In proxy pattern, a class represents functionality of another class.

This type of design pattern comes under structural pattern.

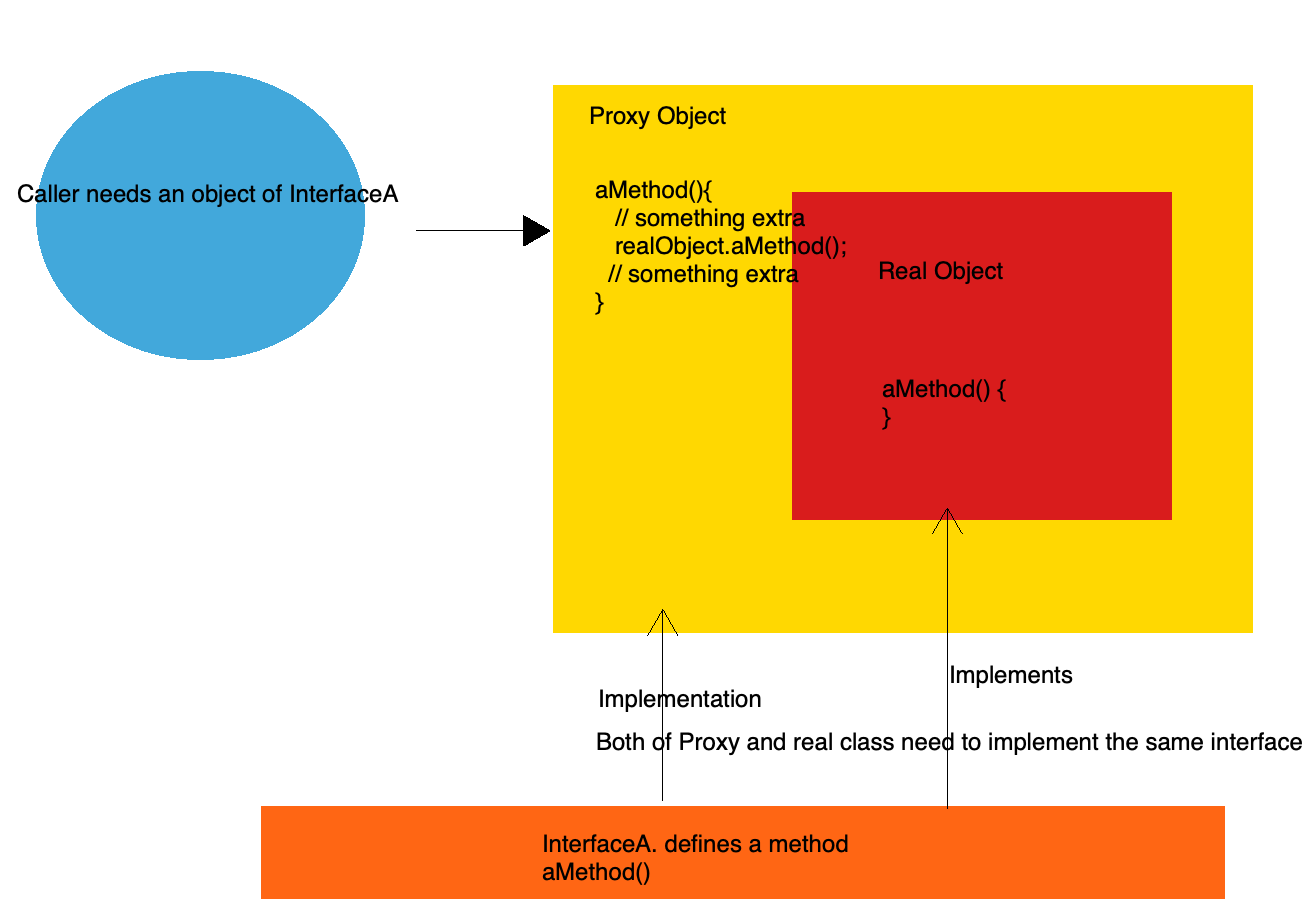
In proxy pattern, we create object having original object to interface its functionality to outer world.

In proxy pattern, we have the ****Proxy Class****, ****Target Class, Interface**** and the ****caller Class****.

Proxy object will contain a target object.

Both Proxy and target implement the same interface.

Caller refers the object using interface type. So both proxy and real object can be set to it.



##### **Static Proxy**

Means the proxy class is generated staticly. This requires the developer to know the target class before writing the code.

// Interface public interface DepartmentService {

public void saveDept();}

// Target Class public class DepartmentServiceImpl implements DepartmentService {

@Override

public void saveDept() {

System.out.println("Do save department logic.");

}

}

// Static proxy classpublic class DepartmentServiceProxy implements DepartmentService{

// Contains a target object

private DepartmentServiceImpl realSubject = new DepartmentServiceImpl();

@Override

public void saveDept() {

System.out.println("Do something extra before saving...");

realSubject.saveDept(); // call the real target method

System.out.println("Do something extra after saving...");

}}

// Caller classpublic class StaticProxyTest {

public static void main(String[] args) {

// DepartmentService ds = new DepartmentServiceImpl();

DepartmentService ds = new DepartmentServiceProxy();

doSomething(ds);

}

// using interface DepartmentService as parameter.

private static void doSomething(DepartmentService ds) {

ds.saveDept();

}}

##### **Dynamic proxy**

In dynamic proxy, developer doesn't have to know the target class or its interfaces. The proxy class is dynamically created by JVM during rumtime.

JDK provide dynamic proxy api.

// same interface and implementation.public interface DepartmentService {

public void saveDept();}public class DepartmentServiceImpl implements DepartmentService {

@Override

public void saveDept() {

System.out.println("Do save department logic.");

}}

// Now create a dynamic proxy. // This code is little bit complicated.// Doesn't require you to write.// Needs knowledge of classloader and reflection api.public class DynamicProxyTest {

public static void main(String[] args) {

DepartmentService ds = new DepartmentServiceImpl();

DepartmentService dsProxy = (DepartmentService) newProxy(ds);

doSomething(dsProxy);

}

private static void doSomething(DepartmentService ds) {

ds.saveDept();

}

public static Object newProxy(Object obj){

Class[] interfaces = obj.getClass().getInterfaces();

return Proxy.newProxyInstance(obj.getClass().getClassLoader(),

interfaces, new ProxyHandler(obj));

}}

public class ProxyHandler implements InvocationHandler{

private Object obj;

public ProxyHandler(Object obj) {

super();

this.obj = obj;

}

@Override

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

Object ret = null;

System.out.println("Before saving..");

if(method.getName() == "saveDept"){

ret = method.invoke(obj, args);

}

System.out.println("After saving..");

return ret;

}}

With Proxy, framworks are able to do fanncy stuff like AOP.