In terms of resources, the newFixedThreadPool will keep all the threads running until they are explicitly terminated. In the newCachedThreadPool Threads that have not been used for sixty seconds are terminated and removed from the cache.

Given this, the resource consumption will depend very much in the situation. For instance, If you have a huge number of long running tasks I would suggest the FixedThreadPool. As for the CachedThreadPool, the docs say that "These pools will typically improve the performance of programs that execute many short-lived asynchronous tasks".

**How JVM Handles abstract classes**

Every object has a pointer to its vtable in its object header. The vtable contains pointers to all virtual and abstract methods defined in the hierarchy of the type of the object. They are ordered and have well-known indices which makes it performant to call such a method.

Lets consider Animal as an abstract class. And Lion as it's subclass. We can't have any Animal class because being an Animal is too abstract, cause there's no way something can be just animal and not Lion, or Tiger, or cow. So for better Object Orientation similar to real world, the classes which are too abstract (such as Animal) are marked with abstract keyword and are not allowed to be instantiated, just because no object can be just Animal(using metaphors here). You need to be more specific. Like which kind of animal are you talking about, is it Giraffe, or Lion, or may be a dog. Of course you can choose your own level of specificity and abstractness. If you're talking about objects, being an Animal is more specific, than being just an object.

**ServletConfig Vs ServletContext**

**Signature: public interface ServletConfig**

ServletConfig is implemented by the servlet container to initialize a single servlet using init(). That is, you can pass initialization parameters to the servlet using the web.xml deployment descriptor. For understanding, this is similar to a constructor in a java class

**Signature: public interface ServletContext**

ServletContext is implemented by the servlet container for all servlet to communicate with its servlet container, for example, to get the MIME type of a file, to get dispatch requests, or to write to a log file. That is to get detail about its execution environment. It is applicable only within a single Java Virtual Machine. If a web applicationa is distributed between multiple JVM this will not work. For understanding, this is like a application global variable mechanism for a single web application deployed in only one JVM.