Concepts of
Object-Oriented Programming

Peter Müller
Chair of Programming Methodology

Autumn Semester 2017
C-Example Revisited

```c
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 );
```
3

2. Types and Subtyping

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: assembly languages

- **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

- 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, Go, O’Caml
Nominal and Structural Types

- **Type membership**
  - Two nominal types
  - One structural type

- **Type equivalence**
  - S and T are **different** in nominal systems
  - S and T are **equivalent** in structural systems
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"
5 + 7
```
```
int a;
boolean equals( Object o )
```
```
a + 7
"A number: " + 7
"A string".equals( null )
```
```
a = "A string";
"A string".equals( 1, 2 )
```
Dynamic Type 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"
5 + 7

a = …;
def foo(o): …

a + 7
"A number: " * 7
foo(None)

a = "A string"
a = 7

a = "A string" / 5
foo(5, 7)
```
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
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

```java
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

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

```javascript
eval("x=10; y=20; document.write( x*y )")
```
Bypassing Static Type Checks

- Some static type systems provide ways to bypass static checks
  - C#, Scala
  - Useful to interoperate with dynamically-typed languages or the HTML Document Object Model (DOM)

- Type safety is preserved via run-time checks

```csharp
dynamic v = getPythonObject();
dynamic res = v.Foo(5);
```
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
- **Tool support**: Types enable auto-completion, support for refactoring, etc.

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

<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>C++, C#, Eiffel, Java, Scala</td>
</tr>
<tr>
<td></td>
<td>Go, O’Caml</td>
</tr>
<tr>
<td></td>
<td>JavaScript, Python, Ruby, Smalltalk</td>
</tr>
</tbody>
</table>

“**When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.**”

[James Whitcomb Riley]

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

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

Only T is a nominal subtype of S

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

- 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

```java
class Super {
    void foo() { … }
    void bar() { … }
}

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

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

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

- An **overriding method must not be less accessible** than the methods it overrides.

```java
class Super {
    public void foo() { … }
    public void bar() { … }
}
class Sub <: Super {
    public void foo() { … }
    private void bar() { … }
}
void m(Super s) { s.bar(); }
```
Overriding: Parameter Types

- 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

```java
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( ));
}
```
Overriding: Result Types

- 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

```java
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();
}
```
Overriding: Fields

- 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

```java
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;
}
```
Overriding: Fields (cont’d)

- 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)

```java
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; }
}
```
Overriding: Immutable Fields

- Immutable fields do not have setters
- Types of immutable fields can be specialized in subclasses \((S <: T)\)
  - Works only if the supertype constructor does not initialize \(f\) for subtype objects (with a \(T\)-value)!
- Not permitted by mainstream languages

```java
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; }
}
```
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“

```eiffel
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
Covariant Arrays

- In Java and C#, arrays are covariant
  - If \( S <: T \) then \( S[ ] <: T[ ] \)

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

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

- Each array update requires a run-time type check

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

```
class String[ ] <: Object[ ] {
    public String 0;
    public String 1;
    ...
}
```
Covariant Arrays (cont’d)

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

```java
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

```java
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)

```java
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)

```java
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

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

```java
class Cell {
    int value;
    int getData( ) { return value; }
}
```
Shortcomings of Nominal Subtyping (2)

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

```java
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

```java
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

```java
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

```java
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

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

**Nominal**

**Structural**
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.

```java
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

```java
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”

```java
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

```java
class Person {
    int age;

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

```java
Person p = new Person( 7 );
...
...
assert 7 <= p.age;
```
# Static vs. Dynamic Contract Checking

<table>
<thead>
<tr>
<th>Static checking</th>
<th>Dynamic checking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Program verification</strong></td>
<td><strong>Run-time assertion checking</strong></td>
</tr>
<tr>
<td>▪ <strong>Static safety</strong>: More errors are found at compile time</td>
<td>▪ <strong>Incompleteness</strong>: Not all properties can be checked (efficiently) at run time</td>
</tr>
<tr>
<td>▪ <strong>Complexity</strong>: Static contract checking is difficult and not yet mainstream</td>
<td>▪ <strong>Efficient bug-finding</strong>: Complements testing</td>
</tr>
<tr>
<td>▪ <strong>Large overhead</strong>: Static contract checking requires extensive contracts</td>
<td>▪ <strong>Low overhead</strong>: Partial contracts are useful</td>
</tr>
<tr>
<td>▪ <strong>Examples</strong>: Spec#, .NET</td>
<td>▪ <strong>Examples</strong>: Eiffel, .NET</td>
</tr>
</tbody>
</table>
Contracts and Subtyping

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

```java
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; }
    ...
}
```
Rules for Subtyping: Preconditions

```java
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';
    }
}
```

- 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 < result
  int foo() {
    return 1;
  }
}

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

**class** Sub extends Super {
  // ensures 0 <= result
  int foo() {
    return 0;
  }
}

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

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

x.crash( new Sub( ) );
```
Rules for Subtyping: Invariants

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

```java
class Sub extends Super {
    // invariant 0 <= n
    Sub() {
        n = 0;
    }
}
```

- Subtypes may have stronger invariants
Rules for Subtyping: History Constraints

```java
public class Super {
    int n;

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

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

```java
public 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

- **UndoNaturalNumber** does not specialize the behavior of **Number**

```java
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;
    }
    ...
}
```
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 $\text{S.m}$ of $\text{T.m}$ check for all parameters, heaps, and results
  - $\text{Pre}_{\text{T.m}} \Rightarrow \text{Pre}_{\text{S.m}}$
  - $\text{Post}_{\text{S.m}} \Rightarrow \text{Post}_{\text{T.m}}$

- Any caller that wants to use the postcondition must establish the precondition
  - $\text{Pre}_{\text{T.m}} \Rightarrow$
    ( $\text{Post}_{\text{S.m}} \Rightarrow \text{Post}_{\text{T.m}}$ )
Static Checking of Behav. Subtyping (c’t)

- For each override \( S.m \) of \( T.m \) check for all parameters, heaps, and results:
  - \( \text{Pre}_{T.m} \Rightarrow \text{Pre}_{S.m} \)
  - \( \text{Post}_{S.m} \Rightarrow \text{Post}_{T.m} \)

- Any caller that wants to use the postcondition must establish the precondition
  - \( \text{old}(\text{Pre}_{T.m}) \Rightarrow (\text{Post}_{S.m} \Rightarrow \text{Post}_{T.m}) \)

```java
class Super {
    int p;
    // requires 0 < p
    // ensures 0 < result
    int foo( ) { ... }  
}

class Sub extends Super {
    // requires true
    // ensures p < result
    int foo( ) {
        p = -2; return -1;
    }
}

int client( Super s ) {
    s.p = 5; int r = s.foo( );
    assert 0 < r;
}
```
Static Checking of Behav. Subtyping (c’t)

- For each subtype S <: T check for all heaps:
  - Inv$_S$ => Inv$_T$
  - Cons$_S$ => Cons$_T$

- But: entailment is undecidable

```
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 ) { … }          }
```

- For all p ::
  p == p*p  =>  (p == 0 || p == 1)

- For all p, result ::
  old(p == p*p) => ( result == 2 =>  p < result )
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

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

class Sub extends Super {
    // invariant 0 <= 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 <= n && n < 5
    void foo(int n) { ... }
}

class Sub extends Super {
    // requires 0 <= n && n < 3
    void foo(int n) { ... }
}
```
Precondition Inheritance: Shortcomings

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

```java
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) {
    ...
    }
}
```

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

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

class MySet extends Set {
    // requires true
    void remove(Object x) {
    ...
    }
}
```
Precondition Inheritance: Improved Rule

- Clients view an object through a static type

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

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

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

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

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

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

- $\text{PreEff}_{S.m} = \text{Pre}_{S.m} \parallel \text{Pre}_{T.m} \parallel \text{Pre}_{T'.m} \parallel \ldots$

Overriding methods have weaker eff. preconditions.
Shortcomings Revisited

- Improved rule works for **multiple subtyping**

```java
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:** $0 < n \lor n < 0$

- Improved rule allows **precondition weakening**

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

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

**Effective precondition:** $\text{contains}(x) \lor \text{true}$
Postcondition Inheritance: Improved Rule

- Simple postcondition rule becomes too restrictive

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

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

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

- **Rule from previous slide produces a bogus result:**
  - $\text{PostEff}_{\text{MySet.remove}} = (\text{contains}(x) \Rightarrow !\text{contains}(x)) \land (\text{true} \Rightarrow \text{true})$

- **Precondition must be evaluated in prestate:**
  - $\text{PostEff}_{\text{MySet.remove}} = (\text{old}(\text{contains}(x)) \Rightarrow !\text{contains}(x)) \land (\text{old}(\text{true}) \Rightarrow \text{true})$

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

class MySet extends Set {
    // requires true
    // ensures true
    void remove(Object x) {
        ...  
    }
}
```
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 $T.m$
  - $\text{PostEff}_{S.m} = (\text{old}(\text{Pre}_{S.m}) \Rightarrow \text{Post}_{S.m}) \&\& (\text{old}(\text{Pre}_{T.m}) \Rightarrow \text{Post}_{T.m}) \&\& (\text{old}(\text{Pre}_{T'.m}) \Rightarrow \text{Post}_{T'.m}) \&\& \ldots$
- Overriding methods have stronger effective postconditions
Run-Time Checking

- **Checking entailment** for all arguments, heaps, and results is **not possible at run time**
  - For all \( p :: p == p*p \Rightarrow (p == 0 \text{ || } p == 1) \)

- Specification inheritance avoids this problem by defining **effective contracts**, which satisfy behavioral subtyping

- The run-time checker can check **effective contracts**
Behavioral Structural Subtyping

- With **dynamic type checking**, callers have **no static knowledge of contracts**
  - Cannot establish precondition
  - Have no postcondition to assume
- Called method may check its own contract (see above)
  - Precondition failures are analogous to “message not understood”; **caller cannot be blamed**
  - Postcondition failures may reveal error in method implementation (**like an assert**)
Behavioral Structural Subtyping (cont’d)

- With **static structural type checking**, callers could state which **signature and behavior** they require

```java
render({ void draw() 
    requires P 
    ensures Q } p) {
    p.draw();
}
```

- Contract can be checked statically or dynamically
Behavioral Subtyping needs to be checked when the type system determines a subtype relation.

Static checking is possible, but in general not automatic.

Dynamic checking is not possible (see above):
- Caller cannot be blamed for violations of \( P' \)
- Callee cannot be blamed for violations of \( Q \)

```java
class Circle {
    // requires P'
    // ensures Q'
    draw() { … }
}
```

```java
render( { void draw( )
    requires P
    ensures Q } p ) {
    p.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

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

```java
class Types {
    p;
    // invariant type( p ) <: Person
    // requires type( q ) <: Person
    // ensures type( result ) <: String
    foo( q ) { … }
}
```
Types as Contracts: Subtyping

- **Stronger invariant:**
  - \(\text{type}(p) <: S' \implies \text{type}(p) <: S\)
  - requires \(S' <: S\)

- **Weak precondition**
  - \(\text{type}(q) <: T \implies \text{type}(q) <: T'\)
  - requires \(T <: T'\)

- **Stronger postcondition**
  - \(\text{type}(\text{result}) <: U' \implies \text{type}(\text{result}) <: U\)
  - requires \(U' <: U\)

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

class Sub <: Super {
    S' p;
    // invariant type(p) <: S'
    // requires type(q) <: T'
    // ensures type(result) <: U'
    U' foo(T' q) { ... }
}
```
Invariants over Inherited Fields

- 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.

```java
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; }
}
```
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

```java
class ImmutableCell {
    private int value;

    ImmutableCell(int value) {
        this.value = value;
    }

    int get() {
        return value;
    }

    // no setter
}
```
Immutable and Mutable Types

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

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

class Cell {
    int value;
    Cell(int value) {
        ... }
    int get() {
        ... }
    void set(int value) {
        ... }
}
```
2.3 Types and Subtyping – Behavioral Subtyping

Immutable and Mutable Types (cont’d)

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

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

```java
class Cell {
    int value;
    Cell(int value) { … }
    int get() { … }
    void set(int value) { … }
}
```
Immutable and Mutable Types (cont’d)

- Proposal 2: **Mutable type should be subtype**

- **Mutable type has wider interface**
  - Also complies with structural subtyping

- **But:** **Mutable type does not specialize behavior**

```java
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();
}
```
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
References

- Barbara Liskov and Jeannette Wing: *A Behavioral Notion of Subtyping*. ACM Transactions on Programming Languages and Systems, 1994