**SOLID Principles in Java [with Examples]**

By Lokesh Gupta | Filed Under: [Java Best practices](https://howtodoinjava.com/best-practices/)

Classes are the building blocks of any java application. If these blocks are not strong, the building (i.e. application) is going to face the tough time in future. This essentially means that not so well-written can lead to very difficult situations when the application scope goes up or application faces certain design issues either in production or maintenance.

On the other hand, set of well designed and written classes can speed up the coding process by leaps and bounds, while reducing the number of bugs in comparison.

In this tutorial, We will discuss **SOLID principles in Java with examples** which are 5 most recommended design principles, we should keep in mind while writing our classes. They also form the [**best practices**](https://howtodoinjava.com/category/best-practices/) to be followed for designing our application classes.

Table Of Contents

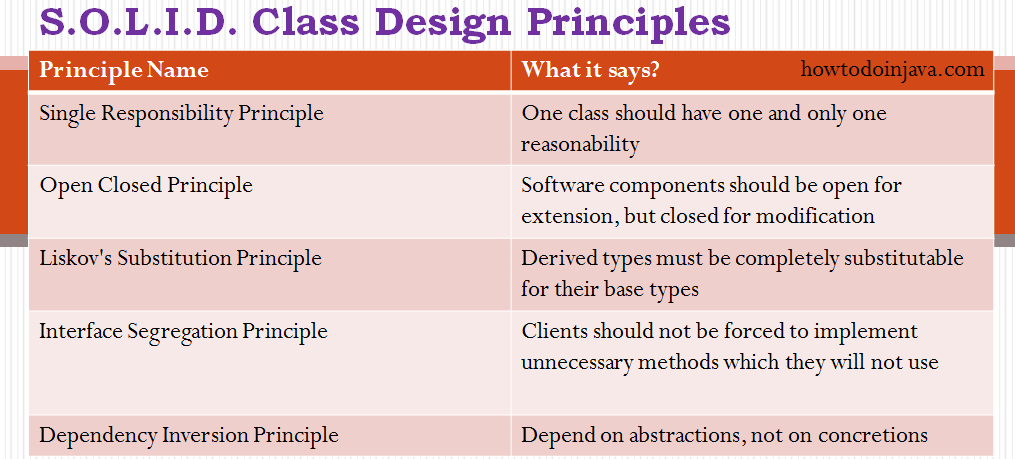
[1. Single Responsibility Principle](https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#SRP)

[2. Open Closed Principle](https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#OCP)

[3. Liskov's Substitution Principle](https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#LSP)

[4. Interface Segregation Principle](https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#ISP)

[5. Dependency Inversion Principle](https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#DI)



**5 java class design principles**

Lets drill down all of them one by one.

**1. Single Responsibility Principle**

The name of the principle says it all:

“One class should have one and only one responsibility”

In other words, we should write, change and maintain a class for only one purpose. If it is model class then it should strictly represent only one actor/ entity. This will give we the flexibility to make changes in future without worrying the impacts of changes for another entity.

Similarly, If we are writing service/manager class then it should contain only that part of method calls and nothing else. Not even utility global functions related to module. Better separate them in another globally accessible class file. This will help in maintaining the class for that particular purpose, and we can decide the visibility of class to specific module only.

**1.1. Single Responsibility Principle Example**

We can find plenty of classes in all popular Java libraries which follow single responsibility principle. For example, in log4j, we have different classes with logging methods, different classes are logging levels and so on.

In our application level code, we define model classes to represent real time entities such as person, employee, account etc. Most of these classes are examples of SRP principle because when we need to change the state of a person, only then we will modify a person class. and so on.

In given example, we have two classes Person and Account. Both have single responsibility to store their specific information. If we want to change state of Person then we do not need to modify the class Account and vice-versa.

|  |
| --- |
| Person.java  public class Person  {      private Long personId;      private String firstName;      private String lastName;      private String age;      private List<Account> accounts;  } |
|  |
| Account.java |
| public class Account  {      private Long guid;      private String accountNumber;      private String accountName;      private String status;      private String type;  } |

**2. Open Closed Principle**

This is second important rule which we should keep in mind while designing our application. **Open closed principle** states:

“Software components should be open for extension, but closed for modification”

What does this mean?? This means that our classes should be designed such a way that whenever fellow developers wants to change the flow of control in specific conditions in application, all they need to extend our class and override some functions and that’s it.

If other developers are not able to design desired behavior due to constraints put by our class, then we should reconsider changing our class. I do not mean here that anybody can change the whole logic of our class, but he/she should be able to override the options provided by software in unharmful way permitted by software.

**2.1. Open Closed Principle Example**

If we take a look into any good framework like struts or spring, we will see that we can not change their core logic and request processing, but we modify the desired application flow just by extending some classes and plugin them in configuration files.

For example, spring framework has class DispatcherServlet. This class acts as **front controller** for String based web applications. To use this class, we are not required to modify this class. All we need is to pass initialization parameters and we can extend it’s functionality the way we want.

Please note that apart from passing initialization parameters during application startup, we can override methods as well to modify the behavior of target class by extending the classes. For example, struts Action classes are extended to override the request processing logic.

|  |
| --- |
| Extending Struts Action |
| public class HelloWorldAction extends Action  {      @Override      public ActionForward execute(ActionMapping mapping,                                  ActionForm form,                                  HttpServletRequest request,                                  HttpServletResponse response)                                  throws Exception      {            //Process the request      }  } |

**3. Liskov’s Substitution Principle**

This principle is a variation of previously discussed open closed principle. It says:

“Derived types must be completely substitutable for their base types”

It means that the classes fellow developer created by extending our class should be able to fit in application without failure. This requires the objects of your subclasses to behave in the same way as the objects of your superclass. This is mostly seen in places where we do run time type identification and then cast it to appropriate reference type.

**3.1. Liskov Substitution Principle Example**

An example of LSP can be [custom property editors](https://howtodoinjava.com/spring-boot/custom-property-editor-example/) in Spring framework. Spring provides property editors to represent properties in a different way than the object itself e.g. parsing human readable inputs from HTTP request parameters or displaying human readable values of pure java objects in view layer e.g. Currency or URL.

Spring can register one property editor for one data type and it is required to follow the constraint mandated by **base class** PropertyEditorSupport. So is any class extend PropertyEditorSupport class, then it can be substituted by everywhere base class is required.

For example, every book has an ISBN number which is in always a fixed display format. You can have separate representations of ISBN in database and UI. For this requirement, we may write property editor in such a way –

|  |
| --- |
| **IsbnEditor.java** |
| import java.beans.PropertyEditorSupport;  import org.springframework.util.StringUtils;  import com.howtodoinjava.app.model.Isbn;    public class IsbnEditor extends PropertyEditorSupport {      @Override      public void setAsText(String text) throws IllegalArgumentException {          if (StringUtils.hasText(text)) {              setValue(new Isbn(text.trim()));          } else {              setValue(null);          }      }        @Override      public String getAsText() {          Isbn isbn = (Isbn) getValue();          if (isbn != null) {              return isbn.getIsbn();          } else {              return "";          }      }  } |

**4. Interface Segregation Principle**

This principle is my favorite one. It is applicable to interfaces as single responsibility principle holds to classes. ISP says:

“Clients should not be forced to implement unnecessary methods which they will not use”

Take an example. Developer Alex created an interface Reportable and added two methods generateExcel() and generatedPdf(). Now client ‘A’ wants to use this interface but he intend to use reports only in PDF format and not in excel. Will he be able to use the functionality easily?

NO. He will have to implement both the methods, out of which one is extra burden put on him by designer of software. Either he will implement another method or leave it blank. This is not a good design.

So what is the solution? Solution is to create two interfaces by breaking the existing one. They should be like PdfReportable and ExcelReportable. This will give the flexibility to user to use only required functionality only.

**4.1. Interface Segregation Principle Example**

The best place to look for IPS examples is Java AWT event handlers for handling GUI events fired from keyboard and mouse. It has different listener classes for each kind of event. We only need to write handlers for events, we wish to handle. Nothing is mandatory.

Some of the listeners are –

* FocusListener
* KeyListener
* MouseMotionListener
* MouseWheelListener
* TextListener
* WindowFocusListener

Anytime, we wish to handle any event, just find out corresponding listener and implement it.

|  |
| --- |
| MouseMotionListenerImpl.java |
| public class MouseMotionListenerImpl implements MouseMotionListener  {      @Override      public void mouseDragged(MouseEvent e) {          //handler code      }        @Override      public void mouseMoved(MouseEvent e) {          //handler code      }  } |

**5. Dependency Inversion Principle**

Most of us are already familiar with the words used in principle’s name. DI principle says:

“Depend on abstractions, not on concretions”

In other words. we should design our software in such a way that various modules can be separated from each other using an abstract layer to bind them together.

**5.1. Dependency Inversion Principle Example**

The classical use of this principle of bean configuration in Spring framework.

In spring framework, all modules are provided as separate components which can work together by simply injected dependencies in other module. This dependency is managed externally in XML files.

These separate components are so well closed in their boundaries that we can use them in other software modules apart from spring with same ease. This has been achieved by dependency inversion and open closed principles. All modules expose only abstraction which is useful in extending the functionality or plug-in in another module.

These were **five class design principles**, also known as **SOLID principles**, which makes the best practices to be followed to design our application classes.

Happy Learning !!