Introduction To

Design Patterns

By

Dr. Sunirmal Khatua,

Visvesvaraya Young Faculty Fellow, Govt. of India

Assistant Professor, University of Calcutta

What is a Design Pattern?

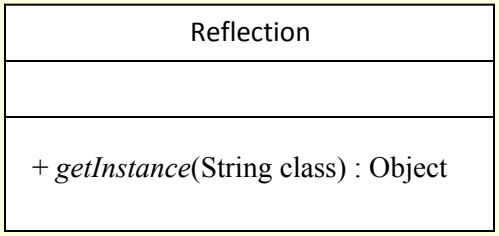
- Design Pattern is a recurring solution to standard problem in design.
- A design pattern has 4 basic parts:
  1. Name
  2. Problem
  3. Solution
  4. Consequences and trade-offs of application
- Language- and Implementation-independent
- The Gang of Four (GoF)
  - “Design Patterns: Elements of Reusable Object-Oriented Software,” Erich Gamma, Richard Helm, Ralf Johnson and John Vlissides, Addison-Wesley, 1995

Classification of Design Pattern

Creational	Structural	Behavioural
Reflections	Adapter	Interpreter
Singleton	Bridge	Template Method
Object Pool	Composite	Chain of Responsibility
Factory	Decorator	Command
Abstract Factory	Flyweight	Iterator
Builder	Facade	Mediator
Prototype	Proxy	Memento
		Observer
		State
		Strategy
		Visitor

Reflection Design Pattern

Allows the creation of objects of a class which is not known at compile time.




```
public class Reflection {

    public static Object getInstance(String class){
        Class c = Class.forName(class);
        return c.newInstance();
    }

}
```

Reflection is the process by which a program can observe and modify its own structure and behavior at runtime.

So where do we start?

- To start manipulating a class we must first get a hold of that class's "blueprint".
  - Using the `java.lang.Class` class
- There are two ways to do this, if the class is already loaded:
  - `Class theClass = ClassName.class;`
- Or if we need to cause it to load:
  - `Class theClass = Class.forName("class.package");`

Inspecting a Class

- After we obtain a Class object `myClass`, we can:
- Get the class name

```
String s = myClass.getName() ;
```

- Get the class modifiers

```
int m = myClass.getModifiers() ;
bool isPublic = Modifier.isPublic(m) ;
bool isAbstract = Modifier.isAbstract(m) ;
bool isFinal = Modifier.isFinal(m) ;
```

- Test if it is an interface

```
bool isInterface = myClass.isInterface() ;
```

- Get the interfaces implemented by a class

```
Class [] itfs = myClass.getInterfaces() ;
```

- Get the superclass

```
Class super = myClass.getSuperClass() ;
```

Class Methods for Locating Members

Member	Class API	List of members?	Inherited members?	Private members?
Field	getDeclaredField()	no	no	yes
	getField()	no	yes	no
	getDeclaredFields()	yes	no	yes
	getFields()	yes	yes	no
Method	getDeclaredMethod()	no	no	yes
	getMethod()	no	yes	no
	getDeclaredMethods()	yes	no	yes
	getMethods()	yes	yes	no
Constructor	getDeclaredConstructor()	no	N/A	yes
	getConstructor()	no	N/A	no
	getDeclaredConstructors()	yes	N/A	yes
	getConstructors()	yes	N/A	no

Example: retrieving **public** fields

```
Class c = Class.forName(“Dtest");

// get all public fields
Field[] publicFields = c.getFields();
for (int i = 0; i < publicFields.length; ++i) {
      String fieldName = publicFields[i].getName();
      Class typeClass = publicFields[i].getType();
      System.out.println("Field: " + fieldName + " of type " +
                                    typeClass.getName());
      }
```

Getting all methods

- Like Fields there are two ways to get Methods
  - `getMethods();`
    - Returns all the public methods for this class and any it inherits from super classes.
  - `getDeclaredMethods();`
    - Returns all the methods for this class only regardless of view.
- To get a specific method you call
  - `getMethod(String name, Class<?>… parameterTypes);`

Getting all methods

- For example, say we have this method:
  - `public int doSomething(String stuff, int times, int max){}`
- If we were trying to get this specific method we would have to call getMethod like this:
  - `getMethod("doSomething", String.class, int.class, int.class);`
  - `Class[] intArgsClass = new Class[]{ String.class, int.class, int.class } ;`
  - `getMethod("doSomething", intArgsClass);`

Building blocks

- To get the constructos we have the methods:
  - `getConstructors()`
    - Returns all public constructors for the class
  - `getDeclaredConstructors()`
    - Returns all constructors for the class, regardless of view
- We can again get specific constructors with:
  - `getConstructor(Class<?>… parameterTypes);`
    - Returns the constructor that takes the given parameters

Creating new objects

- Using Default Constructors
  - java.lang.reflect.Class.newInstance()

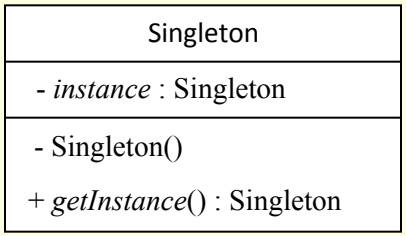
```
Class c = Class.forName("java.awt.Rectangle") ;
Rectangle r = (Rectangle) c.newInstance() ;
```

- Using Constructors with Arguments
  - java.lang.reflect.Constructor.java.lang.reflect.Constructor.newInstance(Object... initargs)

```
Class c = Class.forName("java.awt.Rectangle") ;
Class[] intArgsClass = new Class[]{ int.class, int.class }
  ;
Object[] intArgs = new Object[]{new Integer(12),new
  Integer(24)} ;
Constructor ctor = c.getConstructor(intArgsClass) ;
Rectangle r = (Rectangle) ctor.newInstance(intArgs) ;
```

Singleton Design Pattern

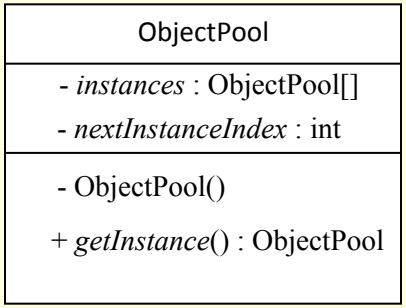
Restricts the creation of more than one object of a class.




```
public class Singleton {

    private static Singleton instance = null;

    private Singleton(){}

    public static Singleton getInstance(){
        if(instance == null){
            instance = new Singleton();
        }
        return instance;
    }

}
```

Object Pool Design Pattern

Allows the creation of more than one object of a class but restricts the number of objects to be created in a limit




```
public class ObjectPool{

    private static ObjectPool[] instances = new ObjectPool[10];
    private static int nextInstanceIndex = 0;

    private ObjectPool(){}

    public static ObjectPool getInstance(){
        ObjectPool instance = null;
        if(instances[nextInstanceIndex] == null){
            instances[nextInstanceIndex]= new ObjectPool();
        }
        instance = instances[nextInstanceIndex];
        nextInstanceIndex = (nextInstanceIndex +1) % instances.size();
        return instance;
    }

}
```