LESSON 14
FRAMEWORK IMPLEMENTATION

REFLECTION

Java reflection

- Reflection allows us to inspect and/or modify attributes of classes, interfaces, fields and methods at runtime.
- Additionally, we can instantiate new objects, invoke methods and get or set field values using reflection

Define your own annotations

Retention: defines the visibility of the annotation

- SOURCE—Annotation is visible only at the source level and will be ignored by the compiler.
- CLASS—Annotation is visible by the compiler at compile time, but will be ignored by the VM.
- RUNTIME—Annotation is visible by the VM so they can be read only at run-time.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

Target: where can I apply this annotation?
@Target(value={TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})

Annotations can have parameters

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Service {
   public String name() default "";
}
```

parameter

```
@Service(name="one")
public class MyServiceOne {
```

Framework classes

```
@Service(name="one")
public class MyServiceOne {
   private String nameOne = "nameOne";
   @Inject
   private String nameTwo = "nameTwo";

   public void print(){
      System.out.println(nameOne);
   }

   @Print
   public void print2(){
      System.out.println(nameTwo);
   }
}
```

```
@Service(name="two")
public class MyServiceTwo {
   private String name = "myServiceTwo";

   private void print(){
      System.out.println(name);
   }
}
```

```
@Retention(RUNTIME)
@Target(METHOD)
public @interface Print {

}
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Service {
   public String name() default "";
}
```

```
@Retention(RUNTIME)
@Target(FIELD)
public @interface Inject {

}
```

Get classes with annotations

```
public class GetClassesWithAnnotation {

  public static void main(String[] args {
    Reflections reflections = new Reflections("");
    Set<Class<?>> serviceClasses = reflections.getTypesAnnotatedWith(Service.class);
    for (Class<?> serviceClass : serviceClasses) {
      System.out.println(serviceClass.getName());
    }
  }
}
```

```
@Service(name="one")
public class MyServiceOne {
   private String nameOne = "nameOne";
   @Inject
   private String nameTwo = "nameTwo";

   public void print(){
      System.out.println(nameOne);
   }

   @Print
   public void print2(){
      System.out.println(nameTwo);
   }
}
```

```
@Service(name="two")
public class MyServiceTwo {
   private String name = "myServiceTwo";

   private void print(){
      System.out.println(name);
   }
}
```

Get fields of a class

```
public class GetFieldsOfAClass {

  public static void main(String[] args {
    Class<?> serviceOne = MyServiceOne.class;
    Field[] fields = serviceOne.getDeclaredFields();
    for (Field field : fields) {
      System.out.println(field.getName());
    }
  }
}
```

```
@Service(name="one")
public class MyServiceOne {
  private String nameOne = "nameOne";
  @Inject
  private String nameTwo = "nameTwo";

  public void print(){
    System.out.println(nameOne);
  }

  @Print
  public void print2(){
    System.out.println(nameTwo);
  }
}
```

```
@Service(name="two")
public class MyServiceTwo {
  private String name = "myServiceTwo";

  private void print(){
    System.out.println(name);
  }
}
```

Get methods of a class

```
public class GetMethodsOfAClass {

  public static void main(String[] args)  {
    Class<?> serviceOne = MyServiceOne.class;
    Method[] methods = serviceOne.getDeclaredMethods();
    for (Method method : methods) {
      System.out.println(method.getName());
    }
  }
}
```

```
@Service(name="one")
public class MyServiceOne {
   private String nameOne = "nameOne";
   @Inject
   private String nameTwo = "nameTwo";

   public void print(){
      System.out.println(nameOne);
   }

   @Print
   public void print2(){
      System.out.println(nameTwo);
   }
}
```

```
@Service(name="two")
public class MyServiceTwo {
   private String name = "myServiceTwo";

   private void print(){
      System.out.println(name);
   }
}
```

Get annotated fields of a class

```
public class GetAnnotatedFieldsOfAClass {

  public static void main(String[] args)  {
    Class<?> serviceOne = MyServiceOne.class;
    Field[] fields = serviceOne.getDeclaredFields();
    for (Field field : fields) {
      if (field.isAnnotationPresent(Inject.class)) {
        System.out.println(field.getName());
      }
    }
  }
}
```

```
@Service(name="one")
public class MyServiceOne {
   private String nameOne = "nameOne";
   @Inject
   private String nameTwo = "nameTwo";

   public void print(){
      System.out.println(nameOne);
   }

   @Print
   public void print2(){
      System.out.println(nameTwo);
   }
}
```

```
@Service(name="two")
public class MyServiceTwo {
   private String name = "myServiceTwo";

   private void print(){
      System.out.println(name);
   }
}
```

Get annotated methods of a class

```
public class GetAnnotatedMethodsOfAClass {

  public static void main(String[] args)  {
    Class<?> serviceOne = MyServiceOne.class;
    Method[] methods = serviceOne.getDeclaredMethods();
    for (Method method : methods) {
      if (method.isAnnotationPresent(Print.class)) {
        System.out.println(method.getName());
      }
    }
  }
}
```

```
@Service(name="one")
public class MyServiceOne {
   private String nameOne = "nameOne";
   @Inject
   private String nameTwo = "nameTwo";

   public void print(){
      System.out.println(nameOne);
   }

   @Print
   public void print2(){
      System.out.println(nameTwo);
   }
}
```

```
@Service(name="two")
public class MyServiceTwo {
   private String name = "myServiceTwo";

   private void print(){
      System.out.println(name);
   }
}
```

Get annotations

```
public class GetAnnotations {

  public static void main(String[] args) {
    Class<?> serviceOneClass = MyServiceOne.class;
    Annotation[] annotations=serviceOneClass.getAnnotations();
    for (Annotation annotation : annotations){
      System.out.println("Class with name "+serviceOneClass.getName()+" has annotation: "+annotation);
    }

    Field[] fields = serviceOneClass.getDeclaredFields();
    for (Field field : fields) {
      Annotation[] fieldAnnotations=field.getAnnotations();
      for (Annotation annotation : fieldAnnotations){
        System.out.println("Field with name "+field.getName()+" has annotation: "+annotation);
      }
    }

    Method[] methods = serviceOneClass.getDeclaredMethods();
    for (Method method : methods) {
      Annotation[] methodAnnotations=method.getAnnotations();
      for (Annotation annotation : methodAnnotations){
        System.out.println("Method with name "+method.getName()+" has annotation: "+annotation);
      }
    }
  }
}
```

```
Class with name application.MyServiceOne has annotation: @application.Service(name="one")
Field with name nameTwo has annotation: @application.Inject()
Method with name print2 has annotation: @application.Print()
```

Instantiate a class

```
public class InstantiateAClass {

  public static void main(String[] args) throws InstantiationException, IllegalAccessException,
                                NoSuchMethodException, InvocationTargetException {
    Class<?> serviceOneClass = MyServiceOne.class;
    MyServiceOne serviceOneObject = (MyServiceOne) serviceOneClass.getDeclaredConstructor().newInstance();
    serviceOneObject.print();
  }
}
```

```
@Service(name="one")
public class MyServiceOne {
   private String nameOne = "nameOne";
   @Inject
   private String nameTwo = "nameTwo";

   public void print(){
      System.out.println(nameOne);
   }

   @Print
   public void print2(){
      System.out.println(nameTwo);
   }
}
```

Get annotation parameter

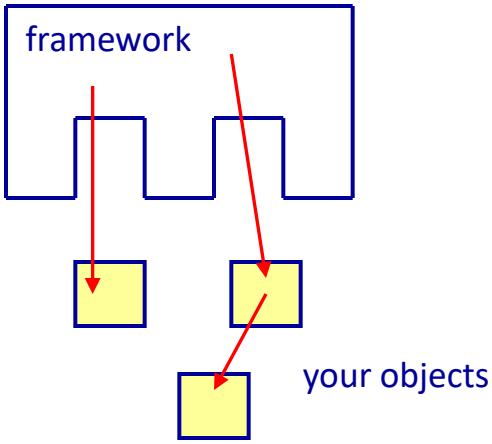
```
public class GetAnnotationParameter {

  public static void main(String[] args) {
    Class<?> serviceOneClass = MyServiceOne.class;
    Annotation[] annotations=serviceOneClass.getAnnotations();
    for (Annotation annotation : annotations){
      System.out.println("Class with name "+serviceOneClass.getName()+" has annotation: "+annotation);
      System.out.println("This annotation: "+annotation+" has a 'name' parameter with value:
"+((Service)annotation).name());
    }
  }
}
```

```
@Service(name="one")
public class MyServiceOne {
   private String nameOne = "nameOne";
   @Inject
   private String nameTwo = "nameTwo";

   public void print(){
      System.out.println(nameOne);
   }

   @Print
   public void print2(){
      System.out.println(nameTwo);
   }
}
```

```
Class with name application.MyServiceOne has annotation: @application.Service(name="one")
This annotation: @application.Service(name="one") has a 'name' parameter with value: one
```

INVERSION OF CONTROL

Inversion of Control (IoC)

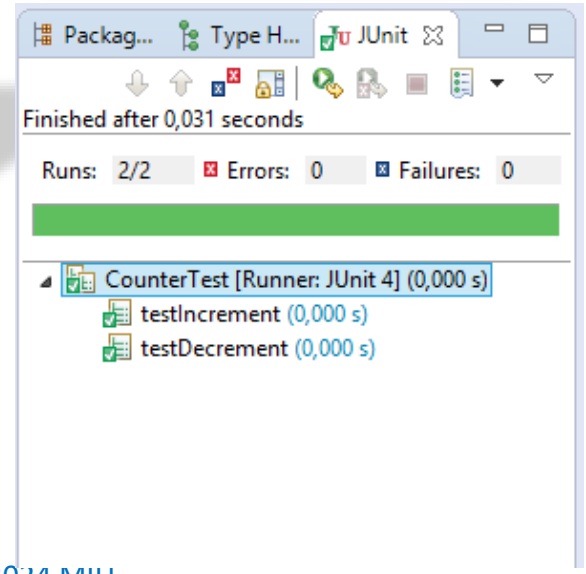


IoC: The framework calls your code

Example of JUnit framework

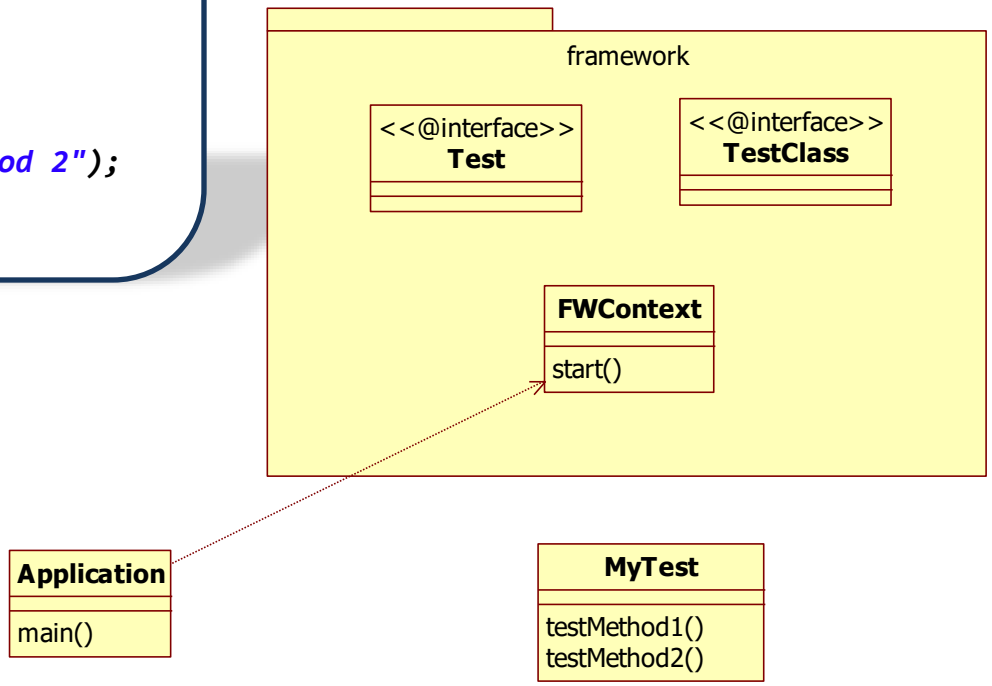
```java
public class CounterTest {
    @Test
    public void testIncrement(){
        Counter counter = new Counter();
        assertEquals(1,counter.increment());
        assertEquals(2,counter.increment());
    }

    @Test
    public void testDecrement(){
        Counter counter = new Counter();
        assertEquals(-1,counter.decrement());
        assertEquals(-2,counter.decrement());
    }
}
```

```java
public class Counter {
    private int counterValue=0;

    public int increment(){
        return ++counterValue;
    }
    public int decrement(){
        return --counterValue;
    }
}
```



Inversion of Control framework

```java
@TestClass
public class MyTest {

  @Test
  public void testMethod1() {
    System.out.println("perform test method 1");
  }

  @Test
  public void testMethod2() {
    System.out.println("perform test method 2");
  }
}
```

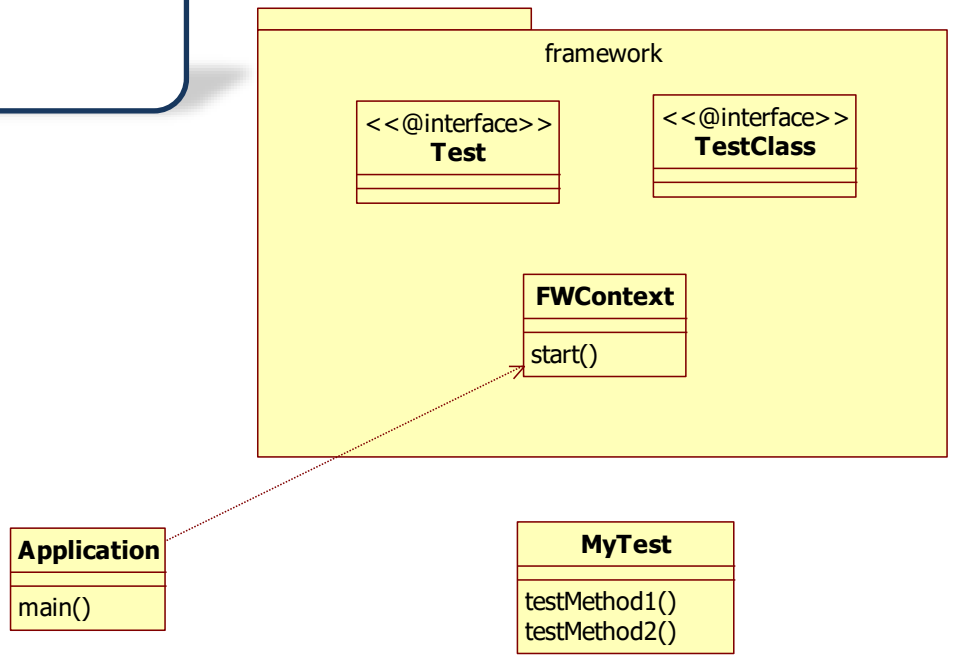


Inversion of Control framework

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

Create your own annotations in Java

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface TestClass {
}
```



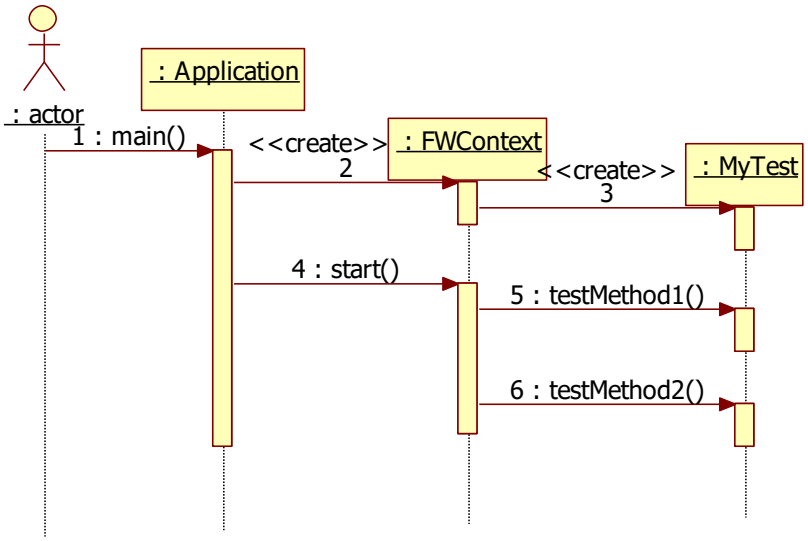
Classpath scanning

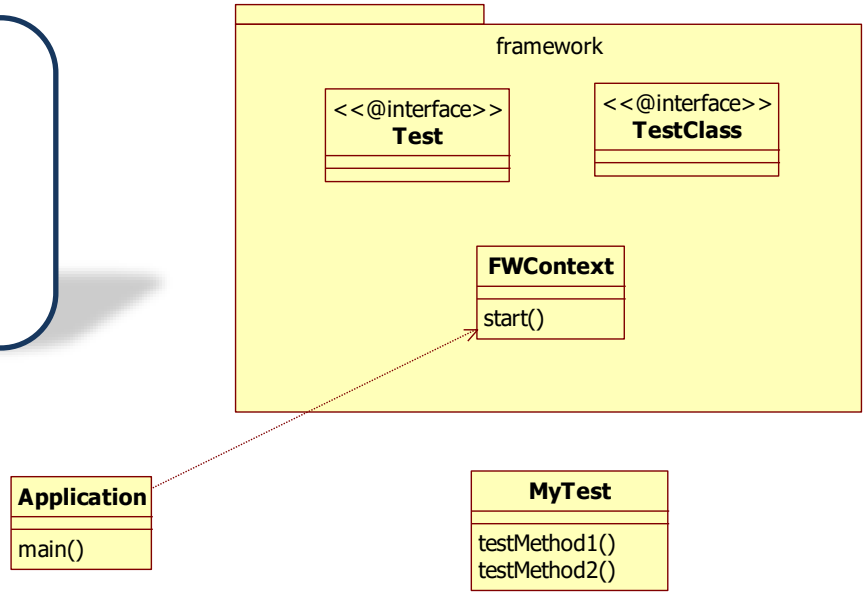
```
public class FWContext {
  private static List<Object> objectMap = new ArrayList<>();

  public FWContext() {
    try {
      // find and instantiate all classes annotated with the @TestClass annotation
      Reflections reflections = new Reflections("");
      Set<Class<?>> types = reflections.getTypesAnnotatedWith(TestClass.class);
      for (Class<?> implementationClass : types) {
        objectMap.add((Object) implementationClass.newInstance());
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void start() {
    try {
      for (Object theTestClass : objectMap) {
        // find all methods annotated with the @Test annotation
        for (Method method : theTestClass.getClass().getDeclaredMethods()) {
          if (method.isAnnotationPresent(Test.class)) {
            method.invoke(theTestClass);
          }
        }
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Classpath scanning

Inversion of Control framework

```
public class Application {

  public static void main(String[] args) {
    FWContext fWContext = new FWContext();
    fWContext.start();
  }
}
```






```
perform test method 2
perform test method 1
```

Let's add the @Before annotation

```
@TestClass
public class MyTest {

  @Before
  public void init() {
    System.out.println("perform initialization");
  }

  @Test
  public void testMethod1() {
    System.out.println("perform test method 1");
  }

  @Test
  public void testMethod2() {
    System.out.println("perform test method 2");
  }

}
```

@Before

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Before {

}
```

Let's add the @Before annotation

```
public class FWContext {
  private static List<Object> objectMap = new ArrayList<>();

  public FWContext() {
      // find and instantiate all classes annotated with the @TestClass annotation
  }

  public void start() {
    try {
      for (Object theTestClass : objectMap) {
        Method beforeMethod = null;
        //find the @before method
        for (Method method : theTestClass.getClass().getDeclaredMethods()) {
          if (method.isAnnotationPresent(Before.class)) {
            beforeMethod = method;
          }
        }
        // find all methods annotated with the @Test annotation
        for (Method method : theTestClass.getClass().getDeclaredMethods()) {
          if (method.isAnnotationPresent(Test.class)) {
            //first call the before method
            if (beforeMethod!= null) beforeMethod.invoke(theTestClass);
            method.invoke(theTestClass);
          }
        }
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

First call @Before before every test method

Let's add an assert method

```
public class CalculatorImpl implements Calculator {
  private int calcValue=0;

  public void reset() {
    calcValue=0;
  }

  public int add(int newValue) {
    calcValue=calcValue+newValue;
    return calcValue;
  }

  public int subtract(int newValue) {
    calcValue=calcValue-newValue;
    return calcValue;
  }
}
```

Static import

```
import framework.Before;
import framework.Test;
import framework.TestClass;
import static framework.Asserts.*;

@TestClass
public class MyTest {
  Calculator calculator;

  @Before
  public void init() {
    calculator = new CalculatorImpl();
  }

  @Test
  public void testMethod1() {
    assertEquals(calculator.add(3),3);
    assertEquals(calculator.add(6),9);
  }

  @Test
  public void testMethod2() {
    assertEquals(calculator.add(3),3);
    assertEquals(calculator.subtract(6),-1);
  }
}
```

assertEquals

Let's add an assert method

```
public class Asserts {

  public static void assertEquals(int x, int y) {
    if (x != y)
      System.out.println("Fail: result = "+x+" but expected "+y);
  }

}
```

Static method

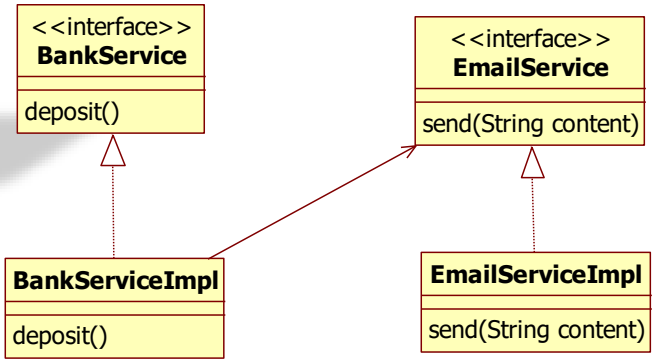
DEPENDENCY INJECTION

Instantiate an object directly

```
public interface BankService {
  public void deposit() ;
}
```

```
public class BankServiceImpl implements BankService{
  private EmailService emailService= new EmailServiceImpl();

  public void deposit() {
    emailService.send("deposit");
  }
}
```

Instantiate the EmailService




```
public interface EmailService {
  void send(String content);
}
```

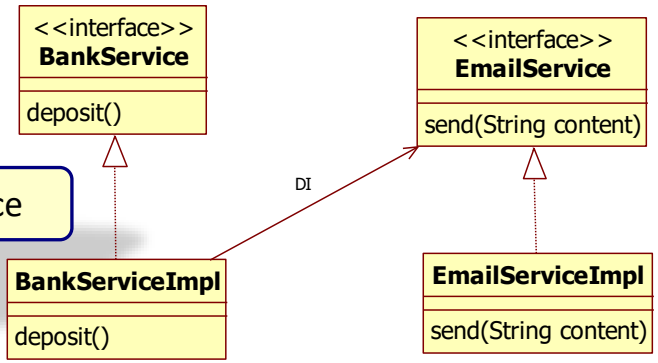
```
public class EmailServiceImpl implements EmailService{

  public void send(String content) {
    System.out.println("sending email: "+content);
  }
}
```

Dependency Injection

```java
public interface BankService {
  public void deposit() ;
}
```

```java
public class BankServiceImpl implements BankService{
  private EmailService emailService;

  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }

  public void deposit() {
    emailService.send("deposit");
  }
}
```

We can inject any EmailService

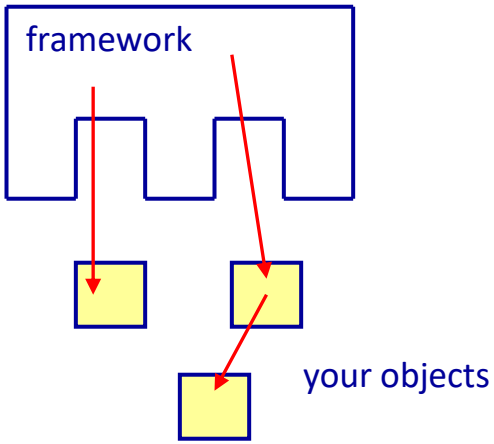



```java
public interface EmailService {
  void send(String content);
}
```

```java
public class EmailServiceImpl implements EmailService{

  public void send(String content) {
    System.out.println("sending email: "+content);
  }
}
```

Framework implementation

- IoC: The framework instantiates our application classes
- Dependency injection: The framework wires our objects together



Dependency injection

@Service

```
@Service
public class CalculatorImpl implements Calculator {
  private int calcValue=0;

  public void reset() {
    calcValue=0;
  }

  public int add(int newValue) {
    calcValue=calcValue+newValue;
    return calcValue;
  }

  public int subtract(int newValue) {
    calcValue=calcValue-newValue;
    return calcValue;
  }
}
```

@Inject

```
@TestClass
public class MyTest {
  @Inject
  Calculator calculator;

  @Before
  public void init() {
    calculator.reset();
  }

  @Test
  public void testMethod1() {
    assertEquals(calculator.add(3),3);
    assertEquals(calculator.add(4),7);
  }

  @Test
  public void testMethod2() {
    assertEquals(calculator.add(3),3);
    assertEquals(calculator.subtract(6),-1);
  }

}
```

Dependency injection

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Service {

}
```

```
@Retention(RUNTIME)
@Target(FIELD)
public @interface Inject {

}
```

FWContext (1/4)

```
public class FWContext {

  private static List<Object> testObjectMap = new ArrayList<>();
  private static List<Object> serviceObjectMap = new ArrayList<>();

  public FWContext() {
    try {
      Reflections reflections = new Reflections("");
      // find and instantiate all classes annotated with the @TestClass annotation
      Set<Class<?>> testtypes = reflections.getTypesAnnotatedWith(TestClass.class);
      for (Class<?> testClass : testtypes) {
        testObjectMap.add((Object) testClass.newInstance());
      }
      // find and instantiate all classes annotated with the @Service annotation
      Set<Class<?>> servicetypes = reflections.getTypesAnnotatedWith(Service.class);
      for (Class<?> serviceClass : servicetypes) {
        serviceObjectMap.add((Object) serviceClass.newInstance());
      }

      performDI();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

All test classes

All service classes

performDI()

FWContext (2/4)

```
private void performDI() {
  try {
    for (Object theTestClass : testObjectMap) {
        // find annotated fields
      for (Field field : theTestClass.getClass().getDeclaredFields()) {
          if (field.isAnnotationPresent(Inject.class)) {
            // get the type of the field
            Class<?> theFieldType =field.getType();
            //get the object instance of this type
            Object instance = getServiceBeanOftype(theFieldType);
            //do the injection
            field.setAccessible(true);
            field.set(theTestClass, instance);
          }
        }
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Loop over all test classes

Check all fields on @Inject

Get the correct @Service bean

Inject the @Service bean

FWContext (3/4)

```
public Object getServiceBeanOftype(Class interfaceClass) {
  Object service = null;
  try {
    for (Object theClass : serviceObjectMap) {
      Class<?>[] interfaces = theClass.getClass().getInterfaces();

      for (Class<?> theInterface : interfaces) {
        if (theInterface.getName().contentEquals(interfaceClass.getName()))
          service = theClass;
      }
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
  return service;
}
```

Loop over all @Service beans

Find the @Service beans with the given type

FWContext (4/4)

```
public void start() {
  try {
    for (Object theTestClass : testObjectMap) {
      Method beforeMethod = null;
      //find the @before method
      for (Method method : theTestClass.getClass().getDeclaredMethods()) {
        if (method.isAnnotationPresent(Before.class)) {
          beforeMethod = method;
        }
      }
      // find all methods annotated with the @Test annotation
      for (Method method : theTestClass.getClass().getDeclaredMethods()) {
        if (method.isAnnotationPresent(Test.class)) {
          //first call the before method
          if (beforeMethod!= null) beforeMethod.invoke(theTestClass);
          method.invoke(theTestClass);
        }
      }
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
 }
}
```

Main point

- Dependency injection gives us flexibility in wiring objects together.
- Daily contact with pure consciousness results in more and more happiness by spontaneous right action.