|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Section** | **Lecture** | **Slide** | **Slide Content** | **Transcript** |
| Introduction | Introduction to Building Automation Frameworks using Selenium and Java | About Me |  | Hi All! My Name is Rajesh Iyer. I am working in the Software Testing industry for over 10 years. I have architected and developed some sophisticated automation frameworks for some renowned clients in my organization. During my tenure I have majorly worked with Selenium RC and Selenium Webdriver, REST API testing and also Performance Testing. Though my educational background does not qualify me to be in the Software Testing industry, I consider myself lucky to get a start. And that was the only luck I had. After that it was all about continuous learning that has lead me to get into more challenging positions and also help some of my junior colleagues in their careers. |
| Why I created this course |  | Right since my school days, I have the passion for teaching. I have conducted trainings for my junior colleagues and also have provided mentorship to a handful of my colleagues. Now I want to widen my horizons and reach out to a lot of people. That is why I have created this course.  The other aspect is, I believe that the Automation is not just about knowing how to use Selenium Webdriver, but there is a science to developing a robust and sophisticated framework, that can be reused over and over to provide:   1. Better ROI for automation 2. Maintainability of the tests 3. Quick Turnaround Time in developing the tests   If you have these goals while developing a framework, your approach will be driven in this direction. The objective of this course is to help you achieve the above goals. |
| What to expect from this course |  | The first thing you can expect from this course is that there won’t be those boring theories about Java, Selenium or any of the different aspect of building frameworks. I don’t want the students of this course get all the information and when they come out of this course wonder “Aww! How do I apply this information now?”  You can expect a lot of interesting stories and real-world examples, that will help you understand every concept in detail and a few recommendations on how to apply the principles you will be learning throughout the course. So are you ready to get started? Let’s first learn Java or for some it will be brush up on Java. |
| Java Programming | OOPS Concepts | Introduction |  | OOP meaning Object Oriented Programming is a concept which is used quite extensively in Java. This is based on the concept of an Object which contains data often referred to as fields and code which is often referred to as methods. It allows you to structure the flow of your code and modularize you code. There are a few concepts that govern Object Oriented Programming. We will be studying them in details in the subsequent lectures. |
|  | Inheritance |  | The word inherit means is “to derive from the previous holder”. You inherit the property of your parent; similarly your son will inherit your property and so on. In Java, a child class inherits all the public and protected fields and methods of its parent class. Let’s try some examples to understand this concept.  Here we have created a class Parent with a field property. We have created a getter and setter method for this field. Now let’s create another class Child with a method inheritProperties and see if we can access the Property methods. To inherit Parent Class, we will be using the keyword “extends” which will allow us to inherit any class. We have used the ‘super’ keyword to access the methods of the Parent class. We will study about the super keyword in detail in the Java Section.  There are two types of inheritance: Single inheritance and multiple inheritance. In Single inheritance, the child class can inherit the features of his parent class. Whereas in Multiple inheritance, the child class can inherit the features of all the parent classes. To above example was Single inheritance. For multiple inheritance, let’s add a class GrandParent and this GrandParent class is inherited by the Parent Class. Here’s how we do it. Let’s say your GrandParent had a 100 Acre Farm Land, since your Parent has inherit it, you can inherit both your Parent’s as well as your GrandParent’s property. |
|  | Polymorphism |  | Polymorphism means many forms. In real world, every object moves in different ways. A car moves on road, a train moves on tracks, a plan moves in the air, a ship moves in the water. So the action is the same but the way it is done is different. In Java, there are two types of Polymorphisms: Run-time polymorphism also known as Method Overriding or Dynamic Polymorphism and Compile-time polymorphism also known as Method Overloading or Static Polymorphism. Let’s discuss each of them in detail  Run-time polymorphism or Method Overriding or Dynamic Polymorphism is used when you have inherited a class, but the same method will be used with a different implementation. For example, assume you are Software Developer, your Father was an Income Tax Consultant and your GrandFather was a Government Clerk. There is a method called getProfession in each of the classes. Then what will be outcome of each of the implementation. Let’s try this out.  You will notice the @Override annotation above getProfession method for Parent and Children class. This indicates that the method has been overridden from the parent. Now let’s try it out in the main class by creating objects for each of the above classes and see what each of them return. So you can see each of the professions represented for the given object.  Compile-time polymorphism or Method Overloading or Static Polymorphism is used when you have different parameters for the same method. For this let’s talk about the geometrical shapes. The area of the circle is πr2 and that of a rectangle is length x breadth. So to find the area of the circle you need to know the radius of the circle, but to find the area the rectangle you need length and breadth. In this case your Area method should have methods one with 1 parameter and the other with 2 parameters. Let’s try this out.  So we have created two functions with same method name getArea, but the parameters are 2 in the first case and 1 in the second case. The parameters can be different data types as well like Integer, String, Float or any object. The examples above may be good enough for you to understand when to use Overriding and when to use Overloading. But I will just summarize this discussion by saying that, use Overriding when you want the implementation of the child class to be different from the parent class and use Overloading when you have the same method but the number of parameters are different. |
|  | Abstraction |  | Now let’s move on to abstraction. Using this feature of the Object oriented programming, one can hide the implementation from the other. Let’s take the example of a Car. Every company has a different way of implementing the way the car accelerates or turns or the way the engine functions. But you as a user know that there is an accelerating system or the turning system or an engine system for your car. You don’t know how it works. So for you the accelerating system or the turning system or the engine system is an abstract. In java, there are two ways in which you can hide the implementations from the others. The first is called the abstract class, where you can have both concrete methods as well as abstract methods. The second one is called the interface, where you have only abstract methods which can be implemented by the class that implements this interface. I will be discussing both of them in detail in the Java section. |
|  | Encapsulation |  | Another way of hiding or gating the implementation is called Encapsulation. It can be referred to as access modifier. In every class, you can define the access modifier for each of the fields and methods.  The keywords often used to specify the access modifiers are: public, protected and private. In java, if an access modifier is not specified for a class, method or field, then the class, method or field is considered private. Now let’s talk one by one about each of these modifiers:  Public modifier: If the class, method or field is public, then it can be accessed by everyone in any package if they have access to that package or class. Let’s take an example:  Protected modifier: If the method or field is protected, then it can be accessed by all the classes within the package where this class is declared. In Java, you cannot have a protected class. Let’s take an example.  Private modifier: If the class, method or field is private, then, the method and field can be accessed only within the class. If the class is private, the it can be accessed only by the Classes in the same package.  I have created a table for your reference so that you can put it somewhere near your desk so that you can refer to it whenever you have to decide on the access modifiers for a class, method or field. |
| Basic Java | Introduction |  | At the beginning of the course, I said I will not be giving the boring theories of Java or Selenium, but I thought it worthwhile to share the concept of how Java actually runs your code.  The program that you write is a .java file.  There is a java compiler or javac program which converts the .java file to a .class file which is a bytecode data.  The core of the Java is the JVM i.e Java Virtual Machine. This JVM is responsible for running your application. The JVM contains a Byte Code Verifier and Class Loader, which will check the byte codes of the class file. Once the check passes, there is another section called the Java Intrepreter and JIT compiler.  Java Interpreter reads the bytecode and performs the functions accordingly. The disadvantage of the Java Interpreter is that it slow as it has to look up the meaning of each byte during execution.  This is overcome by the Just-In-Time Compiler, which compiles the bytecode to the native code just before the execution.  Java can be installed with Java Development Kit (JDK) or Java Runtime Environment (JRE). Both come with JVM packaged into it. However, JDK has some additional Java APIs which will assist the developers in |
| Installation |  |  |
| Creating a class and object |  | I want to make learning Java fun. So, assume you are planning to start a business of building cars of Mercedes and Porsche. Now before that, let’s think of a car. What are the different parts of a car? A car will have an engine, steering system, clutching system, braking system, accelerating system, mirrors, tyres, seats, body etc. All the cars have the same system, isn’t it? So how do you find same kinds of car on road? It is because they have been developed using the same blue print. This blue print is called a class. Once the class is created you can use it to create an object. Let’s say I want to create a Mercedes AMG 63 car. How do I do that? Here’s an example. I have created a car with different systems. If you look at this part, these are termed as fields of the class. They can also be considered the parameters of the class. This is called the methods in the class. Methods are majorly responsible to set or get the value of the fields or perform certain operations in the class like printing or converting. For example, if you look at getName method. It is responsible to give you the value of the name field. And the setName method is responsible to set the value of the name field. So our car blue print is ready. Now let’s create a Mercedes AMG 63 car. I have created different classes for different systems in the car and then assembled them in the car class. You can define each of the specification thoroughly and define your dream car. So here’s how we create a class and an object of that class. |
| Constructor |  | A class can have a method which has a same name as that of the class. This is called a constructor. So in case of a Car, the constructor method will have a name “Car”. You may notice that the class that I created does not have a constructor. In java, when you create a class, the java interpreter understands that there is a method Car() within that class which does nothing. OK, so now let’s create a Car constructor which will take the name of the car as a parameter. We will talk about this keyword later. So what this constructor does is, it stores the name that you will pass while creating an object of class Car. So, well modify our code with this new constructor. The output will remain the same. You can have multiple constructors for a class. This is called Class Overloading, which we will discuss in a later section. |
| static keyword |  | Now let’s say, we want to produce the car in our Car Factory and count the number of cars that were produced using the Car Class. So, this is where static variable is used. A static variable maintains its state irrespective of the instance of the object. So we need to make sure that every time a Car is created, it should be counted. So we will introduce a static variable noOfCars and every time a Car object is created, this variable will add by one. So let’s change the Car class as follows:   1. Add a static variable noOfCars 2. In the constructor, increment the noOfCars by 1 3. Add a static method getCarCount, to fetch the noOfCars value.   You may notice that I have created all the fields as private and have provided methods to access them. This is part of the encapsulation principle which we will learn in the next Section.  So let’s make these changes to our code.  Ok now let’s test it. To check if two cars were created, let’s create another car a Porsche. So we have two cars. Now let’s run the test  Fair enough we have two cars created and the noOfCars shows 2. |
| this keyword |  | You must have noticed the use of this keyword in the process of adding methods for the Car class. So what does “this” keyword mean?  The keyword “this” means whichever class instance you are in right now, access its method or fields and change its value. The operative word being “class instance” which means that it can be used for an instance. Meaning, you cannot access the static fields or methods using the “this” keyword.  The other usages of this is to invoke the current class constructor or a current class method or pass the current class instance as a parameter in a function or it can even return the current class instance from a method.  Some of the usages will be covered during the framework design. So for now just remember the different usages. |
| final keyword |  | In your Car Factory, you only manufacture cars which have a maximum speed limit of 100 as an example. And you do not want to change this maximum speed limit. This is when you can use the final keyword. Here’s how you achieve it. You will add a,   1. private final int maxSpeed and set it to 100.   Now, you will not be able to change anywhere in the methods where the Car class is invoked. Now you can only create a getMaxSpeed method, but you won’t be able to create a setMaxSpeed method. If you try to do it, it will give you a compile time error “cannot assign value to a final variable maxSpeed”. |
| Working with Conditional, Looping and Controlling statements and arrays |  | Let’s now work with Conditions and Looping and Controlling statements and Arrays. I will first define an array; so that we can build on them as well go along.  An array is a collection of objects of the same type. So it could be a collection of text or numbers or objects like car in our case. So how to instantiate an array of Cars? Here’s how we do it  Now, let’s work with the conditional statements if, if-else, if-else if.  In your Car Factory, you will provide texture design and black color to the car body if your customer is a Special Customer then this is how you will be doing it. Let’s introduce a variable customerType in the CarProduction class. So here’s how the code will be. Here we are accessing the Body of the Car and setting the color for it.  Next, let’s assume that if the customerType is an ordinary customer you will provide only White Color body car. Then here’s how you code will look like.  We will go one step further and introduce another customerType let’s say “Privileged” customer. For a privileged customer you are going to provide texture design for the car as well as give them a choice of color. So in this case, let’s introduce another variable, say “myColor”, which the user will specify. So assuming he specifies Black, we will create a Black Mercedes Car with texture design for that car. So we will use else-if between the “if” and “else” statements to achieve this. Now let’s try do it with switch-case which is other way of condition formatting. This is how it is done. We will pass the customerType to the switch parameter, then add the two cases “Privileged” and put statements there with a break at the end. Similarly for case “Special” and finally a default block which means the customer is of ordinary nature. The reason why we put break after each case or default block is that we will need to execute only any one of the 3 blocks. There may be situations where we may need to execute all three blocks or 2 of the three blocks etc. In such case the break is not required. In the if-else if world, these will be if statements alone. I am now going to move this complete code into a class CarFactory as a createCar method to which we will pass the customerType and color. This method will return an object of Type Car, So we can call this CarFactory in our future implementation.  Now let’s work with loops- for and while  As a worker, now you are given a register containing 5 customers and you need to create a Mercedes car for them. Each of those customers are different: 1 is a privileged customer, 1 is a special customer and remaining 3 are ordinary customers. Now you need to roll out 5 cars based on the specifications. The privileged customer wants an Ivory Silver car. So what do you do here?  You will add 2 Arrays one to store the customer type, the other to store the created cars. Let’s first achieve it using a simple for loop.  First let’s create an Array customerType and add the necessary values. This is an hard-coded array. Now our Car array will have a length equal to the customerType, so let’s create another Array cars with the length equal to customerType. The length of array means the number of objects the Array can hold.  Now let’s add a “for” loop. The first “for” loop has parts: initializer, condition, increment/decrement. We have initialized an integer “i”with 0 value. Next, we need to run through the complete customerType array, so let us run this loop till the value of the “i” is less than customerType array length. The incrementer statement will increment the value of “i” by 1. Within the loop let’s add the code for creating the car based on the customerType. We can use instantiate the CarFactory class and invoke the createCar method with the necessary paramaters and get a car which we will store in a Car instance. This instance will be added to the cars Array. Just before the loop starts, let’s add a variable cnt which has an initial value of 0 and increment after every execution. So for every car that is created we will add it to the cars array. So how does this work,  The value of “i” is 0 at the beginning, the array will get the first value of the customerType i.e. Privileged, now it will get into the if,else-if,else conditions and will satisfy the else-if condition, and the texture and color will be added accordingly. Now this car will be added to cars[0]. So we now have a car added to the car array. Next, cnt will be incremented by 1 and “i” will be incremented by 1. The second customerType is “Special”, so the if condition will be satisfied and the Car will be created for the Special customer. This way the loop will continue till it reaches the last element and finally the 5 cars will be added to the cars array.  Now let’s work with the alternative for loop. This loop was introduced in Java version 5. It will operate on iterable elements like Arrays and Collections, which we will be studying in a different section. Now let’s say, we want to print out the colors of each car in our cars array. So let’s use the alternative for loop for this. Here the first part before the colon is the single item object in the array and the second part is the array itself. So when we write Car car: cars, it means that pick up each car from the cars array and perform following operations. Using System.out.println we can print out the body color of the cars.  Now let’s achieve the same thing using a while and do-while loop. The while loop is similar to for loop, except that it has just the conditions part. The initializer is above the while loop statement and the increment/decrement is within the loop, mostly at the last statement of the loop.  The difference between a while and a do-while loop is that in the while loop we first provide the condition and then the statements to perform whereas in do-while, we first provide the statements to perform and then the condition. Here’s how we do it.  You can try working with the above loops for decrementing. Try printing the colors of the cars in reverse order. I am sharing the code snippet for you to verify. But do try it yourself. |
| Creating an Enum |  | Let’s move on to Enums. In your Car Factory, you produce only White, Black and Ivory Silver and Red cars. So how will you restrict you customer from choosing from the options available. So in such cases we can use Enum class which provides you with pre-defined constants. Here’s how you create an Enum. You can see that we use public enum and the class name. Here I have used ColorType as the class name. Now let’s add all the types. Since these are constants, let’s use all upper case for the values and the string corresponding to them. Once done, we need to add a constructor to the Enum. This is a private constructor, hence we don’t need to mention the modifier type. We will create a class field “color” which will store the “color” value. We will make this color field final, so that we don’t modify them. The number of parameters that a constructor will have will be the number of parameters that you set for each Enum constants. In this case the enum BLACK has just one parameter “Black” hence the constructor will have just one parameter. In java, one of the advantages of using the enum is that, it provides a static method values() for the enum, which can be used to iterate over the enum values. Let’s add a method which will print all the enum values. We will use the for loop to iterate over each enum. Also let’s add a method to get the color which we will use to print the color. So here’s how our enum will look like.  Now in our Car factory, let’s provide the Customer the access to this Enum. So our myColor will not be of type String, but it will be of Type ColorType. So here are the modifications, we need to do to our code. Similarly, try a scenario where only Black and Ivory Silver come with texture. I have shared the code snippet for it as well. But it is better to try it yourself first before inferring into the snippet. |
| Creating an abstract class |  | Assume that in your Car factory, you have 2 units: one producing a Mercedes cars and another unit producing Porsche cars. The mechanism for producing both cars is same except the system for producing the logo of the car. In this case, each of these units will have their independent workflows. So our CarFactory will have two different units. Let’s call them Mercedes Unit and Porsche Unit. They will have the Logo workflow implemented differently for each other. In order to do this in our Java code. We will have to make our Car Factory class abstract and provide an abstract method logoImplementation which will have its own implementation in each of these units. Since Car Factory is an abstract class, we need a concrete class which will have the logoImplementation. This is covered in much in the OOPS concepts section. So let’s create two classes: MercedesUnit and PorscheUnit and implement the logoImplementation method. Here’s how it is done. I have created an instance of CarFactory which creates the MercedesUnit or PorscheUnit. This is Liskov’s Substitution Principle which I will cover as part of the SOLID principles Section. This means that your parent class instance can hold the instance of your child class. This is how we work with Abstract Class.  To conclude you can use Abstract class when you have all the methods common except one or two methods which have different implementations. |
| Creating an interface and implementation |  | Now, you have two production units – one for Mercedes and the other for Porsche. The systems used in each of the cars may be different, but they should be able operate in same manner. They should be able to provide the same features, isn’t it. For this there should be a common agreement between the two units. Your car should have these capabilities:   1. accelerate 2. brake 3. changeGears 4. signalTurn 5. turn   to name a few. So doesn’t matter if you create a Mercedes car or a Porsche Car, these capabilities should be available. This can be achieved in Java with the use of an interface. The second advantage is that you can hide the way, the above capabilities are achieved because you don’t need to detail how the car will accelerate. You can hide that mechanism from the others. So now let’s define an interface called DriveCar with the above 5 capabilities.  Ok now let’s create a MercedesCar class which is a child class of Car and implement the methods in the interface. We will use simple print statements for these methods. We need to add a constructor for the MercedesCar with one parameter name because the parent class has the constructor method with one parameter. And below that are the implemented methods of the interface DriveCar. We will do the same for the Porsche Car. This is an example of Polymorphism, which we covered in the OOPS concepts Section.  Some of you may be wondering, this can be achieved using abstract as well. So why do we need an interface. I would agree with you, but the other advantage of interface is that you can implement multiple interfaces for a class, whereas you can extend only one abstract method. So the next question may go, so when to use an interface and when to use an abstract? This is what I usually do:  Whenever, there are common methods and few distinct methods provided that the distinct methods are in the same pretext of implementation then I use an abstract. By same pretext, what I mean is that it is one of the functions of the class but can be implemented differently. Now in this case, the base class Car deals with the different specifications of the car, so it is not concerned with the different operations of the car. Hence, we will differentiate the specifications and operations and define it differently. This follows the 1st principle of the SOLID principles which is called Single Responsibility Principle which we will discuss in detail in later sections.  Interfaces can be used, when you have different operations which need to be available for a given class. There can be a case Turning and Signalling Turn is part of one interface, Accelerating is part of another interface and so on, then in that case you will need to have multiple implementations which can be achieved by interface and not abstract. I know what question you have now. Why should we have different interfaces for each of these functions. Good question. Hold it till you learn the Interface Segregation Principle in the SOLID principles.(giggle)  This concludes the topic of Interfaces. |
| File Handling |  |  |
| String and String Manipulations |  |  |
| Regular Expression handling |  |  |
| Advanced Java | Collections framework |  |  |
| Annotations |  |  |
| Generics |  |  |
| Reflections |  |  |
| Exception Handling |  |  |
| SOLID principles | Introduction |  | In this section, I am going to talk about some of the basic principles of designing. In automation testing, it pays to develop a robust framework. Remember, what I discussed in the introduction. The main goals of automation:   1. Better ROI for automation 2. Maintainability of the tests 3. Quick Turnaround Time in developing the tests   To achieve this, it is very important to create a good design for a framework. So how do we achieve a good framework design. There are certain principles that govern this. One such design principle is called SOLID principles. Each letter of the word SOLID is a principle. Let’s study them one by one. |
| Single Responsibility Principle |  | The letter “S” stands for Single Responsibility Principle, which means that it is a good design if the Class or a method that you create is assigned only one responsibility.  Let’s take the example of the Car class that we created previously. All the methods in the class had a specific task: to get or set the value of the fields. What if we gave it both the responsibilities? One to set the method and two to get it as well. Now what will you do, if you want to just know the color of the car. The car class will say, input your color, only then I will give you my color. You would say, “No, I just want to know the color so that I can decide whether to buy you or not”. The car would say, “No but I can’t tell you my color unless you tell me what color you want”. And this will go on and on. Instead, how about this: “Hey car, tell me what color you are?” The car replies: “I am white”. You say “Ok thank you. I will buy you” or “Ok thank you. I will check for some other color”. If the color is not set, the car would reply “I don’t have any color, you can set me with your favorite color and I will be yours”. You could say “Ok thank you. So please color yourself Red, I will buy you”. Would that be a great car you would like to buy?  Make sure that the class also has a single responsibility. Like the objective of a Car class is to define the specifications of a car. It is not concerned about how to operate the car. |
| Open Closed Principle |  | Let’s move on to the next principle. The Letter O stands for Open Closed Principle. The Open Closed Principle states that the method or class should be open for extension but Closed for modification. Again let’s take a real life example and then bring talk about this principle in terms of Java language.  You are taking this course on your laptop/computer right now. Can you take a print-out of any of the resources that I have added to in the Resources section? Done? What did you do? You simply attached a printer cable to your laptop/computer and printed out the code. Did anyone of you do this? Opened your laptop/computer and placed your printer inside this laptop/computer (can’t imagine how this would work?), then closed it and went ahead with printing? No right? That’s because your computer is open for extension, but closed for modification. Similarly to design an automation framework, your classes should be open for extension and closed for modification. Now let’s look at this from the Java example. Suppose, your car needs an Air Conditioning Unit what could you do? Two things:   1. Modify your Car class and add to it Air Conditioning field and variables, or 2. Extend your Car class and add to it Air Conditioning field and variables.   Will both of them work? Yes of course. But here’s the deal, can you use the Car class and develop a non-AC car? No right? So it is always a good idea to Extend than to modify you class. Modification is suggested only if there is a defect or you are sure that the feature you are adding should be the core feature of that class. |
| Liskov Substitution Principle |  | The letter “L” stands for “Liskov Substitution Principle” which states a Class A can be substituted by its subclass B without altering any of the desirable properties of the program.  Let’s take the example of Car class. If I create an instance of Car or create an instance of MercedesCar which extends Car, then getColor() of the car will return me the color of the car, isn’t it? So let’s try this with an example. Here’s a code snippet that shows you how it works. |
| Interface Segregation Principle |  | The letter “I” stands for “Interface Segregation Principle” which states that a client can use only those features that he needs and should not be forced to use those features that he does not want to use.  Let’s say, you have an interface for your Car class, say Operate, which has pressClutch(), shiftGear(), brake(), signalTurn(Direction direction), turn(Direction direction) as shown in the code snippet. So in order to operate the car, you will use this Operate interface. Sounds good?  Hey but wait. What if I have an automatic gear car? What if there is automatic Clutching system for my car? I am forced to implement the pressClutch/ shiftGear method to operate the car which violates the Interface Segragation Principle. So the smart option is to have these interfaces segregated like this, OperateClutch, OperateGear, OperateBrake or OperateTurns and implement only those interfaces that need to be implemented. So for an auto Clutch system car, you may want to exclude OperateClutch or OperateGear can be excluded for an Auto Gear car. |
| Dependency Inversion Principle |  | The last letter “D” stands for “Dependency Inversion Principle” which states that High level modules should not depend on low level modules. Both should depend on Abstractions. Also, the Abstractions should not depend on Details, but details should depend on Abstractions. I know this is a bit difficult to understand. It was for me too. But if you understand how to use it, it will become very easy.  The switch board is an example of Dependency Inversion Principle. Now here’s how it works. You have a Fan, Fridge, Laptop plugged into your switch-boards. Now, each switch on the switch-board corresponds to any of the appliances. Does the appliance have to care about what the operator has to do operate the appliance? No, isn’t it? All the appliance has to know is if someone operates the switch, I will have to start. Now does the operator of the appliance have to care, what exactly happens when he turns the fan on? No isn’t it? So the operator and the appliance are connected with each other through an interface i.e. switch board. Now let’s translate it into java code. Here’s how you do it. We will first create 3 appliances:   1. Fan with methods turnOn and turnOff 2. Fridge with methods startCooling and saveEnergy 3. Laptop with methods chargeBattery and useBattery   Now we have a Switchboard with turnOn and turnoff methods. We will be using a Mechanical Switchboard which will implement the Switchboard. Now let’s try a main class and see how this works.  So, now you can see that using the dependency inversion, we can do the following:   1. Add many more appliances to the switchboard 2. Have any type of switchboards: mechanical, electronic and what not. |
| Design Patterns | Introduction |  | Now that we have studied the different SOLID principles, Let’s study some of the standard design patterns that have been devised. These are best practices design pattern, which have helped develop robust and extendible application systems. From a testing perspective, I don’t want to confuse you with a huge list of design patterns, but I will choose only those which can probably be used to develop a robust Test Automation Framework. Some of the developers out there might think at the end of this chapter, “Why did he not cover that design pattern?” My reason upfront is pretty straight forward. I have probably not used it or I don’t consider it important from the Testing perspective. Having said that, I may come back on this and add a few more patterns, just in case, I use it and feel it is of utmost importance for the budding Testing Architects like you. Let’s get started:  The Design patterns are divided majorly into 3 different categories:   1. Creational Patterns 2. Behavioral Patterns, and 3. Structural patterns   Now let’s take them one by one and study different patterns within these categories |
|  |  | Creational Patterns |  | As the name suggests, Creational Patterns deals with the way an Object is created. In your car factory, there are different ways of creating a car. So your process of creating a car is what we call as Creational Patterns. The output of creational pattern will more or less be an instance of a class that is created. Let’s look at some of the types of Creational Patterns.  Singleton Pattern: A singleton pattern means creating only one instance of Class. In Singleton pattern, there is a static method within the class which will create an instance of this Class. This provides a global point of access to the application. Let’s take an example of your Car Factory. You have 5 employees in your factory who are producing a car. At this point it does not matter which car he produces either Mercedes or Porsche. You want to maintain the report of how many cars they have produced for the day. So you have a Whiteboard, where everytime an employee produces a car, you add up a number. Now do we need multiple Whiteboard to maintain this count? No, right? So, this is an example of a singleton pattern. Let’s represent it in the java class. We will name this class as Counter. We will create a static method createInstance. Then return this counter instance. We will have a static field called instance which is of type Counter. We will add the constructor as private. We will add another static integer called counter which will be initiated to zero when we create the singleton instance. Everytime a car is produced, we will update the counter using the method updateCounter and finally, we will do a getCounter to find how many cars where produced. So you can see here that you get a count 10 here.  The advantages of using such a pattern are it provides a shared resource for the application. It consumes memory only when the call to the constructor is made, hence allowing better management of the memory. This will help when the class has a lot of resources. Compared to having static classes, it is better in terms of deriving the class from an interface. Which means if you have an interface which needs to be implemented by this Singleton class, you can do it, but not with static classes. The disadvantages however are, it violates the Single Responsibility Principle. Why? Because it does two things: 1. It creates an instance of itself and 2. It does various operations that it is responsible to do. Singleton classes cannot be sub-classed meaning you cannot have a class which extends Singleton classes. Having said that, the best example to use the Singleton pattern is when you have shared resources like in our case the Whiteboard.  The next design pattern I wish to cover is Factory Pattern. This is the most extensively used Pattern in Creational patterns. As the name indicates it is a factory which creates object from a class. The difference only is it does not expose the instantiation logic to the client and it refers to the newly created object through an interface.  We will try this out with an example. In this example, I am going to create an interface Car which has only one method create(). I will implement this interface for 2 new classes: Mercedes Car and Porsche Car. For now, I am just doing a System.out.println, we can write the complete implementation in it. Now I need to have a factory to create these cars. So let’s introduce a CarFactory which based on the request creates either a Mercedes Car or a Porsche Car. In the CarFactory, we have a method createCar(String name) which based on the name will create the appropriate car. So using if-elseif-else statements we have created our condition. Since our factory only makes Mercedes and Porsche car, any other request will return null. So, this is how Factory pattern works. So what are the advantages? Suppose you as the owner of the Car factory tomorrow decides to produce Ferrari cars in your Factory, then all you need is to add the Ferrari Car Class which will extend Car interface. Ok, now for the disadvantage, you have added Ferrari Car class, now your factory should know how to take request and create Ferrari cars, for that you will have to modify the CarFactory to add this condition as well. Which means it violates the Open Closed Principle, isn’t it?  So to overcome this limitation, let’s study the Factory Method Pattern. How does it work? Well, in Factory Method Pattern, we will have a method that is implemented in the Concrete Class and have a concrete method which will call this method. Let’s take the same example that we took in Factory Pattern. You as the owner of the Car Factory, hopes to add more and more car brands to your production portfolio and want to set up different manufacturing units within your Car Factory. So your Car Factory is now an abstract class which will have concrete production factories within itself. So when you call the in-charge of Mercedes and tell him create a car, he will produce a Mercedes car. So let’s achieve it using our Java code. Here’s how. So the advantages are that, it separates the family of classes from the main application meaning, if I tell the Mercedes Car Factory to create a car, it is going to create a Mercedes Car. It won’t ask me, “Which car to make?” Instead, I decide and call the respective supervisor. By doing so, we have also left how to create the car to the specific Factory itself. So how to do is up to the factory and what to create is up to the application. The only disadvantage however, is that I cannot use this factory to anything other than Car. Suppose, Mercedes and Porsche decide to venture into Motor bikes, you won’t be able to achieve that using this factory. Meaning the Objects and the Factory cannot be used to create some other Objects which do not extend the base class or factory.  To overcome this, we will study the Abstract Factory Pattern. In the abstract factory pattern, you can have 2 different factories with their products, but all integrated together and available to the client to decide which ones to pick. Let’s work with the same Car Factory example, but this time we will name it as Vehicle Factory. This vehicle factory will have 2 abstract methods: createCar and createBike. We have added Mercedes Car. Now let’s add Mercedes Bike. We will also add Porsche Bike. Now we modify the Mercedes Car Factory and name it to Mercedes Vehicle Factory and extend it to Vehicle Factory. We will have to implement createCar and createBike methods. These will return a new instance of Mercedes Car and Mercedes Bike. Same would be the case with Porsche Vehicle Factory. Now, we will add another Class called Factory maker and use Singleton Pattern to create an instance of Vehicle Factory which will either be a Mercedes Vehicle Factory or Porsche Vehicle Factory based on the string that we pass to the static method getFactory(String brand). Now let’s test this code. We will change the Car Factory Demo to first create the VehicleFactory instance and pass “Mercedes” to the FactoryMaker.getFactory method. Now let’s call the 2 methods and create a car and a bike instance. After that we will create car and bike using the create method of the car and bike. This way we can have any class family yet have a factory pattern create instances of each of the classes.  So we covered Singleton patterns and all the different Factory patterns. There are many more creational patterns like Prototype, Builder, or Object Pool pattern. But I am excluding them for this course, since I don’t feel they fit in any of our Framework design. But if you personally need any help implementing this pattern. Do reach out to me. |
|  |  | Behavioral Patterns |  | The creational patterns dealt with the way an Object was created. So the outcome of all of the Creational Patterns was an instance of an object. The behavioral pattern more or less deals with the communication and processes within the application. Let’s take an example, In your Car Factory, you need to produce cars based on the specification. There are different processes involved in the production of a car. So how you design the communications and the processes involved is what we will study in Behavioral Pattern. Let’s study some of the most popular Behavioral Patterns.  The chain of responsibility design pattern is one of the most popular behavioral patterns. This behavioral pattern attaches neither with the sender nor the receiver of the request, but processes the request to either check which process will actually handle the request or the request goes through all the processes to become a final product. Let’s work this out using an example. We will head to your Car Factory once again. Here, you will have different processes in producing the car, there will be one process which will be adding an engine, one process which will be adding the accelerating system, another one adding the braking system, and one more to add the clutching system. Finally, the body of the car is painted and texture is added. All this is a part of the process. So we now create a class called Process which has 2 methods: setNextProcess and processCar. We will write processes specific to each process so let’s say EngineProcess, AccelaratingSystemProcess, BrakingSystemProcess, ClutchingSystemProcess and BodySystemProcess. We will then align these processes in our Client(in our case, the main class). So this is how we do it. We will start with the EngineProcess and set the next processes using the setNextProcess Method. Then in our implementation of each method, we will call the next process’s processCar method. This way, we can achieve updates to a single object through different processes. This is how Chain of Responsibility is used. Now let’s move on to Command Pattern.  A collection of instructions is called Command. This design pattern is based on this definition. Now let’s take an example of the implementation of this design pattern. You have decided to start your own showroom where you will be showcasing and selling the car that you manufacture. Now in this showroom, you will buy the car from your Car Factory and sell it to your customer. There are only two things that your showroom will do: buy and sell. Now with this background let’s see how command pattern works. We will create an interface Order which will have the abstract method execute. We will create another Class named Trade, where we will have 2 methods: buy() and sell(). And then our Showroom Class will have two methods called takeOrder with parameter Order and placeOrders(). In the Showroom class let’s have a field of type List with type Order. Now, let’s create two classes which implement the Order interface: BuyOrder and SellOrder. Both of them have the a field of type Trade. The constructor will take in a Trade parameter and will set this to this trade field. So, every time there is a transcation, the Order will be inserted into the List, which will then queue this orders and perform the respective orders. This allows encapsulation of the request into a single action. So when you execute a buy request it buys the car and when you execute a sell request it will sell the car. The queuing helps in taking the request at one point and allowing you to execute it at other time. So when to use Command Pattern? Here’s when,   1. If you want to parameterize an Object depending on the action. 2. When you want to queue the request and execute it at a different time. 3. As a result of queuing, it acts as a memory to perform undoable actions.   This is about Command Design Pattern. |
|  |  | Structural Patterns |  |  |
| Frameworks | Types of Automation Frameworks | Data driven Framework |  |  |
| Keyword driven Framework |  |  |
| Behaviour Driven Framework |  |  |
| Hybrid Framework |  |  |
| Different components of an Automation Framework | Selenium Webdriver |  |  |
| TestNG Framework |  |  |
| Maven |  |  |
| Dataproviders |  |  |
| Custom Page Creation |  |  |
| Object Repository Management |  |  |
| Reporting |  |  |
| Continuous Integration with Jenkins |  |  |
| Selenium Webdriver | Introduction |  |  |  |
| How does Selenium Work? |  |  |  |
| Different implementations of Webdriver |  |  |  |
| Working with UI Components Using Webdriver |  |  |  |
| Handling Waits in Selenium Webdriver |  |  |  |
| Taking Screenshots using Remote Webdriver |  |  |  |
| Working with Actions API |  |  |  |
| Using JavascriptExecutor |  |  |  |
| Conclusion |  |  |  |
| Maven | Introduction |  |  |  |
| What is Maven? |  |  |  |
| What is pom.xml? |  |  |  |
| How to add dependencies to your project |  |  |  |
| How to add plugins to your project |  |  |  |
| Conclusion |  |  |  |
| TestNG Framework | Introduction |  |  |  |
| What is TestNG |  |  |  |
| Difference between TestNG and Junit |  |  |  |
| Different annotations in TestNG |  |  |  |
| Working with Dataprovider annotation |  |  |  |
| Dependent tests using TestNG |  |  |  |
| Grouping of tests in TestNG |  |  |  |
| Running tests in parallel |  |  |  |
| Writing Sample scripts with TestNG |  |  |  |
| Dataproviders | How to manage data in your framework? |  |  |  |
| Understanding the Apache POI API |  |  |  |
| Creating an utility to read an excel file using Apache POI API |  |  |  |
| How to pass data to the TestNG test |  |  |  |
| Object Repository Management | Identifying Web Elements |  |  |  |
| Advanced CSS and XPATH identifiers |  |  |  |
| Using Firebug and Firepath to identify elements |  |  |  |
| Creating Custom Page Methods | What is a Page Factory Pattern? |  |  |  |
| Creating a Page using Page Factory pattern |  |  |  |
| Creating Object Repository for a Page |  |  |  |
| Reporting and Logging | How to report the execution status |  |  |  |
| Adding logs to your tests |  |  |  |
| Integrating the reports using ATU reports |  |  |  |
| Conclusion | Final Thoughts |  |  |  |