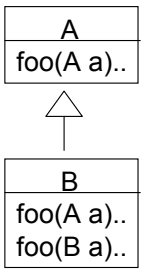
OOP 2005
Lecture 11
Inheritance & Polymorphism

Kasper B. Graversen
(minor changes by Kasper Østerbye)
IT-University of Copenhagen

1

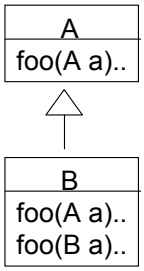
Today's program



In short, today's program is about some of the consequences of introducing inheritance into the language. We shall look at references, method calls, overloading methods and scratch the surface of type theory. In other words, this lecture cuts to the bone of programming... as the picture illustrates :-)

Today's program

- Types
  - variance
  - type safety
  - java variance
- Generic types and variance
- Method dispatching
  - virtual methods
  - overloading
- Interface based software development
- Perspectives
  - Run-time inheritance
  - Languages "without" types



Inheritance

Can be explained from two points of view

- Code re-use: inheritance = $\Delta x \oplus o$
  - Just specify changes (delta x) and merge them ($\oplus$) with the old (o)
- Scandinavian school of OOP
  - we see phenomena and create hierarchical views of these all the time
    - flower, food, bank accounts,...
  - Concept from the real world.
  - Since 1968 maybe even earlier
- Was a key concept in getting OO to be accepted in industry (30 years after its invention!)

4

Each tradition has its own definitions on what proper use is.

Other than that, it was the concept of inheritance which, along with a lot of hype, was the key concept that made OO take over procedural programming in the mid 90's (with the language C++ which supported both OO and procedural style of programming).

Simula was the first OO language released in 1967 (a prior version from 1964 was not OO). Both Ole-Johan Dahl (implementor) and Kristen Nygaard (inventor) later received the Turing award around 30 years later---luckily just before they passed away.

Inheritance and types

```
class A {
    void foo() ..
}

class B extends A {
    void foo() ..  // overwrite
    void bar() ..  // new stuff
}
```

- We say B is a subtype of A. That is, B is of type B as well as A. Sometimes written as A <: B or $A \subseteq B$

```
class C implements OOPIterator { .. }
```

- Simply means, “C is has the type OOPIterator” – C thereby commit itself to implement the methods of that interface.

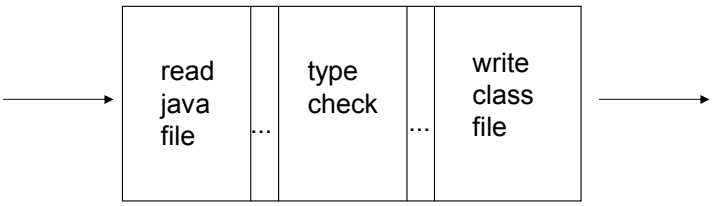
Types

- A type is a sort of a contract.
- A type specifies **a set** of signatures and fields
  - A signature is the return type, name of the method, and the types of the arguments.
- **If some object has type `A`, you are guaranteed to be able to call any method on the object, which has a signature given in `A`.**
- Why?
  - May seem rigid compared to how we think and speak, but for computers I think it's ok. (Eg, an Emu is a bird, even though it can not fly).
  - Helps the programmer in that he can specify expectations to input and promises on output.
  - Helps compilers (and other tools) in finding errors, making presentations, refactoring etc.

6

Type checking

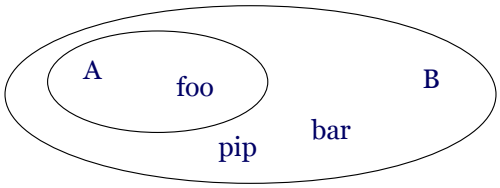
- Who issues the guarantees?

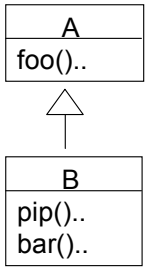


- Affordable check compared to run-time tests
- Types are much less expressive (ie. can express integers, but not positive integers).
- The generality enables a static (pessimistic) check that the contracts are not broken.

Subtypes

- A subtype is an extension of an existing contract (the subtype's supertype's contract).
- The extension always has 0..*n* additional signatures and fields and 0..*n* re-definitions of methods.
- The subtype automatically inherits the contract of its supertype, thus the following property always hold
  - A ⊆ B (A is a subset of B signature- and field- wise)
  - when is it also A = B ?



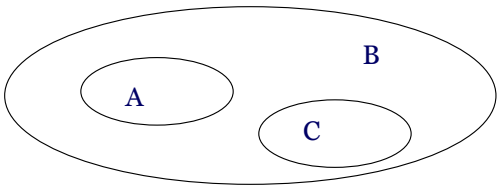


8

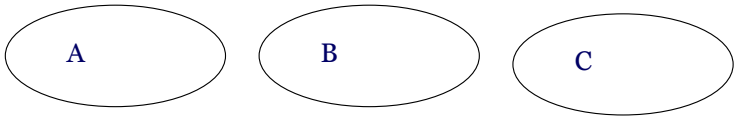
Subtypes

- How do we specify the following type relationships

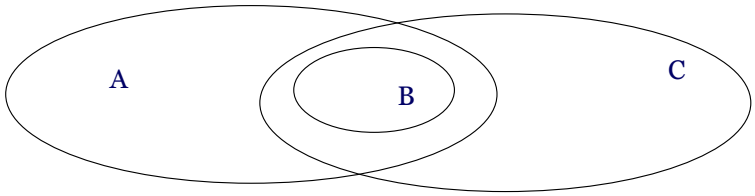
1.



2.



3.



9

Polymorphism

- A key benefit of inheritance is polymorphic references.
- Allows usage of subtypes in all the places of an application where a type is expected.
- Pragmatically you already know this from playing with collections… every element is stored using `Object` references (thats also how generic types in Java works!).

Polymorphism

- Using knowledge on subtypes, we can deduce some rules.
  - Object o = new A() ??
  - A a = new A() ??
  - B b = new A() ??
  - A a = new B() ??
- this applies to all references-including arguments in method headers.
- Since a reference can hold subtypes it is said to be *polymorphic* (many shaped).

11

Polymorphism also goes under the name “subsumption”

Cancellation

- Can a subtype have less methods/fields? No. But can we cheat the type system. Yes.
  - A method in a subclass can be implemented just to throw an exception. `Iterator.remove()` throws `UnsupportedOperationException`
  - A method may return the status of the implementation eg. `BufferedReader.markSupported()` – return boolean tells if `mark()/reset()` are usable.
- The exceptions i could find
  - `CloneNotSupportedException`
  - `OperationNotSupportedException`
  - `SAXNotSupportedException`
  - `UnsupportedOperationException`
  - `AuthenticationNotSupportedException`
- Type checker cannot check anything, only run-time exceptions are raised
- Why can't we just remove the methods we don't like? (slide 6)

Type variance

- Forget Java for now...
- Looking at methods overwritten in subclasses, their arguments and return type can vary in three ways
  - *Invariance* : no change
  - *Covariance* : more specific
  - *Contravariance* : less specific

```
class Office {
       void meet(Person p) ...
}

class PrincipalOffice extends Office
       void meet(Student s)...
}
```

- covariant specialization of the argument of `meet`

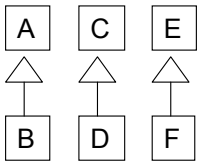
13

Type safe variance

- The rules of type safety are equal to the ones used on polymorphic references

```
class A {
    C foo(F e) ...
}

class B extends A {
    D foo(E f) ...
}
```



- Contravariance for arguments.
- Covariance for return values.
- Invariance is trivially type safe.

Do not confuse this variance with

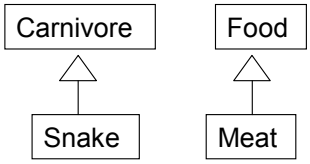
1. what is possible in java
2. what makes sense functionality wise. In most practical situations contravariance is uninteresting, whereas covariance is interesting.

14

See also the discussion on the Liskov substitution principle
http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple

Type safe variance (contravariant arguments)

```
class Carnivore {
    void feed(Meat m)..
}
class Snake extends Carnivore {
    void feed(Food f)..
}
```



- Type wise ok since the contract of carnivor is not broken in snake.. i.e.

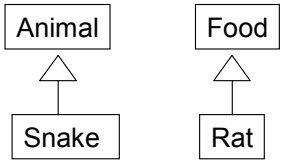
```
Carnivore c = new Snake()
c.feed(new Meat())
```

- If X expect something blue, and we substitute X with Y (which expects any color). Y will not break when we feed it blue.

15

Type unsafe variance (covariant arguments)

```
class Animal {
    void feed(Food m)..
}
class RatSnake extends Carnivore {
    void feed(Rat r)..
}
```

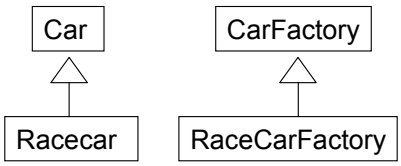


- Type wise *not* ok since the contract of animal is broken in snake.. i.e.

```
Animal a = new RatSnake()
a.feed(new Food())  // failure!
```

16

Type safe variance (covariant return type)




```
class CarFactory {
    Car produce()..
}
class RaceCarFactory extends CarFactory {
    Racecar produce()..
}
```

- Type wise ok since a race car is a car, hence we can substitute a car factory with a racecar factory

```
CarFactory f = new RacecarFactory()
Car c = f.produce()
```

- And note, that covariant return type can be very useful

17

The example is valid in that if we expect something blue, and we substitute it with something which expects any color, it will not break to feed blue to such a thing.

Solving the Covariance problem

- Covariance is conceptually pleasing – but it isn't typesafe!
- How to solve this issue?
  1. Disallow polymorphic references
  2. Hide the specialized class and ensure proper usage
  3. Ignore typecheck for covariant specialization (the Eiffel language)

1.

```
Animal a = new Snake()              // fail
class List { void add(Object o) } // accept only Object objects
```

2.

```
collaboration A {
    class animal {
        void eat(Food f)
     }
     class Food { }
}
```

```
collaboration B extends A {
    class Snake extends Animal {
        void eat(Rat f)
    }
    class Rat extends Food {}
}
```

Inheritance in Java

- Java only supports **invariance**!

```
class A {
   void foo(A a)..
}
```

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

  - `foo` simply becomes overloaded

```
class A {
   C foo()..
}
```

```
class B extends A {
    D foo()..
}
```

- Is not possible..

Inheritance in Java

- We can specialize what we return (type wise) in subclasses, the type checker just does not take this into consideration.

```
class CarFactory {
   Car build() { return new Car(); }
}
class RacecarFactory extends CarFactory {
   Car build() { return new Racecar(); }
}
```

- We are not allowed to say

```
RacecarFactory fac = new RacecarFactory();
Racecar r = fac.build();    // r = (Racecar) fac.build();
r.driveLikeCrazy();
```

- Remember type casts (down) are unsafe! We overrule the typechecker

20

Note: yes indeed we can typecast the return value of the build() but this is not type safe, hence a run-time error may occur if we are not careful: e.g. a subclass of RacecarFactory could return a Car object.

A typecast basically is telling the compiler that you know better than it does. But since it didn't know the particular situation, there is no compiler support (no checking) done for you.. hence a cast in a code is a potential run-time error.

Remember that we've seen this need-to-cast-problem re-occurring all the time when using the collection library. (at least before generic types ;-)

Inheritance in Java (arrays)

- Java has covariance for arrays. How do we see this?

```
Car[] ca = new Car[1];
ca[0] = new Car();     // pass
ca[0] = new Object(); // fail
Car c = ca[0];         // pass

Racecar[] ra = new Racecar[1];
ra[0] = new Racecar();
ra[0] = new Car();
Racecar r = ra[0];
ca = ra;
```

```
class CarArray {
    void add(Car c) ...
    Car get(int i) ...
}

class RacecarArray
            extends CarArray {
    void add(Racecar c) ...
    Racecar get(int i) ...
}
```

- Which parts are covariant? contra variant? invariant?
- Which parts that vary are type safe?

Inheritance in Java (arrays)

- When are we in trouble?

```
Racecar[] ra = new Racecar[] { new Racecar() };
Car[] ca = ra;       // is ok, but should we allow it?
ca[0] = new Car();   // yields run-time error
Racecar r = ra[0];
```

- Note that the whole idea about static type checking at compile time is to detect and report errors on programs. The above example shows that there are some programs which cannot be type checked properly.. and hence will break at run-time.
- Bill Joy suggests that this behaviour is a mistake more than intended semantics.

22

Afterthoughts about arrays, which seems to suggest that covariant arrays are more a hack than a feature.

Type parameters as implemented in Java 5.0 will provide a more acceptable solution to this.

Note that all the information below is taken from John Mitchell (http://www.stanford.edu/class/cs242/slides/java.ppt)

Date: Fri, 09 Oct 1998 09:41:05 -0600

From: bill joy

Subject: ...[discussion about java genericity]

actually, java array covariance was done for less noble reasons ...: it made some generic "bcopy" (memory copy) and like operations much easier to write...

I proposed to take this out in 95, but it was too late (...).

i think it is unfortunate that it wasn't taken out...

it would have made adding genericity later much cleaner, and [array covariance] doesn't pay for its complexity today.

wnj

Comparing to C++ we see that Java is more flexible:

Access by pointer: you can't do array subtyping.

```
B* barr[15];
A* aarr[] = barr;   // not allowed
```

Direct naming: allowed, but you get garbage !!

```
B barr[15];
A aarr[] = barr;
```


Generic types
23

Generic types

- Are generic references polymorphic?

```
import java.util.*;
class GenTry {
   public static void main(String[] args) {
      ArrayList<Integer> list = new ArrayList<Integer>();
      list.add(new Integer(42)); // can't use autoboxing!
      ArrayList<Number> l2 = list;
      ArrayList<Integer> l3 = list;

      // covariant parameter - can't be used
      ArrayList<? extends Number> ln = list;
      Number n = ln.get(0);
      ln.add(new Integer(43));

      // contravariant parameter - can be used
      ArrayList<? super Integer> li = list;
      Integer i = li.get(0);
      li.add(new Integer(43));
      li.add(new SpecialInteger(43));

       ArrayList<? super SpecialInteger> si = list;
} }
```

24

Invariant references seems very limited. Fortunately the wildcards allows us to vary the parameter co- or contra-wise

l2 is not legal as we would then be able to insert any Number into the integerlist.. eg. l2.add(new Float(4.2)) and hence we would no longer have a list of Integer.

ArrayList<? extends Number> ln = list;

Means a list of elements which all are guaranteed to inherit Number... we can read it as Number.. but we can't write it as it could be a list of Integer.. in which we cannot insert a Float

ArrayList<? super Integer> li = list;

Means a list of elements which are Integer or any super type. Hence we can instert elements of type Integer or subtypes (here the imaginary SpecialInteger)

But we cannot cheat and say ? super SpecialInteger.. as our list only promised to contain Integer

And notice how poorly the autoboxing mechanism is implemented in Java...

Method dispatching

Method dispatching (how methods are called)

```
class A {
    void foo() { this.bar(); }
    void bar() { print("aaa"); }
    void foe() { print("foe"); }
}
class B extends A {
    void foo() { super.foo(); }
    void bar() { print("bbb"); }
}

new B().foo();
```

- What is printed on the screen?

Method dispatching

- Instance methods in Java are *virtual*.
  - Methods are "late bound" or'
  - "late binding of self" (in Java "self" is called "this")

- It is always the most specific (most specialized) method which is called. That method can then choose to invoke a less specific method using `super`

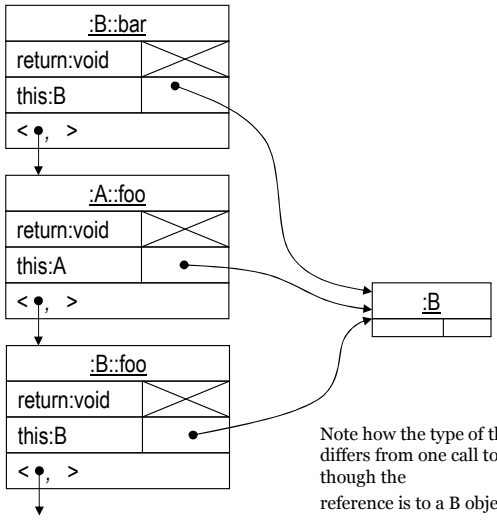
```
A a = new B()
a.bar(); // "bbb" is still printed
```

Call stack drawing

```
class A {
    void foo() { this.bar(); }
    void bar() { print("aaa"); }
}
class B extends A {
    void foo() { super.foo(); }
    void bar() { print("bbb"); }
}

new B().foo();
```

- `this` varies covariantly



Note how the type of the this reference differs from one call to the other, even though the reference is to a B object at all times.
The this reference is polymorphic, except if the class is declared final.

Virtual methods

- Why?? so subclasses have 100% freedom to re-define functionality.

```
class Car {
    void shiftGearUp() ..
    void automaticGear() {
        if(motor.rpm > 7800)
            shiftGearUp();
    }
}
```

```
class Racecar extends Car {
    void shiftGearUp() {
        checkSuperInfusion();
        igniteRockets();
        .. // may call super
    }
}
```

- Note no change to `automaticGear()`

Non-virtual method dispatching

- All instance methods in Java are defined virtual. In other languages this must be explicitly declared (hence they are non-virtual by default – e.g. C++, C#).
- This means that there is difference between a and b in:

```
A a = new B();
B b = a;
a.foo();
b.foo();
```

when `foo` is overwritten in B, and B extends A

30

virtual methods

- Advantages
  - More flexible subclasses
- Disadvantages
  - Harder to read code as 'this' may be bound to any subclass
  - Calling virtual methods from within the constructor of a super class
    - the method of the super class may use fields only defined in the subclass… which are not initialized yet!

Overloading

- Nice in that it avoids branching over types

```
class A {
   void process(Video v)..
   void process(House h)..
}
```

```
class A {
    void process(Object o) {
        if(o instanceof Video)..
        if(o instanceof House)..
    }
}
```

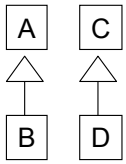
- And the interface (what the method really accept) is much clearer

Overloading

- The most specific method is called
  - foo(B a) rather than foo(A a) on the `call xx.foo(new B())`
- But sometimes it is not defined which is the most specific method!

```
class A {
  void foo(A x, D y) { .. }
  void foo(B x, C y) { .. }
}

new A().foo(new A(), new C());
new A().foo(new B(), new D());
```

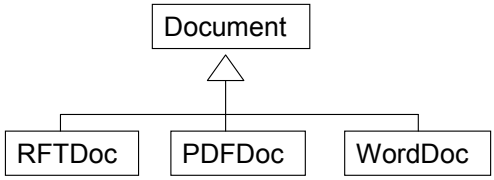


Overloading

- "Fails" when overloading on subclasses

```
class Processor {
    void process(Document d)..
    void process(RTFDoc d)..
}

Document doc = new RTFDoc();
RTFDoc rtf = (RTFDoc) doc;
Processor p = new Processor();
p.process(doc);
p.process(rtf);
```



- Who is called?
- Explanation: *Static binding of method selection*
- Other languages have dynamic binding of method selection
  - calls are more expensive as its the run-time type which is used for the selection
  - Sometimes there may not be a unique method = run-time error

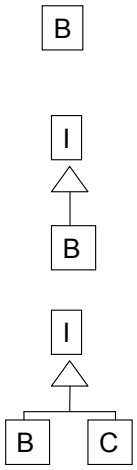

Interface based development
35

Interface based software development

- Goal of OO programs are
  - modular
  - loose coupling of entities
  - preferably substitution of entities
- One approach is rather simple
  - Substitutable entities has the same type
  - Implements the same interface (contract)

Interface based software development

- Code against interfaces
- All references should be of an interface type rather than a concrete type
  - `B r = new B()`
  - `I r = new B()`

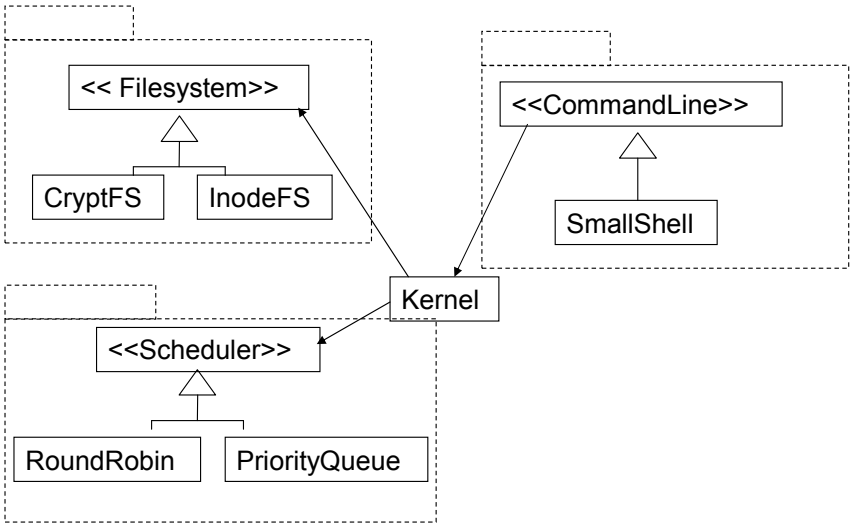


37

When reference r is of type I rather than B we can type wise substitute the B instance with any other instance implementing I, e.g. a C instance.

Interface based software development

- An example of modules of a system



38

This examples shows the implementation of a lager operating system. The dotted boxes denote the possibility of hiding each bigger part of the system inside a package. The beauty of this design is that when starting the kernel (the most basic of the OS) it can be chosen which file system to use, which task scheduler etc. This can be done since references are of interface types rather than concrete types. This way we know that we do not call methods using the references which any other class implementing the interface does not contain.

Interface based software development

- Instantiating is a little different
  `Filesystem fs = new INodeFS()`
- rather than
  `INodeFS fs = new INodeFS()`
- Ensures only methods defined in the interface is called
  - is implemented by all entities of that interface.
  - allows for substitutions like
    `fs = new CryptFS()`
- Statements such as `instanceof(INodeFS)` becomes dangerous as it tightly connected to a certain implementation.

Dynamically typed languages

- No type check phase
- No type declaration in the programs!
  - Shorter code
  - More flexible code
    - A reference is just a reference (no person reference, document reference etc)
    - An object is useable based on its structure rather than its type. If it has the method foo and I need to call a foo, fine!
    - Calling a method on an object which does not have such a method is not necessarily an error. Language support for redirecting the call.
- The real (tough) errors are not type errors. So why waste time fighting a rigid type system?
- What errors can a compiler detect a unit test cannot?

Perspectives..

- A small python program

```
class Person(object):
    def __init__(self, name, age):
        self.age = age          # create an age field
        self.name = name        # create a name field

        def setMom(self, mom):
            """mom is a female person"""
            self.mom = mom      # create a mom field

        def whosYourMom(self):
            print("my mom is " + self.mom.name)

p1 = Person("palle", 22)
p2 = Person("Anna", 44)
p1.setMom(p2)
p1.whosYourMom()          # works ok
p2.whosYourMom()          # run-time error..no mom field
p1.whosYourMom = p1.foo # name now points to foo()
```

41

This is just a small example. Since there is no type check, the language is very dynamic. E.g. one can add methods to objects, to classes etc. at run-time. It is also possible to change the content of a method run-time or rename it. This can lead to some very neat code since branching (if-statements) to some extend can be left out and instead rely on changing the content of the method instead.

In many communities such as python, it is often cited that you will code 5-10 times faster than in Java. I'm not sure of the validity of such claims... but being fond of programming languages as I am, I say it won't hurt you to try alternatives to Java ;-)

What did you learn in school today?

- Polymorphic references
- Static method dispatching
- Variance of arguments and return types
- Overloading
- Arrays
- Generic types
- Interface based development
- What we missed
  - Static methods (again, different rules – how fun! you should really consider looking at exercise 7!)
  - Field access virtual or non-virtual? (exercise 2)
  - The Beta language has a completely different definition of virtual –it is the least specific method which is invoked first. This method then can call inner (instead of super)
  - Inheritance pr. object rather than pr. type

42

4 week project proposals

- Learn Beta
  - Look into the Beta language and compare it to Java
    - class definitions, method dispatch and inheritance, exceptions, virtual inner classes, ...
- Learn python
  - Dynamically typed languages, better event model, functions, multiple inheritance
  - Try its GUI – much nicer and easier. E.g. play with an editor component and create your own editor with plugins. Or create a multi-user editor.
  - Try PyGame a framework for making 2d games REALLY easily
  - http://c2.com/cgi/wiki?BenefitsOfDynamicTyping