Aspects of object orientation

Jan Niklas Rösch

Institute for Software Engineering and Programming Languages

01. February 2016

Table of contents

Introduction

- Many different aspects that make up a language
- Defining relationship between objects
  - Fundamental facet of OOP
- These aspects contribute to an overall behaviour of the language

Variance - Overview

- Describes behaviour of complex structures
  - Lists
  - Arrays
  - Functions
  - ...
- Covariance and Contravariance
- Invariance and Bivariance

Variance - Definitions

Complex Structure

$I\langle A\rangle$

Subtyping

$A \leq B \quad , \quad I\langle A\rangle \leq I\langle B\rangle$

Variance - Covariance

Covariance

$A \leq B \rightarrow I\langle A\rangle \leq I\langle B\rangle$

- Ordering of types is preserved
- Most common approach

Covariant Array

```
String[] str = new String[1];
Object[] obj = str;
```

Variance - Broken Array Covariance

Runtime Error

```
String[] str = new String[1];
Object[] obj = str;

obj[0] = 2;
```

- safe to read but not safe to write
- not caught at compile time

Variance - Contravariance

Contravariance

$A \leq B \rightarrow I\langle B\rangle \leq I\langle A\rangle$

- Ordering of types is reversed
- Unintuitive but comes with some benefits

Variance - Contravariance

Contravariant comparator

```
void CompareCats(IComparer<Cat> comparer){
    var cat1 = new Cat("Mittens");
    var cat2 = new Cat("Oliver");
    if(comparer.Compare(cat1, cat2) > 0)
        Console.WriteLine("Mittens wins!");
}

IComparator<Animal> compAnimals =
        new AnimalComparator();
CompareCats(compAnimals);
```

- Using a base class instead of a more derived one

Variance - Invariance

Invariance

$A \leq B \rightarrow I\langle A\rangle \nleq I\langle B\rangle \ \wedge \ I\langle B\rangle \nleq I\langle A\rangle$

- Prohibits variant behaviour of complex structures
- Regardless of underlying type hierarchy
- Used to prevent type errors

Variance - Invariance

Invariant list

```
void MammalReadWrite(IList<Mammal> mammals){
    Mammal mammal = mammals[0];
    mammals[0] = new Tiger();
}
```

- Covariance: List of giraffes
  - Put a tiger in it
- Contravariance: List of animals
  - Animals do not need to be mammals

Variance - Bivariance

Bivariance

$I\langle A\rangle \leq I\langle B\rangle \quad , \quad I\langle B\rangle \leq I\langle A\rangle$

- Either impossible or not allowed
- Only listed for the sake of completeness

Instantiation, Subtyping and Subclassing

- Instantiation
  - Creation of a new object
- Subtyping
  - Describes relationship of objects
  - Objects share a common interface
  - 'Is-a' relationship $\rightarrow$ Liskov substitution principle
- Subclassing
  - Does not alter type hierarchy
  - Reuse of code
- Class-based vs Prototype-based

Class-based Programming - Instantiation

- Classes as blueprints
  - Can not change at runtime
  - Easier to optimize compiler tasks
- Implicit/Explicit constructors
  - Creates new instance of a class
  - Allocates memory
  - Initialize all fields

Class-based Programming - Instantiation

Class-based instantiation

```
class NumBox{
        private int number;

        public numBox(int num){
                this.number = num;
        }
        public int getNum(){
                return this.number;
        }
}

NumBox num3 = new NumBox(3);
print(num3.getNum());
```

Class-based Programming - Subtyping

- Type hierarchy has to be explicitly declared

Class-based subtyping

class NumBox { . . . }

class PNatBox **extends** NumBox { . . . }

- No subclassing without subtyping
- Example: Square vs Rectangle

Prototype-based Programming - Instantiation

- No classes
- Constructor functions
  - Explicitly declared

Constructor

```
function ExampleObject(){...}
```

- Ex-nihilo
  - Using object literals

Ex-nihilo

```
var cat = {
    name : "Tardar_Sauce",
    follower : 8403156
}
```

Prototype-based Programming - Subtyping

- Cloning
  - Objects inherit from objects
  - Copy fields into clone
  - Add more specialized fields
- Ex-nihilo
  - Root object

Cloning

```
function ParentClass(){...}
function ChildClass(){...}

ChildClass.prototype = new ParentClass();
```

Prototype-based Programming - Pure Prototyping

- Prototypes and objects can be changed at runtime
- Links between prototype and clones
  - Change in prototype will update clones
- Pure prototyping
  - No delegation but much more memory is used
  - Different versions of same type

Type systems - Overview

- Ensure type safety
- Define equality and compatibility of types
- Can vary widely depending on the language
- Not exclusive to OOP

- Structural typing
- Nominal / nominative typing

Type systems - Structural Typing

- Elements with the same structure are compatible
  - Attributes with their names
  - Functions with their names and parameter/return types
- The name of the type does not matter
- Nor does the place of declaration

Type systems - Structural Typing

- Automatism of type compatibility
- Very flexible and convenient
  - Type hierarchy does not need to be declared beforehand
  - Programmer does not need to maintain the common interfaces himself

Implicit common interface

```
type A = {
    foo();
    bar();
}
```

```
type B = {
    foo();
    baz();
}
```

Type systems - Structural Typing

Problem with compatible types

```
record DistanceInInches{
    double dist;
};

record DistanceInCentimeters{
    double dist;
};
```

- Equivalent in structure
- Different in meaning

Type systems - Nominal Typing

- Subset of structural typing
- Much more type-safe
- No accidental inheritance
- Subtyping has to be explicitly declared

Type systems - Nominal Typing

Nominal subtyping

```
class Animal{
    void feed(){...}
}

class Cat extends Animal{
    int age = 5;
}
```

- Without 'extends' these classes would be completely distinct from each other

Conclusion

- Defining relationship between objects
  - Based on many pieces
  - All come together to make up the specific language
- Flexibility vs Safety
  - Finding the right mix can be difficult
  - Trade-offs are hard to make
- In the end it comes down to personal preference

Thank you for your attention!
QUESTIONS ?