# Most important things to know about classic TDD

### TDD in general

- Motivation
  - Never been afraid to touch some mazy legacy code fearing you'd break it?
  - Unit tests are like a manual.
  - "Any fool can write code that a computer can understand. Good programmers write code that **humans** can understand" (Martin Fowler)
- Precondition
  - Domain knowledge must be available (p.e. by a domain expert or a domain specific language)
- Some Rules
  - No upfront design assumptions. Design emerges completely from the code, hence it solves over-engineering problems.
  - Tests must be *mutually independent*
  - Never refactor with failing tests (or no tests)
  - Only use what you can control
  - Rule of 3 for duplication cleanup (duplication is cheaper than abstraction in terms of coupling)

### TDD approach

- Baby Steps
  - Only write as much of code for a unit test to fail.
  - Only write as much of production code for making a failing unit test pass
- Internal test structure
  - Arrange
  - Act
  - Assert
- The three ways forward in Test Code
  - Red
  - All Green
  - Refactor
- The three ways forward in Production Code
  - Fake implementation
  - Obvious implementation
  - Generalization trough triangulation
- Only one execution path per test
- Naming Test-Classes
  - Test a single Class: [ClassName]Sould vs. [ClassName]Tests
  - Test a feature: [FeatureName]Tests
- Naming Test-Methods
  - [behave]With[Inputs] vs. test[MethodName][expected behaviour]With[Inputs]
  - Beispiel: BankAccountSould.have\_the\_balance\_increased\_after\_a\_deposit vs. BankAccountTests.testDepositIncreasesBalance
  - => It dosen't really matter witch naming convention is used, but it is important, that name of the testmethod mirrors the business rule under test.

### Transformation priority premises

• Ruleset to avoid overengineering

#

TRANSFORMATION	STARTING CODE	FINAL CODE
{} => nil		return nil
nil => constant	return nil	return "1"
constant => constant+	return "1"	return "1" + "2"
constant => scalar	return "1" + "2"	return argument
statement => statements	return argument	return arguments
unconditional => conditional	return arguments	if(condition)return arguments
scalar => array	dog	[dog, cat]
array => container	[dog, cat]	$\{ dog = "DOG", cat = "CAT" \}$
statement => tail recursion	a + b	a + recursion
conditional => loop	if (condition)	while(condition)
tail recursion => full recursion	a + recursion	recursion
expression => function	today – birthday	CalculateAge()
variable => mutation	day	var day = 10; day = 11;
switch case		

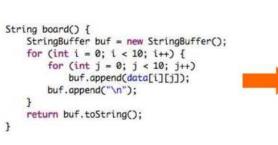
- Application of the Rules
  - Rule 1 is sufficient to satisfy the first failing test case
  - If rule n isn't adequate anymore, then apply rule n+1

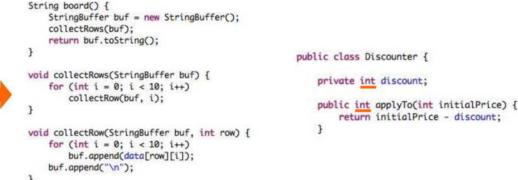
## Object Calisthenics (beauty & strength)

Rule	Why	
Only one level of indentation per method	<ul><li>Focus (single responsibility)</li><li>Size</li></ul>	
Don't use the ELSE keyword	<ul> <li>Single execution line</li> <li>Handle complex cases by polymorphism (Stragegy-Pattern)</li> <li>Use a Map</li> </ul>	
Wrap all primitives and strings	<ul><li>Better readability</li><li>Bundle behaviour and data</li></ul>	
First class collections (wrap all collections)	<ul> <li>Bundle behaviour and data</li> <li>Encapsulation</li> <li>Streaming</li> </ul>	
No getters/setters/properties	Bundle behaviour and data	
One dot per line	<ul> <li>Readability</li> <li>Hiding implementation (by not passing attribut values of an Object)</li> <li>Only talk to friends (Law of Demeter)</li> <li>Tell, don't ask moves behavoiur from the calling class to the called class</li> </ul>	
Don't abbreviate	<ul><li>Long names are indicators for missing concepts</li><li>Avoid confusion</li></ul>	
Keep entities small	<ul> <li>Single responsibility</li> <li>Complexity (no class over 50 lines, no package over 10 files)</li> </ul>	
No classes with more than two instance variables	<ul><li>Low cohesion for actuator classes</li><li>High cohesion only for orchestrators classes</li></ul>	

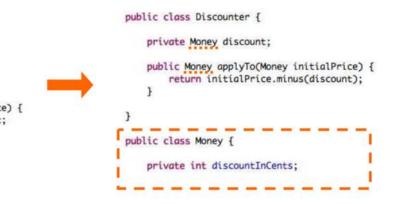
### Some Code snippets for visualisation

#### Only one level of indentation per method





#### Wrap all primitives and strings



### No getters/setters/properties



#### First class collections (wrap all collections)

#### public class Accounts {

private Map<AccountId, Account> accounts = new HashMap<AccountId, Account>();