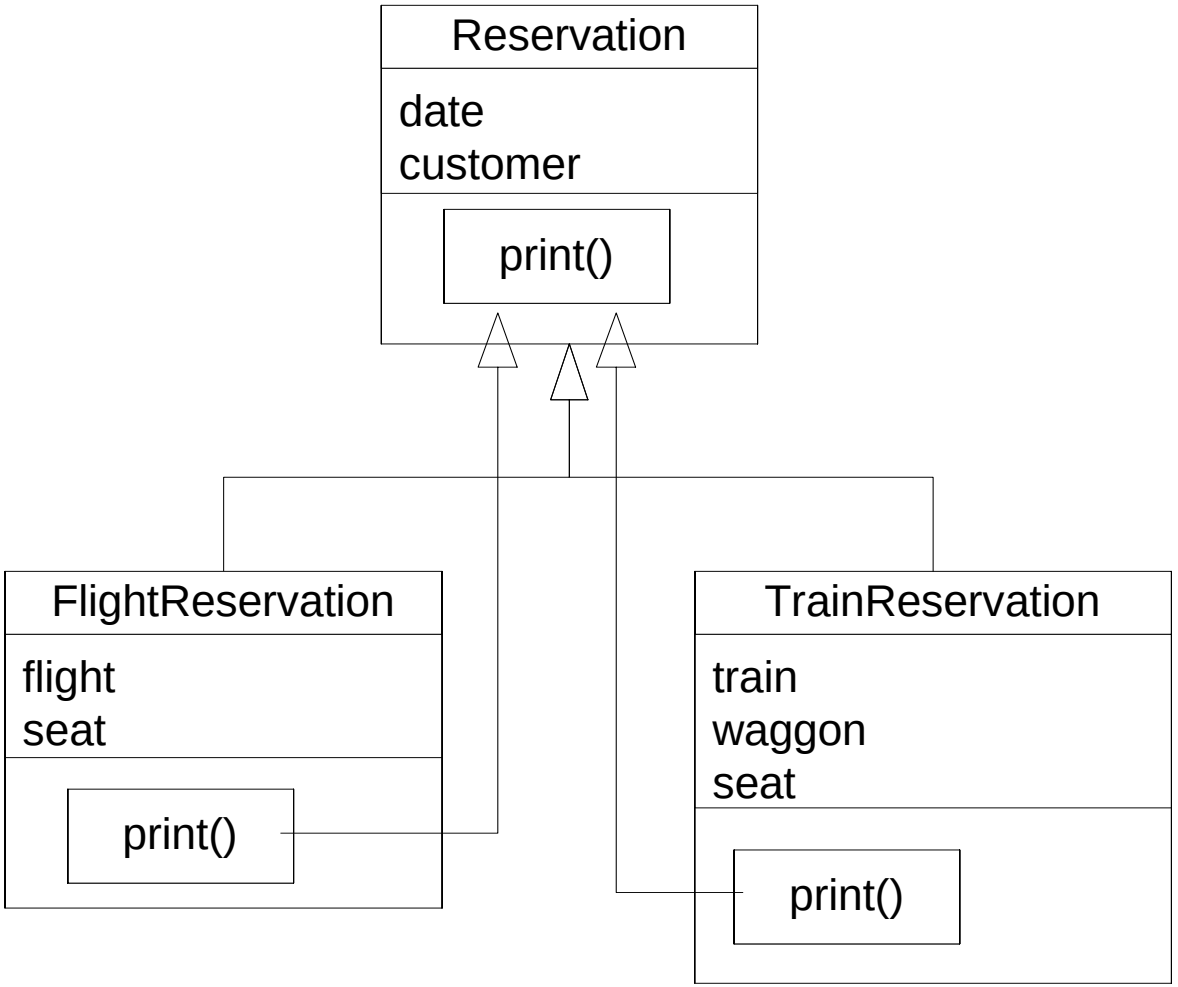
’Subtyping’ for behaviour?



’Subtyping’ for behaviour – the inner style

```
class Reservation {
   date . . . ;  customer . . . ;
   void print()  {
      // print Date and Customer
      inner;
   }
}

class FlightReservation
   extends Reservation {
      flight . . .;  seat . . .;
      void print  extended {
         // print flight and seat
         inner;
   }
}
```

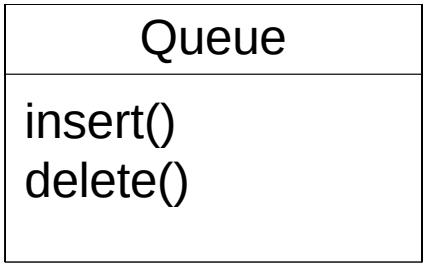
’Subtyping’ for behaviour – the super style

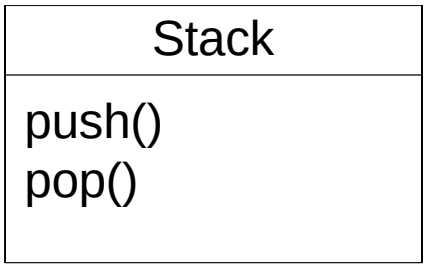
```
class Reservation {
   date . . . ;  customer . . . ;
   void print()  {
      // print date and Customer
   }
}

class FlightReservation
   extends Reservation {
      flight. . .;  seat. . .;
      void print {
         super.print();
         // print Flight and Seat
    }
}
```

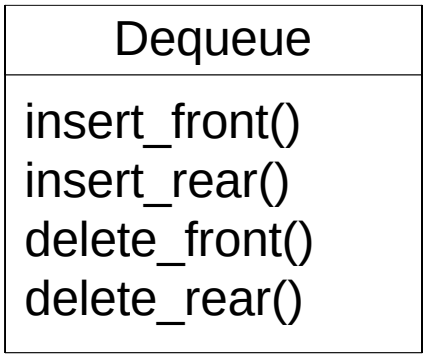
- super.print() == (Reservation)this.print()
- Does the inner style give ’behavioral compatibility’?
- What if we turn print into a static method?

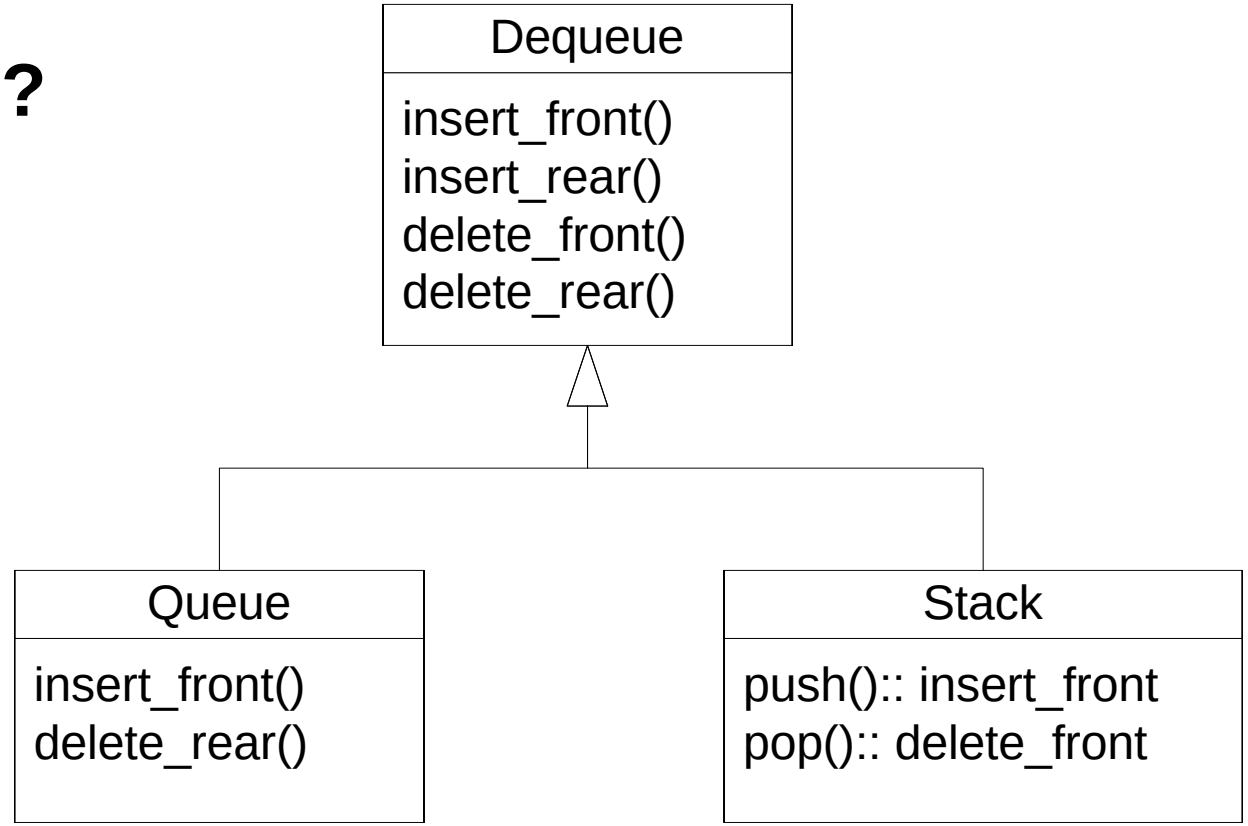
```
class Point { int x, y; }

is equivalent to the declaration:

class Point { int x, y;
   Point() { super(); }
}
```

Subtyping = subclassing??



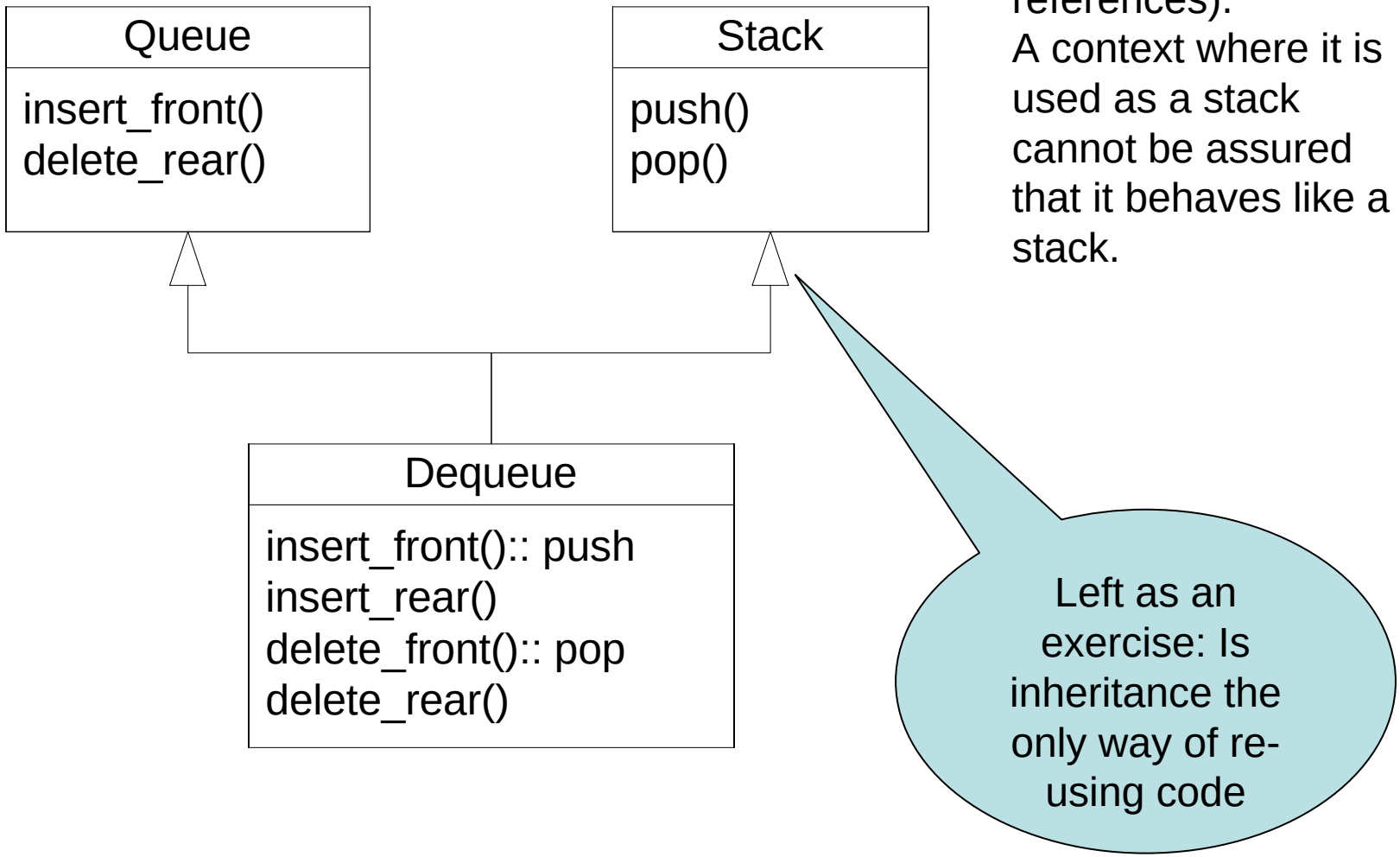







```
Dequeue d; Stack s; Element e;
void f(Dequeue dp, Element ep) {
  dp.insert_front(ep); dp.insert_rear(ep) }
...
f(s, e)
```

The opposite any better?



Can be substituted for both a Queue and a Stack (via different references).
A context where it is used as a stack cannot be assured that it behaves like a stack.

Subtyping = subclassing (Smalltalk)

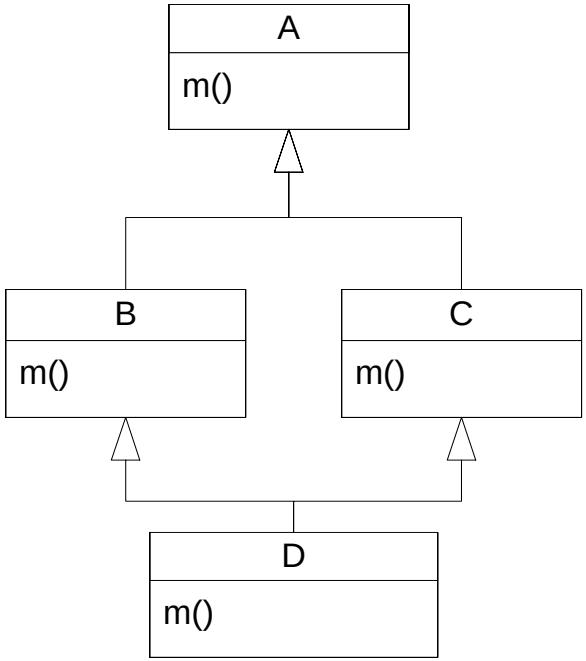
- Mitchell 11.7
- Smalltalk untyped, so how?
  - Subtyping as a relation between interfaces, substitutability
- Class Set
  - Set_interface ={isEmpty, size, includes, add}
- Class ExtensibleCollection
  - Set_interface ={isEmpty, size, includes, add}

- An ExtensibleCollection object can take the place of a Set object
  - There will be no ’message not understood’

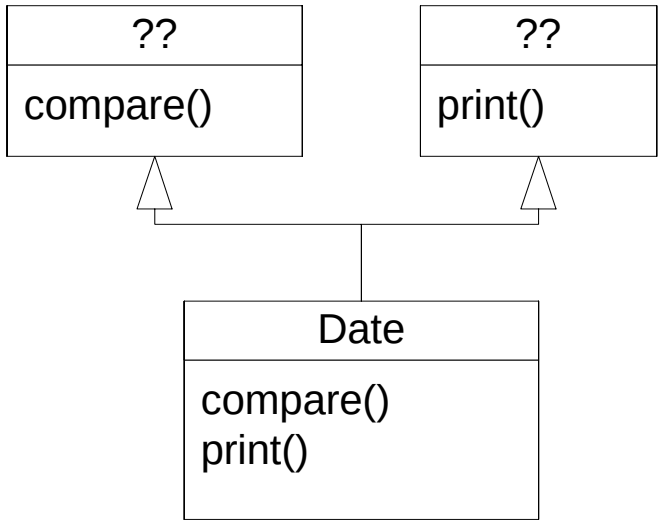
- Remember the cowboy ...; r.draw(); ...,

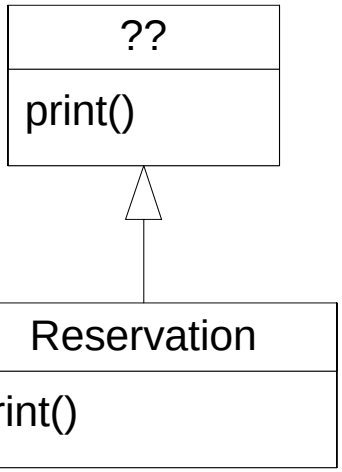
Multiple inheritance I

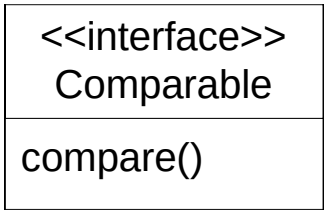
- Multiple supertypes or just multiple implementations?
- Name conflicts - m()
  - Take the leftmost (i.e. 'B.m()')
  - Not allowed
  - Renaming
  - Explicit identification 'B.m()'
    - In definition of class D
    - In every use of m()
- One or two A's?
- Overriding

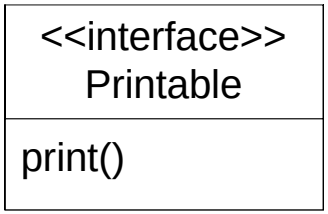


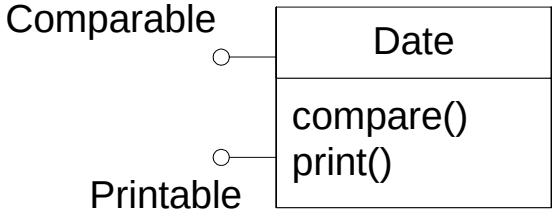
Multiple inheritance II

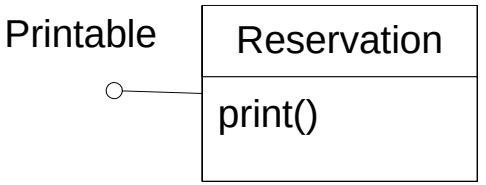




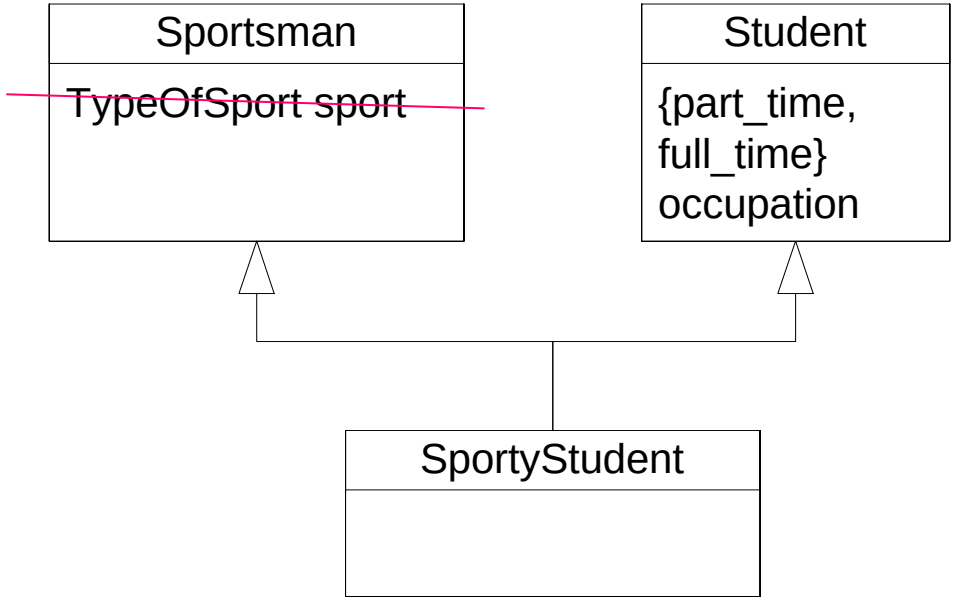






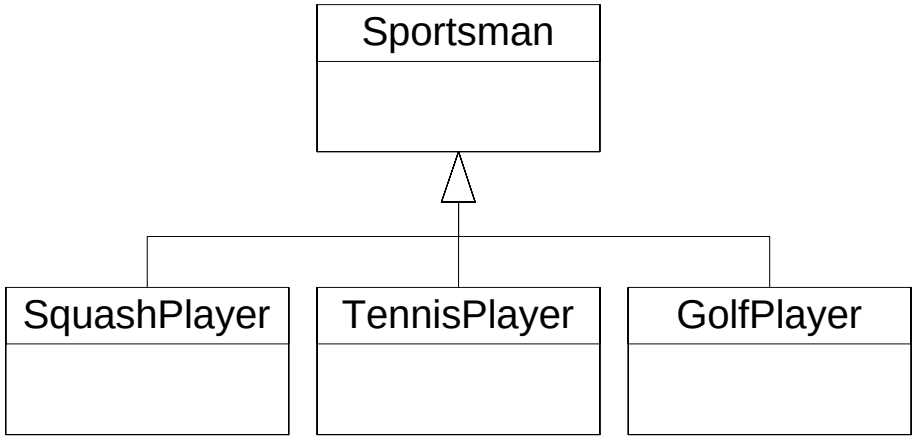


Multiple classification



Squash and Tennis playing student??

Squash playing student??



Multi-methods, ’dynamic overloading’

```
class Point { int x,y; }

class ColorPoint extends Point { Color c; }

bool equal(Point p1, Point p2) {
   return p1.x=p2.x and p1.y=p2.y };

bool equal(ColorPoint p1, ColorPoint p2) {
   return p1.x=p2.x and p1.y=p2.y and p1.c=p2.c };


equal (pt1, pt2);

        - equal (aPoint, aPoint);
        - equal (aPoint, aColorPoint);
        - equal (aColorPoint, aColorPoint);
```

Constraining type parameters

- C++ polymorphic sort function

```
template <typename T>
void sort( int count, T * A[count] ) {
    for (int i=0; i<count-1; i++)
        for (int j=i+1; j<count-1; j++)
            if (A[j] < A[i]) swap(A[i],A[j]);
}
```

- What parts of implementation depend on type?

  Meaning and implementation of <

Generic classes in Java (and C#)

```
interface I{…}
class C{…}
```

- Generic class

```
class G<T extends C implements I> {
   … T is known as a C and an I …  }
```

```
class G<T> where T:C, I {
  … T is known as a C and an I …   }
```

- Actual parameter

```
class AC extends C implements I{ … }
```

- Used like this:

```
G<AC> g = new G<AC>( );
```

- Java:Generic parameters can be classes and interfaces
- C#: Generic parameters can be classes and interfaces, and predefined types

Java

- Implementation
  - Type parameters removed (erased), substituted by `Object`.
  - Castings inserted according to the actual parameters.
- Restrictions
  - Can not cast to G<A> or use `instanceof G<A>`
  - Can not cast to `T` or use `instanceof T`.
  - `new T( )` not allowed.
  - All classes G<A>, where A is an actual parameter have a common set of static variables and methods; therefore:
    - Types of these can not depend on `T`
  - A1 subtype of A2 does not imply G<A1> subtype of G<A2>

C#

- Implementation
  - Makes a runtime descriptor for all actual uses of a generic class.
  - Uses the same code for all G<A>s as far as possible (i.e. as long as acatual parameters are classes and interfaces)
- Fewer restrictions
  - casting and `instanceof` are allowed, both with `T` (within `G`) and with `G<A>`.
  - `new T( )` allowed if `T` specified like this:

```
class G<T> where T: C, new( ) { … }
```

- Naming classes with actual parameters:

```
using GA = G<A>;
```

Java and C#

- T cannot be super class of an inner class.
- F-bounded polymorphism

```
class G<T extends G<T>>
```

  - Actual parameter for T is e.g.:

```
class A extends G<A>
```

- Generic methods (including static methods):
  - Java: `<U extends B>U getValue  (int i, U u) {…}`
  - C#: `U getValue <U> (int i, U u) where U: B {…}`

Modularity - Chapter 9 : Basic Concepts

- Component
  - Meaningful program unit
    - Function, data structure, module, …
- Interface
  - Types and operations defined within a component that are visible outside the component
- Specification
  - Intended behavior of component, expressed as property observable through interface
- Implementation
  - Data structures and functions inside component
  - Representation independence

Example: Function Component

- Component
  - Function to compute square root
- Interface
  - float sqroot (float x)
- Specification
  - If x>1, then sqrt(x)*sqrt(x) ≈ x.
- Implementation

```
float sqroot (float x){
   float y = x/2; float step=x/4; int i;
   for (i=0; i<20; i++){if ((y*y)<x) y=y+step; else y=y-step; step = step/2;}
   return y;
}
```

’programming-in-the-small’ versus ’programming-in-the large’

Module language concept

```
module Set
    interface
        type set
        val empty : set
        fun insert : elt * set -> set
        fun union : set * set -> set
        fun isMember : elt * set -> bool
    implementation
        type set = elt list
        val empty = nil
        fun insert(x, elts) = ...
        fun union(…) = ...
        ...
end Set
```

- Can define ADT
  - Private type
  - Public operations
- More general
  - Several related types and operations
- Some languages
  - Separate interface and implementation
  - One interface can have multiple implementations

Modules in object oriented languages

- Classes?
  - Interfaces
    - Types?
    - Operations?
  - Implementation
    - Representation independence?
    - State?
- Packages?
  - Interfaces?
    - Types?
    - Operations?
  - Implementation
    - State?
- EJB/.NET Components?

-
  - Yes
    - Inner classes?
    - Methods
  -
    - Depends: Interface or implementation inheritance
    - Yes
-
  - No, but
    - Public classes
    - Static methods
  -
    - No
- Special-made classes

Encapsulation versus composition

```
class Apartment {
      Kitchen theKitchen = new Kitchen();
      Bathroom theBathroom = new Bathroom();
      Bedroom theBedroom = new Bedroom ();
      FamilyRoom theFamilyRoom = new FamilyRoom ();
      . . .
      Person Owner;
      Address theAddress = new Address()
};



...; myApartment.theKitchen.paint(); ...
```

```
class Point {
      int x,y;
      Point(int i,j) {
      }
};
```

Inner classes - locally defined classes

```
class Apartment {
    Height height;
    Kitchen theKitchen = new Kitchen {… height  …}();
    class ApartmentBathroom extends Bathroom {… height …}
    ApartmentBathroom Bathroom_1 = new ApartmentBathroom ();
    ApartmentBathroom Bathroom_2 = new ApartmentBathroom ();
    Bedroom theBedroom = new Bedroom ();
    FamilyRoom theFamilyRoom = new FamilyRoom ();
    . . .
    Person Owner;
    Address theAddress = new Address()
};
```

Software engineering/typing issue

From Kim Bruce: Some Challenging Typing Issues
in Object-Oriented Languages

$$S \in SExp ::= n \mid -S$$

$$E \in Exp ::= n \mid -E \mid E + E$$

Functional solution I

$S \in SExp ::= n \mid -S$

```
datatype sexp = SConst of int | SNeg of sexp;

fun sinterp (SConst n) = n
| sinterp (SNeg t) = ~(sinterp t);
```

Easy to add a new operation on expression:

```
fun sformatter (SConst n) = Int.toString(n)
| sformatter (SNeg t) = "-" ^ (sformatter t);
```

Functional solution II

$$E \in Exp ::= n \mid -E \mid E + E$$

Not so easy to extend with + (i.e. new data structure):

```
datatype exp = Const of int | Neg of exp | Plus of exp*exp;

fun interp (Const n) = n
  | interp (Neg t) = ~(interp t)
  | interp (Plus t u) = (interp t) + (interp u);

fun formatter (Const n) = Int.toString(n)
  | formatter (Neg t) = "-" ^ (formatter t)
  | formatter (Plus t u) = (formatter t) ^ " + " ^
                           (formatter u);
```

Object-oriented solution I

$S \in SExp ::= n \mid -S$

```
public interface Form {int interp(); // Interpret formula}
```

```
public class ConstForm implements Form {
  public int value; // value of constant
  public ConstForm(int value) {this.value = value;}
  public int interp() { return value; }
}

class NegForm implements Form {
  public Form base; // formula being negated
  public NegForm(Form basep) {base = basep;}
  public int interp() { return (- base.interp()); }
}
```

Object-oriented solution II

$$E \in Exp ::= n \mid -E \mid E + E$$

Easy to extend with +:

```
public class PlusForm implements Form {
  public Form first, second;
  public PlusForm(Form firstp, Form secondp) {
    first = firstp;
    second = secondp;
  }
  public int interp() {
    return first.interp() + second.interp();
  }
}
```

Object-oriented solution III

$$E \in Exp ::= n \mid -E \mid E + E$$

Not so easy to extend with new operation:

```
public interface FForm extends Form {
  String formatter();
}
public class FConstForm extends ConstForm implements FForm
{
  public FConstForm(int value) {super(value);}
  public String formatter() {
    return "" + value;
  }
}
```

Object-oriented solution IV

```
class FNegForm extends NegForm implements FForm { ... }

public class FPlusForm extends PlusForm implements FForm {
  public FPlusForm(FForm firstp, FForm secondp) {
    super(firstp,secondp);
  }
  public String formatter() {
    return "(" + ((FForm)first).formatter() + " + " +
                 ((FForm)second).formatter() + ")";
  }
}
```