

CSC 347 - Concepts of Programming Languages

Unit Tests and Contracts

Instructor: Stefan Mitsch



Compute the Factorial of a Number



```
1 Implement an iterative Scala 3 program  
2 that computes the factorial of a number
```



```
1 def factorial(n: Int) : Int =  
2   var result = 1  
3   for i <- 1 to n do  
4     result *= i  
5   result
```

? How do we use `factorial` ?

? Is the code correct?

? How can we convince ourselves whether code is correct?



Learning Objectives

- ❓ How can we build trust in (auto-generated) code?
 - Read and express unit tests
 - Read and express contracts



Unit Tests

💡 Unit tests provide sample inputs and test expected outputs of code in isolation

- Unit tests specify **expected behavior**
- Unit tests are **documentation**
 - How to initialize and use some code
 - What to expect as a result on legal inputs
 - What to expect as an exception on illegal inputs
- Unit tests **prevent quality regression**



Unit Tests with ScalaTest

```
1 def factorial(n: Int) : Int =  
2   var result = 1  
3   for i <- 1 to n do  
4     result *= i  
5   result
```

```
1 class FactorialTests  
2   extends AnyFlatSpec with should.Matchers:  
3  
4   "Factorial" should "compute 0! correctly" in {  
5     factorial(0) should be (1)  
6   }  
7  
8   it should "compute 1! correctly" in {  
9     factorial(1) should be (1)  
10  }  
11  
12  it should "compute 3! correctly" in {  
13    factorial(3) should be (6)  
14  }
```

- Matchers `should be` etc. express expected results
- ⚠ Can only test a finite number of examples



Contracts

💡 Contracts are formal and verifiable interface specifications

- **Preconditions** are expectations that must be met by a client
- **Postconditions** are guarantees made by the component
- **Invariants** are maintained (e.g., every loop iteration)



Formal Specification and Verification

Specification

- Hoare triples: $\{P\} \ C \ \{Q\}$
- Modal logic: $P \rightarrow [C]Q$
- Code:

```
1 def C() = {  
2   require (P)  
3   // ...  
4 } ensuring (Q)
```

Verification

- **Theorem proving** shows correctness mathematically for all possible inputs and all possible executions
- **Model checking** shows correctness for finite-length executions
- **Runtime verification** checks contracts while executing the software and throws exceptions when violated



Contracts in Scala

```
1 def factorial(n: Int) : Int = {  
2   require (n>=0)  
3   var result = 1  
4   for i <- 1 to n do  
5     result *= i  
6   result  
7 } ensuring {  
8   (result: Int) =>  
9     result == factorialContract(n)  
10 }
```

⚠ Useful?

```
1 def factorialContract(n: Int) : Int =  
2   var result = 1  
3   for i <- 1 to n do result *= i  
4   result
```

💡 Alternative solution!

```
1 def factorialContract(n: Int) : Int =  
2   if n<=1 then 1  
3   else n * factorialContract(n-1)
```

- Contract specifies input assumptions (`require`) and output guarantees (`ensuring`)
- Contract can be an alternative implementation
- ⚠ Runtime checking adds computational overhead



Edge Cases

- Open project `hw1` in VSCode, create a file `week2-contracts.sc` in folder `play`

- `max` by normalized comparison `x - y > 0`

```
1 def max(x: Int, y: Int): Int =  
2   // x > y normalized to x - y > 0  
3   if x - y > 0 then x else y  
4 end max
```

- Write some unit tests, can you find a bug?

- Generate a `Rational` case class (use chat or inline suggestions)
 - for rational numbers `n/d`
 - numerator `n`, denominator `d`
- Generate and test `equals`, e.g.,
 - `6/2 == 9/3`
 - `1/2 == 2/4`
 - Can you find a bug?

- Software verification with [Scala Stainless](#)



Summary

Unit Tests

- Execute code with some inputs and test for expected outputs
 - **+** documents known and expected use and behavior
 - **-** finite number of tests
 - Best for: regression testing

Contracts

- Express input assumptions and output guarantees
 - **+** show correctness for all possible inputs and executions
 - **-** computational overhead, extra effort
 - Best for: correctness-critical applications