1. Tutorial

Tutor: Markus Ast

Organization

- Tutorial room may change
- Check out VSR page / OPAL announcements for room schedule

Exams

- In most cases in written form
- No computer allowed
- C# as language to write code examples

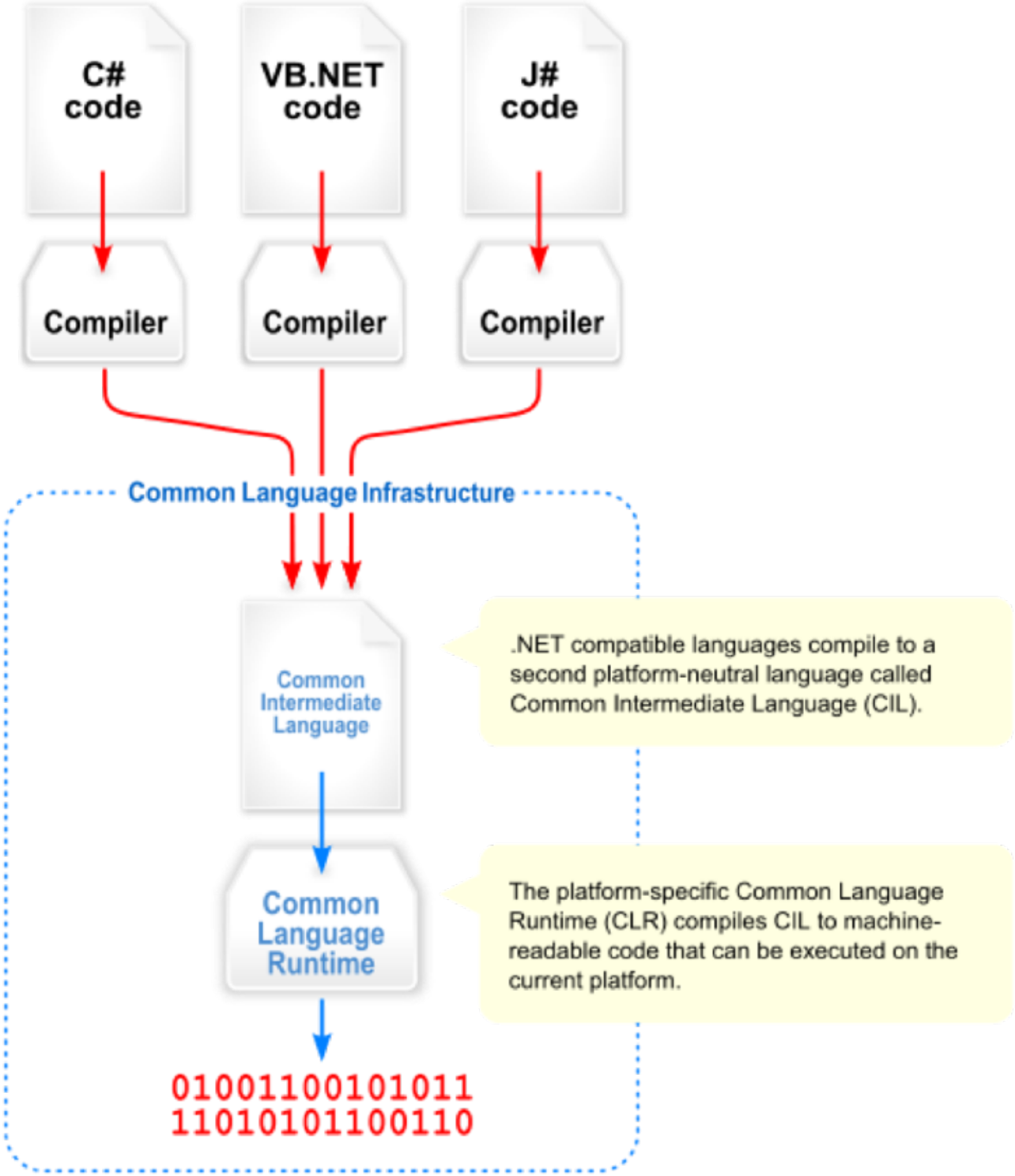
Tools

- Visual Studio Code with C# extension
- C# / .NET Core

.NET Framework

- Development using .NET-Framework
  - Runtime Environment
    - Memory and resource management
  - Class Library
    - More than 12.000 classes and datatypes
    - Grouping into namespaces

    - Tools and services
- Outcomes:
    - Console, Desktop, Web Applications
    - (Web Services)
    - Class Libraries
    - Components
- .NET Family:



C# Introduction

- **Characteristics**

- Very similar to Java
- Strict type system
- Object-oriented
- Automatic Garbage Collection

→ **Focus on robustness, durability and productivity**

- **Data types**
  - Value types: `int`, `double`, `bool`, ...
  - Reference types: `string`, `object`, `Exception`, ...
  - Generic types: `List<string>`, `Stack<int>`
  - Boxing and Unboxing

> *Boxing is the process of converting a value type to the type `object` or to any interface type implemented by this value type.*

```
// Value types
int i = 1;
double d = 0.25;
bool b = true;

// Reference types
string s = @"This is escaped \string";
object obj = new StringBuilder();

// Boxing and unboxing
Int32 boxedInt = i;
object boxedInt2 = i;
int unboxedInt = (int) boxedInt;
int unboxedInt2 = (int) boxedInt2;
```

- Control structures

```
if(a == b)
{
   foreach (var item in items)
   {
      switch (item)
      {
         case "a":
           a++;
           break;
         default:
```

```
            b++;
            break;
        }
    }
}
```

```
for(int i=0; i<5; i++)
{
    while(i > 2)
    {
        do
        {
          Console.WriteLine("hello");
        } while (i < 3);
    }
}
```

- Inheritance and interfaces

```
public interface INetwork
{
   NetworkAddress ResolveHostName(string serverName);
}
public abstract class AbstractNetwork : INetwork
{…}
public class EthernetNetwork : AbstractNetwork
{…}
public class WirelessNetwork : AbstractNetwork
{…}
```

- Class library
  - `IEnumerable`
  - `object[]`
  - `List<…>`
  - `Dictionary<…, …>`
  - `Exception`
  - `Regex`
  - `Random`
  - `Console`

- Lambda-Expressions and LINQ

```
Func<int, int> transformer = x => x*x;
Func<int, int, int> transformer2 = (x,y) => x + y;
Console.WriteLine(transformer(3));
Console.WriteLine(transformer2(4,5));

IEnumerable<Point> points = GetAllPoints()
IEnumerable<string> labels = points.Where(point => point.X > 10).
                              Select(point => point.Label);

labels = from point in points
         where point.X > 10
         select point.Label;
```

Task 1

> *Create a simple C# Hello World application.*

Task1.zip
Shared on Dropbox

Task 1 Solution

Create a new console application into the directory `task1` with

```
dotnet new console -o task1
```

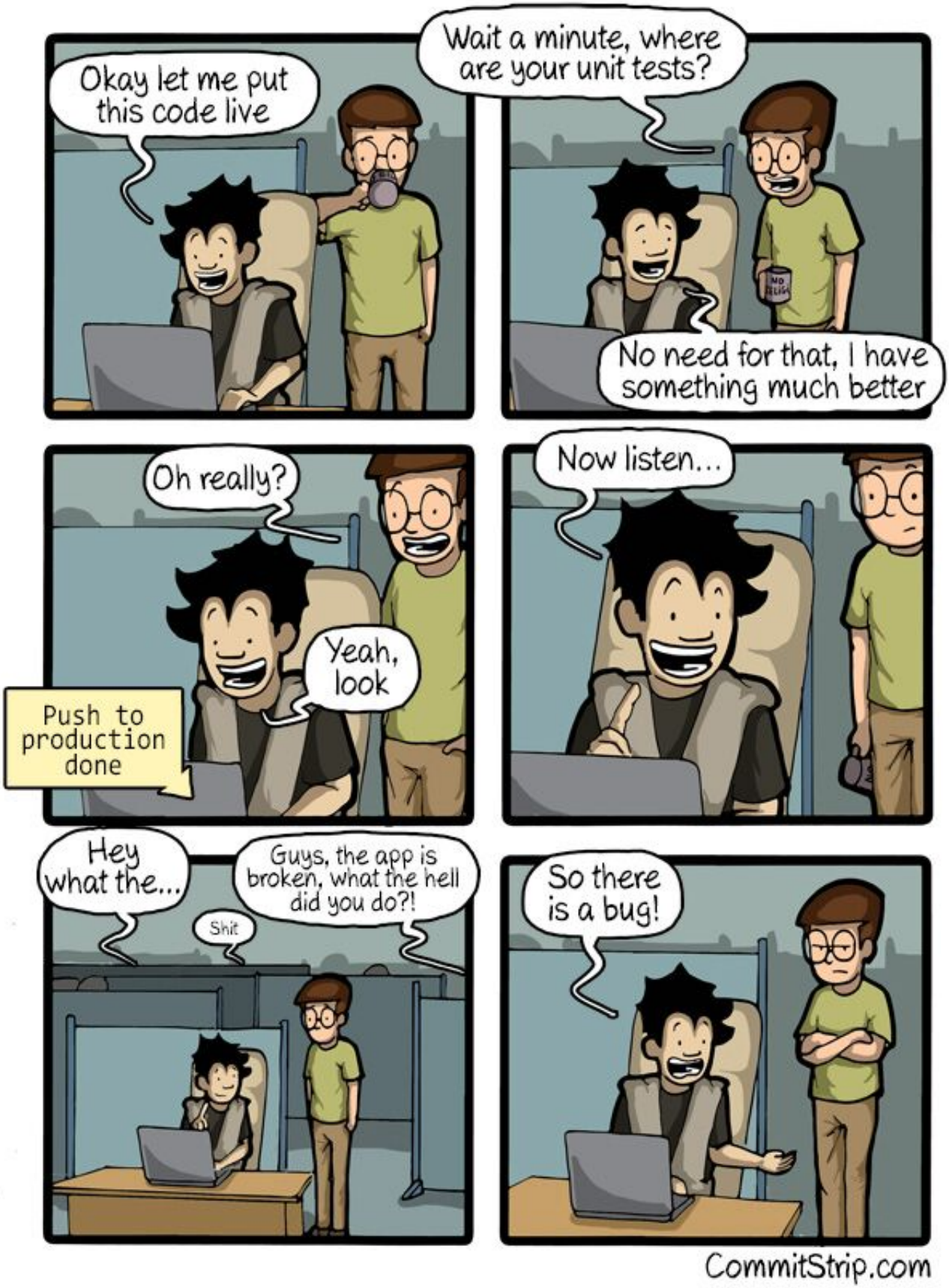
Run with `F5` in Visual Studio Code or in the console with

```
dotnet run
```

Task 2.1.

> *Get informed about Unit Testing and Test Driven Development (TDD). Answer the following questions:*
> - *What are the advantages and disadvantages of writing unit tests?*
> - *What is the difference between unit, integration and system tests?*
> - *How does the lifecycle of TDD look like?*

Unit Testing



- Technique to programmatically verify expected code behaviour
- Automatic check of system integrity at any time
- Facilitates clean design and separation of concerns
- But:

- Tests are code and thus must be maintained as well
- Testing tests is hard

Types of Tests

- Unit Tests
  - Testing of isolated code parts
- Integration Tests
  - Testing of integrated components
- System Tests
  - Verification of the system compliance with specified requirements (incl. usability, security, scalability, etc.)

Why integration tests?

Test Driven Development

A process of writing code starting from a test:

1. Write a test that describes the behaviour of a function under the test
2. Make sure the test **fails** (the function doesn't exist or is not implemented yet)
3. Implement the function (do not change or edit other code)
4. Make sure the test **passes**
5. Perform refactoring (if needed)
6. Make sure the test still passes

Task 2.2

> *Implement a simple* Calculator *application in the TDD manner. The application should enable the following computations:*
> *• Multiplication of two integers*
> *• Division of two integers*

Task2.zip
Shared on Dropbox

Task 2 Solution

Tips:

- Extension: .NET Core Test Explorer
- Create new project with XUnit template: `dotnet new xunit -o task2`

Task 3 - Hometask

> *Inform yourself about when Mock Objects should be used. Extend the* Calculator *class from the Task 2 to write the result of the computation to a file on a local hard drive. Use TDD and Mock Objects to simulate exceptional situations (e.g., drive is not ready or file is locked)*

Task3.zip
Shared on Dropbox

Task 3 Solution

Tips:

- Use Moq NuGet package

Mock Objects

Simulate behaviours of real objects if they:

- Are slow (e.g. database connections or networks)
- Do not yet exist
- Provide results which are not predictable or hard to reproduce (network errors)
- Avoid placing test data into real objects