Classes, Objects, Subtyping

Objects

- What objects are:
  - Aggregate structures that combine data (fields) with computation (methods)
  - Fields have public/private qualifiers (can model ADTs)
- Need special support in many compilation stages:
  - Type checking
  - Static analysis and optimizations
  - Implementation, run-time support
- Features:
  - inheritance, subclassing, polymorphism, subtyping, overriding, overloading, dynamic dispatch, abstract classes, interfaces, etc.

Inheritance

- Inheritance = mechanism that exposes common features of different objects
- Class B extends class A = "B has the features of A, plus some additional ones", i.e., B inherits the features of A
  - B is subclass of A; and A is superclass of B

```
class Point {
        float x, y;
        float getx(){ ... };
        float gety(){ ... };
}

class ColoredPoint extends Point {
        int color;
        int getcolor(){ ... };
}
```


Single vs. Multiple Inheritance

- Single inheritance: inherit from at most one other object (Java)
- Multiple inheritance: may inherit from multiple objects (C++)

```
class A {
 int a;
 int geta(){…};
}
```

```
class B {
 int b;
 int getb(){…};
}
```

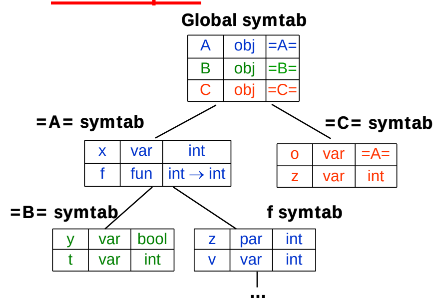
```
class C : A, B {
 int c;
 int getc(){…};
}
```

Inheritance and Scopes

- How do objects access fields and methods of:
  - Their own?
  - Their superclasses?
  - Other unrelated objects?
- Each class declaration introduces a scope
  - Contains declared fields and methods
  - Scopes of methods are sub-scopes
- Inheritance implies a hierarchy of class scopes
  - If B extends A, then scope of A is a parent scope for B

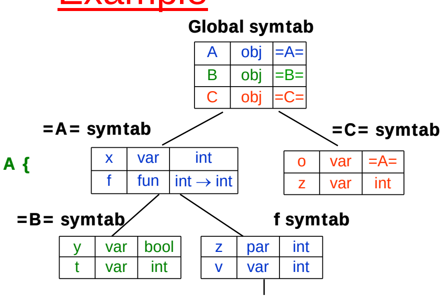
Example

```
class A {
  int x;
  int f(int z) {
    int v; ...
  }
}
class B extends A {
  bool y;
  int t;
}
class C {
  A o;
  int z;
}
```



Example

```
class A {
  int x;
  int f(int z) {
    int v; ...
  }
}
class B extends A {
  bool y;
  int t;
}
class C {
  A o;
  int z;
}
```



Class Scopes

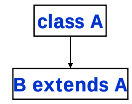
- Resolve an identifier occurrence in a method:
  - Look for symbols starting with the symbol table of the current block in that method
- Resolve qualified accesses:
  - Accesses o.f, where o is an object of class A
  - Walk the symbol table hierarchy starting with the symbol table of class A and look for identifier f
  - Special keyword this refers to the current object, start with the symbol table of the enclosing class

Class Scopes

- Multiple inheritance:
  - A class scope has multiple parent scopes
  - Which should we search first?
  - Problem: may find symbol in both parent scopes!
- Overriding fields:
  - Fields defined in a class and in a subclass
  - Inner declaration shadows outer declaration
  - Symbol present in multiple scopes

Inheritance and Typing

- Classes have types
  - Type is Cartesian product of field and method types
  - Type name is the class name
- What is the relation between types of parent and inherited objects?

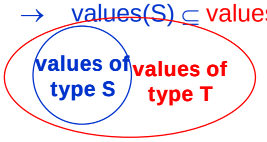


- Subtyping: if class B extends A then
  - Type B is a subtype of A
  - Type A is a supertype B
- Notation: B <: A

Subtype ≈ Subset

"A value of type S may be used wherever a value of type T is expected"

$S <: T \quad \rightarrow \quad values(S) \subset values(T)$



Subtype Properties

- If type S is a subtype of type T (S <: T), then:
  a value of type S may be used wherever a value of type T is expected (e.g., assignment to a variable, passed as argument, returned from method)

```
Point x;
ColoredPoint y;
x = y;
```

ColoredPoint <: Point

subtype (ColoredPoint) **supertype** (Point)

- Polymorphism: a value is usable as several types
- Subtype polymorphism: code using T's can also use S's; S objects can be used as S's or T's.

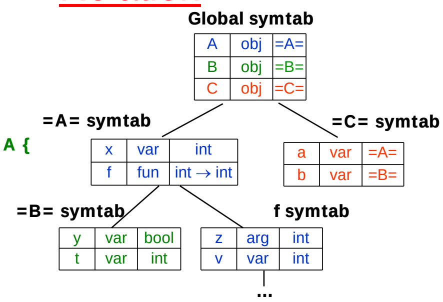
Assignment Statements (Revisited)

$$\frac{A,\ id{:}T\ \vdash E : T}{A,\ id{:}T\ \vdash id = E : T}\ \text{(original)}$$

$$\frac{A,\ id{:}T\ \vdash E : S \quad \text{where } S <: T}{A,\ id{:}T\ \vdash id = E : T}\ \text{(with subtyping)}$$

How To Test the SubType Relation

```
class A {
  int x;
  int f(int z) {
    int v; ...
  }
}
class B extends A {
  bool y;
  int t;
}
class C {
  A a;
  B b;
  ...
  a = b;
}
```

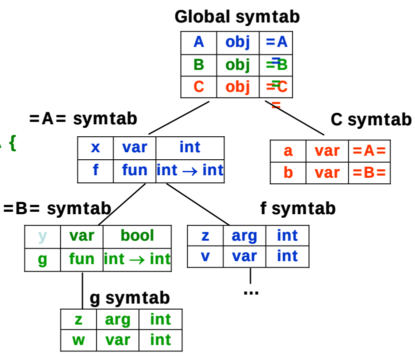


Implications of Subtyping

- Type of object may be different from the declared type of reference
  - Can be the declared class or any subclass
  - Precise types of objects known only at run-time
- Problem: overriden fields / methods
  - Declared in multiple classes in hierarchy. Don't know statically which declaration to use at compile time
  - Java solution:
    - statically resolve fields using declared type of reference; no field overriding
    - dynamically resolve methods using the object's type (dynamic dispatch); in support of static type checking, a method m overrides m' only if the signatures are "nearly" identical --- the same number and types of parameters, and the return type of m a subtype of the return type of m'

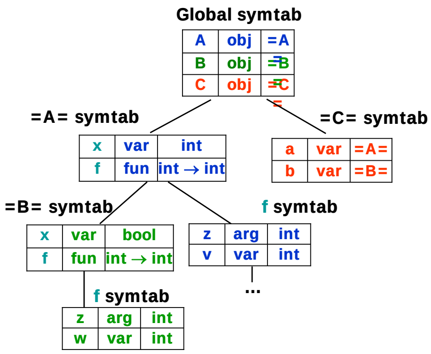
Example

```
class A {
  int x;
  int f(int z) {
    int v; ...
  }
}
class B extends A {
  bool y;
  int g(int z) {
    int w; ...
  }
}
class C {
  A a = new B();
  B b = new B();
  ... a.x ...
  ... b.y ...
}
```



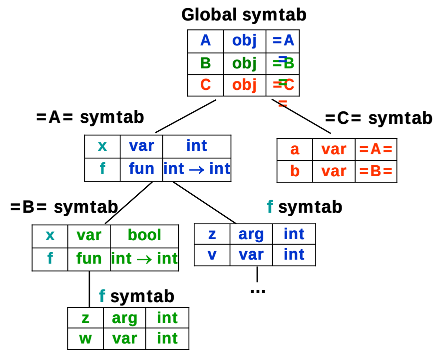
Example

```
class A {
   int x;
   int f(int z) {
      int v; …
   }
}
class B extends A {
   bool x;
   int f(int z) {
      int w; …
   }
}
class C {
   A a = new B();
   B b = new B();
   … a.x …
   … b.x …
}
```



Example

```
class A {
   int x;
   int f(int z) {
      int v; …
   }
}
class B extends A {
   bool x;
   int f(int z) {
      int w; …
   }
}
class C {
   A a = new B();
   B b = new B();
   … a.f(1) …
   … b.f(1) …
}
```



Objects and Typing

- Objects have types
  - … but also have implementation code for methods
- ADT perspective:
  - Specification = typing
  - Implementation = method code, private fields
  - Objects mix specification with implementation
- Can we separate types from implementation?

Interfaces

- Interfaces are pure types; they don't give any implementation

implementation

```
class MyList implements List {
  private int len;
  private Cell head, tail;

  public int length() {…};
  public List append(int d) {…};
  public int first() {…} ;
  public List rest() {…};
}
```

specification

```
interface List {
  int length();
  List append(int d);
  int first();
  List rest();
}
```

Multiple Implementations

- Interfaces allow multiple implementations

```
interface List {
   int length();
   List append(int);
   int first();
   List rest(); }
```

```
class SimpleList implements List {
   private int data;
   private SimpleList next;
   public int length()
      { return 1+next.length() } ...
```

```
class LenList implements List {
      private int len;
      private Cell head, tail;
      private LenList() {...}
      public List append(int d) {...}
      public int length() { return len; }
      ...
```


Implementations of Multiple Interfaces

```
interface A {
        int foo();
        }
interface B {
        int bar();
        }
class AB implements A, B {
        int foo(){ … }
        int bar(){ … }
        }
```


Subtyping vs. Subclassing

- Can use inheritance for interfaces
  – Build a hierarchy of interfaces

  interface A {…}
  interface B **extends** A {…}

B <: A

- Objects can implement interfaces

  class C **implements** A {…}

C <: A

- Subtyping: interface inheritance
- Subclassing: object (class) inheritance
  – Subclassing implies subtyping

Abstract Classes

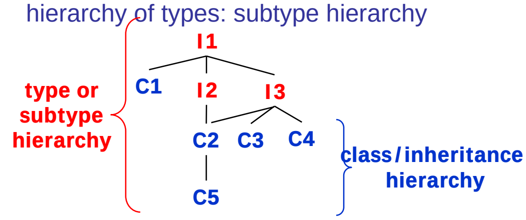
- Classes define types and some values (methods)
- Interfaces are pure object types
- Abstract classes are halfway:
  – define some methods
  – leave others unimplemented
  – no objects (instances) of abstract class

Subtypes in Java

interface I_1 extends I_2 { … }	class C implements I { … }	class C_1 extends C_2
I_2	I	C_2
I_1	C	C_1
I_1 <: I_2	C <: I	C_1 <: C_2

Subtype Hierarchy

- Introduction of subtype relation creates a hierarchy of types: subtype hierarchy



Type-checking

- Problem: what are the valid types for an object?
- Subsumption rule connects subtyping relation and ordinary typing judgements

$$\frac{A \vdash E : S \quad S <: T}{A \vdash E : T} \qquad S <: T \rightarrow \text{values}(S) \subseteq \text{values}(T)$$

- "If expression E has type S, it also has type T for every T such that S <: T"

Type-checking

- Rules for checking code must allow a subtype where a supertype was expected
- Old rule for assignment:

$$\frac{id : T \in A \quad A \vdash E : T}{A \vdash id = E : T}$$

What needs to change here?

Type-checking Overview

- Rules for checking code must allow a subtype where a supertype was expected
- New rule for assignment:

$$\frac{A \vdash E : T_p \quad T_p <: T \quad id : T \in A}{A \vdash id{=}E : T} = \frac{A \vdash E : S \quad S <: T}{A \vdash E : T} + \frac{id : T \in A \quad A \vdash E : T}{A \vdash id = E \; : T}$$

Type-checking Code

```
class Assignment extends ASTNode {
  Variable var; ExprNode E;
  Type typeCheck() {
      Type Tp = E.typeCheck();
      Type T = var.getType();
      if (Tp.subtypeOf(T)) return T;
      else throw new TypecheckError(E); }}
```

$$\frac{A \vdash E : T_p \qquad T_p <: T \qquad id : T \in A}{A \vdash id{=}E : T}$$

Issues

- When are two object/record types identical?
  - Do struct foo { int x,y; } and struct bar { int x,y; } have the same type?
- We know inheritance (i.e. adding methods and fields) induces subtyping relation
- Issues in the presence of subtyping:
  1. Types of records with object fields
     class C1 { Point p; }   class C2 { ColoredPoint p; }
  2. Is it safe to allow fields to be written?
  3. Types of functions (methods)
     Point foo(Point p)   ColoredPoint bar(ColoredPoint p)

Type Equivalence

- Types derived with constructors have names
- When are record types equivalent?
- When they have the same fields (i.e. same structure)?
  struct point { int x,y; } = struct edge { int n1, n2; } ?
- ... or only when they have the same names?
  - Types with the same structure are different if they have different names

Type Equivalence

- Name equivalence: types are equal if they are defined by the same type constructor expression and bound to the same name
  - C/C++ example:

```
struct foo { int x; };
struct bar { int x; }
```

struct foo ≠ struct bar

- Structural equivalence: two types are equal if their constructor expressions are equivalent
  - C/C++ example:

```
typedef struct foo t1[ ];
typedef struct foo t2[ ];
```

t1 = t2

Named vs. Structural Subtyping

- Name equivalence of types (e.g. Java): direct subtypes explicitly declared; subtype relationships inferred by transitivity
- Structural equivalence of types (e.g., Modula-3): subtypes inferred based on structure of types; extends declaration is optional
- Java: still need to check explicit interface declarations similarly to structural subtyping

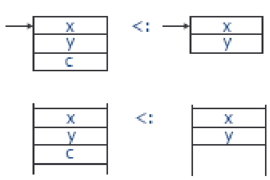
The Subtype Relation

For records:

S <: T

{int x; int y; int color; } <: { int x; int y; } ?

- Heap-allocated:
- Stack allocated



Width Subtyping for Records

- Example:

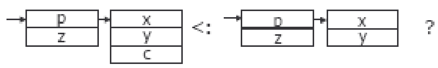
{int x; int y; int color; } <: { int x; int y; }

- General rule:

$$\frac{n \le m}{A \vdash \{a_1: T_1, \ldots, a_m: T_m\} <: \{a_1: T_1, \ldots, a_n: T_n\}}$$

Object Fields

- Assume fields can be objects
- Subtype relations for individual fields
- How does it translate to subtyping for the whole record?
- If ColoredPoint <: Point, allow
  { ColoredPoint p; int z; } <: { Point p; int z; } ?

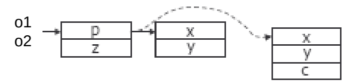


Field Invariance

- Try { p: ColoredPoint; int z; } <: { p: Point; int z; }

```
class C1 { Point p; int z; }
class C2 { ColoredPoint p; int z; }
C2 o2 = new C2();
C1 o1 = o2;
o1.p = new Point( );
o2.p.c = 10;
```

- Mutable (assignable) fields must be type invariant!



Immutable Record Subtyping

- Rule: corresponding immutable fields may be subtypes; exact match not required

$$\frac{A \vdash T_i <: T_i' \quad (i \in 1..n)}{A \vdash \{a_1: T_1 \ldots a_n: T_n\} <: \{a_1: T_1' \ldots a_n: T_n'\}}$$

$$\frac{n \le m}{A \vdash \{a_1: T_1, \ldots, a_m: T_m\} <: \{a_1: T_1, \ldots, a_n: T_n\}}$$

Signature Conformance

- Subclass method signatures must conform to those of superclass
  - Argument types
  - Return type
  - Exceptions
  - How much conformance is really needed?

[redacted]

Example 1

- Consider the program:
  interface List { List rest(int); }
  class SimpleList implements List
  { SimpleList rest(int); }
- Is this a valid program?
- Is the following subtyping relation correct?

{ rest: int→SimpleList } <: { rest: int→List }

int→SimpleList <: int→List ?

Example 2

- Consider the program:
  class Shape { int setLLCorner(Point p); }
  class ColoredRectangle extends Shape
  { int setLLCorner(ColoredPoint p); }
- Legal in language Eiffel
- Is this safe?

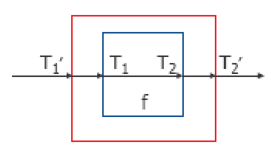
ColoredPoint → int <: Point → int ?

Function Subtyping

- From definition of subtyping: F: $T_1 \to T_2$ <: F': $T_1' \to T_2'$ if a value of type $T_1 \to T_2$ can be used wherever $T_1' \to T_2'$ is expected
- Requirement 1: whenever result of F' is used, result of F can also be used
  - Implies T_2 <: T_2'
- Requirement 2: any argument to F' must be a valid argument for F
  - Implies T_1' <: T_1

General Rule

- Function subtyping: $T_1 \to T_2$ <: $T_1' \to T_2'$
- Consider function f of type $T_1 \to T_2$:



Contravariance/Covariance

- Function argument types may be contravariant
- Function result types may be covariant

$$\frac{T_1' <: T_1 \qquad T_2 <: T_2'}{T_1 \rightarrow T_2 \;<:\; T_1' \rightarrow T_2'}$$

- Java is conservative!

{ rest: int→SimpleList } <: { rest: int→List }

Unification

- Some rules more problematic: if
- Rule:

$$\frac{A \vdash E : bool \qquad A \vdash S_1 : T \qquad A \vdash S_2 : T}{A \vdash \text{if (E) } S_1 \text{ else } S_2 : T}$$

- Problem: if S_1 has type T_1, S_2 has type T_2. Old check: $T_1 = T_2$. New check: need type T. How to unify T_1, T_2?
- Occurs in Java: ?: operator

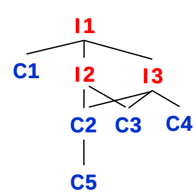
General Typing Derivation

$$\frac{A \vdash E : bool \qquad \dfrac{A \vdash S_1 : T_1 \quad T_1 <: T}{A \vdash S_1 : T} \qquad \dfrac{A \vdash S_2 : T_2 \quad T_2 <: T}{A \vdash S_2 : T}}{A \vdash \text{if (E) } S_1 \text{ else } S_2 : T}$$

How to pick T ?

Type unification

- Unified type is least common ancestor in type hierarchy
- Least common ancestor is also known as least upper bound (lub)
- What if lub does not exist?



What is lub(C2,C3)?

Conclusions

- Adding objects and subtyping can complicate type checking
- No consensus yet on subtyping
  - Many design choices
  - What is best in practice?