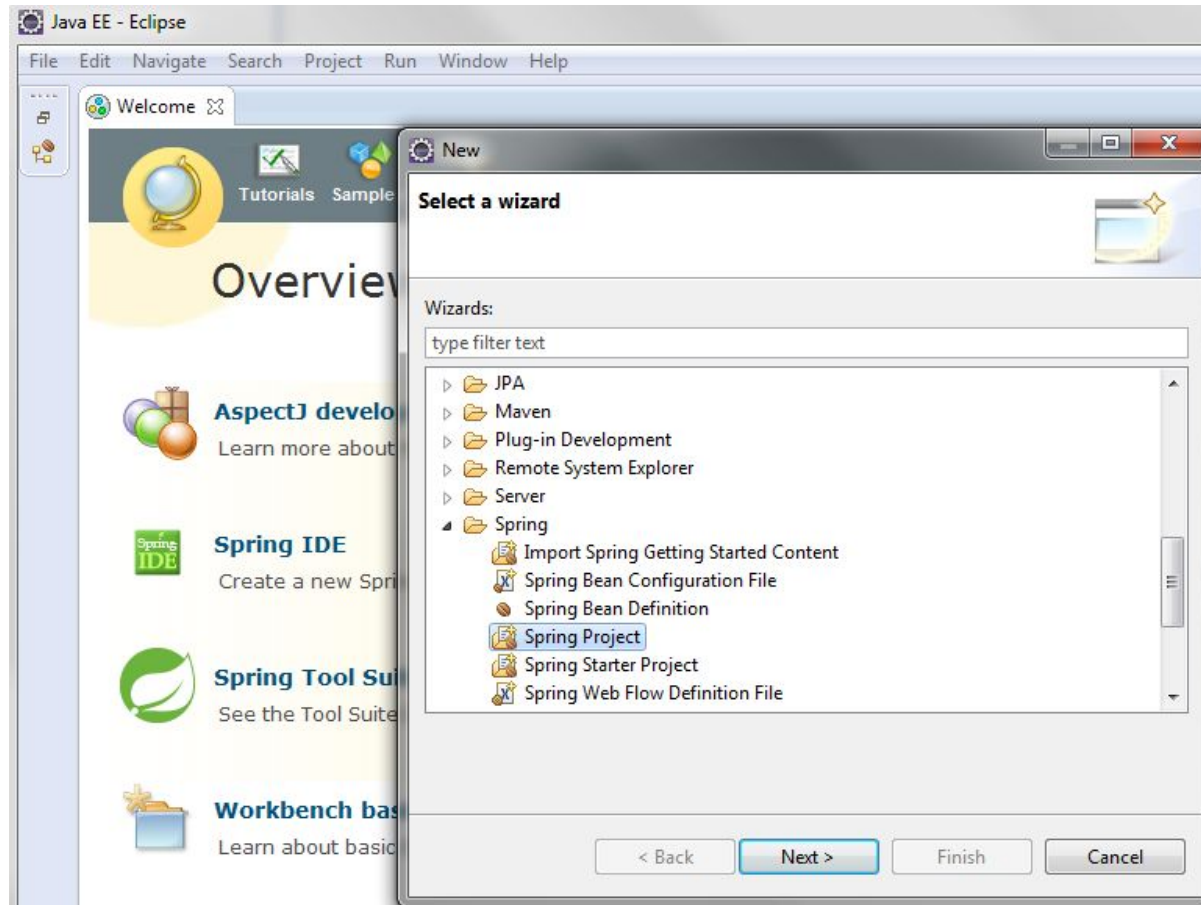
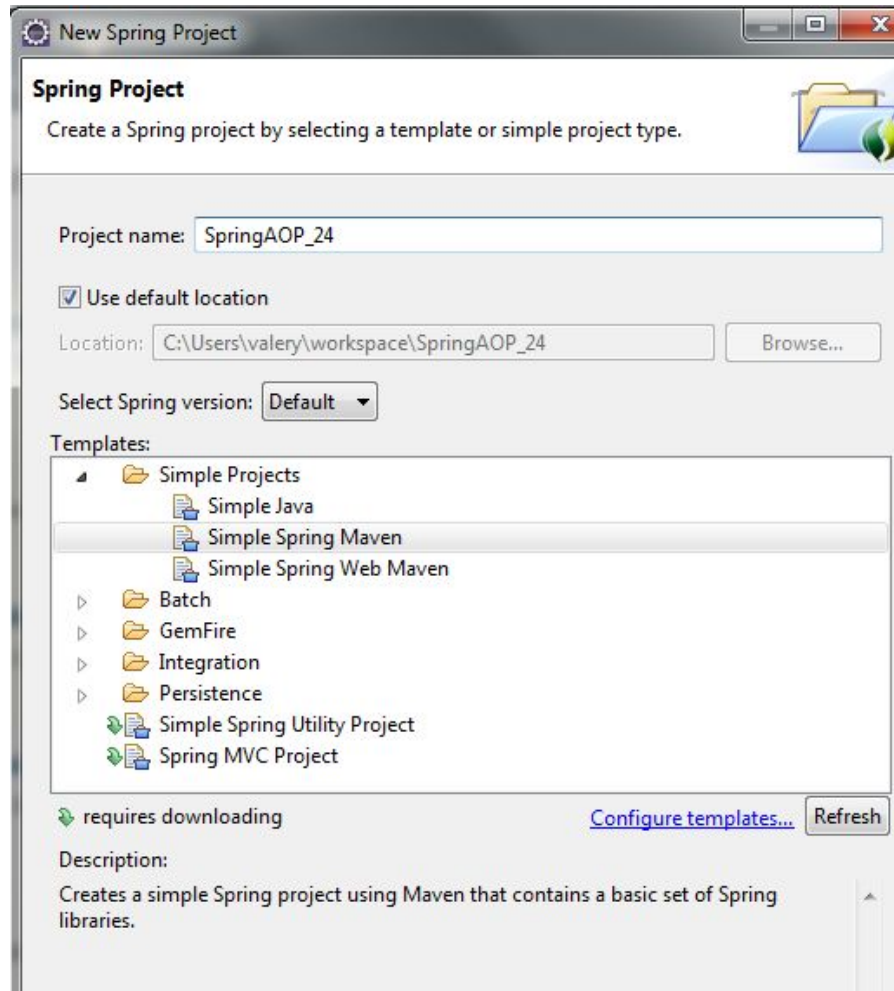


Аспектно - ориентированное программирование

Создание простого Spring примера

- создадим в **Eclipse** проект **Maven** и назовем его **SpringAOPExample**. В файле **pom.xml** будет содержать следующие зависимости:





Пример из жизни:

- Вы приходите в библиотеку и просите выдать вам книгу. В программе данное действие будет представлено функцией *getBook(String author, String book_name)*.
 - Перед тем как выдать вам книгу, необходимо проверить, есть ли такая в наличии: *checkBook(String author, String book_name)*.
 - Помимо этого было бы неплохо проверить, нет ли у вас задолженностей по книгам, ведь без возврата всех книг нельзя брать новые: *checkReader(String reader_name)*.
 - Если все необходимые условия выполнены, то вам выдается книга. Однако после этого надо бы пометить, что данная книга находится теперь у вас на руках: *booked(String author, String book_name, String reader_name)*.
-

ОСНОВНЫЕ ПОНЯТИЯ:

- *Аспект (aspect)* — модуль или класс, реализующий сквозную функциональность. Если в ООП базовым элементом является класс, то в АОП — это аспект;
 - *Точка соединения (join point)* — определяется как любая логическая точка в процессе выполнения программы, где встречаются основная программа и аспект. В **Spring AOP** точка соединения всегда соответствует вызову метода;
-

Напоминалка

Для языка **Java** парадигма АОП реализуется с помощью такого фреймворка, как **Spring AOP** , который заключает всю сквозную функциональность в аспекты. Проще говоря, он способен улавливать выполнение какого-либо метода и добавлять до или после него выполнение других методов. Делается это с помощью **Advice**(совет, рекомендация). В **Spring AOP** есть 4 вида рекомендаций:

- Рекомендация *before* — запускается до выполнения метода;
- Рекомендация *after* — запускается после выполнения метода;
- Рекомендация *throws* — выполняется после того, как метод выбросит исключение;
- Рекомендация *around* — окружает точку соединения. Объединяет в себе три вышеперечисленные рекомендации;

Добавляем в проект зависимости

```
<dependency>  
  <groupId>cglib</groupId>  
  <artifactId>cglib</artifactId>  
  <version>2.2.2</version>  
</dependency>  
</dependencies>  
</object>
```

III			
Dependencies	Dependency Hierarchy	Effective POM	pom.xml

Создаём класс сервисов пользователя

```
SpringAOP_1/pom.xml CustomerService.java ✕  
  
public class CustomerService {  
    private String name;  
    private String url;  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public void setUrl(String url) {  
        this.url = url;  
    }  
  
    public void printName() {  
        System.out.println("Customer name : " + this.name);  
    }  
  
    public void printURL() {  
        System.out.println("Customer website : " + this.url);  
    }  
  
    public void printThrowException() {  
        throw new IllegalArgumentException();  
    }  
  
}
```




помещаем конфигурационный файл **Spring** с именем **Spring-Customer.xml**



```
SpringAOP_1/pom.xml  CustomerService.java  *Spring-Customer.xml ✕
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
  <bean id="customerService" class="CustomerService">
    <property name="name" value="Iuri" />
    <property name="url" value="http://www.sut.ru" />
  </bean>
</beans>
```

New Java Class

Java Class

 The use of the default package is discouraged. 

Source folder:

Package:

Enclosing type:

Name:

Modifiers: public default private protected
 abstract final static

Superclass:

Interfaces:

Which method stubs would you like to create?

public static void main(String[] args)
 Constructors from superclass
 Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))
 Generate comments

```
public class App {  
    public static void main(String[] args) {  
        ApplicationContext appContext = new ClassPathXmlApplicationContext(  
            new String[] { "Spring-Customer.xml" });  
  
        CustomerService cust = (CustomerService) appContext  
            .getBean("customerServiceProxy");  
  
        System.out.println("*****");  
        cust.printName();  
        System.out.println("*****");  
        cust.printURL();  
        System.out.println("*****");  
        try {  
            cust.printThrowException();  
        } catch (Exception e) {  
  
        }  
    }  
}
```

Вывод программы

```
*****
```

```
Customer name : Iuri
```

```
*****
```

```
Customer website : http://www.sut.ru
```

```
*****
```

Spring AOP Advices (Рекомендации)

Рекомендация before

```
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class BeforeMethod implements MethodBeforeAdvice {
    @Override
    public void before(Method method, Object[] args, Object target)
        throws Throwable {
        System.out.println("BeforeMethod : До метода!");
    }
}
```

В конфигурационном файле **Spring** (**Spring-Customer.xml**) создаем бин для класса **BeforeMethod**, а также **НОВЫЙ ОБЪЕКТ** с именем **customerServiceProxy**:

```
<bean id="BeforeMethodBean" class="ru.aop.BeforeMethod" />

<bean id="customerServiceProxy"
      class="org.springframework.aop.framework.ProxyFactoryBean">

  <property name="target" ref="customerService" />

  <property name="interceptorNames">
    <list>
      <value>BeforeMethodBean</value>
    </list>
  </property>
</bean>
```

Свойство с именем *target* определяет бин класса, с которым мы будем работать. Свойство с именем *interceptorNames* определяет какие классы (рекомендации) будут работать с классом, находящемся в свойстве *target*. Теперь при запуске программы вы увидите следующее:

Вывод программы

```
*****  
BeforeMethod : До метода!  
Customer name : Iuri  
*****  
BeforeMethod : До метода!  
Customer website : http://www.sut.ru  
*****  
BeforeMethod : До метода!
```

до выполнения каждого метода класса **CustomerService** выполняется метод *before* рекомендации **BeforeMethod**

Рекомендация after

```
import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;

public class AfterMethod implements AfterReturningAdvice {
    @Override
    public void afterReturning(Object returnValue, Method method,
        Object[] args, Object target) throws Throwable {
        System.out.println("AfterMethod : После метода!");
    }
}
```


Конфигурационный файл Spring-Customer.xml:

```
<bean id="AfterMethodBean" class="ru.aop.AfterMethod" />

<bean id="customerServiceProxy"
      class="org.springframework.aop.framework.ProxyFactoryBean">

  <property name="target" ref="customerService" />

  <property name="interceptorNames">
    <list>
      <value>AfterMethodBean</value>
    </list>
  </property>
</bean>
```

```
*****  
Customer name : Iuri  
AfterMethod : После метода!  
*****  
Customer website : http://www.sut.ru  
AfterMethod : После метода!  
*****
```

Рекомендация `throws`

Выполняется после того, как метод *выбросит исключение*.

```
import org.springframework.aop.ThrowsAdvice;

public class ThrowException implements ThrowsAdvice {
    public void afterThrowing(IllegalArgumentException e) throws Throwable {
        System.out.println("ThrowException : После выброса исключения!");
    }
}
```

```
<bean id="ThrowExceptionBean" class="ru.aop.ThrowException" />

<bean id="customerServiceProxy"
      class="org.springframework.aop.framework.ProxyFactoryBean">

    <property name="target" ref="customerService" />

    <property name="interceptorNames">
        <list>
            <value>ThrowExceptionBean</value>
        </list>
    </property>
</bean>
```

Customer name : Iuri

Customer website : <http://www.sut.ru>

ThrowException : После выброса исключения!

Рекомендация around

Сочетает в себе три
вышеприведенных рекомендации и
выполняется во время выполнения
метода.

```
import java.util.Arrays;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class AroundMethod implements MethodInterceptor {
    @Override
    public Object invoke(MethodInvocation methodInvocation) throws Throwable {

        System.out.println("Method name : "
            + methodInvocation.getMethod().getName());
        System.out.println("Method arguments : "
            + Arrays.toString(methodInvocation.getArguments()));

        // до метода
        System.out.println("AroundMethod : Вместо BeforeMethod!");

        try {
            // выполняем оригинальный метод
            Object result = methodInvocation.proceed();

            // после метода
            System.out.println("AroundMethod : Вместо AfterMethod!");

            return result;
        } catch (IllegalArgumentException e) {
            // если был выброс исключения
            System.out
                .println("AroundMethod : Вместо ThrowMethod!");
            throw e;
        }
    }
}
```

Конфигурационный файл Spring-Customer.xml

```
<bean id="AroundMethodBean" class="ru.aop.AroundMethod" />

<bean id="customerServiceProxy" class="org.springframework.aop.framework

    <property name="target" ref="customerService" />

    <property name="interceptorNames">
        <list>
            <value>AroundMethodBean</value>
        </list>
    </property>
</bean>
```



```
*****
Method name : printName
Method arguments : []
AroundMethod : Bmectro BeforeMethod!
Customer name : Iuri
AroundMethod : Bmectro AfterMethod!
*****
Method name : printURL
Method arguments : []
AroundMethod : Bmectro BeforeMethod!
Customer website : http://www.sut.ru
AroundMethod : Bmectro AfterMethod!
*****
Method name : printThrowException
Method arguments : []
AroundMethod : Bmectro BeforeMethod!
AroundMethod : Bmectro ThrowMethod!
```