Concepts of Object Oriented Programming
Summary HS 2012

Pascal Spörri
pascal@spoerri.io

February 21, 2013

Part I.
Introduction

1. Requirements

With time programming languages received more and more requirements.

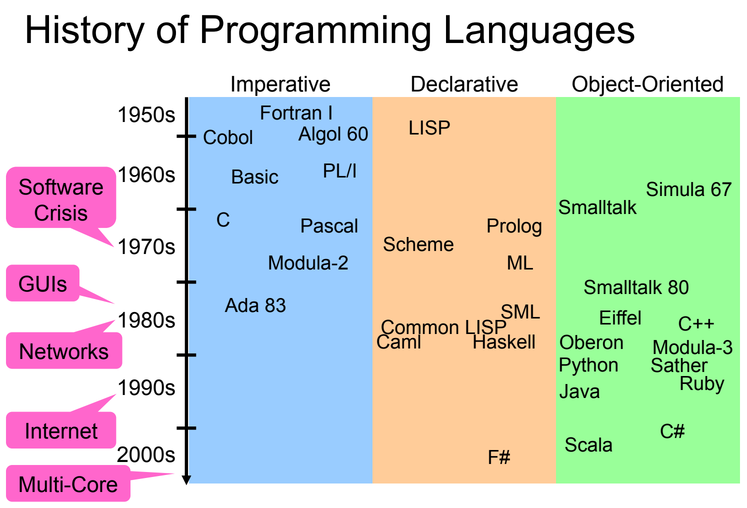


Figure 1: History of Programming Languages

Which forced a rethinking process.

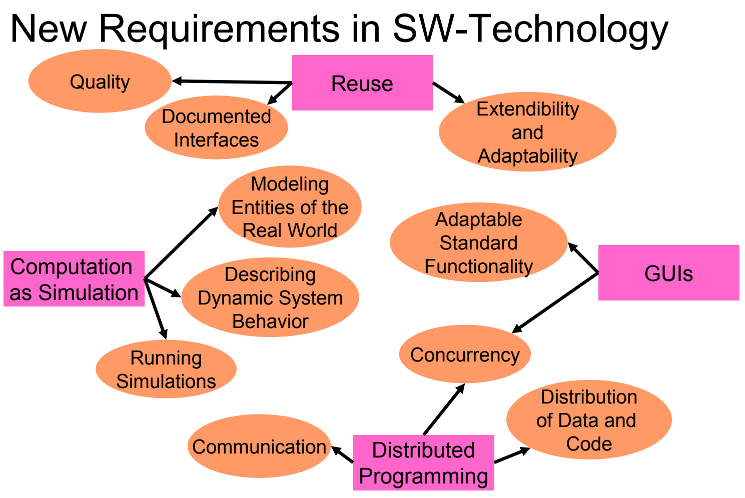

New Requirements in SW-Technology
Reuse
Quality
Documented Interfaces
Extendibility and Adaptability
Computation as Simulation
Modeling Entities of the Real World
Describing Dynamic System Behavior
Running Simulations
Adaptable Standard Functionality
GUIs
Concurrency
Distribution of Data and Code
Communication
Distributed Programming

Figure 2: New Requirements in Software Technology

1.1. Study: Reusing Imperative Programs

We try to model a University Administration System:

- Which models students and professors
- Stores one record for each student and each professor in a repository
- A procedure printAll prints all records in the repository

```c
typedef struct {
  char *name;
  char *room;
  char *institute;
} Professor;

typedef struct {
  char *name;
  int  regnum;
} Student;

void printStudent(Student *s) {...}
void printProf(Professor *p) {...}

typedef struct {
  enum { STU,PROF } kind;
  union {
    Student *s;
    Professor *p;
  } u;
} Person;

typedef Person **List;

void printAll(List l) {
  int i;
  for(i=0; l[ i ] != NULL; i++)
    switch(l[ i ]->kind) {
    case STU:
      printStudent(l[ i ]->u.s);
      break;
    case PROF:
      printProf(l[ i ]->u.p);
      break;
  }
}
```

Listing 1: Implementation in C

Extending the System in order to extend the system with assistants on has to:

- Add a record and print function for the assistants
- Reuse old code for repository and printing

```c
// Student and Professor structs
typedef struct {
  char *name;
  char PhD_student; /* 'y', 'n' */
} Assistant;

// Student and Professor print code
void printAssi(Assistant *a) {...}

// Change the Person struct
typedef struct {
  enum { STU,PROF,ASSI } kind;
  union {
    Student *s;
    Professor *p;
    Assistant *a;
  } u;
} Person;

// Change the printAll function
void printAll(List l) {
  int i;
  for(i=0; l[ i ] != NULL; i++)
    switch(l[ i ]->kind) {
    case STU:
      printStudent(l[ i ]->u.s);
      break;
    case PROF:
      printProf(l[ i ]->u.p);
      break;
    case ASSI:
      printAssi(l[ i ]->u.a);
      break;
  }
}
```

Listing 2: Extending the system

1.2. Reuse in Imperative Languages

- Imperative languages don't have an explicit language support for extension and adaption
- Adaption usually requires modification reused code
- Code adaption requires **Copy-and-paste reuse** which leads to
  - Code duplication
  - Is difficult to maintain
  - Is *Error-prone*

1.3. Core Requirements

All of this leads to these core requirements for object oriented programming languages:

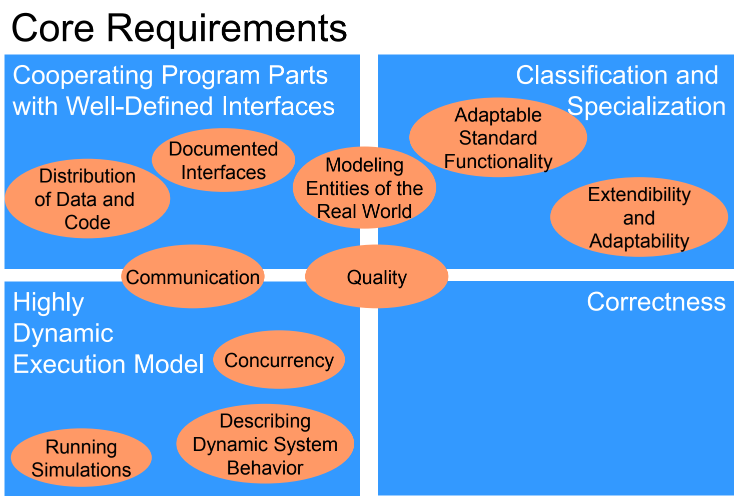

Core Requirements
Cooperating Program Parts with Well-Defined Interfaces
Classification and Specialization
Adaptable Standard Functionality
Documented Interfaces
Distribution of Data and Code
Modeling Entities of the Real World
Extendibility and Adaptability
Communication
Quality
Highly Dynamic Execution Model
Correctness
Concurrency
Running Simulations
Describing Dynamic System Behavior

Figure 3: Core Requirements in Software Technology

2. Core Concepts

> The basic philosophy underlying object-oriented programming is to make the programs as far as possible reflect that part of the reality they are going to treat. It is then often easier to understand and to get an overview of what is described in programs. The reason is that human beings from the outset are used to and trained in the perception of what is going on in the real world. The closer it is possible to use this way of thinking in programming, the easier it is to write and understand programs.
>
> *Object-oriented Programming in the BETA Programming Language*

2.1. The Object Model

- A software system is a set of cooperating object
- Objects have state and processing ability
- Objects exchange messages

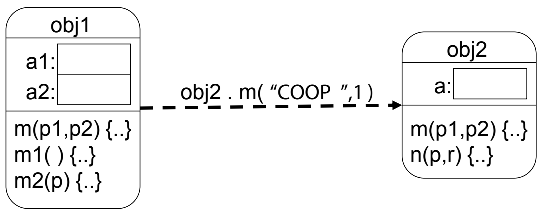


Figure 4: The object model

2.1.1. Characteristics of Objects

- State
- Identity
- Lifecycle
- Location
- Behavior

Compared to imperative programming,

- Objects lead to a *different program structure*
- Objects lead to a *different execution model*

2.2. Interfaces and Encapsulation

- Objects have **well-defined interfaces**
  - Publicly accessible fields
  - Publicly accessible methods
- Implementation is hidden behind interface
  - Encapsulation
  - Information hiding
- Interfaces are the basis for **describing behavior**

2.3. Classification and Polymorphism

- **Classification**: Is a hierarchical structuring of objects
- Objects belong to different classes simultaneously
- **Substitution principle**: Subtype objects can be used wherever supertype objects are expected.

Definition (Classification)**.** *Classifying is a general technique to hierarchically structure knowledge about concepts, items, and their properties.*
The result is called classification.

Characteristics of Classifications We can classify objects or fields:

- Classifications can be *trees* or *DAGs*
- Classifications of objects form *"is-a" relation*.
- Classes can be *abstract* or *concrete*.

Definition (Substitution principle)**.** *Objects of subtypes can be used wherever objects are expected.*

2.4. Polymorphism

> "The quality of being able to assume different forms.
> *Merriam-Webster Dictionary*"

Definition. *A program part is polymorphic if it can be used for objects of several types.*

Subtype Polymorphism is a direct consequence of the substitution principle.

- Program parts working with supertype objects work as well with subtype objects
- Example: `printAll` can print objects of class Person, Student, Professor, etc.

2.5. Other forms of polymorphism

Parametric Polymorphism Generic types

- Uses *type parameters*
- One implementation can be used for different types
- *Type mismatches can be detected at compile time*

```
class List<G> {
  G[ ] elems;
  void append(G p) { ... }
}

List<String> myList;
myList = new List<String>();
myList.append("String");
```

Ad-hoc Polymorphism Method overloading

- Allows several methods with the *same name but different arguments*
- Also called *overloading*

- No semantic concept: Can be modeled by *renaming*

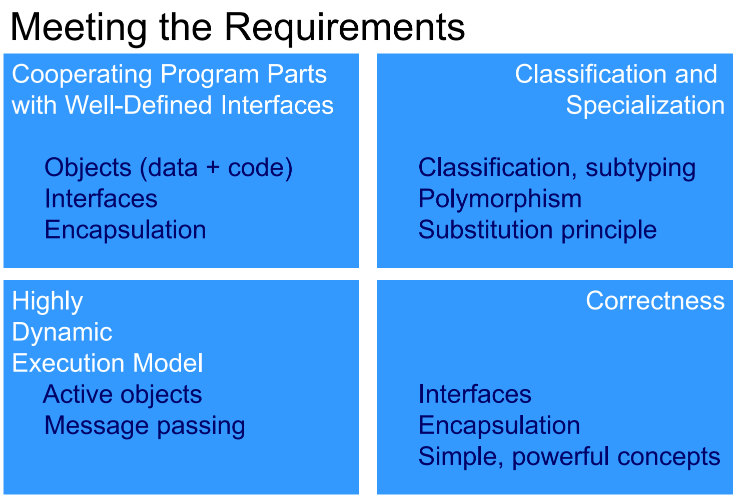
```
class Any {
  void foo(Polar p) { ... }
  void foo(Coord c) { ... }
}
x.foo(new Coord(5, 10));
x.foo(new Polar(5, 10));
```

2.6. Specialization

Definition. *Adding specific properties to an object or refining a concept by adding further characteristics.*

- Start from general objects or types
- Extend these objects and their implementations(add properties)
- Requirement: Behavior of specialized objects is compliant to behavior of more general objects
- Program parts that work for the more general objects work as well for specialized objects
- Implementation inheritance, reuse.

2.7. Meeting the Requirements



Meeting the Requirements

Cooperating Program Parts with Well-Defined Interfaces
- Objects (data + code)
- Interfaces
- Encapsulation

Classification and Specialization
- Classification, subtyping
- Polymorphism
- Substitution principle

Highly Dynamic Execution Model
- Active objects
- Message passing

Correctness
- Interfaces
- Encapsulation
- Simple, powerful concepts

Figure 5: Meeting the Requirements

2.8. Summary

Core concepts of the OO-paradigm:

- Object model
- Interfaces and encapsulation
- Classification and polymorphism

Core concepts are *abstract concepts* to meet the new requirements. To apply the core concepts we need ways to *express them in programs*. *Language concepts* enable and facilitate the application of the core concepts.

3. Language Concepts

Why use an OO-language when we can describe a program in an imperative language?

Example Previous Problem in C.

- Type declaration:

```
typedef char* String;
typedef struct sPerson Person;
```

- Record declaration with fields and methods (function pointers)

```
struct sPerson {
  String name;
  void (*print)(Person*);
  String (*lastName)(Person*);
};
```

- Method definitions:

```
void printPerson(Person *this) {
  printf("Name: %s\n", this->name);
}
String LN_Person(Person *this)
  {...}
```

- Constructor definitions:

```
Person *PersonC(String n) {
  Person *this    = (Person *) malloc(sizeof(Person));
  this->name      = n;
  this->print     = printPerson;
  this->lastName  = LN_Person;
  return this;
}
```

- Using constructors, fields, and methods

```
Person *p;
p = PersonC("Tony Hoare");
p->name = p->lastName(p);
p->print(p);
```

- Inheritance
  - Copy code
  - Adapt function signatures
  - Define Specialized Methods
  - Reuse Person for Student
  - View Student as Person (cast)

```
typedef struct sStudent Student;
struct sStudent {
  String name;
  void (*print)(Student*);
  String (*lastName)(Student*);
  int regNum;
};
void printStudent(Student *this) {
  printf("Name: %s\n", this->name);
  printf("No: %d\n", this->regNum);
}
Student *StudentC(String n, int r) {
  Student *this  = (Student *) malloc(sizeof(Student));
  this->name     = n
  this->print    = printStudent;
  this->lastName = (String (*)(Student*)) LN_Person;
  this->regNum   = r;
  return this;
}
```

- Subclassing and Dynamic Binding
  - Student has all fields and methods of Person
  - Casts are necessary

```
  Student *s;
  Person *p;
  s = StudentC("Susan Roberts", 0);
  p = (Person *) s;
  p->name = p->lastName(p);
  p->print(p);
```

- Methods are selected dynamically

```
  void printAll( Person **l ) {
    int i;
    for (i=0; l[i] != NULL; i++)
      l[i]->print(l[i]);
    }
  }
```

3.0.1. Discussion of the C Solution

Pros We can express *objects*, *fields*, *methods*, *constructors*, and *dynamic method binding* by imitating OO-programming. The union in Person and the switch statement in `printAll` (previous example) became dispensable. The behavior of reused code (`Person`, `printAll`) can be *adapted* to introduce Student *without changing the implementation*.

Cons Inheritance has to be replaced by *code duplication*. Subtyping can be simulated, but it requires:

- Casts, which are *not type safe*
- *Same memory layout* of super and subclasses (same fields and function pointers in same order), which is *extremely error-prone*.

Appropriate language support is needed to apply object-oriented concepts.

3.0.2. A Java Solution

```
class Person {
  String name;
  void print() {
    System.out.println("Name: " + name);
  }
  String lastName() {...}
  Person(String n) {name = n}
}
class Student extends Person {
  int regNum;
  void print() {
    super.print();
    System.out.println("No: "+ regNum);
  }
  Student(String n, int i ) {
    super(n);
    regNum = i;
  }
}
void printAll(Person[] l) {
  for (int i=0; l[i] != null; i++)
    l[i].print();
}
```

The Java solution uses

Inheritance to avoid code duplication.

Subtyping to express classification.

Overriding to specialize methods.

Dynamic Binding to adapt reused algorithms.

Java supports the OO-language concepts and the Java solution is

- Simpler and smaller

- Easier to maintain (no duplicate code)
- Type safe

3.0.3. Concepts Summary

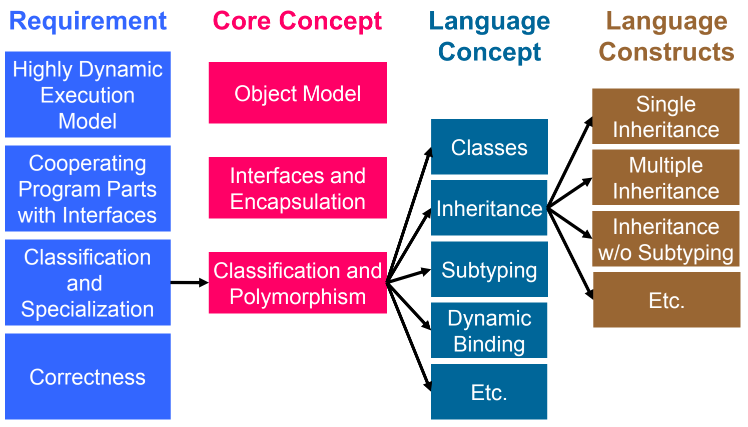

Requirement
Core Concept
Language Concept
Language Constructs
Highly Dynamic Execution Model
Cooperating Program Parts with Interfaces
Classification and Specialization
Correctness
Object Model
Interfaces and Encapsulation
Classification and Polymorphism
Classes
Inheritance
Subtyping
Dynamic Binding
Etc.
Single Inheritance
Multiple Inheritance
Inheritance w/o Subtyping
Etc.

Figure 6: Concepts Summary

4. Language Design

A good language should resolve design trade-offs in a way *suitable for its application domain*.

Design Goals

Simplicty

- Syntax and semantics can be easily understood by users and implementers of the language.
- This doesn't mean that the number of constructs should be limited.
- Simple languages: BASIC, Pascal, C.
- It is not know whether the Java 5 type system (generics) is deciable.

Expressiveness

- Language can (easily) express complex processes and structures
- Expressive languages: C#, Scala, Python
- Expressiveness is often conflicting with simplicity:

(Static) Safety

- Language discourages errors and allows errors to be discovered and reported. Ideally at compile time.

- Safe languages: Java, C#, Scala.
- Often conflicting with expressiveness and performance.

Modularity

- Language allows modules to be compiled separately
- Modula languages: Java, C Scala
- Often conflicting with expressiveness and performance.

Performance

- Programs written in the language can be executed efficiently
- Efficient languages: C, C++, Fortran.
- Often conflicting with safety and productivity:
  - C arrays:
    * Sequence of memory locations
    * Access is a simple look-up with only 2-5 machine instructions
  - Java arrays
    * Sequence of memory locations *plus length*
    * Access is look-up *plus bound-check*

Productivity

- Language leads to low costs of writing programs
- Closely related to expressiveness
- Languages for high productivity:
  - Visual Basic
  - Python
- Often conflicting with static safety

Backwards Compatibility

- Newer language version work and interface with programs in older version
- Backwards compatible languages:
  - Java

```java
class Tuple<T> {
  T first; T second;
  void set(T first, T second) {
    this.first = first;
    this.second = second;
  }
}
class Client {
  static void main(String[] args) {
    Tuple t = new Tuple();
```

```
        t.set("Hello", new Client());
      }
    }
```

  - C

- Often in conflict with simplicity and performance.

Part II.
Types and Subtyping

5. Types

5.1. Type Systems

Definition (Type System)**.** *A type system is a tractable syntactic method for proving absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*

Syntactic Rules are based on form, not behavior

Phrases Expressions, methods, etc... of a program.

Kinds of values Types

5.2. Weak and Strong Type Systems

Untyped Languages Do not classify values into types.
Example: Assembler

Weakly-typed Languages Classify values into types, but do not srictly enforce additional restrictions.
Examples: C, C++

Strongly-typed Languages Enforce that all operations are applied to arguments of the appropriate types.
Examples: C#, Eiffel, Java, Python, Scala, Smalltalk

Strongly-typed languages prevent certain erroneous or undesirable program behavior Comparison between strong and weak typing:

```
int main( int argc, char** argv ) {
  int i = ( int ) argv[ 0 ];
  printf( "%d", i );
}
```

Listing 3: C Example

```
int main( String[ ] argv ) {
  int i = ( int ) argv[ 0 ];
  System.out.println( i );
}
```

Listing 4: Java Example

The Java example results in a Compile-time error.

5.3. Types

Definition (Types). *A type is a set of values sharing some properties. A value v has type T if v is an element of type.*

Nominal Types are based on *type names*
Examples: C++, Eiffel, Java, Scala

Structural Types are based on *availability of methods and fields*
Examples: Python, Ruby, Smalltalk

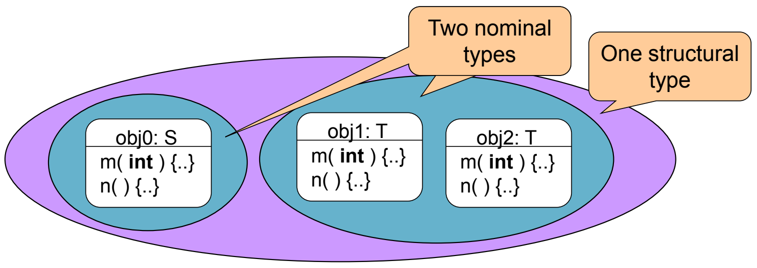


Figure 7: Type Membership

5.3.1. Example

```
class S {
  m( int ) { ... }
  n( )     { ... }
}

class T {
  m( int ) { ... }
  n( )     { ... }
}
```

- **S** and **T** are ***different*** *in nominal systems*
- **S** and **T** are ***equivalent*** *in structural systems*

5.4. Static Type Checking

Each expression of a program *has a type*. *Types* of variables and methods are *declared explicitly* or *inferred*. Types of expressions can be derived from the types of their constituents. *Type rules* are used *at compile time* to check whether a program is *correctly typed*.

Examples:

```
int a;
boolean equals(Object o)
a + 7
```

```
"A number: " + 7
"A string".equals(null)
```

Listing 5: compiles

```
a = "A string";            // Assignment of a String to an integer
"A string".equals(1, 2);   // Too many arguments
```

Listing 6: results in compile-time errors

5.5. Dynamic Type Checking

Variables, *methods*, and *expressions* of a program *are* typically *not typed*. Every object and value has a type. A *Run-time system* checks that operations are applied to *expected arguments*.

5.6. Static Type Safety

Definition (Static Type Safety). *A programming language is called type-safe if its design prevents type errors.*

> “ In every execution state, the type of the value held by variable v is a subtype of the declared type of v. ”

Most static type systems rely on dynamic checks for certain operations. Such as *type conversion by casts*. *Run-time checks* throw an exception in case of a type error.

```
Object[] oa = new Object[10];
String s    = "A String";
oa[0]       = s;
// ..
if (oa[0] instanceof String) {
  s = (String) oa[0];
}
s = s.concat("Another String");
```

Listing 7: Java example on a dynamic check

Static checkers need to *approximate run-time behavior* (conservative check).

```
def divide(n,d):
  if d != 0:
    res = n/d
  else:
    res = "Division by zero"
  print res
```

Listing 8: Python example on dynamic type checking

Dynamic checkers support *on-the-fly code generation* and dynamic class loading.

```
eval (
  "x=10; y=20; document.write(x*y);"
);
```

Listing 9: JavaScript example on on-the-fly code generation

5.7. Comparison

5.8. Advantages of static checking

Static safety: More errors are found at compile time.

Readability: Types are excellent documentation.

Efficiency: Type information allows optimizations.

5.9. Advantages of dynamic checking

Expressiveness: No correct program is reject by the type checker.

Low overhead: No need to write type annotations.

Simplicity: Static type systems are often complicated.

	Static	**Dynamic**
Nominal	C++, C#, Eiffel, Java, Scala	For certain features of statically-typed languages.
Structural	Research languages such as Moby, PolyToil, O'Caml.	JavaScript, Python, Ruby, Smalltalk.

Dynamically typed languages with a structural types are also called *duck typed languages*.

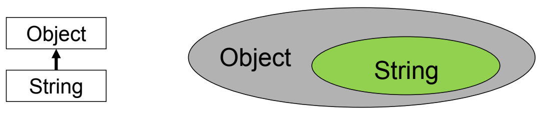
6. Subtyping

“Objects of subtypes can be used wherever objects of supertypes are expected.”
Substitution principle

Syntactic classification Subtype objects can *understand at least the messages* that supertype objects can understand.

Semantic classification Subtype objects *provide at least the behavior* of supertype objects.

The *subtype relation* corresponds to the *subset relation* on the values of a type.



Nominal type systems Determine *type membership* based on *type names* and determine *subtype relations* based on *explicit declarations*:

```
class S {
  m(int) {...}
}
class T extends S { // nominal subtype because of extend declaration
  m(int) {...}
}
class U {           // not a subtype of S
  m(int) {...}
  n()    {...}
}
```

Listing 10: nominally typed language

Structural type systems Determine *type membership* and *subtype relations* based on *availability of methods and fields*:

```
class S {
  m(int) {...]
}
class T {          // Subtype of S
  m(int) {...}
}
class U {          // Subtype of U
  m(int) {...}
  n()    {...}
}
```

Listing 11: structurally typed language

6.1. Nominal Subtyping and Substitution

Subtype objects can *understand at least the messages* that supertype objects can understand:

- Method calls
- Field accesses

Subtype objects have *wider interfaces* than supertype objects:

- Existence of methods and fields
- Accessibility of methods and fields
- Types of methods and fields

6.1.1. Rules

Existence *Subtypes may add, but not remove methods and fields*

Accessibility An *overriding method must not be less accessible* than the methods it overrides:

```
class Super {
  public void foo() {...}
  public void bar() {...}
}
```

```
class Sub <: Super {
  public void foo() {...}
  private void bar() {...}
}

void m( Super s) { s.bar(); }
```

At run time, `m` could access a private method of `Sub`, thereby violating information hiding.

Contravariant parameters An *overriding method must not require* ***more specific*** *parameter types* than the method it overrides.

Covariant results An *overriding method must not have* ***more genera*** *result type* than then the ethod it overrides.

Fields *Subtypes must not change the type of fields* Regard field as pair of getter and setter methods:

- *Specializing a field type* `S<:T` corresponds to specializing the argument of the setter (*violates the contravariant parameters*)
- *Generalizing a field type* `T<:S` corresponds to generalizing the result of the getter (*violates covariant results*).

Immutable Fields Immutable fields do not have setters. *Types of immutable fields can be specialized in subclasses* `S<:T`. This only works in the absence of inheritance.

Covariant Arrays In Java and C# *arrays are covariant*. If `S<:T` then `S[] <: T[]`. Each *array update requires* a *run-time check*. Covariant arrays allow one to write methods that work for all arrays such as:

```
Class Arrays {
    public static void fill(Object[] a, Object val) { ... }
}
```

Generics allow a solution that is expressive and statically-safe.

6.2. Reuse: Adapter Pattern

Nominal Subtyping can impede reuse. Consider two library classes:

```
class Resident {
    String  getName( ) { ... }
    Data    dateOfBirth( ) { ... }
    Address getAddress( ) { ... }
}
class Employee {
    String  getName( ) { ... }
    Data    dateOfBirth( ) { ... }
    int     getSalary( ) { ... }
}
```

Now we would like to store Resident and Employee objects in a collection of type Person[]. Implement an Adapter:

- Subtype of Person
- Delegate calls to adaptee
- Adapter requires boilerplate code
- Adapter causes memorz and run-time overhead
- Works also if Person is reused.

```
interface Person {
    String  getName();
    Data    dateOfBirth();
}
class EmployeeAdapter implements Person {
    private Employee adaptee;
    String  getName() {
        return adaptee.getName();
    }
    Data dateOfBirth() {
        return adaptee.dateOfBirth();
    }
}
```

Listing 12: Adapter Pattern

6.3. Generalisation

Most OO-languages support specialisation of superclasses (top-down development). Some research languages also support *generalisation* (bottom-up development).

```
interface Person generalizes Resident, Employee {
    String  getName();
    Data    dateOfBirth();
}
```

A supertype can be declared after a subtypes have been implemented. Generalisation does not match well with inheritance: A Subclass-to-be already has a superclass.

6.4. Reuse: Structural Subtyping

Subtype objects can *understand at least the messages* that supertype objects can understand:

- Method calls
- Field accesses

Structural subtypes have *by definition wider interfaces* then their supertypes.

All types are "automatically" subtypes of types with smaller interfaces. No support for inheritance (like generalization).

```
interface Person {
    String  getName();
    Data    dateOfBirth();
}

class Resident {
    String  getName()       {...}
    Data    dateOfBirth()   {...}
    ...
}

class Employee {
    String  getName()       {...}
    Data    dateOfBirth()   {...}
    ...
}
```

Listing 13: Structural Subtyping Example

Person is a supertype of Resident and Employee. We observe that generalisation doesn't match well with inheritance.

6.5. Shortcomings

Nominal subtyping can impede code reuse. Consider these two library classes:

```
class Resident {
    String  getName()       { ... }
    Date    dateOfBirth()   { ... }
    Address getAddress()    { ... }
}

class Employee {
    String  getName()       { ... }
    Date    dateOfBirth()   { ... }
    int     getSalary()     { ... }
}
```

Now we would like to store Resident and Employee-objects in a collection of type Person[]. Neither Resident nor Employee is a subtype of Person. So we implement and Adapter:

- Subtype of Person
- Delegate calls to adaptee (Resident or Employee)

```
interface Person {
    String  getName()       { ... }
    Date    dateOfBirth()   { ... }
}
class EmployeeAdapter implements Person {
    private Employee    adaptee;
    String  getName() {
        return adaptee.getName();
    }
    Data    dateOfBirth() {
        return adaptee.dateOfBirth();
    }
}
```

The adapter requires boilerplate code and causes memory and run-time overhead. The adapter works also if Person is reused.