

Concepts of Object-Oriented Programming

Peter Müller

Chair of Programming Methodology

Autumn Semester 2011



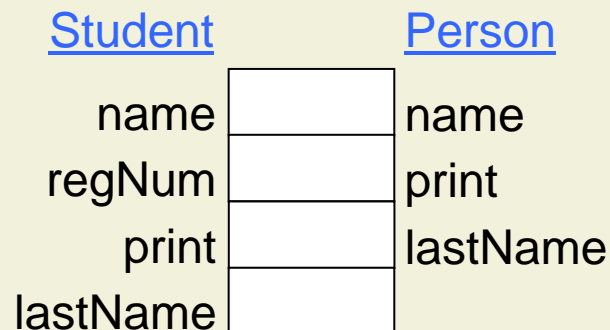
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

C-Example Revisited

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

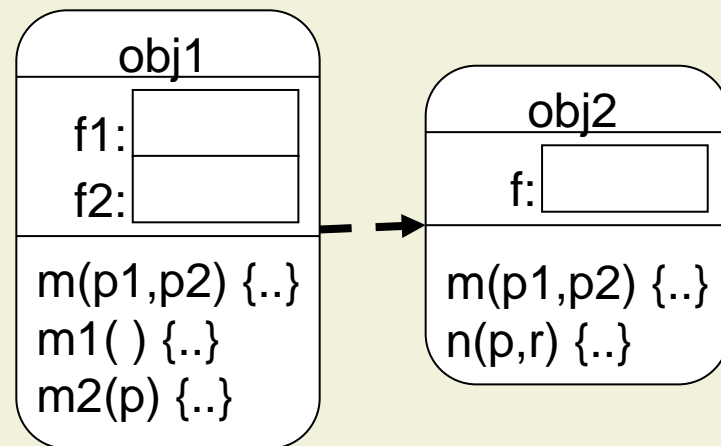
```
typedef struct sStudent Student;
struct sStudent {
    String name;
    int regNum;
    void  ( *print )( Student* );
    String ( *lastName )( Student* );
};
```

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



Message not Understood

- Objects access fields and methods of other objects
- A safe language **detects situations** where the receiver object does not have the accessed field or method
- Type systems** can be used to **detect such errors**

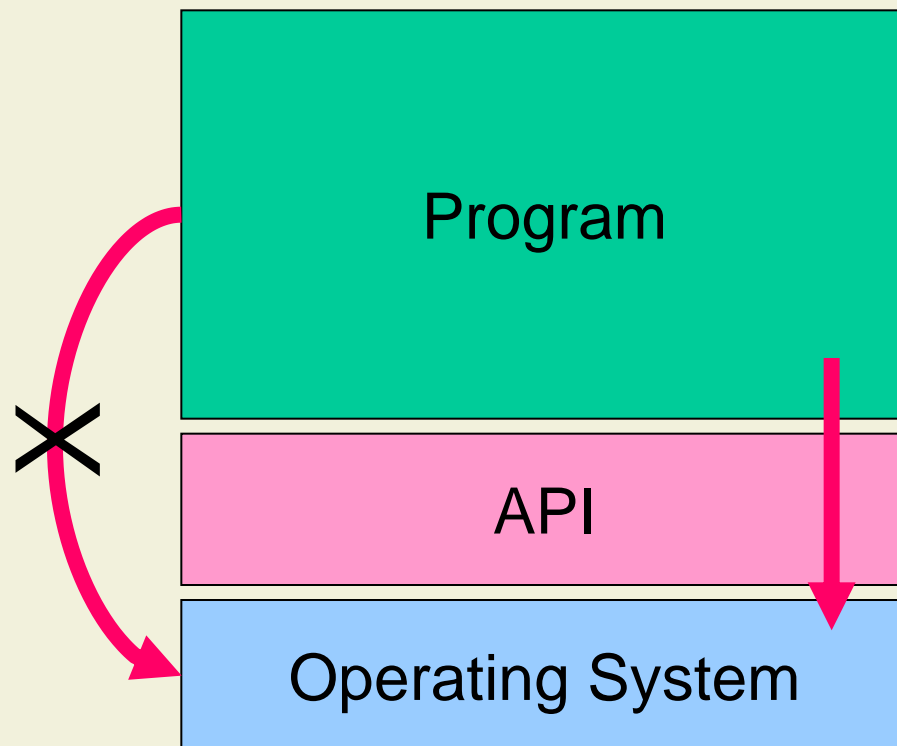


```
...  
r = obj2.m( 0, 1 );  
s = obj2.f;
```

```
r = obj2.m( );  
r = obj2.anotherMethod( 0, 1 );  
s = obj2.anotherField;
```

Java Security Model (Sandbox)

- Applets get access to system resources **only through an API**
- Access control can be implemented in API (security manager)
- **Code must be prevented from by-passing API**



2. Types and Subtyping

2.1 Types

2.2 Subtyping

2.3 Behavioral Subtyping

Type Systems

- Definition:

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.

[B.C. Pierce, 2002]

- *Syntactic*: Rules are based on form, not behavior
- *Phrases*: Expressions, methods, etc. of a program
- *Kinds of values*: Types

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 strictly enforce additional restrictions
 - Example: C, C++
- Strongly-typed languages
 - Enforce that all operations are applied to arguments of the appropriate types
 - Examples: C#, Eiffel, Java, Python, Scala, Smalltalk

Weak vs. Strong Typing: Example

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

C

1628878672

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

Java

Compile-time error:

inconvertible types

found : java.lang.String

required: int

- Strongly-typed languages prevent certain erroneous or undesirable program behavior

Types

- Definition:

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

- Question: what are the “*properties*” shared by the values of a type?

- Nominal types:

based on **type names**

Examples: C++, Eiffel, Java, Scala

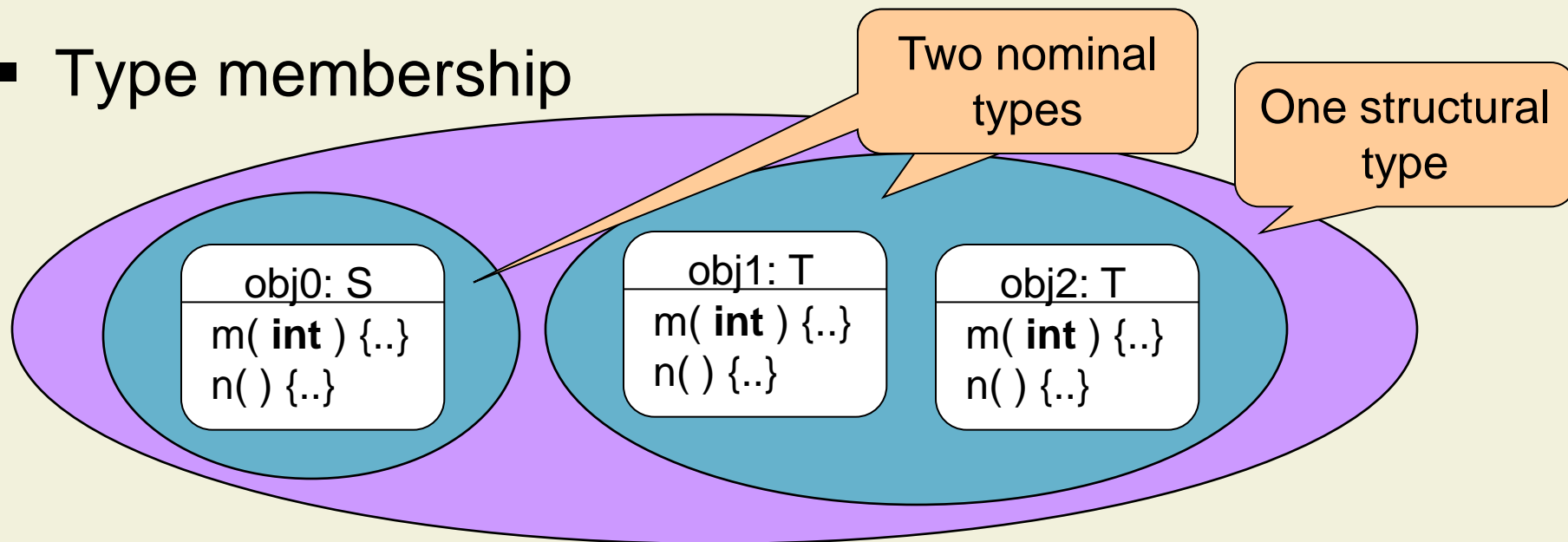
- Structural types:

based on **availability of methods and fields**

Examples: Python, Ruby, Smalltalk

Nominal and Structural Types

■ Type membership



■ Type equivalence

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

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

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

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

“A string”

Java

5 + 7

int a;

Java

boolean equals(Object o)

a + 7

Java

“A number: “ + 7

“A string”.equals(**null**)

a = “A string”;

Java

“A string”.equals(1, 2)

Compile-time
errors

DynamicType Checking

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

“A string”

Python

5 + 7

a = ...;

Python

def foo(o): ...

a + 7

Python

“A number: “ * 7

foo(**None**)

a = “A string”

Python

a = 7

a = “A string” / 5

Python

foo(5, 7)

Run-time
errors

Static Type Safety

- Definition:

A programming language is called type-safe if its design prevents type errors.

- Statically type-safe object-oriented languages guarantee the following type invariant:

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

- Type safety guarantees the absence of certain run-time errors

Run-Time Checks in Static Type Systems

- Most static type systems rely on dynamic checks for certain operations
- Common example: **type conversions 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" );
```

Expressiveness of Dynamic Type Systems

- Static checkers need to **approximate run-time behavior** (conservative checks)
- Dynamic checkers support **on-the-fly code generation** and dynamic class loading

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

Python

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

JavaScript

Static vs. Dynamic Type Checking

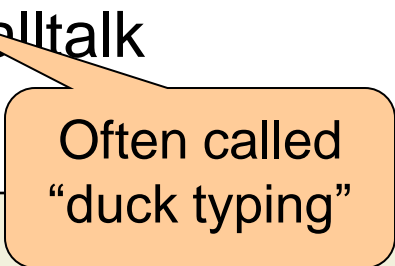
Advantages of static checking

- **Static safety**: More errors are found at compile time
- **Readability**: Types are excellent documentation
- **Efficiency**: Type information allows optimizations

Advantages of dynamic checking

- **Expressiveness**: No correct program is rejected by the type checker
- **Low overhead**: No need to write type annotations
- **Simplicity**: Static type systems are often complicated

Type Systems in OO-Languages

	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  Often called “duck typing”

2. Types and Subtyping

2.1 Types

2.2 Subtyping

2.3 Behavioral Subtyping

Classification in Software Technology

- Substitution principle

Objects of subtypes can be used wherever objects of supertypes are expected

- 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

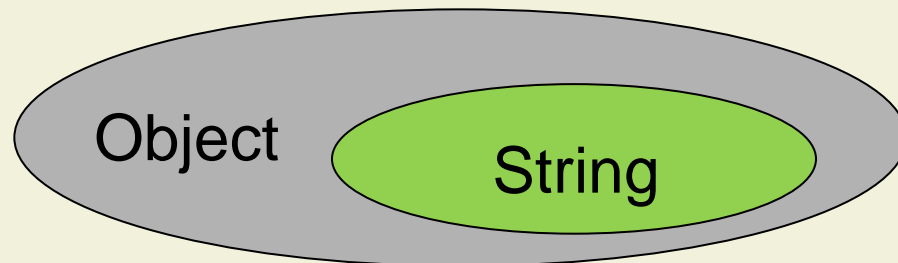
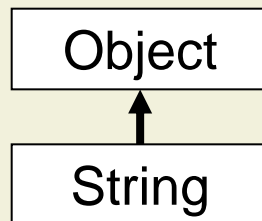
Subtyping

- Definition of “Type”:

A type is a set of values sharing some properties.

A value v has type T if v is an element of T .

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



Nominal and Structural Subtyping

- Nominal type systems
 - Determine type membership based on type names
 - Determine **subtype relations based on explicit declarations**
- Structural type systems
 - Determine type membership and **subtype relations based on availability of methods and fields**

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

```
class T  
extends S {  
  m( int ) { ... }  
}
```

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

Only T is a nominal
subtype of S

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

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

T and U are structural
subtypes of S

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

Existence

```
class Super {  
    void foo( ) { ... }  
    void bar( ) { ... }  
}  
  
class Sub <: Super {  
    void foo( ) { ... }  
    // no bar( )  
}
```

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

- Sub narrows Super's interface
- If m is called with a Sub object as parameter, execution fails
- Subtypes may add, but not remove methods and fields

Accessibility

```
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 private method of Sub, thereby violating information hiding
- An **overriding method must not be less accessible** than the methods it overrides

Overriding: Parameter Types

```
class Super {  
  void foo( String s ) { ... }  
  void bar( Object o ) { ... }  
}
```

```
class Sub <: Super {  
  void foo( Object s ) { ... }  
  void bar( String o ) { ... }  
}
```

```
void m( Super s ) {  
  s.foo( "Hello" );  
  s.bar( new Object( ) );  
}
```

- Calling m with a Sub object demonstrates a violation of static type safety
 - o in Sub.bar is not a String
- **Contravariant parameters:**
An overriding method must not require more specific parameter types than the methods it overrides

Overriding: Result Types

```
class Super {  
  Object foo( ) { ... }  
  String bar( ) { ... }  
}  
  
class Sub <: Super {  
  String foo( ) { ... }  
  Object bar( ) { ... }  
}
```

```
void m( Super s ) {  
  Object o = s.foo( );  
  String t = s.bar( );  
}
```

- Calling m with a Sub object demonstrates a violation of static type safety
 - t in m is not a String
- **Covariant results:**
An overriding method must not have a more general result type than the methods it overrides
 - Out-parameters and exceptions are results

Overriding: Fields

```
class Super {  
  Object f;  
  String g;  
}  
  
class Sub <: Super {  
  String f;  
  Object g;  
}
```

```
void m( Super s ) {  
  s.f = new Object( );  
  String t = s.g;  
}
```

- Calling m with a Sub object demonstrates a violation of static type safety
 - s.f is not a String
 - t is not a String
- Subtypes must not change the types of fields
 - Fields are bound statically

Overriding: Fields (cont'd)

```
class Super {  
  T f;  
  void setF( T f ) { this.f = f; }  
  T getF( ) { return f; }  
}  
  
class Sub <: Super {  
  S f;  
  void setF( S f ) { this.f = f; }  
  S getF( ) { return f; }  
}
```

- Regard field as pair of getter and setter methods
 - Specializing a field type
($S <: T$) corresponds to specializing the argument of the setter (**violates contravariant parameters**)
 - Generalizing a field type
($T <: S$) corresponds to generalizing the result of the getter (**violates covariant results**)

Overriding: Immutable Fields

```
class Super {  
  final T f;  
  void setF(T f) { this.f = f; }  
  T getF() { return f; }  
}
```

```
class Sub <: Super {  
  final S f;  
  void setF(S f) { this.f = f; }  
  S getF() { return f; }  
}
```

- Immutable fields do not have setters
- Types of immutable fields can be specialized in subclasses ($S <: T$)
 - Works only in the absence of inheritance (supertype constructor initializes f with a T -value)!
- Not permitted by mainstream languages

Narrowing Interfaces in Eiffel

- Eiffel permits the “illegal” narrowing of interfaces
 - Changing the existence of methods
 - Overriding with covariant parameter types
 - Specializing field types
- Run-time exception
“catcall detected for argument #1 'o' expected STRING but got ANY”

```
class SUPER
feature
  bar ( o: ANY ) do ... end
end

class SUB inherit SUPER
  redefine bar end
feature
  bar ( o: STRING ) do ... end
end
```

```
m ( s: SUPER )
do
  s.bar ( create {ANY} )
end
```

Narrowing Interfaces in Eiffel (cont'd)

- With attached (non-null) types, covariant overriding **requires a detachable (possibly-null) type**
- Run-time system **passes null** when an argument is not of the expected type
- Method must **check for null-ness explicitly**

```
class SUPER
feature
  bar ( o: ANY ) do ... end
end

class SUB inherit SUPER
  redefine bar end
feature
  bar ( o: ?STRING )
  do
    if { o: STRING } s then s.foo;
    else ... end
  end
end
```

Covariant Arrays

```
class C {  
  void foo( Object[ ] a ) {  
    if( a.length > 0 )  
      a[ 0 ] = new Object( );  
  }  
}
```

```
void client( C c ) {  
  c.foo( new String[ 5 ] );  
}
```

- In Java and C#, **arrays are covariant**
 - If $S \leq T$ then $S[] \leq T[]$

```
class Object[ ] {  
  
  public Object 0;  
  public Object 1;  
  ...  
}
```

```
class String[ ]  
  <: Object[ ] {  
  public String 0;  
  public String 1;  
  ...  
}
```

- Each **array update requires a run-time type check**

Covariant Arrays (cont'd)

- Covariant arrays allow one to write methods that work for all arrays such as

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

- Here, the designers of Java and C# **resolved** the **trade-off** between expressiveness and static safety **in favor of expressiveness**
- Generics allow a solution that is expressive and statically-safe (more later)

Shortcomings of Nominal Subtyping (1)

- 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[]
 - Neither Resident nor Employee is a subtype of Person

Reuse: Adapter Pattern

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

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

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

- Adapter requires boilerplate code
- Adapter causes memory and run-time overhead
- Works also if Person is reused

Reuse: Generalization

- Most OO-languages support specialization of superclasses (top-down development)
- Some research languages (e.g., Sather, Cecil) also support **generalization** (bottom-up development)

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

- Supertype can be declared after subtype has been implemented

Reuse: Generalization (cont'd)

- Generalization does not match well with inheritance
- Subclass-to-be already has a superclass
 - Single inheritance: exchanging the superclass might affect the subclass
 - Multiple inheritance: additional superclass may cause conflicts

```
class Cell {  
    int value;  
    int getData( ) { return value; }  
}
```

```
abstract class DataPoint  
    generalizes Cell {  
    abstract int getData( );  
    boolean equals( Object o ) {  
        ... // check type of o  
        return getData( ) ==  
            ( (DataPoint) o ).getData( );  
    }  
}
```

Shortcomings of Nominal Subtyping (2)

- Nominal subtyping can limit generality
- Many method signatures are overly restrictive

```
void printData( Collection<String> c ) {  
    if( c.isEmpty() ) System.out.println( "empty" );  
    else {  
        Iterator<String> iter = c.iterator( );  
        while( iter.hasNext() ) System.out.println( iter.next() );  
    }  
}
```

- printData uses only two methods of c, but requires a type with 13 methods

Generality: Additional Supertypes

- Make type requirements weaker by declaring interfaces for useful supertypes
- But: many useful subsets of operations
 - Read-only collection
 - Write-only collection (log file)
 - Convertible collection
 - Combinations of the above
- Overhead for declaring supertypes and subtyping

```
interface Iterable<E> {  
    Iterator<E> iterator( );  
}
```

```
interface Collection<E>  
    extends Iterable<E> {  
    // 13 methods  
}
```

Generality: Optional Methods

- Java documentation marks some methods as “optional”
 - Implementation is allowed to throw an unchecked exception
 - For Collection: all mutating methods
- Static safety is lost

```
interface Collection<E>  
    extends Iterable<E> {  
    /* 13 methods, out of which 6 are  
       optional */  
}
```

```
class AbstractCollection<E>  
    implements Collection<E> {  
    boolean add( E e ) {  
        throw new  
            UnsupportedOperationException( );  
    }  
    ...  
}
```


Structural Subtyping and Substitution

- Subtype objects can **understand at least the messages** that supertype objects can understand
 - Method calls
 - Field accesses
- Structural subtypes have **by definition wider interfaces** than their supertypes

Reuse: Structural Subtyping

- All types are “automatically” subtypes of types with smaller interfaces
 - No extra code or declarations required
- No support for inheritance (like generalization)
- Person is a supertype of Resident and Employee

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

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

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

Generality: Structural Subtyping

```
void printData( Collection<String> c ) {  
    // uses only c.isEmpty() and c.iterator()  
}
```

- Static type checking
 - Additional supertypes approach applies
 - Additional supertypes must be declared, but not the subtype relation
- Dynamic type checking
 - Arguments to operations are not restricted
 - Similar to optional methods approach (possible run-time error)

Type Systems in OO-Languages

		Static	Dynamic
Structural	Nominal	<p>Sweetspot: Maximum static safety</p>	<p>Why should one declare all the type information but then not check it statically?</p>
	Structural	<p>Overhead of declaring many types is inconvenient; Problems with semantics of subtypes (see later)</p>	<p>Sweetspot: Maximum flexibility</p>

2. Types and Subtyping

2.1 Types

2.2 Subtyping

2.3 Behavioral Subtyping

Types

- Definition:

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

- Question: what are the “*properties*” shared by the values of a type?
 - So far we focused on syntax
- “*Properties*” should also include the **behavior of the object**
 - Expressed as interface specifications (**contracts**)

Method Behavior

- **Preconditions** have to hold in the state before the method body is executed
- **Postconditions** have to hold in the state after the method body has terminated
- **Old-expressions** can be used to refer to prestate values from the postcondition

```
class BoundedList {  
    Object[ ] elems;  
    int free; // next free slot  
    ...  
    // requires free < elems.length  
    // ensures elems[ old( free ) ] == e  
    void add( Object e ) { ... }  
}
```

Object Invariants

- Object invariants describe **consistency criteria** for objects
- **Invariants** have to hold in all states, in which an object can be accessed by other objects

```
class BoundedList {  
    Object[ ] elems;  
    int free; // next free slot  
    /* invariant  
       elems != null           &&  
       0 <= free               &&  
       free <= elems.length   */  
  
    ...  
    // requires free < elems.length  
    // ensures elems[ old( free ) ] == e  
    void add( Object e ) { ... }  
}
```


Visible States

- Invariants have to **hold in pre- and poststates** of methods executions but may be **violated temporarily** in between
- Pre- and poststates are called “**visible states**”

```
class Redundant {  
    private int a, b;  
    // invariant a == b  
  
    public void set( int v ) {  
        // invariant of this holds  
        a = v;  
        // invariant of this violated  
        b = v;  
        // invariant of this holds  
    }  
}
```

History Constraints

- History constraints describe **how objects evolve over time**
- History constraints **relate visible states**
- Constraints must be **reflexive** and **transitive**

```
class Person {  
    int age;  
  
    // constraint old( age ) <= age  
  
    Person( int age ) {  
        this.age = age;  
    }  
  
    ...  
}
```

```
Person p = new Person( 7 );  
...  
...  
assert 7 <= p.age;
```

Static vs. Dynamic Contract Checking

Static checking

Program verification

- **Static safety:** More errors are found at compile time
- **Complexity:** Static contract checking is difficult and not yet mainstream
- **Large overhead:** Static contract checking requires extensive contracts
- **Examples:** Spec#, JML

Dynamic checking

Run-time assertion checking

- **Incompleteness:** Not all properties can be checked (efficiently) at run-time
- **Efficient bug-finding:** Complements testing
- **Low overhead:** Partial contracts are useful
- **Examples:** Eiffel, JML

Contracts and Subtyping

```
class Number {  
  
    int n;  
    // invariant true  
  
    // requires true  
    // ensures n == p  
    void set( int p )  
        { n = p; }  
  
    ...  
}
```

```
class UndoNaturalNumber  
    extends Number {  
  
    int undo;  
    // invariant 0 < n && 0 < undo  
  
    // requires 0 < p  
    // ensures n == p && undo == old( n )  
    void set( int p )  
        { undo = n; n = p; }  
  
    ...  
}
```

- Subtypes specialize the behavior of supertypes
- What are legal specializations?

Rules for Subtyping: Preconditions

```
class Super {  
  // requires 0 <= n && n < 5  
  void foo( int n ) {  
    char[ ] tmp = new char[ 5 ];  
    tmp[ n ] = 'X';  
  }  
}
```

```
class Sub extends Super {  
  // requires 0 <= n && n < 3  
  void foo( int n ) {  
    char[ ] tmp = new char[ 3 ];  
    tmp[ n ] = 'X';  
  }  
}
```

```
void crash( Super s ) {  
  s.foo( 4 );  
}
```

```
x.crash( new Sub( ) );
```

- Subtype objects must **fulfill contracts** of supertypes
- Overriding methods of subtypes may have **weaker preconditions** than corresponding supertype methods

Rules for Subtyping: Postconditions

```
class Super {  
  // ensures  $0 < \text{result}$   
  int foo( ) {  
    return 1;  
  }  
}
```

```
class Sub extends Super {  
  // ensures  $0 \leq \text{result}$   
  int foo( ) {  
    return 0;  
  }  
}
```

```
void crash( Super s ) {  
  int i = 5 / s.foo( );  
}
```

```
x.crash( new Sub( ) );
```

- Overriding methods of subtypes may have **stronger postconditions** than corresponding supertype methods

Rules for Subtyping: Invariants

```
class Super {  
  int n;  
  // invariant  $0 < n$   
  Super( )    { n = 5; }  
  int crash( ) { return 5 / n; }  
}
```

```
new Sub( ).crash( );
```

```
class Sub extends Super {  
  // invariant  $0 \leq n$   
  Sub( ) {  
    n = 0;  
  }  
}
```

- Subtypes may have **stronger invariants**

Rules for Subtyping: History Constraints

```
class Super {  
  int n;  
  
  // constraint old( n ) <= n  
  
  int get( ) { return n; }  
  
  void foo( ) { }  
}
```

```
class Sub extends Super {  
  // constraint true  
  
  void foo( ) {  
    n = n - 1;  
  }  
}
```

```
int crash( Super s ) {  
  int cache = s.get( ) - 1;  
  s.foo( );  
  return 5 / ( cache - s.get( ) );  
}
```

```
x.crash( new Sub( ) );
```

- Subtypes may have **stronger history constraints**

Natural Numbers Revisited

```
class Number {  
  
    int n;  
    // invariant true  
  
    // requires true  
    // ensures n == p  
    void set( int p )  
        { n = p; }  
  
    ...  
}
```

```
class UndoNaturalNumber  
    extends Number {  
  
    int undo;  
    // invariant 0 < n && 0 < undo  
  
    // requires 0 < p  
    // ensures n == p && undo == old( n )  
    void set( int p )  
        { undo = n; n = p; }  
  
    ...  
}
```

- UndoNaturalNumber does not specialize the behavior of Number

Rules for Subtyping: Summary

- Subtype objects must **fulfill contracts** of supertypes, but:
 - Subtypes can have **stronger invariants**
 - Subtypes can have **stronger history constraints**
 - Overriding methods of subtypes can have
weaker preconditions
stronger postconditions
than corresponding supertype methods
- Concept is called **Behavioral Subtyping**
 - Often implemented via **specification inheritance**

Static Checking of Behavioral Subtyping

- For each override $S.m$ of $T.m$ check for all parameters, heaps, and results
 - $\text{Pre}_{T.m} \Rightarrow \text{Pre}_{S.m}$ and $\text{Post}_{S.m} \Rightarrow \text{Post}_{T.m}$
- For each subtype $S <: T$ check for all heaps:
 - $\text{Inv}_S \Rightarrow \text{Inv}_T$ and $\text{Cons}_S \Rightarrow \text{Cons}_T$
- But: entailment is undecidable

```
class Super {  
  // requires  $p == p * p$   
  // ensures  $p < \text{result}$   
  int foo( int p ) { ... }  
}
```

```
class Sub extends Super {  
  // requires  $p == 0 \parallel p == 1$   
  // ensures  $\text{result} == 2$   
  int foo( int p ) { ... }  
}
```

- For all $p :: p == p * p \Rightarrow (p == 0 \parallel p == 1)$
- For all $p, \text{result} :: \text{result} == 2 \Rightarrow p < \text{result}$

Run-Time Checking of Behav. Subtyping

- Checking entailment for all parameters, heaps, and results is **not possible at run time**
- Idea: check **those properties** subsequent code **relies on**
- For each call `o.m(...)`
 - Check **precondition** of `m` in `o`'s **dynamic type** (which the executed body relies on)
 - Check **postcondition** of `m` in `o`'s **static type** (which the caller relies on)

```
class Super {  
    // requires p == p*p  
    // ensures p < result  
    int foo( int p ) { ... }  
}
```

```
class Sub extends Super {  
    // requires p == 0 || p == 1  
    // ensures result == 2  
    int foo( int p ) { ... }  
}
```

```
void bar( Super s ) {  
    // check 0 == 0 || 0 == 1  
    r = s.foo( 0 );  
    // check 0 < r  
}
```

Assume `s` is
a Sub object

Specification Inheritance

- Behavioral subtyping can be enforced by **inheriting specifications** from supertypes
- Rule for invariants
 - The invariant of a type *S* is the **conjunction** of the invariant **declared in *S*** and the invariants **declared in the supertypes of *S***
 - Subtypes have **stronger invariants**
 - Analogous for history constraints

```
class Super {  
  int n;  
  // invariant  $0 < n$   
  Super() { n = 5; }  
  int crash() {  
    return 5 / n;  
  }  
}
```

```
class Sub extends Super {  
  // invariant  $0 \leq n$   
  Sub() { n = 0; }  
}
```

Violates
inherited
invariant

Simple Inheritance of Method Contracts

- An overriding method **must not declare additional preconditions**
 - The overriding and the overridden method have **identical preconditions**
- The postcondition of an overriding method is the **conjunction** of the postcondition **declared for the method** and the postconditions **declared for the methods it overrides**
 - Overriding methods have **stronger postconditions**

```
class Super {  
    // requires  $0 \leq n \ \&\& \ n < 5$   
    void foo( int n ) { ... }  
}
```

```
class Sub extends Super {  
    // requires  $0 \leq n \ \&\& \ n < 3$   
    void foo( int n ) { ... }  
}
```

Precondition Inheritance: Shortcomings

- Simple rule does not work for **multiple subtyping**

```
interface I {  
  // requires 0 < n  
  int foo( int n );  
}
```

```
interface J {  
  // requires n < 0  
  int foo( int n );  
}
```

```
class C implements I, J {  
  int foo( int n ) { ... }  
}
```

What is the
resulting
precondition?

- Simple rule does not allow **precondition weakening**

```
class Set {  
  // requires contains( x )  
  void remove( Object x )  
  { ... }  
}
```

```
class MySet extends Set {  
  // requires true  
  void remove( Object x )  
  { ... }  
}
```

Precondition Inheritance: Improved Rule

```
interface I {  
    // requires 0 < n  
    // ensures result == n  
    int foo( int n );  
}
```

```
interface J {  
    // requires n < 0  
    // ensures result == -n  
    int foo( int n );  
}
```

```
class C implements I, J {  
    int foo( int n ) { ... }  
}
```

- Clients fetch an object through a static type

```
void client1( I x ) {  
    // assert 0 < 5  
    int y = x.foo( 5 )  
    // assume y == 5  
}
```

```
void client2( J x ) {  
    // assert -3 < 0  
    int y = x.foo( -3 )  
    // assume y == 3  
}
```

- Idea: method implementation may assume only the **disjunction of all inherited and declared preconditions**

Effective Preconditions

- Let $\text{Pre}_{T.m}$ denote the precondition of method m declared in class T
- The **effective precondition** $\text{PreEff}_{S.m}$ of a method m in class S is the **disjunction** of the precondition $\text{Pre}_{S.m}$ declared for the method and the preconditions $\text{Pre}_{T.m}$ declared for the methods it **overrides**
 - $\text{PreEff}_{S.m} = \text{Pre}_{S.m} \parallel \text{Pre}_{T.m} \parallel \text{Pre}_{T'.m} \parallel \dots$
- Overriding methods have **weaker eff. preconditions**

Shortcomings Revisited

- Improved rule works for **multiple subtyping**

```
interface I {  
  // requires 0 < n  
  int foo( int n );  
}
```

```
interface J {  
  // requires n < 0  
  int foo( int n );  
}
```

```
class C implements I, J {  
  int foo( int n ) { ... }  
}
```

Effective
precondition:
 $0 < n \parallel n < 0$

- Improved rule allows **precondition weakening**

```
class Set {  
  // requires contains( x )  
  void remove( Object x )  
  { ... }  
}
```

```
class MySet extends Set {  
  // requires true  
  void remove( Object x )  
  { ... }  
}
```

Effective
precondition:
 $\text{contains}(x) \parallel \text{true}$

Postcondition Inheritance: Improved Rule

- Simple postcondition rule becomes too restrictive

```
class Set {  
  // requires contains( x )  
  // ensures size() == old( size() - 1 )  
  void remove( Object x )  
  { ... }  
}
```

```
class MySet extends Set {  
  // requires true  
  void remove( Object x )  
  { ... }  
}
```

- Idea: method implementation needs to satisfy **each postcondition for which the corresponding precondition holds**
 - $\text{PostEff}_{S.m} = (\text{Pre}_{S.m} \Rightarrow \text{Post}_{S.m}) \ \&\& \ (\text{Pre}_{T.m} \Rightarrow \text{Post}_{T.m}) \ \dots$

Postcondition Inheritance: Improved Rule

```
class Set {  
  // requires contains( x )  
  // ensures !contains( x )  
  void remove( Object x )  
  { ... }  
}
```

```
class MySet extends Set {  
  // requires true  
  // ensures true  
  void remove( Object x )  
  { ... }  
}
```

- Rule from previous slide produces bogus result:
 - $\text{PostEff}_{\text{MySet.remove}} = (\text{contains}(x) \Rightarrow \neg \text{contains}(x)) \ \&\& \ (\text{true} \Rightarrow \text{true})$
- Precondition must be evaluated in prestate:
 - $\text{PostEff}_{\text{MySet.remove}} = (\text{old}(\text{contains}(x)) \Rightarrow \neg \text{contains}(x)) \ \&\& \ (\text{old}(\text{true}) \Rightarrow \text{true})$

Effective Postconditions

- Let $\text{Post}_{T.m}$ denote the postcondition of method m declared in class T
- The **effective postcondition** $\text{PostEff}_{S.m}$ of a method m in class S is the **conjunction** of implications $(\text{old}(\text{Pre}_{T.m}) \Rightarrow \text{Post}_{T.m})$ for all types T such that T declares $S.m$ or $S.m$ overrides $T.m$
 - $\text{PostEff}_{S.m} = (\text{old}(\text{Pre}_{S.m}) \Rightarrow \text{Post}_{S.m}) \ \&\& \ (\text{old}(\text{Pre}_{T.m}) \Rightarrow \text{Post}_{T.m}) \ \&\& \ (\text{old}(\text{Pre}_{T'.m}) \Rightarrow \text{Post}_{T'.m}) \ \&\& \ \dots$
- Overriding methods have **stronger eff. postconditions**

Behavioral Structural Subtyping

- Structural type systems determine subtype relation automatically
- Automatic static checking
 - Not possible, see above
- Dynamic checking
 - Not possible for all contracts
 - **Postcondition cannot be checked**
because we have no static type for receiver, so we do not know what we can expect
(just like result types are not checked)

```
class Circle {  
    draw( ) { ... }  
}
```

```
render( p ) {  
    p.draw( );  
}
```

```
class Cowboy {  
    draw( ) { ... }  
}
```

Types as Contracts

- Types can be seen as a special form of contract, where **static checking is decidable**
- Operator `type(x)` yields the **type of the object** stored in `x`
 - (The dynamic type of `x`)

```
class Types {  
    Person p;  
    String foo( Person q ) { ... }  
    ...  
}
```

```
class Types {  
    p;  
    // invariant type( p ) <: Person  
    // require type( q ) <: Person  
    // ensure type( result ) <: String  
    foo( q ) { ... }  
    ...  
}
```

Types as Contracts: Subtyping

```
class Super {
  S p;
  // invariant type( p ) <: S
  // require type( q ) <: T
  // ensure type( result ) <: U
  U foo( T q ) { ... }
}
```

```
class Sub <: Super {
  S' p;
  // invariant type( p ) <: S'
  // require type( q ) <: T'
  // ensure type( result ) <: U'
  U' foo( T' q ) { ... }
}
```

■ Stronger invariant:

- $\text{type}(p) <: S' \Rightarrow \text{type}(p) <: S$
requires $S' <: S$

Covariance

■ Weaker precondition

- $\text{type}(q) <: T \Rightarrow \text{type}(q) <: T'$
requires $T <: T'$

Contravariance

■ Stronger postcondition:

- $\text{type}(\text{result}) <: U' \Rightarrow$
 $\text{type}(\text{result}) <: U$
requires $U' <: U$

Covariance

Invariants over Inherited Fields

```
package Library;  
public class Super {  
    protected int f;  
}
```

```
package Client;  
public class Sub  
    extends Super {  
    // invariant 0 <= f  
}
```

```
package Library;  
class Friend {  
    void foo( Super s ) { s.f = -1; }  
}
```

- Invariants over inherited field f can be **violated by all methods that have access to f**
- Static checking of such invariants is **not modular**
- Even without qualified field accesses ($x.f = e$), one needs to **re-check all inherited methods**

Immutable Types

- Objects of immutable types **do not change their state** after construction
- Advantages
 - No unexpected modifications of shared objects
 - No thread synchronization necessary
 - No inconsistent states
- Examples from Java
 - String, Integer

```
class ImmutableCell {  
    int value;  
  
    ImmutableCell( int value ) {  
        this.value = value;  
    }  
  
    int get( ) {  
        return value;  
    }  
  
    // no setter  
}
```

Immutable and Mutable Types

```
class ImmutableCell {  
  int value;  
  ImmutableCell( int value ) { ... }  
  int get( ) { ... }  
  // no setter  
}
```

- What should be the subtype relation between mutable and immutable types?

```
class Cell {  
  int value;  
  Cell( int value ) { ... }  
  int get( ) { ... }  
  void set( int value ) { ... }  
}
```

Immutable and Mutable Types (cont'd)

```
class ImmutableCell extends Cell {  
    ImmutableCell( int value ) { ... }  
    void set( int value ) {  
        // throw exception  
    }  
}
```

```
class Cell {  
    int value;  
    Cell( int value ) { ... }  
    int get( ) { ... }  
    void set( int value ) { ... }  
}
```

- Proposal 1: **Immutable type should be subtype**
- Not possible because mutable type has wider interface

Immutable and Mutable Types (cont'd)

```
class ImmutableCell {  
  int value;  
  // constraint old( value ) == value  
  ... // no setter  
}
```

```
class Cell extends ImmutableCell {  
  Cell( int value ) { ... }  
  void set( int value ) { ... }  
}
```

```
foo( ImmutableCell c ) {  
  int cache = c.get( );  
  ...  
  assert cache == c.get( );  
}
```

- Proposal 2: **Mutable type should be subtype**
- Mutable type has **wider interface**
 - Also complies with structural subtyping
- But: **Mutable type does not specialize behavior**

Immutable and Mutable Types: Solutions

- Clean solution
 - No subtype relation between mutable and immutable types
 - Only exception: **Object**, which has no history constraint
- Java API contains immutable types that are subtypes of mutable types
 - AbstractCollection and Iterator are mutable
 - All mutating methods are optional

