

Basic Designs and How We Got Them Wrong

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM

This talk may not change how you do things

BUT WILL EXPLAIN HOW TO MAKE THEM BETTER

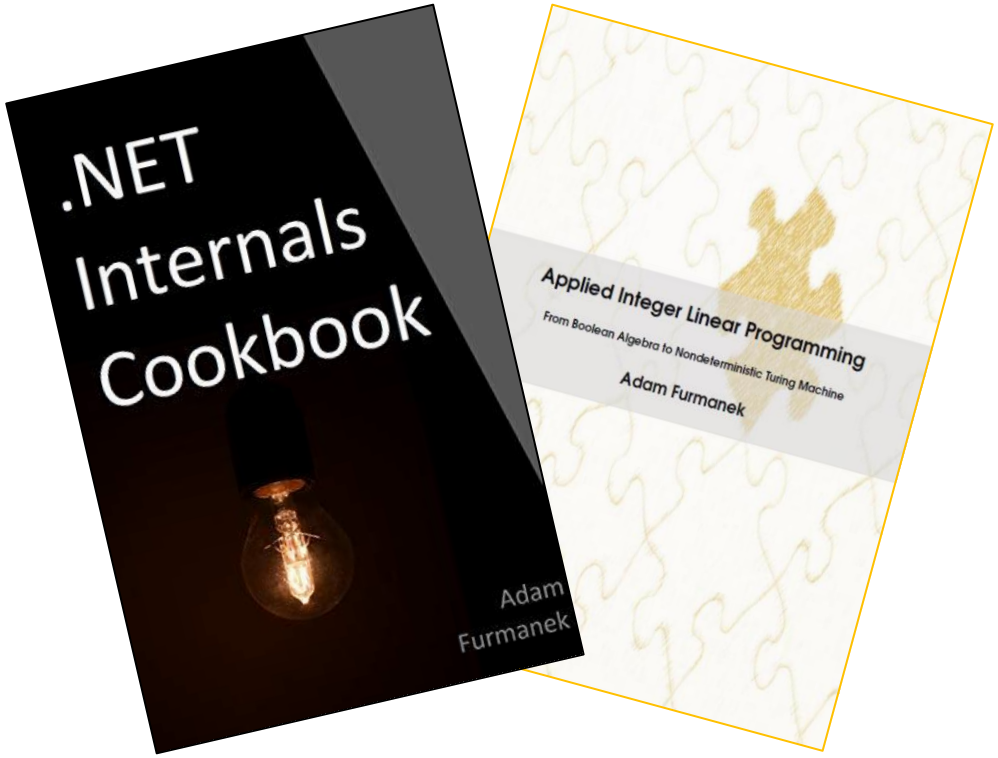
About me

Software Engineer, Blogger, Book Writer, Public Speaker.
Author of ***Applied Integer Linear Programming*** and ***.NET Internals Cookbook***.

http://blog.adamfurmanek.pl

contact@adamfurmanek.pl

furmanekadam



Random IT Utensils

IT, operating systems, maths, and more.

Agenda

Liskov Substitution Principle.

Dependency Inversion (and async...).

Inheritance.

Exceptions.

Encapsulation.

Objects.

Pure functions.

Summary.

Liskov Substitution Principle

Liskov Substitution Principle

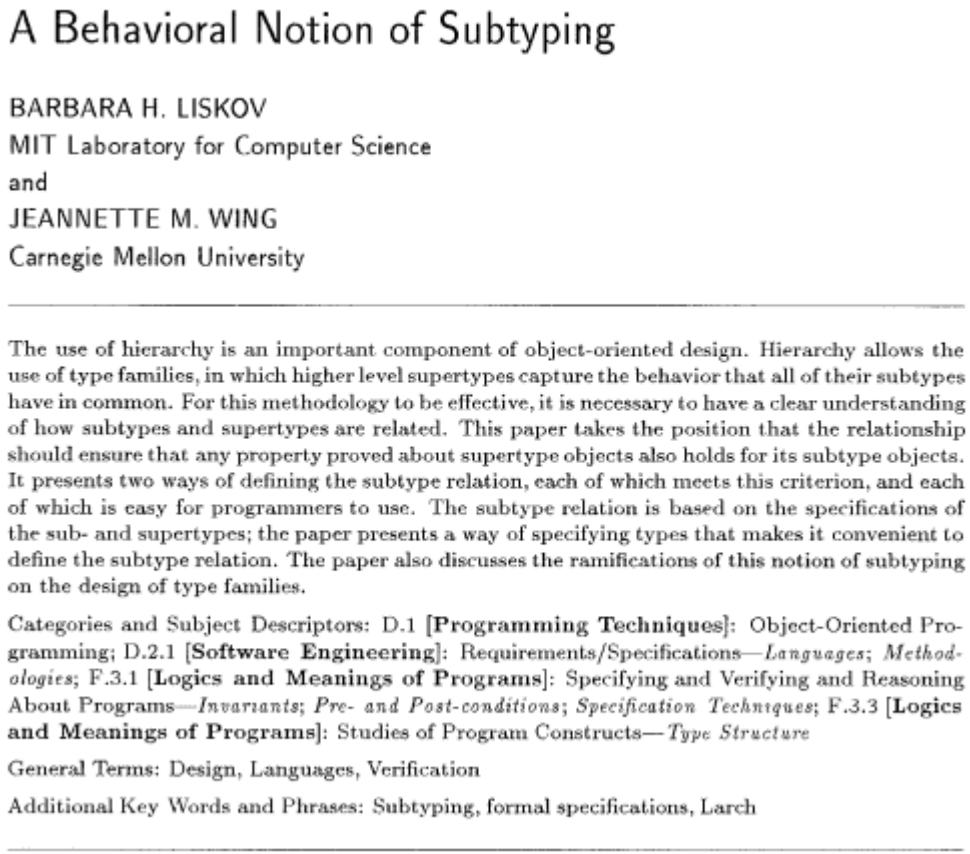
Part of SOLID principles.

The principle comes from A Behavioral Notion of Subtyping paper from 1994.

It considers issues with covariance, contravariance, subtyping, contracts, invariants, and history.

„This paper takes the position that the relationship should ensure that any property proved about supertype objects also holds for its subtype objects".

Many articles have been written, including the one from Robert C. Martin.



A Behavioral Notion of Subtyping

BARBARA H. LISKOV
MIT Laboratory for Computer Science
and
JEANNETTE M. WING
Carnegie Mellon University

The use of hierarchy is an important component of object-oriented design. Hierarchy allows the use of type families, in which higher level supertypes capture the behavior that all of their subtypes have in common. For this methodology to be effective, it is necessary to have a clear understanding of how subtypes and supertypes are related. This paper takes the position that the relationship should ensure that any property proved about supertype objects also holds for its subtype objects. It presents two ways of defining the subtype relation, each of which meets this criterion, and each of which is easy for programmers to use. The subtype relation is based on the specifications of the sub- and supertypes; the paper presents a way of specifying types that makes it convenient to define the subtype relation. The paper also discusses the ramifications of this notion of subtyping on the design of type families.

Categories and Subject Descriptors: D.1 [**Programming Techniques**]: Object-Oriented Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages*; *Methodologies*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning About Programs—*Invariants*; *Pre- and Post-conditions*; *Specification Techniques*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type Structure*

General Terms: Design, Languages, Verification

Additional Key Words and Phrases: Subtyping, formal specifications, Larch

Square and Rectangle

According to Wikipedia:

1) A typical example that violates LSP is a ***Square*** class that **derives** from a ***Rectangle*** class, assuming getter and **setter** methods exist **for both width and height**.

2) The ***Square*** class always **assumes** that the **width is equal with the height**.

3) If a ***Square*** object is **used** in a context **where** a ***Rectangle* is expected**, unexpected behavior may occur because the dimensions of a *Square* cannot (or rather should not) be modified independently.

Ratio-maintaining component

We implement a framework for UI applications (let's call it *WPF* = *Widely Popular Framework*).

We have a top class called ***Component*** that can be put on the canvas.

Component has ***Width*** and ***Height***.

Let's say that we want to implement a component that maintains the image ratio. Let's call it ***FixedImage***.

Any decent framework would requre ***FixedImage*** **inherit from (or implement)** ***Component***.

Does it break LSP?

This Doesn’t Break LSP!

Let’s see the paper

Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T.

The $\phi(y)$ should be $true$ for objects y of type S where S is a subtype of T.

Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Let’s see the paper

In other words: we have a property that can be proved for the objects of the base type.

We should ensure that we can prove that property for the objects of the subtype.

Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Easy!

```
class BaseType{
    public virtual void Foo(){
        Console.WriteLine("Literal");
    }
}
```

```
BaseType.Foo()
    L0000: mov ecx, [0x19d27e88]
    L0006: call System.Console.WriteLine(System.String)
    L000b: ret
```

My ϕ is:

The compiled code of the method *Foo* is as in the bottom snippet.

And completely weird!

My ϕ is: the compiled code of the method *Foo* is as in the bottom snippet.

This must be true for every subtype of *BaseType*.

Effectively, we can't change the method implementation.

```
BaseType.Foo()
    L0000: mov ecx, [0x19d27e88]
    L0006: call System.Console.WriteLine(System.String)
    L000b: ret
```

LSP is about the explicit contract.

Contract

According to Wikipedia:

It [Design by contract] prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants. These specifications are referred to as "**contracts**", in accordance with a conceptual metaphor with the conditions and obligations of business contracts.

First coined by Bertrand Meyer in 1986 (8 years before LSP).

Square and Rectangle revisited

According to Wikipedia:

1) A typical example that violates LSP is a ***Square*** class that **derives** from a ***Rectangle*** class, assuming getter and **setter** methods exist **for both width and height**.

2) The ***Square*** class always **assumes** that the **width is equal with the height**.

3) If a ***Square*** object is **used** in a context **where** a ***Rectangle*** **is expected**, unexpected behavior may occur because the dimensions of a *Square* cannot (or rather should not) be modified independently.

The problem is: there is no contract specifying that we can't change one side without changing the other.

We tend to assume such a contract because of what we learned at school.

However, **contracts must be explicit**.

Contracts

Must be explicit.

Must be precise, formal, and verifiable.

The authors of the contract may not be able to verify them on their own.

The contract may be specified outside of the source code:

- Documentation
- Wikis
- Diagrams
- Organization’s patterns

Mocks based on contracts

We need to mock production components (for whatever reason).

Mocks do not represent the actual components – whitebox and London (Mockist) TDD.

Don't use mocks if possible

- Follow Detroit TDD
- Run production code as much as possible
- Design by contract

Any non-production component, no matter how smart, is a mock.

Solution – **Liskov Substitution Principle**.

It's not (only) about the inheritance.

It's about the contract

- Preconditions
- Invariants
- Postconditions
- History principle

Contracts cannot be verified by the callee! **Only the caller can verify them**.

Contracts don't need to be written down in the code.

It’s easy to break contract with mocks

```
public <T, Client> T doWithOptionalBackupConnection(Client mainClient, Client backupClient, Function<Client, T> action){
    if(rand()%2 == 0){
       // First main, then backup
       return doInternal(mainClient, backupClient, action);
    }else {
       // First backup, then main
       return doInternal(backupClient, mainClient, action);
    }
}

private <T, Client> T doInternal(Client mainClient, Client backupClient, Function<Client, T> action){
    try {
        return action.apply(mainClient);
    } catch(Exception e){
        logger.log(e);
    }

    return action.apply(backupClient)
}
```

```
interface Client {
    String getData(String parameter);
}

public String doWork(Client mainClient, Client backupClient){
    String first = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("First"));
    String second = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Second"));
    String third = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Third"));
    return first + second + third;
}
```

It’s easy to break contract with mocks

```
interface Client {
    String getData(String parameter);
}

public String doWork(Client mainClient, Client backupClient){
    String first = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("First"));
    String second = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Second"));
    String third = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Third"));
    return first + second + third;
}
```

```
@Test
public void test(){
    assert(doWork(throwingClient, workingClient) == "Expected");
}
```

It’s easy to break contract with mocks

```
public <T, Client> T doWithOptionalBackupConnection(Client mainClient, Client backupClient, Function<Client, T> action){
    if(rand()%2 == 0){
       // First main, then backup
       return doInternal(mainClient, backupClient, action);
    }else {
       // First backup, then main
       return doInternal(backupClient, mainClient, action);
    }
}

private <T, Client> T doInternal(Client mainClient, Client backupClient, Function<Client, T> action){
    try {
        return action.apply(mainClient);
    } catch(Exception e){
        logger.log(e);
    }

    return action.apply(backupClient)
}
```

```
interface Client {
    CompletableFuture<String> getData(String parameter);
}

public String doWork(Client mainClient, Client backupClient){
    CompletableFuture<String> first = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("First"));
    CompletableFuture<String> second = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Second"));
    CompletableFuture<String> third = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Third"));
    return first.WaitForResult() + second.WaitForResult() + third.WaitForResult();
}
```

It’s easy to break contract with mocks

```
interface Client {
    CompletableFuture<String> getData(String parameter);
}

public String doWork(Client mainClient, Client backupClient){
    CompletableFuture<String> first = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("First"));
    CompletableFuture<String> second = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Second"));
    CompletableFuture<String> third = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Third"));
    return first.WaitForResult() + second.WaitForResult() + third.WaitForResult();
}
```

```
@Test
public void test(){
    assert(doWork(throwingClient, workingClient) == "Expected");
}
```

It’s easy to break contract with mocks

```
interface Client {
    String getData(String parameter);
}

public String doWork(Client mainClient, Client backupClient){
    String first = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("First"));
    String second = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Second"));
    String third = utilInstance.doWithOptionalBackupConnection(mainClient, backupClient, client -> client.getData("Third"));
    return first + second + third;
}
```

```
@Test
public void test(){
    // Whatever set up needed for client
    prepareFailingConditions();

    assertThrows((client -> client.getData("First"))(clientUnderTest));
}
```

You can put anything in the contract Interesting case of Collection.add in Java

`boolean add(E e)`

Ensures that this collection contains the specified element (optional operation). Returns `true` if this collection changed as a result of the call. (Returns `false` if this collection does not permit duplicates and already contains the specified element.)

Collections that support this operation may place limitations on what elements may be added to this collection. In particular, some collections will refuse to add `null` elements, and others will impose restrictions on the type of elements that may be added. Collection classes should clearly specify in their documentation any restrictions on what elements may be added.

If a collection refuses to add a particular element for any reason other than that it already contains the element, it *must* throw an exception (rather than returning `false`). This preserves the invariant that a collection always contains the specified element after this call returns.

Parameters:

`e - element whose presence in this collection is to be ensured`

Returns:

`true if this collection changed as a result of the call`

Throws:

`UnsupportedOperationException - if the add operation is not supported by this collection`

`ClassCastException - if the class of the specified element prevents it from being added to this collection`

`NullPointerException - if the specified element is null and this collection does not permit null elements`

`IllegalArgumentException - if some property of the element prevents it from being added to this collection`

`IllegalStateException - if the element cannot be added at this time due to insertion restrictions`

Immutable vs readonly

What happens when we accept „immutable” collection as a function parameter

- Do we say that „we won’t modify the collection”?
- Or do we say that „this collection can’t be modified ever”?

While we can encode the first in the type system, we can’t encode the second.

Encoding the first is like saying „I need to show that I can’t do something”.

Encoding the second is like saying „I need to show that nobody can do something”. We can’t do that in general.

Mutable and immutable collections (Java)

1. **Why don't you support immutability directly in the core collection interfaces so that you can do away with *optional operations* (and UnsupportedOperationException)?**

   This is the most controversial design decision in the whole API. Clearly, static (compile time) type checking is highly desirable, and is the norm in Java. We would have supported it if we believed it were feasible. Unfortunately, attempts to achieve this goal cause an explosion in the size of the interface hierarchy, and do not succeed in eliminating the need for runtime exceptions (though they reduce it substantially).

   Doug Lea, who wrote a popular Java collections package that did reflect mutability distinctions in its interface hierarchy, no longer believes it is a viable approach, based on user experience with his collections package. In his words (from personal correspondence) "Much as it pains me to say it, strong static typing does not work for collection interfaces in Java."

   To illustrate the problem in gory detail, suppose you want to add the notion of modifiability to the Hierarchy. You need four new interfaces: ModifiableCollection, ModifiableSet, ModifiableList, and ModifiableMap. What was previously a simple hierarchy is now a messy heterarchy. Also, you need a new Iterator interface for use with unmodifiable Collections, that does not contain the remove operation. Now can you do away with UnsupportedOperationException? Unfortunately not.

   Consider arrays. They implement most of the List operations, but not remove and add. They are "fixed-size" Lists. If you want to capture this notion in the hierarchy, you have to add two new interfaces: VariableSizeList and VariableSizeMap. You don't have to add VariableSizeCollection and VariableSizeSet, because they'd be identical to ModifiableCollection and ModifiableSet, but you might choose to add them anyway for consistency's sake. Also, you need a new variety of ListIterator that doesn't support the add and remove operations, to go along with unmodifiable List. Now we're up to ten or twelve interfaces, plus two new Iterator interfaces, instead of our original four. Are we done? No.

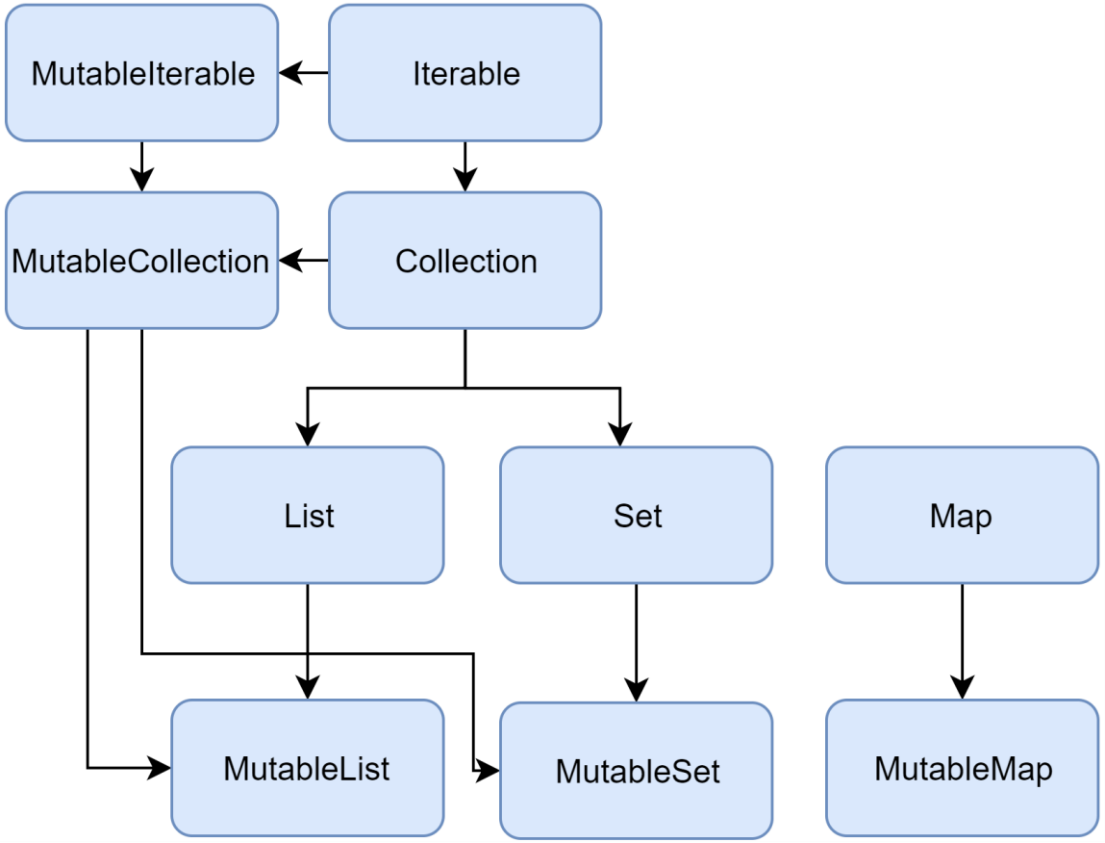
   Consider logs (such as error logs, audit logs and journals for recoverable data objects). They are natural append-only sequences, that support all of the List operations except for remove and set (replace). They require a new core interface, and a new iterator.

   And what about immutable Collections, as opposed to unmodifiable ones? (i.e., Collections that cannot be changed by the client AND will never change for any other reason). Many argue that this is the most important distinction of all, because it allows multiple threads to access a collection concurrently without the need for synchronization. Adding this support to the type hierarchy requires four more interfaces.

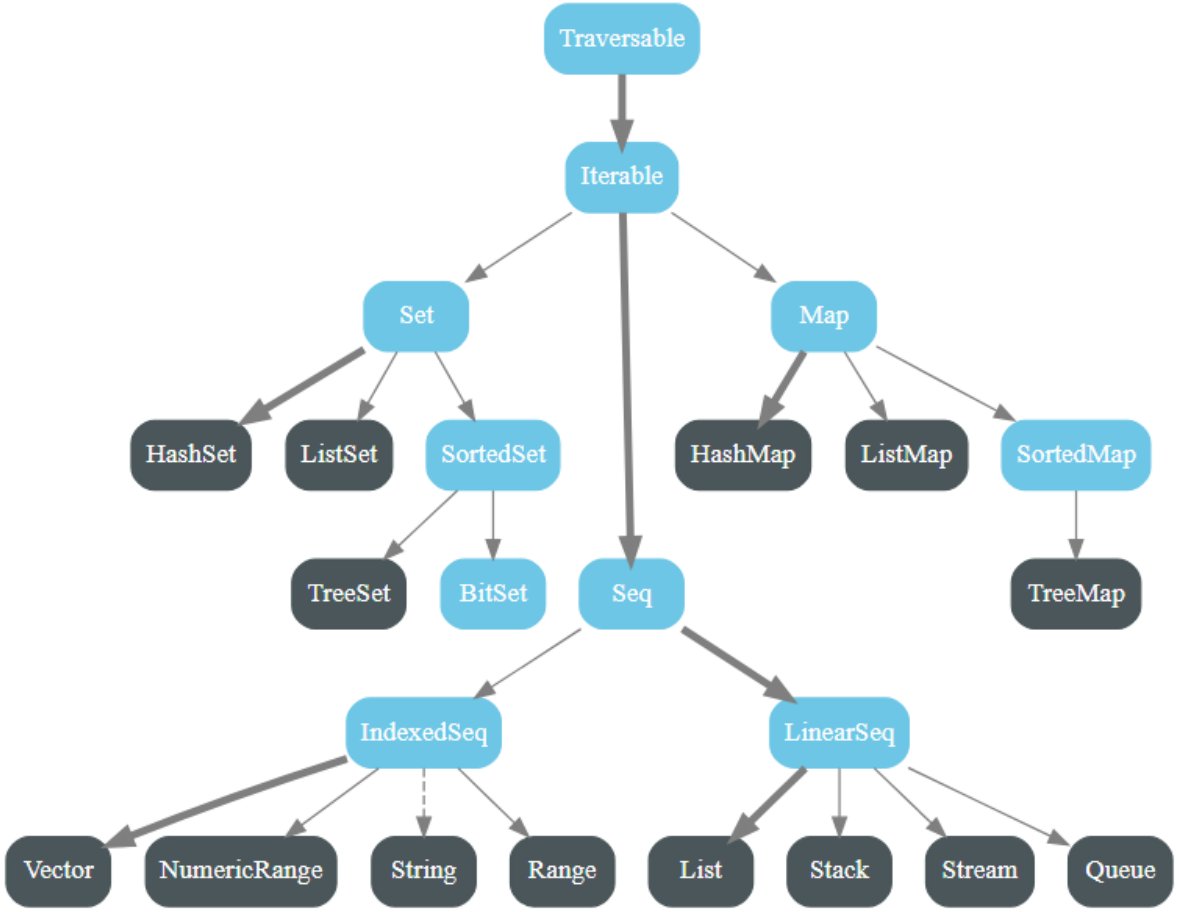
   Now we're up to twenty or so interfaces and five iterators, and it's almost certain that there are still collections arising in practice that don't fit cleanly into any of the interfaces. For example, the *collection-views* returned by Map are natural delete-only collections. Also, there are collections that will reject certain elements on the basis of their value, so we still haven't done away with runtime exceptions.

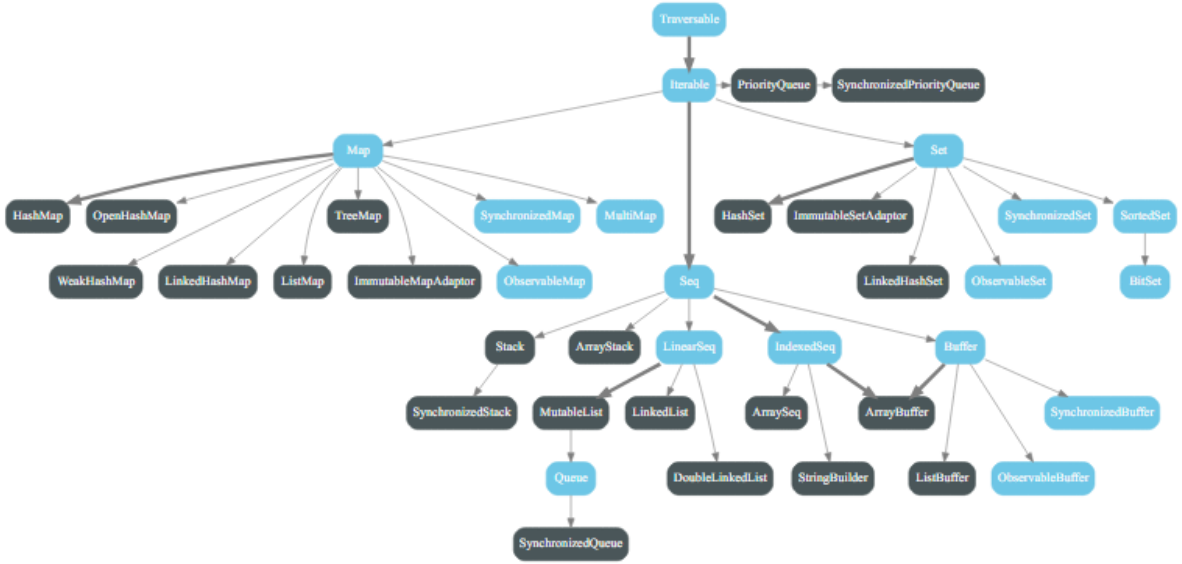
   When all was said and done, we felt that it was a sound engineering compromise to sidestep the whole issue by providing a very small set of core interfaces that can throw a runtime exception.

Mutable and immutable collections (Kotlin)



Mutable and immutable collections (Scala)





Dependency Inversion

HOW DO YOU REPLACE A STRING?

String improvements in Java

Internal structure:

- Originally String was implemented using char array under the hood.
- Java 9 changed it to byte array to allocate 1 byte if string has no unicode characters.

Concatenation performance:

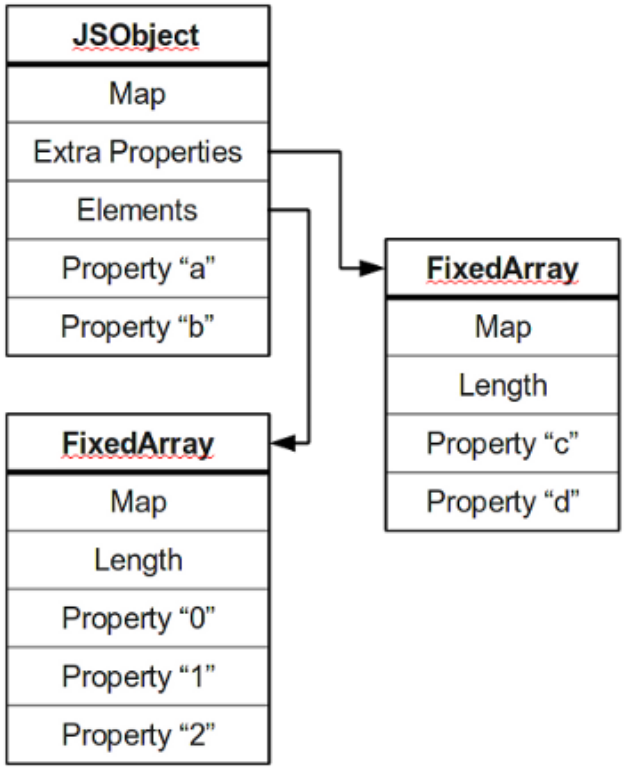
- Before Java 9 concatenations were translated to StringBuilder.append.
- Starting in Java 9 they are translated to invokedynamic and reuse multiple strategies.

String at Facebook – 1% performance win

CppCon 2016: Nicholas Ormrod “The strange details of std::string at Facebook"



Objects in V8



Object is a dictionary — use hash map.

Use maps to optimize access by offset.

Reserve more memory than needed to have room for new properties.

Property names are strings but for arrays we can use… well, arrays.

V8 switches implementation depending on the usage.

How do you replace a string?

You cannot!

String is:

- A class – not an interface
- A sealed class – no inheritance
- Exposed on IL level via string literals
- Highly coupled with managed and unmanaged code (native part relies on memory structure)

This applies to many more things!

Built in things that we might want to override:

- DateTime.Now
- All the things for static configuration
- Environment.Exit

There are even libraries for doing this magic:

- Moles

Colorful functions

	Green	Red
Green		
Red		

Colorful functions

	Green	Red
Green		
Red		

	void Foo()	async Task Foo()
void Foo()		
async Task Foo()		

Colorful functions

	Green	Red
Green		
Red		

	void Foo()	async Task Foo()
void Foo()		
async Task Foo()		

	void Foo()	async ValueTask Foo()	async Task Foo()
void Foo()			
async ValueTask Foo()			
async Task Foo()			

DRY

ASYNC COROUTINES

Do not repeat yourself (DRY)

```
private static string InternalReadAllText(string path, Encoding encoding)
{
    Debug.Assert(path != null);
    Debug.Assert(encoding != null);
    Debug.Assert(path.Length > 0);

    using (StreamReader sr = new StreamReader(path, encoding, detectEncodingFromByteOrderMarks: true))
        return sr.ReadToEnd();
}
```

```
private static async Task InternalReadAllTextAsync(string path, Encoding encoding, CancellationToken cancellationToken)
{
    Debug.Assert(!string.IsNullOrEmpty(path));
    Debug.Assert(encoding != null);

    char[] buffer = null;
    StreamReader sr = AsyncStreamReader(path, encoding);
    try
    {
        cancellationToken.ThrowIfCancellationRequested();
        buffer = ArrayPool.Shared.Rent(sr.CurrentEncoding.GetMaxCharCount(DefaultBufferSize));
        StringBuilder sb = new StringBuilder();
        while (true)
        {
#if MS_IO_REDIST
            int read = await sr.ReadAsync(buffer, 0, buffer.Length).ConfigureAwait(false);
#else
            int read = await sr.ReadAsync(new Memory(buffer), cancellationToken).ConfigureAwait(false);
#endif
            if (read == 0)
            {
                return sb.ToString();
            }

            sb.Append(buffer, 0, read);
        }
    }
    finally
    {
        sr.Dispose();
        if (buffer != null)
        {
            ArrayPool.Shared.Return(buffer);
        }
    }
}
```

Inheritance

Single inheritance

Typically, we can inherit from one base class only. We can implement multiple interfaces, though.

Sometimes we can’t inerit at all – structs in C#.

There is no multiple inheritance in C# nor Java.

Or is there?

Multiple inheritance

A feature of OOP in which a class can inherit from multiple classes.

Some languages (most notably C++) support that.

C# (and others) doesn’t support that because Java decided not to.

Java doesn’t support the multiple inheritance due to **diamond problem**.

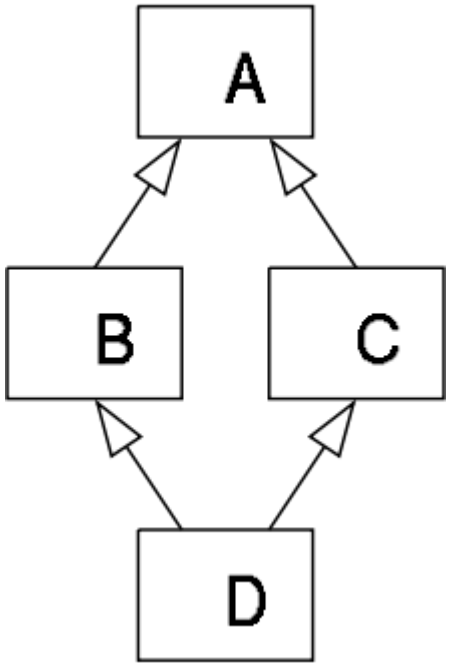
Diamond problem

Situation when **two subclasses** *B* and *C* **inherit from the same base class** *A*, **and then class** *D* **inherits from both** *B* and *C*.

The problem arises when we want to use something from the class *D* that comes from the class *A*. **We have two copies of the element from the base class and we don't know which one to use**.

In other words: the diamond problem is when we have two „similar" things and there is no „best answer" which one to use.

Is this a problem at all?



We’ve solved the diamond problem many times already!

Diamond problem and interfaces

We have two „independent” methods with the same signature.

Is this a problem?

In this case, there is no problem at all. We have two identical signatures and just one implementation.

```
interface A {
    void Foo();
}

interface B {
    void Foo();
}

class C : A, B {
    public void Foo(){
        Console.WriteLine("Foo");
    }
}
```

```
public static void Main()
{
    C c = new C();
    c.Foo();
}
```

```
Foo
```

Diamond problem and arrays

We have two methods with different input parameters.

We call the method passing values that match both of the methods.

Is this a problem?

C# compiler prefers the method with no casting needed and calls „Two integers”.

This can break compatibility.

This is the diamond problem in disguise.

```
class A {
    public void Foo(int a, int b){
        Console.WriteLine("Two integers");
    }

    public void Foo(params int[] array){
        Console.WriteLine("Array");
    }
}

public static void Main()
{
    A a = new A();
    a.Foo(1, 2);
}
```

Diamond problem and type priority

We have two methods with different input parameters.

We call the method passing a value that needs to be casted to one of the types.

Is this a problem?

C# compiler prefers to cast integers to long values, so the first method is picked.

This can break compatibility.

This is the diamond problem in disguise.

```
class A {
    public void Foo(long l){
        Console.WriteLine("Long");
    }

    public void Foo(double d){
        Console.WriteLine("Double");
    }
}

public static void Main()
{
    A a = new A();
    a.Foo(123);
}
```

Diamond problem and interfaces part 2

We have two methods with different return type.

Is this a problem?

The diamond problem is not solved because C# doesn't support covariant methods here.

This works in Java because they use bridge methods.

```
interface A {
    object Foo();
}

interface B {
    string Foo();
}

class C : A, B {
    public string Foo(){
        Console.WriteLine("Foo");
    }
}
```

```
'Program.C' does not implement interface member 'Program.A.Foo()'.
'Program.C.Foo()' cannot implement 'Program.A.Foo()' because it does not have
the matching return type of 'object'.
```

Default interface implementation Multiple inheritance?

We have two interfaces with the same method signature.

Both interfaces provide the default implementation.

We create a class that inherits from both interfaces.

What happens in *c.Foo()*?

Is this a problem?

The code doesn't compile because we don't know which *Foo* to use. We need to cast *c* to either *A* or *B*.

Scala would call *A.Foo()*.

```
interface A{
    void Foo() {
        Console.WriteLine("A");
    }
}

interface B{
    void Foo() {
        Console.WriteLine("B");
    }
}

class C : A, B {
}

public class Program
{
    public static void Main()
    {
        C c = new C();
        c.Foo();
    }
}
```

Diamond problem

Many ways to solve the problem of „what is the best pick”:

- Nobody picks – simply ban muliple class inheritance
- There is just one pick - virtual inheritance in C++ or same signature example
- We pick arbitrarily – cast *integer* to *long* or prefer the method with better match
- We let the user pick – throw an error and ask for a code fix

Diamond problem has been solved in many places and all solutions have been used.

Types of inheritance

Signature inheritance

Implementation inheritance

State inheritance

Identity inheritance

Signature inheritance

We „inherit” the interface which is a set of signatures.

Signature depends on the programming language

- C# - name + parameter types
- Intermediate Language – name + parameter types + return type
- C++ - name + parameters types + return type + calling convention

There is one case in C# where signature includes the returned type – cast operators.

C# and Java support **multiple signature inheritance** thanks to interfaces.

Implementation inheritance

We „inherit” the implementation which is basically a method body.

Typically implemented by traits, mixins, or „multiple inheritance”.

C# and Java support **multiple implementation inheritance** thanks to default interface implementation.

State inheritance

We „inherit” the state which is basically fields.

Can be implemented with traits, mixins, or „multiple inheritance”.

C# and Java support single state inheritance thanks to „regular inheritance”.

C# and Java don’t support multiple state inheritance. It can be implemented with implementation inheritance (so default interface implementation again).

Identity inheritance

We „inherit” the identity which is basically a constructor.

C# and Java support single identity inheritance (that’s the „regular inheritance”).

C# anda Java don’t support multiple identity inheritance. This can be implemented in a hacky way with memory manipulation.

Multiple Inheritance

```
static void Main(string[] args)
{
    MultipleBase child = new FakeChild1
    {
        currentState = new CurrentState(new Base1(), new Base2(), new Base3(), new Base4())
    };

    Console.WriteLine("Base1");
    Base1 base1 = child.Morph<FakeChild1, Base1>();
    base1.field = 123;
    base1.PrintInt();
    Console.WriteLine();

    Console.WriteLine("Base2");
    Base2 base2 = child.Morph<FakeChild2, Base2>();
    base2.field = 456.0f;
    base2.PrintFloat();
    Console.WriteLine();

    Console.WriteLine("Base3");
    Base3 base3 = child.Morph<FakeChild3, Base3>();
    base3.field1 = 789;
    base3.field2 = 987;
    base3.PrintFields();
    Console.WriteLine();

    Console.WriteLine("Base4");
    Base4 base4 = child.Morph<FakeChild4, Base4>();
    base4.field = "Abrakadabra";
    base4.PrintString();
    Console.WriteLine();

    Console.WriteLine("Base3 again");
    base3 = child.Morph<FakeChild3, Base3>();
    base3.PrintFields();
    Console.WriteLine();

    Console.WriteLine("Base2 again");
    base2 = child.Morph<FakeChild2, Base2>();
    base2.PrintFloat();
    Console.WriteLine();

    Console.WriteLine("Base1 again");
    base1 = child.Morph<FakeChild1, Base1>();
    base1.PrintInt();
}
```

Exceptions

Exceptions – what's the output?

```
try{
    try{
        throw new Exception("Exception 1");
    }finally{
        throw new Exception("Exception 2");
    }
}catch(Exception e){
    Console.WriteLine(e);
}
```

Exceptions

C# will lose the first exception.

Python 2 will lose the first exception.

Python 3 will store the first exception as a property on the second one.

Java may lose or keep, depending on the exception type.

Why would we care?

Resource management

```
using(var resource = new Resource()){
    // ...
}
```

```
var resource = new Resource();
try{
    // ...
} finally{
    if(resource != null) resource.Dispose();
}
```

Fixing using

```
Exception exception = null;
Exception exception2 = null;
var resource = new Resource();
try{
    Console.WriteLine("Using");
    throw new Exception("Using failed");
}catch(Exception e){
    exception = e;
}finally{
    try{
        if(resource != null) resource.Dispose();
    } catch(Exception e2){
        exception2 = e2;
    }

    if(exception != null && exception2 != null){
        throw new AggregateException(exception, exception2);
    }else if(exception != null){
        ExceptionDispatchInfo.Capture(exception).Throw();
    }else if(exception2 != null){
        ExceptionDispatchInfo.Capture(exception2).Throw();
    }
}
```

Rethrowing exceptions

```
public static void MethodRethrowingException()
{
    try
    {
        MethodThrowingException();
        MethodThrowingException();
    }
    catch (Exception e)
    {
        if (e.Message == "Custom message")
        {
            Console.WriteLine("Custom exception");
        }

        // Rethrow it here
    }
}
```

throw e may lose the call stack or work well.

Special instructions like *throw* may break the call stack (this happens in C#).

Creating new exception and throwing may break the exception semantics.

async may lose the state machine.

ExceptionDispatchInfo.Capture(e).Throw(); is the solution in C#.

Generally – rethrowing exceptions is hard!

Finally

finally is supposed to be executed „no matter what”, however

- It may not be executed when application dies
- May be skipped when application exits
- May be ignored depending on the exception type
- May be skipped on unhandled exceptions

finally may swallow exceptions!

Never return in *finally*! It breaks the two-pass exception system.

Filtering exceptions is prone to the same issue. It’s better to filter outside of *catch* where possible (use exception filters).

Encapsulation

Encapsulation

Typically, one of the two mechanisms:

- A language mechanism for restricting direct access to some of the object's components (sometimes called **Information Hiding**).
- A language construct that facilitates the bundling of data with the methods (or other functions) operating on those data.

Grady Booch defined it as „the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; **encapsulation serves to separate the contractual interface of an abstraction and its implementation**.”

C# defines it as „hiding the internal state and functionality of an object and only allowing access through a public set of functions.”

Implementation

With access modifiers:

- Private, public, protected, internal, package, ...
- Typically, private is for the instances of the type, not only the same instance!
- Scala has *private[this]* that is „really” private.

With naming conventions:

- Private members start with *_underscore*
- Effectively, everything is public

With lexical closure:

- We use nested lambdas that capture local variables within the closure
- „The best” method to really hide variables as it’s hard to find them even with reflection

```
class Private {
    private void Foo () {
        Console.WriteLine("Foo");
    }

    public void Bar(Private other){
        other.Foo();
    }
}
```

Problem with encapsulation

Testing:

- We can't access „private" state from the tests
- We need to make it available by making „friends" – *friend* in C++, *InternalsVisibleTo* in C#, reusing package name in Java

Serialization:

- We need to access private state to create objects
- Sometimes we even need to create objects without calling their constructors

Performance:

- Reading state via methods is slower
- We may need to get pointers to read the memory directly

Encapsulation is not a mechanism for security!

IT'S JUST A BUNCH OF BYTES ANYWAY

Workarounds

Reflection

Debugging

Memory dumps

IL-level assembly modifications

Pointers

Method hijacking (via pointers or descriptors)

Custom class loaders

Encapsulation helps us avoid errors!

View of the data

View is created to display the specific information that a user needs and to hide the portions that shouldn't be accessed.

```
class Encapsulation {
    private int SomeField;

    public int GetSomeField() {
        return SomeField;
    }

    public void SetSomeField(int value) {
        this.SomeField = value;
    }
}
```

```
interface IView {
    int GetSomeField();
    void SetSomeField(int value);
}

class View : IView {
    public int SomeField;

    public int GetSomeField() {
        return SomeField;
    }

    public void SetSomeField(int value) {
        this.SomeField = value;
    }
}
```

Benefits

There is no case of „accessing hidden field” because the field is simply not in the contract.

We can have many views of the same entity for different purposes (slightly tricky in C# or Java, though).

Most importantly: we simply cast the interface to a more concrete view to access everything.

We have a **full compiler support**!

Objects

Instance methods of objects

Type identity – *GetType(), getClass()*

Cloning – *MemberwiseClone(), clone()*

Cleanups – *Finalize()*

Representation – *ToString()*

Access control – *wait(), notify()*

Equality – *Equals()*

Hashing – *GetHashCode()*

Type identity

Typically, implemented as an internal runtime method. The implementation is not in your programming language.

For reference types, it simply gets the pointer to the type tag. For value types, the value is boxed first (and the compiler inserts the type).

Should it be a method or a property? Think about *GetType()* in C#.

Should it be virtual? Is it virtual?

Why is it an instance method?

Cloning

Typically implemented as protected method. Typically part of the platform (or uses low-level constructs).

May require some mix-in or marker interface. May require some crazy implementation. For instance, Java requires the type to:

- Implement empty interface *Cloneable*
- Override the method to be public
- Call *super.clone()* only

Performs shallow copy.

Should it be public?

Should it be on the object at all?

Many consider the implementation wrong. Josh Bloch says *„The Cloneable interface was intended as a mixin interface for objects to advertise that they permit cloning. Unfortunately it fails to serve this purpose...”*.

Cleanups

Typically protected and empty. Is supposed to release resources owned by the object.

It is allowed to bring the object back to life! The object may be registered again for cleaning up in the future in some platforms.

Typically, any thread may call this method. There is no guarantee which one will do that.

Throwing the exception may break the platform!

Related to other cleanup-interfaces (*AutoCloseable*, *IDisposable*). However, there is **typically no requirement to release the resources early**!

You should behave like the GC is never called!

- First, we may have plenty of memory.
- Second, finalize may be time-limited when exiting.

Representation

Default implementation typically returns the name of the type.

Subclasses should override the method. Typical implementation uses hashcode or something similar.

Unfortunately, there is no single „textual representation” of an object. Think of *DebuggerDisplayAttribute*.

Best practice is not to rely on the textual form returned by the implementation.

Why do we have the method then?

Some platforms don’t have the method built into the object and require some marker type.

Access control

Used for synchronizing threads accessing the same object.

Very low-level mechanism that should be avoided today. Use high-level constructs instead.

Usage must follow specific protocol to avoid spurious wake-ups or lost notifications.

Do not use! Unless you really know what you’re doing!

Equality

Indicates whether two instances are equal based on...

- Memory location?
- Identifier?
- Fields?

Two objects of different classes (even if they inherit from each other) cannot be equal!

Two objects can be equal depending on the context:

- Should cloned objects be equal? Should they have the same identifiers?
- Can we ignore units, denominators, non-observable internal state?
- Can we cast between types?
- Is there a single definition of „equality”?

Should this be on the object?

Hashing

Supported for the benefit of the hash tables in the language.

We would like to be able to put all our objects in dictionaries. However, we can’t do that without „generic hashing solution”. But should it be in the object implementation?

This affects the memory layout of the object. Should it be exposed as a field?

This affects how GC works.

May have different performance characteristic depending on the history of an object!

Purity and Immutability

Pure function

Function that has the following properties:

- Return values are identical for identical arguments
  - Specifically, there is no variation with local static variables, non-local variables, mutable reference arguments, input streams, referential transparency, etc.
- Function has no side effects

Side effect is when we modify some state variable outside of the local environment.

Local environment is basically a function call frame.

Pure functions are „obviously better”?

bool double.TryParse(string, out double)

This function is not pure

- It mutates the input parameter by writing to the second argument
- It uses the global state to check the decimal separator

Is this function bad in any way?

Better TryParse

```
ParseResult TryParse(string input, char decimalSeparator){
    return DoSomeParsing();
}

struct ParseResult {
    double Result;
    bool Success;
}
```

Why pure functions

Easy to test.

Easy to optimize by the compiler.

Support referential transparency.

Can be easily memoized.

Can be easily analyzed.

Each function is „pure”!

„Pure” functions take input parameters I and produce output O

- I is the set of input arguments
- O is the return value (output)

„Impure” functions take input parameters (I, S) and produce output O

- I is the set of input arguments
- S is the external state (global variables, static variables, etc.)
- O is the return value (output) + the new state

Technically, „pure” functions take the same parameters but the state is empty: $(I, \emptyset)$.

There is no difference between these two types of functions on the technical level. The only difference is in reasoning.

Reasoning

We can reason about I much easier than about S.

To prove something about I we just need to trace back how I was prepared (analyze the call site).

To prove something about S we need to analyze how S was prepared (the call site) and how it could have been affected „in the meantime”.

Effectively, with I we just prove what happened, whereas with S we need to show what „didn’t happen” as well.

This is not a problem at all when both $\boldsymbol{I}$ **and** $\boldsymbol{S}$ **are small**.

It’s not about purity or impurity.
It’s about reasoning.

Digression

Technically, each function takes (I, S, E) and produce O

- I is the set of input arguments
- S is the external state
- E is the execution environment that the function uses indirectly (or implicitly)
- O is the return value + the new state + the new environment

Environment is basically anything – memory layout of our process, memory used by the operating system, type of the CPU, electrical interference, cosmic rays, etc.

While we can control S (since the function uses it directly), we can't control E.

Digression 2

Sometimes „impure” functions are „much worse” because we can’t use them at all.

For instance, *constexpr* in C++ requires pure functions, or generally things calculated in the compilation time.

Summary

Our basic building blocks are often inefficient.

Remember about the contracts.

Beware of async.

Do not repeat yourself.

Question all decisions as they are often made arbitrarily.

Q&A



References

https://www.cs.cmu.edu/~wing/publications/LiskovWing94.pdf - A Behavioral Notion of Subtyping

https://web.archive.org/web/20151128004108/http:/www.objectmentor.com/resources/articles/lsp.pdf - The Liskov Substitution Principle

https://blog.adamfurmanek.pl/2021/02/06/types-and-programming-languages-part-4 - Diamond Problem

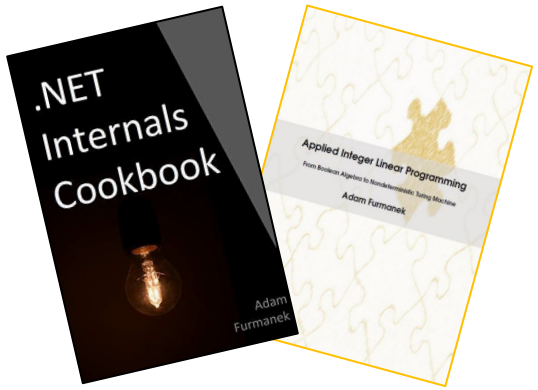
https://blog.adamfurmanek.pl/2021/07/17/types-and-programming-languages-part-6/ - LSP

https://blog.adamfurmanek.pl/2022/07/23/types-and-programming-languages-part-16/ - Encapsulation

https://blog.adamfurmanek.pl/2022/07/30/types-and-programming-languages-part-17/ - LSP in practice

https://blog.adamfurmanek.pl/2021/01/23/types-and-programming-languages-part-3/ - Exception handling

https://blog.adamfurmanek.pl/2020/07/25/net-inside-out-part-21/ - Using is broken



Random IT Utensils

IT, operating systems, maths, and more.

Thanks!

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM