2. Objects, Classes, Inheritance

Overview:

2.1 Objects and classes

- Characteristics of objects
- Describing and implementing objects
- Objects representing program elements

2.2 Subtyping and inheritance

- Classification
- Subtyping and dynamic binding
- Inheritance

2.3 Semantics of object-oriented languages

Programming with objects leads to a different

- program structure
- execution model

when compared to imperative programming

2.1 Objects and Classes

2.1.1 Characteristics of Objects

Objects – in the sense of informatics – have:

- an identity
- a state
- a location
- an interface (or protocol)
- references to other objects
- a lifecycle
- a behavior

Remark:

The object characteristics apply to objects on the conceptual as well as on the programming level.

Identity, State, and Location

Objects have an ***identity***, that is,
objects that are equal w.r.t. state, behavior, and
all other aspects may be different.

Example: (Identity/equality)

The following expressions yield true:

```
new String("a") != new String("a")

new String("a").equals(new String("a"))
```

Objects have a ***state*** that can change over time.
In particular, the following holds:

- Objects of the same type can have different states.
- The behavior of an object can depend on its state.

Remark:

- On the programming level the state is given by the values of the instance variables.
- On the conceptual level the state may be an an abstract property.

© A. Poetzsch-Heffter, TU Kaiserslautern

Objects have a ***location***:

- conceptually
- in distributed and mobile computing
- related to the identity

Remark:

Mathematical objects do not have an identity or location („Where is the integer 7?“. Adding something to a set yields a different set.)

Lifecycle

Objects are created (usually by other objects) at some point in time.

Objects are possibly deleted sometime after their creation (usually by other objects, automatically, or by themselves).

Interfaces & References to other Objects

Objects have ***interfaces***. In particular:

- provided interface
- required interface
- interface for inheritance (considered later)

Explanation: (Provided/required interface)

The ***provided interface*** of an object X describes the services offered by X to „external“ objects.

On the programming level, a *service* is usually realized as an attribute access or as a method.

Objects may provide different interfaces to objects with different access rights.

The ***required interface*** of an object X describes the services that X needs from „external“ objects.

Remarks:

- On the programming level, required interfaces are seldom explicitly described.
- The required interface can often be derived from the references an object has to other objects.

Examples: (Interfaces)

1. Provided interfaces:

```
package p;

public class C {
   private int idno;
   String  name;
   public String getName() {...}
   public void myMethod() {...}
}
```

- for objects of classes outside p:
  - the public methods
- for objects of classes inside p:
  - additionally read and write access to instance variable `name`
- for objects of class C
  - additionally read and write access to instance variable `idno`

2. Required interface:

Observers require a reference to the observed subject.

Subject

attach(Observer)
detach(Observer)
notify()

observers *

Observer

update()

for all o in observers
o.update()

ConcreteSubject

getState()
setState()

subjectState

1 subject

ConcreteObserver

update()

observerState

return subjectState

observerState =
subject.getState()

Behavior

Objects have a ***behavior***. In particular:

- They can be passive, reacting only to messages from the outside.
- They can be passive and „synchronized“ allowing only one method execution at a time.
- They can be active, executing one or several threads.

Object behavior is the most complex aspect of an object.

Examples: (Behavior)

1. Single passive objects:

```
public class C1 {
    private int tmp;
    private int invokeCount = 0;
    public void increment() {
        tmp = invokeCount;
        tmp++;
        invokeCount = tmp;
    }
}
```

```
public class C2 {
   private int tmp;
   private int invokeCount = 0;
   synchronized
   public void increment() {
     tmp = invokeCount;
     tmp++;
     invokeCount = tmp;
   }
}
```

Objects of both classes are passive.
C1 may fail to count the invocations correctly.

2. Single active objects:

```
public class Act1 extends Thread {
  public Act1() { start(); }

  public run() {
    while( true ) {
      System.out.println("Hi di du");
      try{ Thread.sleep(1000); }
      catch( Exception e ){}
    }
  }
}
```

2.1.2 Describing and Implementing Objects

We distinguish between

- *single* objects:
  - implemented as one object of the underlying OO language
- *compound* objects:
  - appear to users as one object
  - implemented by many objects of the underlying OO language

Examples: (Compound objects)

```
class String {
  char[] value; // used for characters
  int offset;   // first used index
  int count;    // number of characters


  String() { value = new char[0]; }
  ...
  int indexOf(int ch) { ... }
  ...
  int length() { return count; }
  char charAt(int index) { ... }
  boolean equals(Object anObject) {...}
}
```

Describing single objects:

Two basic description techniques:

- class concept
- prototype concept

Class concept:

- Programs declare ***classes instead of individual objects.***
- A class is the ***description of the common properties*** of the class' objects.
- Classes define the interface of their instances.
- During program execution, objects of declared classes are created (***instances***).
- Classes cannot be modified at runtime.
- Class declarations correspond to record declarations in imperative languages.
- In typed languages, a class defines as well a type.

© A. Poetzsch-Heffter, TU Kaiserslautern

Prototype concept:

- Programs ***describe individual objects directly.***
- New objects are created by ***cloning*** existing objects and ***modifying*** their properties at runtime:
  - Cloning an object means to create a new object with the same properties (but different identities)
  - Modifying means adding attributes, or adding and replacing methods
- Avoid distinction between classes and instances.
- E.g. used in the language **Self**, **JavaScript**.

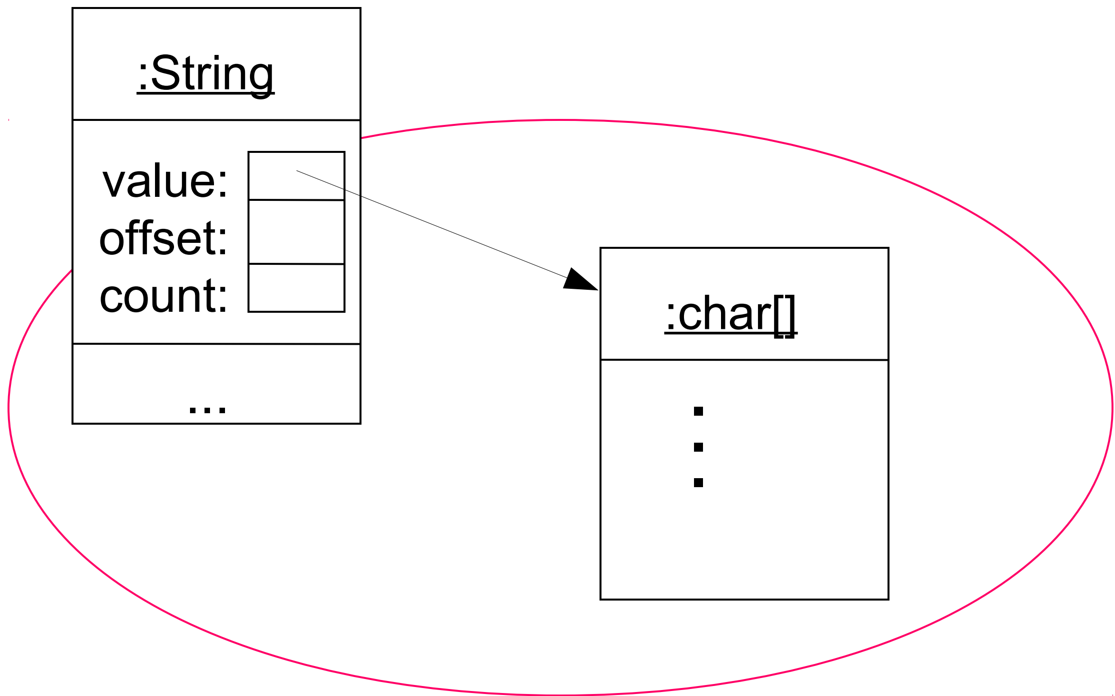
Examples: (Prototype language)

A Self program:

```
Vehicle: (Object copy)
(name <- ' ') Vehicle.
SportsCar: (Vehicle copy).
(driveToWork <-
  |some code, a block|) SportsCar.
Porsche911: (SportsCar copy).
Porsche911 name:'Bobs Car'.
```

Describing compound objects:

In class-based languages the initial structure of complex objects is described by the constructor of the object that „owns“ the compound:



Remarks:

- The implementation of the compounds is usually hidden from the user.
- The behavior of compound objects is often described in an abstract way.

2.1.3 Objects Representing Program Elements

Objects can as well be used to realize or represent elements used in programming like procedures or classes (cf. Self example above). We consider:

- procedure incarnations as objects
- introspection and reflection

Procedures and Classes:

In many respects, procedures are similar to classes:

- They are static program entities, declared by the programmer.
- At runtime:
  - Procedures are patterns for procedure incarnations
  - Classes are patterns for class instances (objects)
- Procedure incarnations and objects have identity, state, life cycle (in particular, coroutines)

Remarks:

BETA uses the analogy to unite procedures and classes to so-called *patterns*.

Introspection and reflection:

To automatically analyse and combine classes and objects within programs, a mechanism is needed that allows to

- inspect classes (*introspection),*
- create objects of classes based on the class name,
- invoke methods of unknown classes,
- construct new classes from given ones.

Typical applications: component frameworks

In OO programming, introspection and reflection can be achieved by modelling classes by objects, that is, having an object for each class:

- representing the information of the class
- allowing to invoke methods on the class

Java supports introspection and a restricted form of reflection:

- There is an object for every type in Java; these objects are instances of class `Class`.
- The static method `forname` of class `Class` takes the name of a class C and yields the corresponding `Class`-object.
- Objects of class `Method` represent methods.

Example: (Introspection)

The following program reads a class name, loads the corresponding class and prints its interfaces:

```
import java.lang.reflect.*;

public class Inspektor {
 public static void main(String[] ss) {
  try {
   Class klasse = Class.forName( ss[0] );
   Method[] methoden = klasse.getMethods();
   for( int i = 0; i < methoden.length; i++ ){
     Method m = methoden[i];
     Class retType = m.getReturnType();
     String methName = m.getName();
     Class[] parTypes = m.getParameterTypes();
     System.out.print( retType.getName() + " "
                              + methName + "(");
     for( int j=0; j < parTypes.length; j++ ){
       if( j > 0 ) System.out.print(", ");
       System.out.print(parTypes[j].getName());
     }
     System.out.println( ");" );
   }
  } catch( ClassNotFoundException e ) {
   System.out.println("Class "
                  + ss[0] + " not found");
}}}
```

Reflection and meta-programming goes beyond introspection by

- calling methods represented by a `Method`-object
- dynamically controlling the program behavior

Examples: (Reflection)

The following program uses reflection for higher-order programming:

```
static
String[] filter( String[] strs, Method test ){
  int i, j = 0;
  String[] aux = new String[ strs.length ];
  for( i = 0; i< strs.length; i++ ) {
    Object[] pars = { strs[i] };
    try{
      Boolean bobj;
      bobj = (Boolean) test.invoke(null,pars);
      if( bobj. booleanValue() ) {
        aux[j] = strs[i];
        j++;
      }
    } catch( Exception e ){
      System.out.println("error in invoke");
    }
  }
  String[] result = new String[ j ];
  System.arraycopy( aux, 0, result, 0, j );
  return result;
}
```

Method `invoke` belongs to class `Method` and has the following signature:

```
Object invoke( Object obj, Object[] args )
```

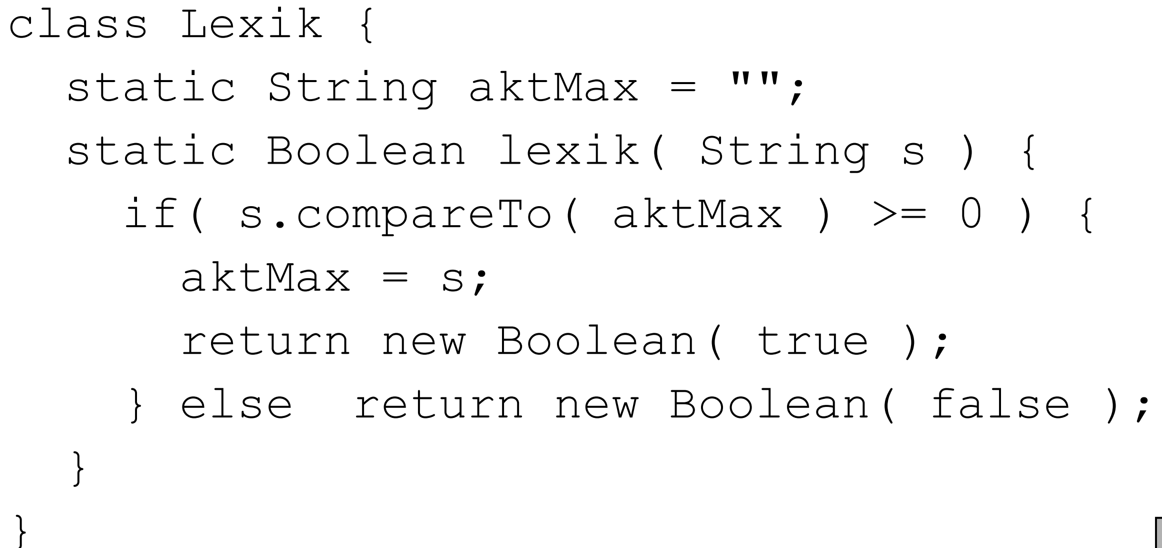
where the first parameter denotes the implicit parameter of the invocation; null is used for static methods. Here is a test scenario:

```
public static void main(String[] args) ... {
 Class cl = InvokeTest.class;
 Class[] parTypes = { String.class };
 Method m =
        cl.getMethod("prefixAorB",parTypes);
 System.out.println("Output of prefixAorB:");
 printStringArray( filter( args, m ) );

 cl = Lexik.class;
 m = cl.getMethod("lexik",parTypes);
 System.out.println("Output of lexik:");
 printStringArray( filter( args, m ) );
}

class InvokeTest {
  public static Boolean prefixAorB(String s) {
   if( s.length() > 0 && ( s.charAt(0)=='A',
                        || s.charAt(0)=='B' ) )
        return new Boolean( true );
    else return new Boolean( false );
  }
}
```

Test scenario continued:



```
class Lexik {
  static String aktMax = "";
  static Boolean lexik( String s ) {
    if( s.compareTo( aktMax ) >= 0 ) {
      aktMax = s;
      return new Boolean( true );
    } else  return new Boolean( false );
  }
}
```

Discussion:

- In the last years, reflection and meta-programming were important research topics, in particular for:
  - adaptation
  - composition
  - aspect separation
- The techniques provide a lot of flexibility at the price of efficiency and lost type safety.

2.2 Subtyping and Inheritance

2.2.1 Classification

Object-orientation has three ***core concepts***:

- Object model
- Object interfaces
- „is-a“ relation between objects enabling
  - polymorphism and classification
  - abstraction and specialization

Remark:

Notice that the object model is a prerequisite to object interfaces which in turn are the basis for the „is-a“ relation.

Explanation: („is-a“ relation, classification)

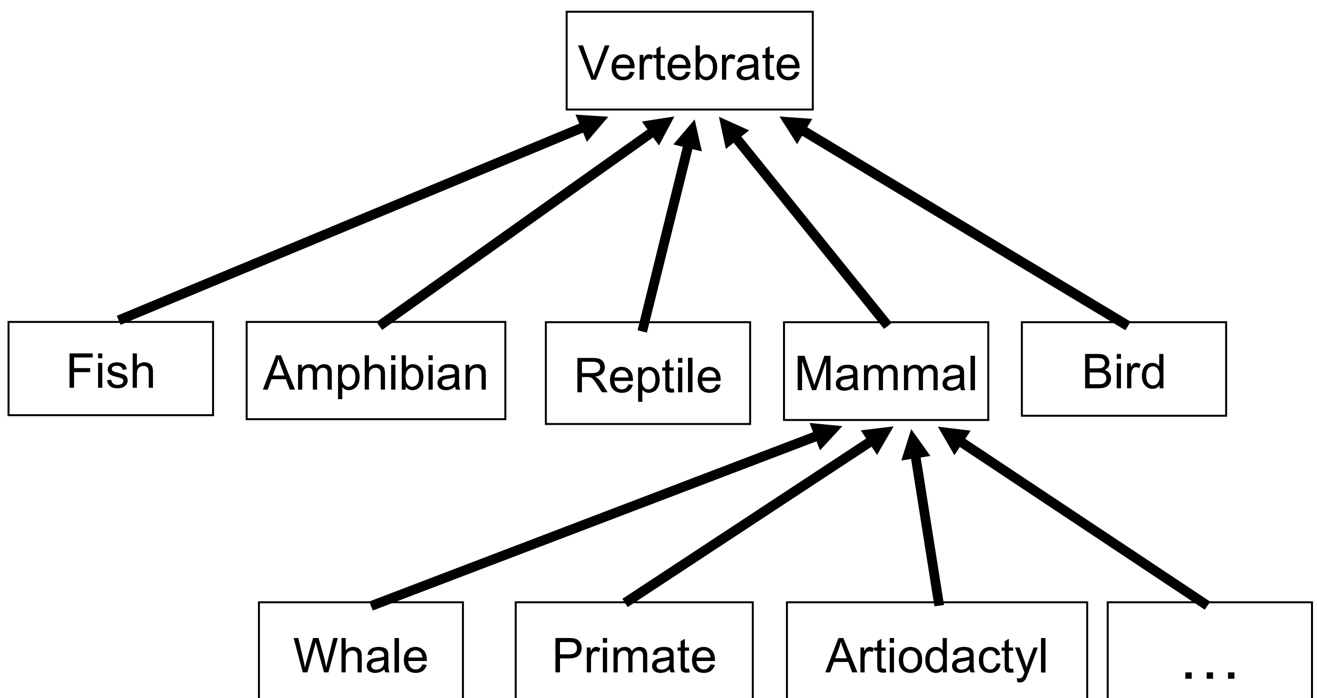
We can say that an object/item/term of type A ***is*** also ***an*** object/item/term of type B iff A-objects have all relevant properties of B-objects.

„is-a“ relations can be used to ***classify*** objects/items/ terms into hierarchical structures according to their properties. The result is often called a classification.

The objects/items/terms higher up in the classification are called more ***general***/***abstract*** than those below them which in turn are more ***special***/***concrete*** than the former.

Example: (classification)

1. Classification of vertebrates:



2. Classification of figures:

Arrows represent the „is-a“ relation

Figure

Ellipse → Figure

Polygon → Figure

Circle → Ellipse

Quadrangle → Polygon

Triangle → Polygon

Parallelogram → Quadrangle

Lozenge → Parallelogram

Rectangle → Parallelogram

Square → Lozenge

Square → Rectangle

Goal: Apply classification to software artefacts!

Classification in Object-Oriented Programming:

- Syntactic classification:

  Subtype objects understand at least the messages that supertype objects understand and have at least the same attributes (*wider interface*).

- Semantic classification:

  Subtype objects provide at least the behavior of supertype objects (*behavioral subtyping*).

- Substitution principle:

  Subtype objects can be used where objects of their supertypes are expected.

Explanation: (subtype polymorphism)

Polymorphism is the quality of being able to assume different forms.

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

Subtyping is a special kind of polymorphism in which types are ordered w.r.t. inclusion relation: Objects of a subtype belong as well to the supertype.

Remark:

Other kinds of polymorphism:

– Parametric polymorphism (generic types)

– Ad-hoc polymorphism (method overloading)

Designing Classifications:

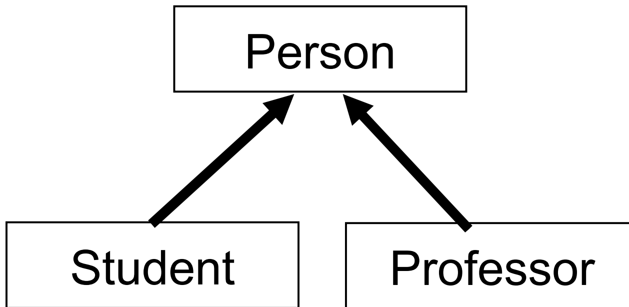
Explanation: (abstraction)

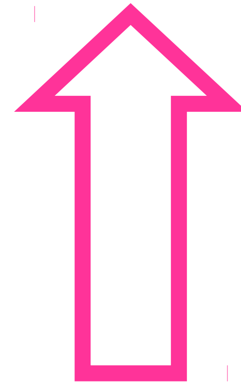
Abstraction means to form a general concept by extracting common features from specific examples.

Abstraction in OO programming:

- Start from different objects or types with common properties
- Develop a more abstract type, extracting the common properties
- Corresponds to making the interface smaller
- Program parts that only rely on the common properties work for all objects of the more abstract type

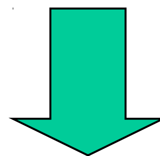
Example: (abstraction)





```
class Student {
  String      name;
  int               regno;
  …
  void print( ) {
   System.out.println(name);
   System.out.println(regno);
  }
}
```

```
class Professor {
  String  name;
  String  room;
  …
  void print( ) {
   System.out.println(name);
   System.out.println(room);
  }
}
```



Abstraction

```
interface Person {
  void print( );
}
```

```
class Student implements Person { … }
```

```
class Professor implements Person { …}
```

Application of abstraction, that is, of type Person:

„Algorithm" based on Person

```
Person[ ] p = new
Person[4];
p[0] = new Student (…);
p[1] = new Professor (…);
…
for ( int i=0; i < 4; i++ )
    p[ i ].print( );
```

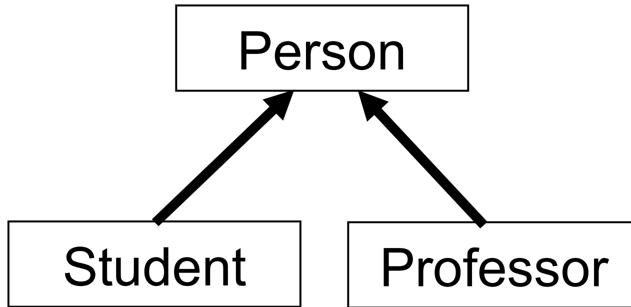
Explanation: (specialization)

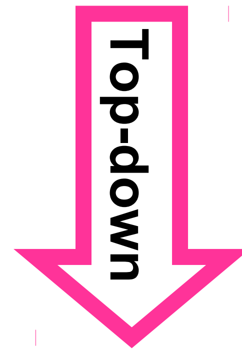
Specialization means to add specific properties to an object or to refine a concept by further characteristics.

Specialization in OO programming:

- Start from general objects or types.
- Extend them and their implementations.
- Requirement: Behavior of specialized objects should *conform* to behavior of more general objects.
- Program parts that work for the more general objects work as well for specialized objects.
- Implementation parts can be reused (inheritance).

Example: (specialization)





Specialization:

- Develop implementation for type Person
- Specialize it
- Inheritance and overriding can be used

```
class Person {
  String        name;
  …
   void print( ) {
   System.out.println( name );
  }
}
```

```
class Student
        extends Person {
  int       regno;
  …
  void print( ) {
    super.print( );
    System.out.println(regno);
  }
}
```

```
class Professor
         extends Person {
  String   room;
   …
  void print( ) {
    super.print( );
    System.out.println(room);
  }
}
```

2.2.2 Subtyping & Dynamic Binding

To enable classification of objects, all typed OO-languages support subtyping (there are as well other languages supporting subtyping).

Explanation: (subtyping)

In a type system with ***subtyping***, the types are partially ordered. If $S \leq T$, we say that S is a ***subtype*** of T and T a ***supertype*** of S.

Elements of the subtype belong as well to the supertype, i.e. types are not disjoint.

Example: (type system with subtyping)

Types:

- primitive datatypes: int, char, byte, ...
- Interface types
- Class types
- Array types

(Interface types, Class types and Array types:) Reference types

Subtype relation:

Declaration: interface S extends T1, T2, ...
implicit: S <T1, S <T2, ...

Declaration: class S extends T implements T1, T2, ...
implicit: S < T, S < T1, S < T2, ...

S < T implies: S[] < T[]

Subtype ordering is the reflexive and transitive closure.

Remarks:

- Syntactic interpretation of types: A type defines the attribute and method interface of its objects.
- Subtyping entails restrictions on method signatures in subtypes.

Typing Problem in OO-Languages:

To enable substitutability, a subtype object must have

- all attributes of supertype objects;
- all methods of supertype objects.

Attributes and methods in the subtype must be *compatible* to those of the supertypes. Compatibility concerns typing and accessibility.

 © A. Poetzsch-Heffter, TU Kaiserslautern

Explanation: (dynamic method binding)

An expression

E . m(p1,p2…);

is evaluated as follows:

1. Evaluate E. The result has to be an object, the so-called *receiver* or *target object* x.
2. Evaluate the actual parameters.
3. Evaluate the method with name m that is either declared in the class of x or the nearest superclass passing x and p1, p2 as pameters.

As the binding of name m to the method happens at runtime, the technique is called ***dynamic binding*** (often as well ***dynamic dispatch*** or ***dynamic method selection***).

Remark:

Subtyping and dynamic binding are the essential language concepts for improved *reusability* and *extensibility*

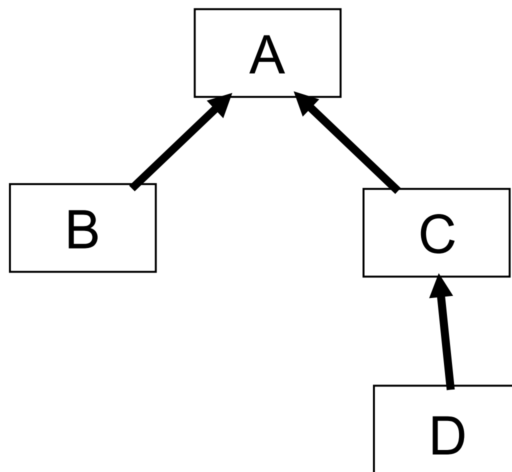
These aspects are further supported by inheritance.

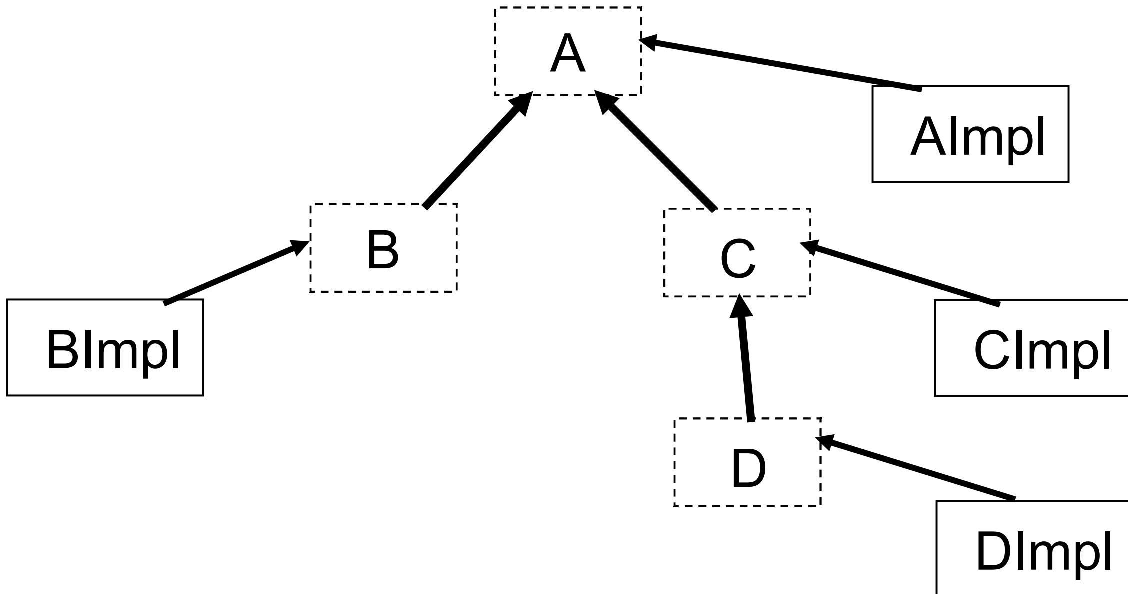
Expressiveness of Subtyping:

Subtyping together with dynamic binding can express classification. Inheritance is not needed:

Every class hierarchy can be expressed by a hierarchy of types where only the minimal types have an implementation.

Example: (refactoring class hierarchies)





2.2.3 Inheritance

Inheritance is a language concept to support the implementation of subclasses. Subclasses are implemented by:

- inheriting all attributes, methods, and inner classes from superclasses
- adding new members/features
- „adapting“ methods of superclasses

Inheritance guarantees that subclasses have at least the features required by subtyping.

Techniques for adapting methods:

- Overriding: provide a new method declaration that may use the overridden method (e.g. in Java)
- Specialization points: Methods contain named statement points at which subclasses can add code (e.g. in BETA).

Example: (Inheritance)

```
class Person {
  String name;
  int  gebdatum; /* Form JJJJMMTT */

  void  drucken() {
    System.out.println("Name: "+ this.name);
    System.out.println("Gebdatum: "+gebdatum);
  }

  boolean hat_geburtstag ( int datum ) {
    return (gebdatum%10000)==(datum%10000);
  }

  Person( String n, int gd ) {
    name = n;
    geburtsdatum = gd;
  }
}

class Student extends Person {
  int  matrikelnr;
  int  semester;

  void  drucken() {
    super.drucken();
    System.out.println("Matnr: "+ matrikelnr);
    System.out.println("Semzahl: "+ semester);
  }

  Student(String n,int gd,int mnr,int sem) {
    super( n, gd );
    matrikelnr = mnr;
    semester = sem;
  }
}
```

Explanation: (Inheritance interface)

The ***inheritance interface*** contains the features of class C that are potentially inherited by subclasses of C.

Problems of Inheritance:

- Subclasses may use the inherited code in a wrong way.
- The inheritance interface may be the interface between two developers (*fragile baseclass* problem).

Example: (Breaking superclass invariants)

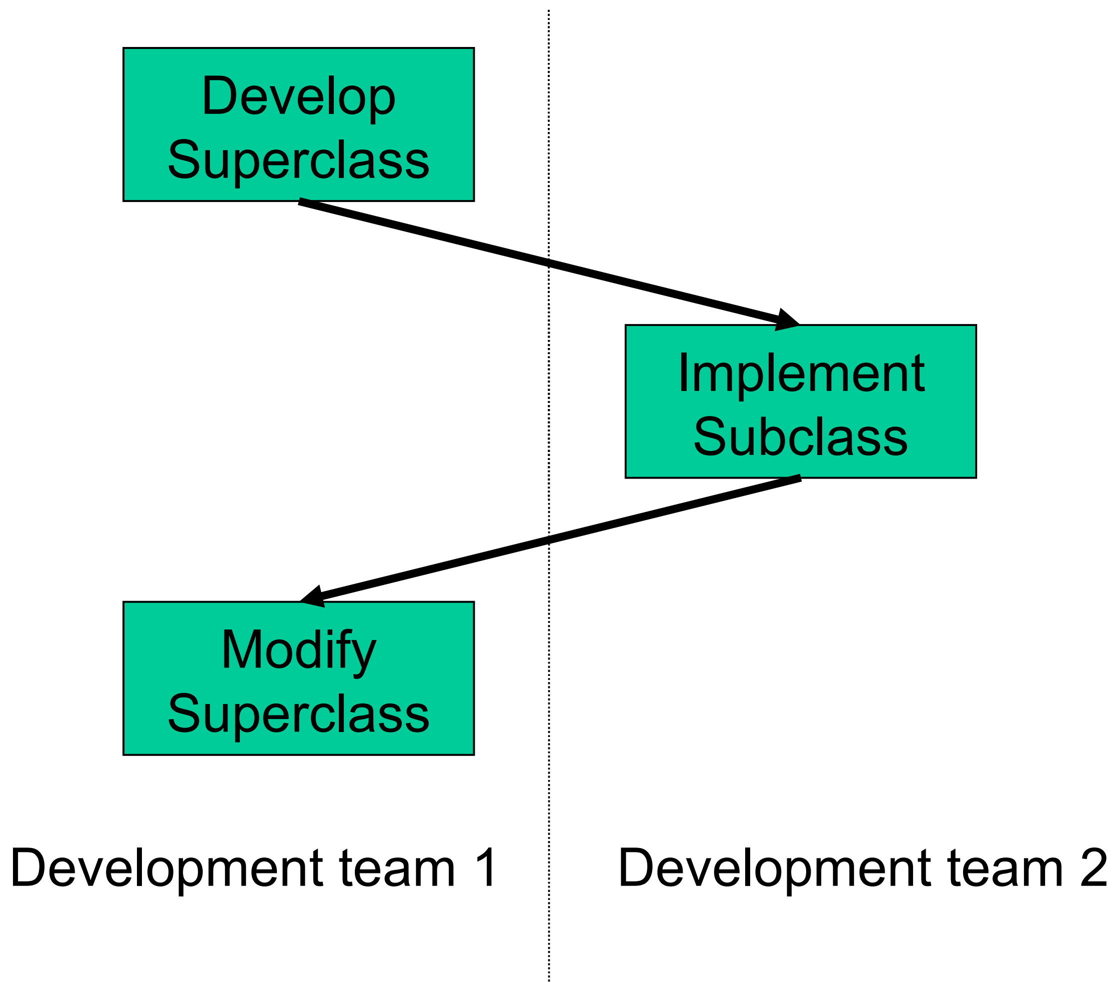
```
public class Superclass {
   protected int a, b, c;
   // invariant:  a == b+c ;
}

public class Subclass
             extends Superclass {
    public incrementC(){ c++; }
}
```

Fragile Baseclass (FBC):

- Software, including reusable components, is often modified over time:
  - Maintenance
  - Bugfixing
  - Reengineering
- Subclasses can be affected by changes to superclasses

How should we apply inheritance to make our code robust against revisions of superclasses?



Example: (Fragile baseclass)

```
class Bag {
  …
  int getSize( ) {
    … // count elements
  }

  void insert( Object o )
    { … }

  void insertAll( Object[ ] arr )    {
    for( int i=0; i < arr.length; i++ )
      insert( arr[ i ] );
  }
}
```

```
class CountingBag extends Bag {
  int size; // invariant: size==super.getSize();

  int getSize( )   { return size; }

  void insert( Object o )
    { super.insert( o ); size++; }
}
```

```
Object[ ]  oa = …  // 5 elements
CountingBag cb = new CountingBag( );
cb.insertAll( oa );
System.out.println( cb.getSize( ) );
```

Works well until implementation of insertAll is modified, e.g. by inserting all array elements without using insert.

Hints to avoid FBC-problem:

- Override all methods that might break the invariant in the subclass.
- In subclass implementation, only rely on the superclass specification, not on its implementation.

Inheritance and Delegation:

If attribute access is realized by getter and setter methods, subclass implementation can be achieved by delegation:

- No inheritance
- Each object has a reference to superclass object
- Methods that are not declared locally are delegated to superclass object.

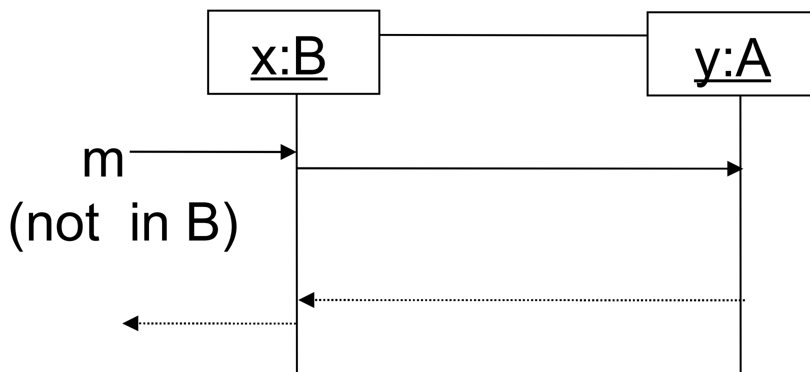
Thus, the basic idea is:

class composition → object composition

We first explain forwarding, an object composition technique weaker than delegation.

Explanation: (forwarding)

Forwarding means to resend messages that are not directly handled by the current object x to an object y that is referenced by x. Object y can be used to provide the functionality that is otherwise inherited by x.



Lemma: (forwarding transformation)

If methods and constructors used in superclasses do not use methods that are overridden in subclasses and if objects only have private fields, then subclasses can be implemented by forwarding instead of inheritance. ■

Remark:

- Using forwarding is less powerful than inheritance.
- Checking the criterion of the lemma helps to detect error-prone inheritance situations (see example on next slide). ■

Transformation of inheritance to forwarding:

- Introduce an interface type for each class.
- Classes implement corresponding interfaces.
- Subclasses do no longer inherit; they get an additional attribute for referencing an object of the former superclass.
- Constructors initialize reference to former super-class object.
- For inherited methods that are not overridden in subclasses a forwarding method is defined.

Example: (Usage of overridable methods)

```
class Oberklasse {
  String a;

  Oberklasse(){
    a = "aha";
    m();
  }
  void m(){
    System.out.print("Laenge a:"+a.length());
  }
}

class Unterklasse extends Oberklasse {
  String b;

  Unterklasse(){
    b = "boff";
    m();
  }
  void m(){
    System.out.print("Laenge b:"+b.length());
  }
}

class KonstruktorProblemTest {
  public static void main( String[] args ){
    new Unterklasse();
  }
}
```

Example: (Forwarding transformation)

```
class A {
  private int i = 5;

  A(){}

  public int m( int ip ) {
    return plusi(ip) + twice(ip);
  }
  public int plusi( int ip ) {
    return ip + i;
  }
  private int twice( int ip ) {
    return 2*i;
  }
}

class B extends A {
  private int i = 10;

  B(){}
}

public class DelegForwInh {
  public static void main(String[] args){

    A a = new B();
    System.out.println(a.m(1));

  }
}
```

```
interface AItf {
  int m( int ip );
  int plusi( int ip );
}

class A implements AItf {
  private int i = 5;

  A(){}

  public int m( int ip ) {
    return plusi(ip) + twice(ip);
  }
  public int plusi( int ip ) {
    return ip + i;
  }
  private int twice( int ip ) {
    return 2*i;
}  }

interface BItf extends AItf {}

class B implements BItf {
  private AItf inhPart;
  private int i = 10;

  B(){ inhPart = new A(); }

  public int m( int ip ){
    return inhPart.m(ip);
  }
  public int plusi( int ip ) {
    return inhPart.plusi(ip);
  }
}
```

The restrictions given in the Lemma are necessary:

If used methods are overridden in subclasses, forwarding does not work. Here is a subclass of B that overrides plusi used in m:

```
class A { ... }  // as above

class B extends A { ... }  // as above

class C extends B {
  private int i = 20;

  C(){}

  public int plusi( int ip ) {
    return ip + i;
  }
}

public class DelegForwInh {
  public static void main( String[] args ) {

    A a = new B();
    System.out.println(a.m(1));

    a = new C();
    System.out.println(a.m(1));
  }
}
```

Transformation yields the following semantically different program:

```
interface AItf {
  int m( int ip );
  int plusi( int ip );
}

class A implements AItf { ... } // as above

interface BItf extends AItf {}

class B implements BItf { ... } // as above

interface CItf extends BItf {}

class C implements CItf {
  private BItf inhPart;
  private int i = 20;

  C(){ inhPart = new B(); }

  public int m( int ip ){
    return inhPart.m(ip);
  }
  public int plusi( int ip ) {
    return ip + i;
  }
}

public class DelegForwInh1 {
   ...
   a = new C();
   System.out.println(a.m(1));
}
```

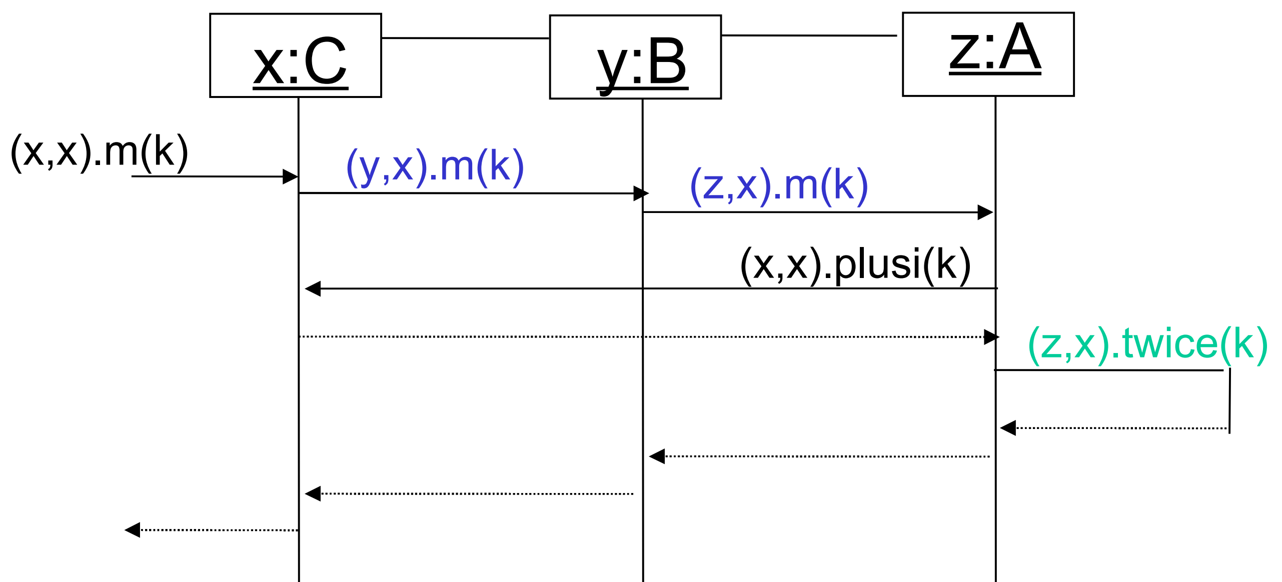
To be able to use object composition instead of inheritance, a message passing technique is needed that is more powerful (and complex) than forwarding:

Explanation: (delegation)

Method passing by ***delegation*** distinguishes three kinds of method invocations:

- statically bound invocations
- dynamically bound invocations
- *delegating* invocations

To handle delegating invocations, each invocation has an additional implicit parameter orig for the original sender of the message. In dynamically bound invocations orig is used instead of this.



Example: (delegating transformation)

We transform the above example using delegation.
As Java does not support a second implicit parameter, we make it explicit.

```
interface AItf {
  int m( AItf orig, int ip );
  int plusi( AItf orig, int ip );
}

class A implements AItf {
  private int i = 5;

  A(){}

  public int m( AItf orig, int ip ) {
    return orig.plusi(orig,ip)
           + this.twice(orig,ip);
  }
  public int plusi( AItf orig, int ip ){
    return ip + i;
  }
  private int twice( AItf orig, int ip ){
    return 2*i;
  }
}
```

```
interface BItf extends AItf {}

class B implements BItf {
  private AItf inhPart;
  private int i = 10;

  B(){ inhPart = new A(); }

  public int m( AItf orig, int ip ){
    return inhPart.m(orig,ip);
  }
  public int plusi( AItf orig, int ip ){
    return inhPart.plusi(orig,ip);
  }
}

interface CItf extends BItf {}

class C implements CItf {
  private BItf inhPart;
  private int i = 20;

  C(){ inhPart = new B(); }

  public int m( AItf orig, int ip ){
    return inhPart.m(orig,ip);
  }
  public int plusi( AItf orig, int ip ){
    return ip + i;
  }
}
```

```
public class DelegForwInh2 {
  public static void main( String[] s ){
    AItf a = new B();
    System.out.println(a.m(a,1));
    a = new C();
    System.out.println(a.m(a,1));
  }
}
```

Remark:

- The explicit formulation of delegation demonstrates as well the often hidden complexity of inheritance.
- Delegation is a general design pattern with different applications. Replacing inheritance by object composition and delegation is certainly not the usual application.

2.3 Semantics of OO Languages

To illustrate semantics definitions for object-oriented languages, we consider a sequential, tiny object-oriented (almost) Java subset (*StoJas*). We define:

- Context-free syntax
- Context conditions
- Static semantics
- Dynamic semantics

Syntax of *StoJas*:

```
Program     = ClassDecl+
ClassDecl   = class ClassId  [ extends ClassId ]
                { FieldDecl* MethodDecl* }
FieldDecl   =   TypeId  FieldId ;
MethodDecl  =  TypeId MethodId(TypeId p) { Statement }
TypeId      =  boolean | int | ClassId
Statement   =  TypeId VarId;
            |  VarId = VarId . FieldId ;
            |  VarId . FieldId = Exp ;
            |  VarId = ( ClassId ) Exp ;
            |  VarId = new ClassId( );
            |  Statement  Statement
            |  if( Exp ){ Statement }else{ Statement }
            |  while( Exp ){ Statement }
            |  VarId = VarId . MethodId( Exp );
            |  VarId = super.ClassId@MethodId( Exp );
Exp         =  Int
            |  Bool
            |  null
            |  VarId
            |  UnOp Exp
            |  Exp BinOp Exp
```

ClassId, FieldId, MethodId, VarId are sets of identifiers where VarId contains `this`.

UnOp and BinOp are suitable operators yielding boolean- or int-values.

A StoJas-program contains a special class Object.

<u>Context Conditions of *StoJas*:</u>

The context conditions of StoJas are those of Java. In particular, the following type conditions have to hold:

- The type of the right-hand side of an assignment is a subtype of the type of the left-hand side.
- The type of the actual parameter expression is a subtype of the formal parameter type of the method.
- Expressions in conditions are of type **boolean**.

In addition, fields in subclasses must have an identifier different from all inherited fields and programs must satisfy the conditions described by the following example:

Examples: (StoJas-program)

```
class Object {}

class Main {
  int main( int p ) {
    List l;
    l = new List();
    l = l.addElems(p);
  }

}

class List {
  int elem;
  List next;

  List addElems( int p ) {
    while( p>0 ) {
      res = new List();
      res.elem = p;
      res.next = this;
      p = p-1;
    }
  }
}
```

As illustrated by the example, a StoJas program must have a startup class `Main` without fields and with a method `main`.

Each method m has an implicitly declared local variable `res` of m's return type. The value of `res` is returned on exit from m.

Static semantics of *StoJas*:

The semantics defines how programs are executed (*operational semantics*). For each statement c, it describes how c modifies the current state.

The ***static semantics*** describes how states look like, that is, the data types and functions to represent states. In addition, it describes needed auxiliary funtions.

```
Value   =   Bool
          |   Int
          |   null
          |   ClassId ObjId

init : TypeId -> Value    //  initial value of a type
```

The set of instance variables and methods:

InstVar = { <c,o,f> | f is a field of class c, o an ObjId }

Method = { c@m | m is a method declared in class c }

```
rtyp :   Method  -> TypeId         // return type of method

body : Method  ->  Statement   // body of a method

_._  :  Value x FieldId  ->  InstVar

    v.f  =    if   v = (c,o)  and  f is a valid field in c
                then   <c,o,f>
                 else   undefined
```

Stores/Heaps:

The heap of objects/object store is modelled by an abstract datatype with sort Store and operations:

write : Store x InstVar x Value → Store

read : Store x InstVar → Value

new : Store x ClassId → Value

alloc : Store x ClassId → Store

isAllocated : Store x Value → Bool

The properties of Store can be specified by axioms. Examples:

$iv1 \neq iv2 \Rightarrow$ read(write(st,iv1,x), iv2) = read(st,iv2)

read(write(st,iv,x), iv) = x

read(alloc(st,T), iv) = read(st, iv)

States:

A state provides

- the values of variables and parameters (stack)
- the states of the objects (heap)

We model states as functions:

State = (VarId U {$}) --> (Value U Store)

such that S(x) is a store iff x=$.

Notation: If S is a state, x,y are variables, v is a value, and e is an expression, we write:

- S(x) to read the value of x in S,
- S($) to denote the current store,
- S[x:= v] to update S at x by v

The evaluation of an expression e in state S is denoted by S(e), in particular, S(x+y) = S(x) + S(y).

Furthermore, we define/assume:

Type = TypeId | NullT

$\leq$: Type x Type

S $\leq$ T $\Leftrightarrow$ (S is NullT and T is a ClassId)
or (S and T are ClassIds and S is a subclass of T)

stype: Exp x StmtOcc → Type

// denotes the type of an expression in a statement

ftype: FieldId x ClassId → Type

ftype(f,C) = if f is a valid field in C
then range type of f else undefined

Notice:

D is a subclass of C and f is a valid field in C
==> ftype(f,D) = ftype(f,C)

vis: StmtOcc → Pow(VarId)

vis(STM) ⊆ VarId denotes the visible variables in STM

impl: MethodId x ClassId → Method

impl(m,C) = if m is valid method in C
then B@m where B = C or
B is the nearest superclass
with a declaration for m
else undefined

Notice:

If impl(m,C) is defined and D is a subclass of C

then impl(m,D) is defined

type : Value → Type

type(v) = **boolean** if v is a bool value

type(v) = **int** if v is an integer

type(null) = NullT

type((C,o)) = C

Definition: (well-typed state)

Let

- STM be a statement of a *StoJas*-program P,
- SV: VarId → Value be a variable state, and
- OS: Store be an object store for P.

SV is ***well-typed*** w.r.t. STM iff for all x in vis(STM):

type(SV(x)) ≤ stype(x,STM)

OS is ***well-typed*** iff for all <c,o,f> in InstVar:

type(read(OS,<c,o,f>)) ≤ ftype(f,c)

A state S = (SV,OS) is ***well-typed*** w.r.t. STM
iff SV is well-typed w.r.t. STM and OS is well-typed.

Lemma: (well-typed expression evaluation)

If S is a well-typed state w.r.t. STM and
e is a well-typed expression occurring in STM, then

type(S(e)) ≤ stype(e,STM)

Proof:

(as exercise)

Dynamic semantics of *StoJas*:

We describe the semantics by a predicate (*judgement*) of the form

S : *stmt* → SQ

that expresses the fact that execution of *stmt* starting in state S terminates in state SQ.

Here are the rules for *StoJas*:

$$\frac{S(y) \neq null}{S\text{: x=y.a; } \rightarrow S[\text{ x:=read(S(\$),S(y).a) }]}$$

$$\frac{S(x) \neq null}{S\text{: x.a=e; } \rightarrow S[\text{ \$:=write(S(\$),S(x).a,S(e)) }]}$$

$$\frac{type(S(e)) \leq C}{S\text{: x=(C) e; } \rightarrow S[\text{ x:=S(e) }]}$$

$$\frac{}{S\text{: x=new T(); } \rightarrow S[\text{ x:=new(S(\$),T), \$:=alloc(S(\$),T) }]}$$

$$\frac{S\text{: s1 } \rightarrow SQ \;,\; SQ\text{: s2 } \rightarrow SR}{S\text{: s1 s2 } \rightarrow SR}$$

$$\frac{S(e) = true \;,\; S\text{: s1 } \rightarrow SQ}{S\text{: if(e)\{s1\}else\{s2\} } \rightarrow SQ}$$

$$\frac{S(e) = false \;,\; S\text{:s2 } \rightarrow SQ}{S\text{: if(e)\{s1\}else\{s2\} } \rightarrow SQ}$$

$$\frac{S(e)=\text{false}}{S: \text{while}(e)\{s\} \rightarrow S}$$

$$\frac{S(e)=\text{true},\ S{:}s \rightarrow SQ,\ SQ{:}\text{while}(e)\{\ s\ \} \rightarrow SR}{S: \text{while}(e)\{s\} \rightarrow SR}$$

$$\frac{}{S: T\ x; \rightarrow S[x{:=}\text{init}(T)]}$$

$$\frac{S(y) \neq \text{null},\ MY = \text{impl}(\ \text{type}(S(y)),\ m),\quad S[\text{this}{:=}S(y), p{:=}S(e), \text{res}{:=}\text{init}(\text{rtyp}(MY))]{:}\text{body}(MY) \rightarrow SQ}{S: x = y.m(\ e\ ); \rightarrow S[\ x{:=}SQ(\text{res}),\ \${:=}SQ(\$)\ ]}$$

$$\frac{S[p{:=}S(e), \text{res}{:=}\text{init}(\text{rtyp}(C@m))] : \text{body}(C@m) \rightarrow SQ}{S: x = \text{super}.C@m(\ e\ ); \rightarrow S[\ x{:=}SQ(\text{res}),\ \${:=}SQ(\$)\ ]}$$

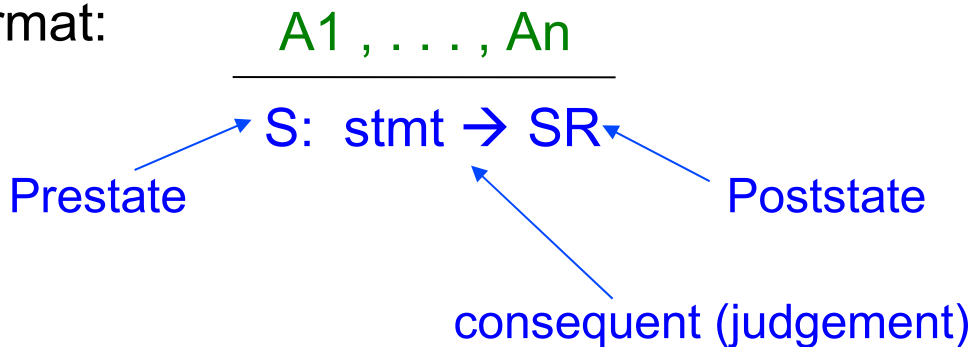
„Execution“ starts in the initial state S0 and executes:

output = mainobj. main(input);

where

- S0 is well-typed
- S0(input) is the input parameter
- type(S0(mainobj)) = Main
- isAllocated(S0($), S0(mainobj)) = true

Rule format:



SP: s → SQ can be derived by the rules iff
there is a rule R and a substitution Φ such that

- Φ(consequent(R)) = SP: s → SQ
- the boolean expressions in the antecedent are satisfied under Φ
- for all judgements A in the antecedent of R Φ(A) can be derived (in a finite number of steps).

Remarks:

- Execution means searching derivations.
- A derivation does not exist in case of nontermination or because the derivation leads to an antecedent to which no rule can be applied.
- The form of operational semantics is called a *big-step* or *natural semantics*.