```java
@Test
public void test1() {
  List<Integer> source = List.of(-10, 5, -3, -11, 33);
  List<Integer> negativeActual = service.filterNegative(source);
  List<Integer> negativeExpected = List.of(-10, -3, -11);
  assertEquals(negativeActual, negativeExpected);
}
```

are extremely coupled

making changes to source requires changing expected

tend to be errored

```java
// this predicate can be moved to the library
Predicate<List<? extends Integer>> lessThan0 = new Predicate<>() {
  @Override
  public boolean test(List<? extends Integer> ints) {
    return ints.stream().allMatch(x -> x < 0);
  }
};

// more general approach - can be used for any type
assertThat(negativeActual)
    .matches(lessThan0);
```

- custom matchers
- can be written EASILY for any type
- can be shared/reused with main codebase

business logic is here

they don't have any deps
they are foundation

① What to test ⟷ Internal Libraries, Service] they contain REAL logic

② How to test

What's responsibility Controller

1. parse request
2. validate request
3. pass data to the service
4. I don't care about service. It's already tested.

```java
@Test
public void isEligibleToLoanTest() {
  Person p1 = new Person( age: 20,  name: "Jim", List.of());
  assertFalse(service.isEligibleToLoan(p1));

  Person p2 = new Person( age: 21,  name: "Jim", List.of());
  assertTrue(service.isEligibleToLoan(p2));
}
```

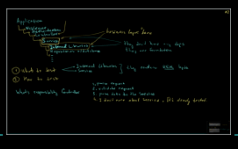
we don't know where the problem is

we separate to different cases

- to deal with the problem
  - faster
  - w/o any ambiguity

one entity per test case

```java
@Test
public void isEligibleToLoanLess21() {
  Person p1 = new Person( age: 20,  name: "Jim", List.of());
  assertFalse(service.isEligibleToLoan(p1));
}

@Test
public void isEligibleToLoanGE21() {
  Person p2 = new Person( age: 21,  name: "Jim", List.of());
  assertTrue(service.isEligibleToLoan(p2));
}
```



how to do the things

custom matches share

what you can do

not how

interface

implementation (test)

actual type

more wide

Loan Service

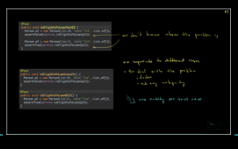
```java
@AllArgsConstructor
public class LoanServiceImpl implements LoanService {

  private final PersonRepository repository;

  @Override
  public boolean isEligibleToLoan(Person p) {
    return p.getAge() >= 21;
  }

  @Override
  public Person getById(int id) {
    return repository.getById(id);
  }

}
```

```java
interface LoanService {

  boolean isEligibleToLoan(Person p);
  Person getById(int id);

}
```

```java
public class LoanServiceTest implements LoanService {

  @Override
  public boolean isEligibleToLoan(Person p) {
    return p.getAge() >= 21;
  }

  @Override
  public Person getById(int id) {
    throw NOT_IMPLEMENTED;
  }

}
```

```java
public class LoanServiceSpec {

  private final LoanService service = new LoanServiceTest();

  @Ugly
  @Test
  public void isEligibleToLoanTest() {
    Person p1 = new Person( age: 20,  name: "Jim", List.of());
    assertFalse(service.isEligibleToLoan(p1));

    Person p2 = new Person( age: 21,  name: "Jim", List.of());
    assertTrue(service.isEligibleToLoan(p2));
  }

  @Good
  @Test
  public void isEligibleToLoanLess21() {
    Person p1 = new Person( age: 20,  name: "Jim", List.of());
    assertFalse(service.isEligibleToLoan(p1));
  }

  @Good
  @Test
  public void isEligibleToLoanGE21() {
    Person p2 = new Person( age: 21,  name: "Jim", List.of());
    assertTrue(service.isEligibleToLoan(p2));
  }

}
```

the whole code isn't aware, what instance is used



mock creation

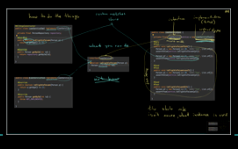
```
private final LoanService service = Mockito.mock(LoanService.class);
{
  Mockito
      .when(service.isEligibleToLoan(Mockito.any()))
      .thenReturn(true);
}
```

method

parameter

behavior configuration

method return result.



```java
interface LoanService {

  boolean isEligibleToLoan(Person p);
  Person getById(int id);

}
```

] 2 methods

```java
public class LoanServiceImplSpec2 {

  // Mock / Stub creation
  private final LoanService service = Mockito.mock(LoanService.class);
  {
    // Mock configuration
    Mockito
        .when(service.getById(Mockito.anyInt()))
        .thenReturn(new Person( age: 33,  name: "Jackson", List.of("Scala")));
  }

  @Test
  public void getById() {
    // mock usage
    Person p0 = service.getById(11111);
    assertThat(p0).isNotNull();
    assertThat(p0)
        .matches(p -> p.getAge() == 33)
        .matches(p -> p.getSkills().contains("Scala"))
        .matches(p -> p.getName().equals("Jackson"));
  }

}
```

anyId Given Mocked With The Same Response

] - we configure only that method which we want to test

- everything else provided by Mockito

we use only one

assertEquals => you almost every time need to use Mockito
you need to rely on consistent response

assertThat(...).matches(proper lambda)

⇒ we almost don't need to use mocbito

I do prefer have more than 1 implement.
than doing mocking

Controller

Service ⟶ is already tested

extract params and call Service.

```
public class Controller {

  private final Service svc;

  public int extractFirst(String request) {...}

  public int extractSecond(String request) {...}

  private String format(int res) {
    return String.format("The result is: %d", res);
  }

  /*
   /calc?x=5&y=7
   */
  public String doAdd(String request) {
    int a = extractFirst(request);
    int b = extractSecond(request);
    int r = svc.add(a, b);
    String result = format(r);
    return result;
  }

}
```

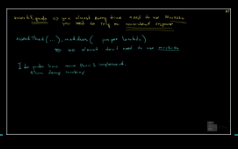
```
private final Service service = Mockito.mock(Service.class);
```

we are in charge of impl.

```
public class Service {

  public int add(int a, int b) {
    return a + b;
  }
}
```

mockito can intercept here and check a=5 b=7

we need to intercept this call



```java
@Test
public void testBridgeControllerService() {
  controller.doAdd(testQuery);
  Mockito
    .verify(service, Mockito.times( wantedNumberOfInvocations: 1))
    .add( a: 5,  b: 7);
}
```

Just a service call (! service made by Mockito)

we are verifying INVOCATION ONLY and PARAMETERS

how many times

with which parameters

assertion is made inside Mockito.verify

```java
@Test
public void testServiceInvocationPreciselyMoreThanOnce() {
  // sending real request. BEWARE: in the middle we have mocked service
  controller.doAdd(testQuery);
  controller.doAdd( request: "calc?x=15&y=17");
  controller.doAdd( request: "calc?x=25&y=27");
  // at that moment Mockito has DONE its job. we need to gather it

  // declaring captors
  ArgumentCaptor<Integer> firstCaptor = ArgumentCaptor.forClass(int.class);
  ArgumentCaptor<Integer> secondCaptor = ArgumentCaptor.forClass(int.class);

  // general check & linking to captors
  Mockito
      .verify(service, Mockito.times( wantedNumberOfInvocations: 3))
      .add(firstCaptor.capture(), secondCaptor.capture());

  List<Integer> first = firstCaptor.getAllValues();
  List<Integer> second = secondCaptor.getAllValues();

  assertThat(first).matches(x -> x.equals(List.of(5, 15, 25)));
  assertThat(second).isEqualTo(List.of(7, 17, 27));
}
```

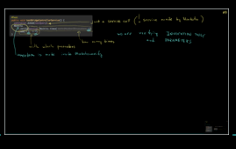
we send request to our Controller but service is made by Mockito

```java
public class Service {

  public int add(int a, int b) {
```

we verify number of calls + providing "BOXES" where our params will reside.

accessing theese "boxes"

make assertions



```java
@Test
public void testServiceInvocationPreciselyMoreThanOnceCombined() {
    // sending real request. BEWARE: in the middle we have mocked service
    controller.doAdd(testQuery);
    controller.doAdd( request: "calc?x=15&y=17");
    controller.doAdd( request: "calc?x=25&y=27");
    // at that moment Mockito has DONE its job. we need to gather it

    // declaring captors
    ArgumentCaptor<Integer> firstCaptor = ArgumentCaptor.forClass(int.class);
    ArgumentCaptor<Integer> secondCaptor = ArgumentCaptor.forClass(int.class);

    // general check & linking to captors
    Mockito
        .verify(service, Mockito.times( wantedNumberOfInvocations: 3))
        .add(firstCaptor.capture(), secondCaptor.capture());

    // access & check combined
    assertThat(firstCaptor)
        .matches(boxContainsValue(5))
        .matches(boxContainsValue(15))
        .matches(boxContainsValue(25));
//          .matches(x -> x.getAllValues().equals(List.of(5, 15, 25)));
    assertThat(secondCaptor)
        .matches(boxContainsAllValues(7, 17, 27));
}
```