



INF3110 – Programming Languages

Types, Subtyping and Object Orientation

Eyvind W. Axelsen

eyvinda@ifi.uio.no | [@eyvindwa](https://twitter.com/eyvindwa) 

<http://eyvinda.at.ifi.uio.no>

Slides adapted from previous years' slides
made by Birger Møller-Pedersen

birger@ifi.uio.no

Object orientation and types

Lecture I (today)

- What is a type and why should we care?
- From predefined (simple) and user-defined (composite) types
 - via
- Abstract data types
 - to
- Classes/OO
 - Type compatibility
 - Subtyping <> subclassing
 - Covariance/contravariance
 - Types of parameters of redefined methods

Lecture II

- Advanced oo concepts
 - Specialization of behaviour?
 - Multiple inheritance - alternatives
 - Inner classes
- Modularity
 - Packages
 - Interface-implementation
- Generics

What is a type?

- A set of values that have a set of operations in common
 - 32 bit integers, and the arithmetic operations on them (simple type)
 - Instances of a Person class, and the methods that operate on them (composite type)
- How is a type *identified*?
 - By its *name* (e.g. Int32, Person, Stack): nominal type checking
 - By its *structure* (fields, operations): structural type checking
- Does this cover everything a type might be? No.
 - Alternative definition of “type”: *A piece of the program to which the type system is able to assign a label.*
 - *(but don't worry too much about this now)*

Why should we care about types?

Remember from last time: *syntax* (program text) and *semantics* (meaning) are two separate things.

Types and type systems help to ascribe meaning to programs:

- What does `"Hello" + " World"` mean?
- Which operation is called when you write `System.out.println("INF3110")`?
- What does the concept of a `Student` entail?

Classification of types

- Simple types
 - Predefined, simple types (not built from other types)
 - boolean, integer, real, ...
 - pointers, pointers to procedures
 - string
 - User-defined simple types
 - enumerations, e.g. `enum WeekDay { Mon, Tue, Wed, ... }`
- Composite types – composed from other types
 - Predefined composite types
 - Arrays, lists/collections (in some languages)
 - User-defined, composite types
 - Records/structs, unions, abstract data types, classes
- Evolution from simple types, via predefined composite types to user-defined types that reflect parts of the application domain.

Properties of simple (primitive) types

- Classifying data of the program
 - E.g. this is a string, this is an integer, etc
- Well-defined operations on values
 - Arithmetic operations
 - String concatenation
 - Etc
- Protecting data from un-intended operations
 - Cannot subtract an integer from a string (in most languages!)
- Hiding underlying representation
 - Does not allow manipulation of individual bits (again, in *most* languages!)
 - Are ints big or small endian?
 - Are strings represented as a character array in memory?

Properties of composite types

- Constructed from other types (simple or composite)
- **Examples:**
- Records, structs
 - (m_1, m_2, \dots, m_n) in $M_1 \times M_2 \times \dots \times M_n$
 - Assignment, comparison
 - Composite values $\{3, 3.4\}$
 - Hiding underlying representation?
- Arrays (mappings)
 - domain \rightarrow range
 - Possible domains, index bound checking, bound part of type definition, static/dynamic?

```
typedef struct {  
    int nEdges;  
    float edgeSize;  
} RegularPolygon;  
RegularPolygon rp={3, 3.4}  
rp.nEdges = 4;
```

C

```
char digits[10]  
  
array [5..95] of integer  
  
array[WeekDay] of T,  
where  
type WeekDay =  
    enum{Monday, Tuesday, ...}
```

C

Algol

Composite types

- Union
 - Alternative representations
 - Run-time type check
- Discriminated union
 - Run-time or compile-time type check
 - Additional discriminator aids checking

```
typedef union {  
    struct {  
        unsigned char byte1;  
        unsigned char byte2;  
        unsigned char byte3;  
        unsigned char byte4;  
    } bytes;  
    unsigned int dword;  
} HW_Register;
```

```
HW_Register reg;  
reg.dword = 0x12345678;  
reg.bytes.byte3 = 4;
```

```
union address {  
    short int offset;  
    long int absolute; }
```

```
typedef struct {  
    address location;  
    descriptor kind;  
} safe_address;  
  
enum descriptor {abs, rel}
```

```
address_type = (absolute, offset);  
  
safe_address =  
record  
    case kind:address_type of  
        absolute: (abs_addr: integer);  
        offset: (off_addr: short)  
    end;
```


Composite types in TypeScript

```
interface Square {  
    kind: "square";  
    size: number;  
}
```

```
interface Circle {  
    kind: "circle";  
    radius: number;  
}
```

```
interface Rectangle {  
    kind: "rectangle";  
    width: number;  
    height: number;  
}
```

```
// Shape is a discriminated union type  
type Shape = Square | Rectangle | Circle;
```

```
function area(s: Shape) {  
    switch (s.kind) {  
        case "square": return s.size * s.size;  
        case "rectangle": return s.width * s.height;  
        case "circle": return Math.PI * s.radius * s.radius;  
    }  
}
```

Type of s is narrowed based on «kind» in the union type – the discriminator

Abstract datatypes (ADTs)

Types so far today: data only (syntactically)

ADT: a user defined datatype that:

- Defines representation and operations in *one syntactical unit*
- *Hides the underlying representation* from the programmer

```
abstype Complex = C of real * real
  with
    fun complex(x, y: real) = C(x, y)
    fun add(C(x1, y1), C(x2, y2)) = C(x1+x2, y1+y2)
  end
```

```
...; add(c1, c2); ...
```

Signature of ADTs:

- Constructor
- Operations

Abstract datatypes versus classes

```
abstype Complex = C of real * real
  with
    fun complex(x, y: real) = C(x, y)
    fun add(C(x1, y1), C(x2, y2)) = C(x1+x2, y1+y2)
  end
```

```
...; add(c1,c2); ...
```

```
class Complex {
  real x,y;
  Complex(real v1,v2) {x=v1; y=v2}
  add(Complex c) {x=x+c.x; y=y+c.y}
}
```

```
...; c1.add(c2); ...
```

Can we make 'add(c1,c2)' with classes?

With abstract data types:
operation (operands)

- meaning of operation is always the same

With classes:

object.operation (arguments)

- meaning depends on object and operation (dynamic lookup, method dispatch)

From abstract data types to classes

- Encapsulation through abstract data types
 - Advantage
 - Separate interface from implementation
 - Guarantee invariants of data structure
 - only functions of the data type have access to the internal representation of data
 - Disadvantage
 - Not extensible in the way classes are

Abstract data types argument of Mitchell

```
abstype queue
with
    mk_Queue: unit -> queue
    is_empty: queue -> bool
    insert: queue * elem -> queue
    remove: queue -> elem
is ...
in
    program
end
```

```
abstype pqueue // priority queue
with
    mk_Queue: unit -> pqueue
    is_empty: pqueue -> bool
    insert: pqueue * elem -> pqueue
    remove: pqueue -> elem
is ...
in
    program
end
```

Cannot apply queue code to pqueue, even though signatures are identical

Type compatibility – is this OK?

```
class A {  
    x: number;  
}  
  
class B {  
    x: number;  
}  
  
var ab: A = new B();  
ab.x = 42;
```

It depends!

- In Java?
- In Python?
- In JavaScript?
- In TypeScript?
- In C#?

Type compatibility

- Nominally compatible
 - Values of types with the same name are compatible
- Structurally compatible
 - Types T1 and T2 are compatible
 - If T1 is nominally compatible with T2, or
 - T1 and T2 have the same signature (functions, variables, including names of such)

```
struct Position {int x, y, z; };  
  
Position pos;  
  
struct Date { int m, d, y; };  
Date today;  
  
void show(Date d);  
  
...; show(today); ...  
  
...; show(pos); ...
```

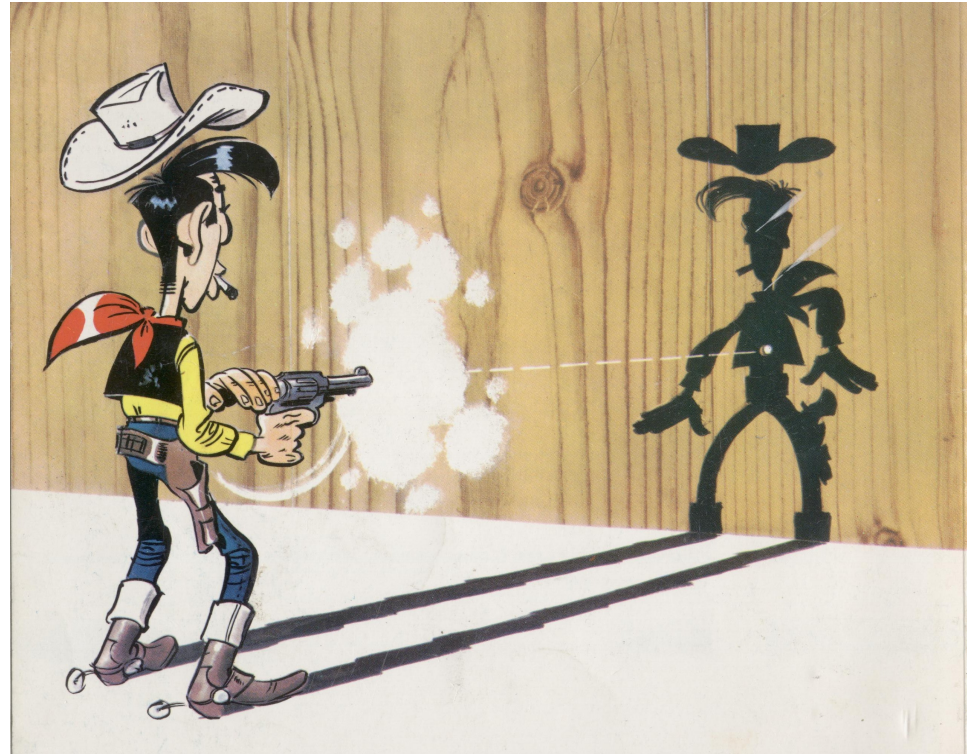
```
struct Complex { real x, y; };  
  
struct Point { real x, y; };
```

Example - Structural compatibility

- Two classes with the same structural type

```
class GraphicalObject {  
    move(dx, dy int) {...}  
    draw() {...}  
};
```

```
class Cowboy {  
    move(dx, dy int) {...}  
    draw() {...}  
};  
...
```



```
class Luke { ... ? } ...; luke.draw();...; luke.draw();
```


Subtyping

- Types can be related through *subtyping*
 - Relationships can, again, be defined nominally or structurally
- A variable of a supertype can at runtime hold a value of a subtype
 - *Without introducing type errors*
 - Enables polymorphism, dynamic dispatch
 - However, *behavioral subtyping* (Liskov) cannot, in general, be enforced by a compiler/type system
- How to best facilitate creation of such hierarchies are subject to much research and debate
 - Single/multiple inheritance
 - Traits/mixins
 - Structural/nominal subtyping
 - etc

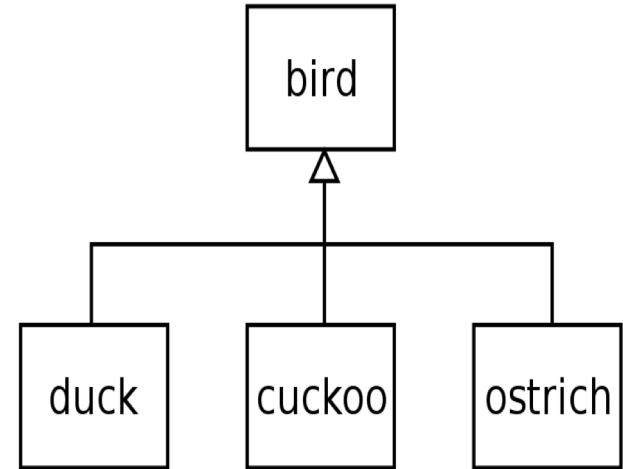
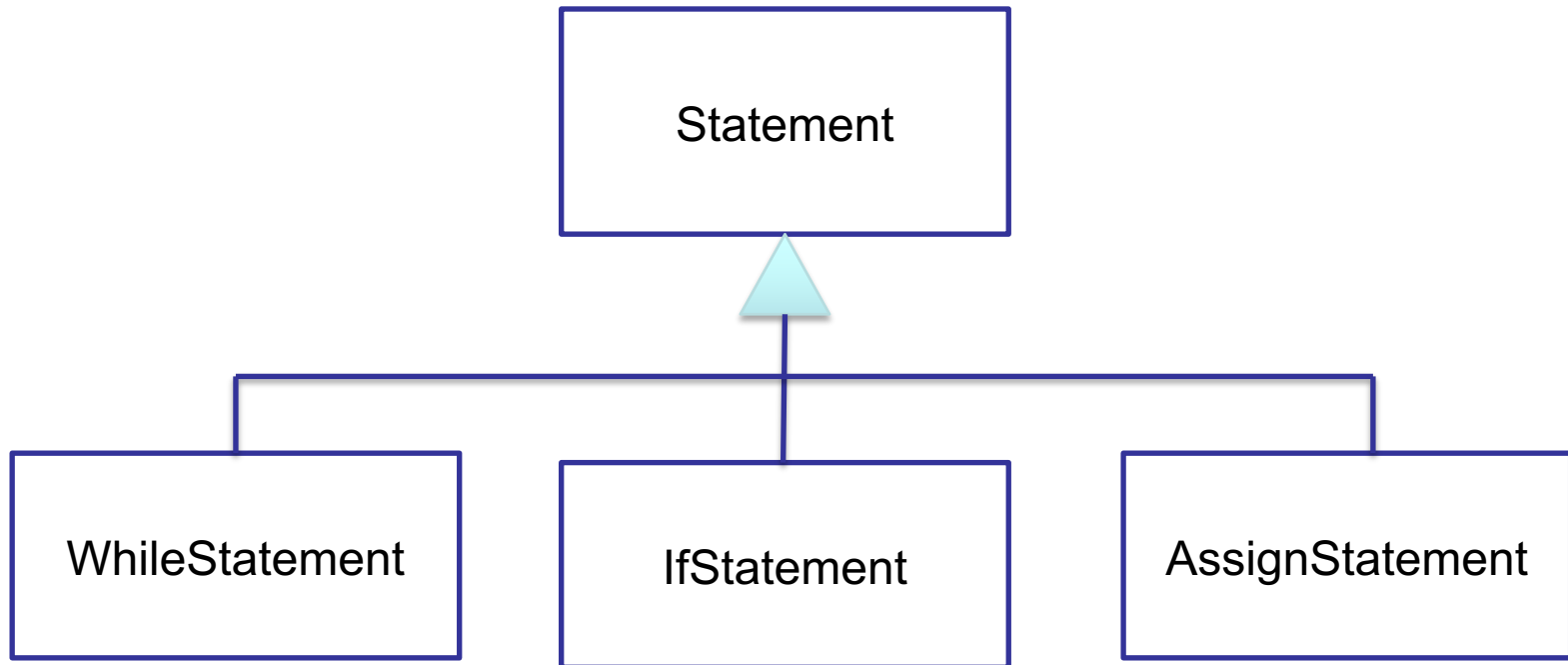


Image from Wikipedia

Language grammars can naturally be expressed through the help of subtyping

Statement ::= WhileStatement | IfStatement | AssignStatement



```
abstract class Statement { ... }  
class WhileStatement extends Statement { ... }  
class IfStatement extends Statement { ... }  
class AssignStatment extends Statement { ... }
```

Look out for this in your oblig exercise!

So, what is the proper object oriented way to get rich?

Inherit!

Point and ColorPoint

```
class Point {  
    int x, y;  
    move(int dx, dy) {  
        x=x+dx; y=y+dy  
    }  
}
```

Point

x
y
move

```
class ColorPoint extends Point {  
    Color c;  
    changeColor(Color nc) {c= nc}  
}
```

ColorPoint

x
y
c
move
changeColor

◆ ColorPoint interface contains Point interface

- ColorPoint is a *subtype* of Point

Could not form list of points and colored points if done by abstract data types

Subclassing

- Two approaches
 - So-called 'Scandinavian'/Modeling Approach
 - Classes represent concepts from the domain
 - Subclasses represent specialized concepts
 - Overriding is specialization/extension
 - Subclass is subtype
 - Reluctant to multiple inheritance (unless it can be understood as multiple specialization)
 - So-called 'American'/Programming Approach
 - Classes represent implementations of types
 - Subclasses inherit code
 - Overriding is overriding
 - Subclassing not necessarily the same as subtyping
 - Multiple inheritance as long as it works



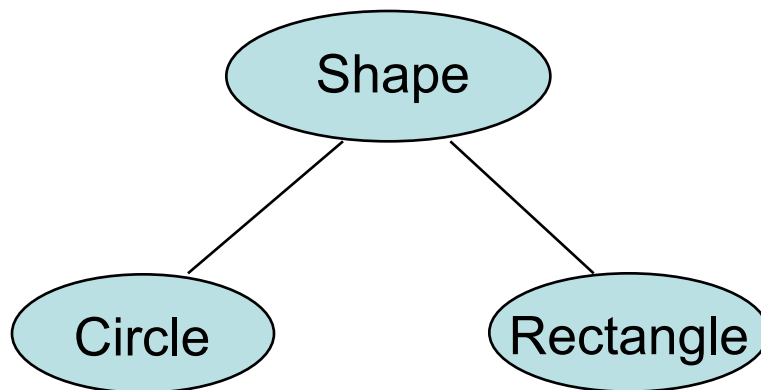
Kristen Nygård and Ole-Johan Dahl

<http://simula67.at.ifi.uio.no/50years/>

Example: Shapes

Interface of every **shape** must include **center**, **move**, **rotate**, **print**

- 'American'/Programming Approach
 - General interface only in Shape
 - Different kinds of shapes are implemented differently
 - **Square**: four points, representing corners
 - **Circle**: center point and radius
- 'Scandinavian'/Modeling Approach
 - General interface *and* general implementation in shape
 - Shape has center point
 - A Shape moves by changing the position of the center point
 - 'To be or not be' virtual
 - e.g. move should not be redefined in subclasses



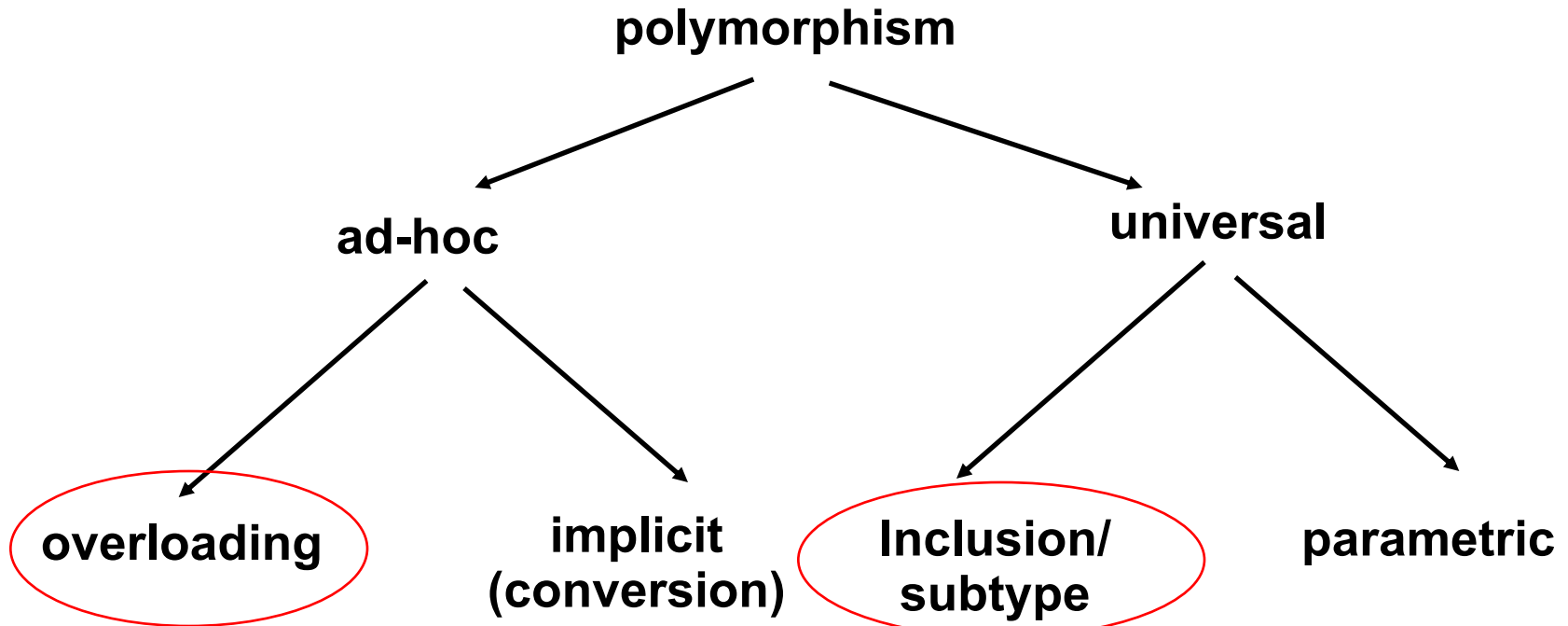
In Simula, C++, C#, a method specified as **virtual** may be overridden.

In Java, a method specified as **final** may *not* be overridden.

Classification of polymorphism

"Polymorphism: providing a single interface to entities of different types"

- "Bjarne Stroustrup's C++ Glossary".



Subtype polymorphism

```
Point p;  
ColorPoint cp;  
  
...; p.equals(cp); ...
```

'equals' works for cp because
ColorPoint is a subtype of type Point

```
class Shape {  
    void draw() {...}  
    ...  
};  
class Circle extends Shape {  
    void draw() {...}  
    ...  
};  
  
...; aShape.draw(); ...
```

Override

Will draw a Circle if aShape is a Circle
This is "dynamic dispatch" →

Subtype polymorphism and dynamic dispatch

- When a message is sent to an object (i.e., a method is called), the receiver of the message determines at *runtime* which operation to perform
- **`o.M(arguments)`**
 - If *M* is a *virtual method*, the compiler cannot know which method to call, this will be determined at runtime.
 - Because *o* might hold objects of a subtype of its compile-time type
 - **Single (dynamic) dispatch**: determined by runtime type of *o*, and the compile-time type¹ of the rest of the arguments
 - Most common, found in e.g. Java, C#, C++, Go, JavaScript, etc.
 - **Multiple (dynamic) dispatch**: determined by runtime type of *o* *and* of the rest of the arguments.
 - Examples: Common Lisp, Julia, Dylan, MultiJava

Overloading – two methods with the same name (“*ad hoc* polymorphism”)

```
class Shape {  
    ...  
    bool contains(point pt) {...}  
    ...  
};  
  
class Rectangle extends Shape {  
    ...  
    bool contains(int x, int y) {...}  
    ...  
}
```

- only within the same scope {...}, or
- across superclass boundaries

Overloading vs Overriding (Java and similar languages)

```
class C {  
    ...  
    bool equals(C myC) {  
        ...           // C_equals_1  
    }  
}  
  
// SC is subclass of C:  
class SC extends C {  
    ...  
    bool equals(C myC) {  
        ...           // SC_equals_1  
    }  
  
    bool equals(SC mySC) {  
        ...           // equals_2  
    }  
}
```

```
C c      = new C();  
SC sc    = new SC();  
C c2     = new SC();
```

```
c.equals(c)      //1    C_equals_1  
c.equals(c2)     //2    C_equals_1  
c.equals(sc)     //3    C_equals_1
```

```
c2.equals(c)     //4    SC_equals_1  
c2.equals(c2)    //5    SC_equals_1  
c2.equals(sc)    //6    SC_equals_1
```

```
sc.equals(c)     //7    SC_equals_1  
sc.equals(c2)    //8    SC_equals_1  
sc.equals(sc)    //9    equals_2
```

Covariance/contravariance/novariance

```
class C {  
    T1 v;  
    T2 m(T3 p) {  
        ...  
    }  
}  
  
class SC extends C {  
    T1' v;  
    T2' m(T3' p) {  
        ...  
    }  
}
```

- Covariance:
 - T1' must be a subtype of T1
 - T2' must be a subtype of T2
 - T3' must be a subtype of T3
- Contravariance:
 - The opposite
- Nonvariance: must be the same types
- Most(?) languages have no-variance
- Statically type-safe:
 - Contravariance on parameter types
 - Covariant on result type
- Some languages provide covariance in both: most intuitive?

Example: Point and ColorPoint – I: no variance

```
class Point {  
    int x,y;  
    move(int dx, dy) {  
        x=x+dx; y=y+dy}  
  
    bool equals(Point p) {  
        return x==p.x and y==p.y  
    }  
}  
  
class ColorPoint  
    extends Point {  
    Color c;  
    bool equals(Point p) {  
        return x==p.x and  
                y==p.y and  
                c==p.c  
    }  
}
```

Problem??

Point p1, p2;

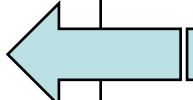
ColorPoint c1,c2;

p1.equals(p2)

c1.equals(c2)

p1.equals(c1)

c1.equals(p1)



```
return  
super.equals(p) and  
c==p.c
```

Example: Point and ColorPoint – II: covariance

```
class Point {
    int x,y;
    move(int dx,dy) {
        x=x+dx; y=y+dy}

    bool equals(Point p) {
        return x=p.x and y=p.y
    }
}

class ColorPoint
    extends Point {
    Color c;
    bool equals(ColorPoint cp) {
        return super.equals(cp)
            and
            c=cp.c
    }
}
```

Point p1, p2;

ColorPoint c1,c2;

Which of these may
be OK, and when can a
language check?

p1.equals(p2) OK run-time

c1.equals(c2) OK compile-time

p1.equals(c1) OK compile-time

c1.equals(p1) OK run-time

Example: Point and ColorPoint – III: casting

```
class Point {
    int x,y;
    move(int dx,dy) {
        x=x+dx; y=y+dy}

    bool equals(Point p) {
        return x==p.x and y==p.y
    }
}

class ColorPoint
    extends Point {
    Color c;
    bool equals(Point p) {
        return super.equals(p) and
                c==(ColorPoint)p.c
    }
}
```

Point p1, p2;

ColorPoint c1,c2;

p1.equals(p2)

c1.equals(c2)

p1.equals(c1)

c1.equals(p1)

Example: Point and ColorPoint –

```
class Point {  
    int x,y;  
    virtual class Type < Point;  
  
    bool equals(Type p) {  
        return x=p.x and y=p.y  
    }  
}  
  
class ColorPoint  
    extends Point {  
    Color c;  
    Type:: ColorPoint;  
    bool equals(Type p) {  
        return super.equals(p) and  
            c=p.c  
    }  
}
```

- Alternative to casting:
 - Virtual classes with constraints (OOPSLA '89)
 - Still run time type checking

Example: Contravariant parameter type

```
class A {  
    void m(A a) {  
        ...  
    }  
}  
  
class B extends A {  
    void m(Object a) {  
        ...  
    }  
}
```

- Statically type safe
- Not allowed in Java

Example: Covariant return type

```
class A {  
    A m() {  
        return new A();  
    }  
}  
  
class B extends A {  
    B m() {  
        return new B();  
    }  
}
```

- Statically type safe
- Allowed in e.g. Java

Practical info

- Mandatory 1 out today
 - Deadline September 21st.
 - Go to the group session for more info!
- Next lecture: ML with Daniel
- Have a nice weekend!