Objektorientierung

by

Dr. Günter Kolousek

Ursprung der Objektorientierung

- Programmiersprachen (programming language, PL)
  - Simula (1967)
  - Smalltalk (1972)
- AI (70er Jahre): Frames
  - Artificial intelligence (Künstliche Intelligenz)
- Datenmodellierung (80er Jahre)
- Benutzerinterface (80er Jahre)
- führte zu:
  - object-oriented programming (OOP)
  - object-oriented design (OOD)
  - object-oriented analysis (OOA)

Idee

- Struktur der Software orientiert sich an Realiät
  - Abbildung eines Ausschnittes bzw. Sichtweise der realen Welt
- Objekt: Gegenstand oder das Ziel des Denkens oder Handelns
  - Reale Welt enthält *Objekte*, die *interagieren*
- $\rightarrow$ OOP: Objekte, die zur Laufzeit exisistieren und interagieren
  - empfangen Nachrichten
  - verarbeiten und speichern Daten
  - senden Nachrichten

Konzepte

- Objekt
- Identität
- Klassifizierung
- Kapselung (Datenkapselung)
- Vererbung
- Polymorphie

Objekt (object)

- Konkretes oder abstraktes Ding
- Jedes Objekt hat
  - eine eindeutige Identität
    - außer… Wertobjekte
  - einen aktuellen Zustand
    - Instanzvariablen (instance variable, data member, attribute, Eigenschaft)
  - ein Verhalten
    - Operationen: meist Methoden (methods, member function)
- Objekte kommunizieren durch Nachrichten (message)
  - meist: Aufruf von Methoden
- In objektorientierten Programmiersprachen: Verhalten und Struktur des Zustandes $\rightarrow$ Klasse & Vererbung
  - … alternativ
    - objekt-basiert (keine Vererbung)
    - prototypen-basiert (Prototypenobjekte)

Identität – (object) identity

- eindeutig!
  - kein anderes Objekt hat gleiche Identität
- unabhängig vom internen Zustand
- Objektidentität $\neq$ Objektgleichheit
  - dasselbe vs. das Gleiche
- Seichte Gleichheit vs. tiefe Gleichheit
  - seichte Gleichheit (shallow equality)
    - interne Zustände sind identisch
  - tiefe Gleichheit (deep equality)
    - interne Zustände sind gleich (aber nicht identisch)
- Seichte Kopie vs. tiefe Kopie
  - shallow copy, deep copy

Klassifizierung – classification

- Objekte werden gruppiert
  - $\rightarrow$ Klassifikation
    - sinnvoll: nach Verhalten
    - alternativ: nach Eigenschaften, Struktur, Aussehen,…
- Objektklasse (object class)
  - Namensgebung:
    - Name der Klasse $\rightarrow$ Substantiv
    - Namen der Methoden $\rightarrow$ Verben
  - beschreibt Struktur des Zustandes und Verhalten von Objekten
  - realisiert (implementiert) einen *Typ*
  - Klasse selber kann
    - Verhalten aufweisen (Klassenmethoden)
    - Zustand aufweisen (Klassenvariablen)
  - Abstrakte Klasse hat keine Instanzen (abstract class)
  - in manchen PL: Klasse kann selbst Objekt sein!
    - $\rightarrow$ Klasse der Klasse = Metaklasse

Klassifizierung – Typ vs. Klasse

Designing good classes is hard because designing good types is hard. Good types have a natural syntax, intuitive semantics, and one or more efficient implementations

– Scott Meyers

Objekt – 2

- es können beliebig viele Instanzen (engl. instance, Exemplar, Ausprägung) einer Klasse erzeugt werden (Instanziierung)
- in OO: ist Instanz einer Klasse
  - Einfach- bzw. Mehrfachklassifikation
    - single inheritance: Java, C#
    - multi inheritance: C++, Python
- kann mehrere Typen haben
  - zeitlich gleichzeitig bzw. zeitlich änderbar (Rollen)

Kapselung – encapsulation

auch: information hiding

- Schutz vor unerlaubten Zugriff auf internen Zustand
  - durch eindeutig definierte Schnittstelle
  - Zugriff über Methoden
- Unterstützung in Programmiersprachen
  - durch Schlüsselwörter: `private`, `protected`, `public` (Java, C#, C++)
    - u.U. auch package-visibility (z.B. Java, C#)
  - Smalltalk: Instanzvariablen privat, Methoden öffentlich
  - Python: Instanzvariablen und Methoden öffentlich
    - außer: quasi-private beginnend mit `__`
  - Properties: Python, C#
- bezieht sich nicht nur auf Klassen!

Kapselung – 2

- Vorteile
  - Implementierung kann geändert werden
    - ohne andere Programmteile ändern zu müssen
  - weniger Abhängigkeiten zu anderen Programmteilen
  - verbesserte Übersichtlichkeit
  - besser zu testen
- Nachteile
  - Performanceverlust durch Funktionsaufrufe
  - Programmieraufwand

Schnittstelle – interface

- legt Menge von Signaturen fest
- unterspezifiziert
- keine Instanzen eines Interfaces
- keine Implementierung
- ”eine Klasse implementiert ein Interface”
  - Interface unterspezifiziert…
    - daher Implementierung der Signaturen ”irgendwie”
  - ADT ist voll spezifiziert…
    - $\rightarrow$ ”eine Klasse implementiert einen Typ”

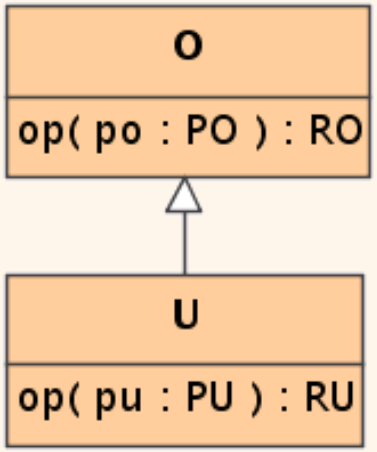
Vererbung – inheritance

- Mechanismus, um neue Klassen (bzw. Typen, Interfaces) aus existierenden Klassen (bzw. Typen, Interfaces) zu definieren
- Unterklasse erbt Instanzvariablen der Oberklasse
- Unterklasse (bzw. Typ, Interface) erbt Methoden der Oberklasse (bzw. Typ, Interface)
- Unterklasse kann
  - neue Instanzvariable oder Methoden (auch überladene[1]) definieren
  - geerbte Methoden überschreiben (overriding[2])
- Begriffe
  - Unterklasse (U): Subklasse, abgeleitete Klasse, Kindklasse
  - Oberklasse (O): Superklasse, Basisklasse, Elternklasse

[1] gleicher Name, aber unterschiedliche Anzahl bzw. Typen der Parameter
[2] gleiche Signatur wie Methode aus Oberklasse

Arten der Vererbung

- Mittels Vererbung wird eine Generalisierung erreicht



- 3 Arten der Generalisierung
  - Implementierungsvererbung
  - Spezialisierungsvererbung
  - Spezfikationsvererbung oder Subtypbeziehung

Begriffe

- Kovarianz: Der deklarierte Typ eines Elements im Untertyp ist ein Untertyp des deklarierten Typs des entsprechenden Elements im Obertyp
  - d.h. $PO \geq PU$ bzw. $RO \geq RU$
    - d.h.: Typhierarchie hat gleiche Richtung zur Vererbungshierarchie
- Kontravarianz: Der deklarierte Typ eines Elements im Untertyp ist ein Obertyp des deklarierten Typs des Elements im Obertyp
  - d.h. $PU \geq PO$ bzw. $RU \geq RO$
    - d.h.: Typhierarchie entgegen der Richtung der Vererbungshierarchie
- Invarianz: Der deklarierte Typ eines Elements im Untertyp ist gleich dem deklarierten Typ des entsprechenden Elements im Obertyp
  - d.h.: Typen sind in gleich: $PU = PO$ bzw. $RU = RO$

”Varianz” in C++: Vererbung

- Methode `RO O::op()` wird durch `RU U::op()` überschrieben
- $\rightarrow$ Signaturen müssen gleich sein!!!
- Typ des Rückgabewertes
  - `RO` = `RU` oder
  - `RO` ist `B*` bzw. `B&` $\rightarrow$ `RU` ist `D*` bzw. `D&`, wenn $B \geq D$
    - d.h. Pointer und Referenzen sind *kovariant*!

”Varianz” in C++: Vererbung – 2

```
struct Coat {};  struct DogCoat : Coat {};

struct Animal {
    virtual void make_noise() { puts(”beep beep”); }
    //virtual Coat coat() { return coat_; }
    virtual Coat& coat() { return coat_; }
    virtual void set_coat(Coat*) {};
    virtual ~Animal() = default;
  private:
    Coat coat_;
};
struct Dog : Animal {
    void make_noise() override { puts(”bow-wow”); }
    // DogCoat coat() override { return DogCoat{}; }
    // ->  invalid covariant return type for
    //         ‘virtual DogCoat Dog::coat()’
    DogCoat& coat() override { return coat_; }
    //void set_coat(DogCoat*) override;
    //  ‘void Dog::set_coat(DogCoat*)’ marked ‘override’,
    //    but does not override
  private:
    DogCoat coat_;
};
```

”Varianz” in C++: Vererbung – 3

```
int main() {
    Dog golu;
    golu.make_noise();  // -> bow-bow
    Animal* animal{&golu};
    animal->make_noise();  // -> bow-bow
}
```

”Varianz” in C++: Arrays

- Array von Subtypen $\rightarrow$ object slicing!
- keine Arrays von Referenzen!
- Array von Pointer $\rightarrow$ Polymorphie ✓
- keine Zuweisung von Arrays mit unterschiedlichen Typen
  - d.h. *invariant*
  - auch nicht, wenn diese in einer Vererbungsbeziehung stehen

”Varianz” in C++: Arrays – 2

```
// object slicing:
Animal animals[5]{ Dog{}, Dog{} };
animals[0].make_noise(); // -> beep beep
// why? object slicing:
Animal rex{golu};
rex.make_noise(); // -> beep beep

// keine Arrays von Referenzen:
// Animal& animals2[5];
// -> declaration of ‘animals2’ as array of references

// Array von Pointer:
Animal* animals2[5]{ new Dog{}, new Dog{} };
animals2[0]->make_noise();  // -> bow-bow

// keine Zuweisung von Arrays mit unterschiedlichen Typen
Dog* dogs[5];
//animals2 = dogs;  // incompatible types in assignment
//dogs = animals2;  // incompatible types in assignment
```

”Varianz” in C++: Templates

- Templates sind ebenfalls *invariant*!
  - jedes Mal ein neuer Typ

```
vector<Dog> dogs3;
vector<Animal> animals3;
//animals3 = dogs3;  //  no match for ‘operator=’...
//dogs3 = animals3;  //  no match for ‘operator=’...
```

- Abhilfe? → copy constructor, assignment operator!
  - z.B. `std::function` der Standardbibliothek

```
using AnimalDoctor = function<void(Animal*)>;
using DogDoctor = function<void(Dog*)>;

auto maxi{[](Animal*){}};
auto mini{[](Dog*){}};

DogDoctor a{mini};
DogDoctor b{maxi};
//AnimalDoctor c{mini};  // no matching function for call...
AnimalDoctor d{maxi};
```

”Varianz” in Java und C#

- Arrays in Java: *kovariant*

```java
class Animal {
}

class Dog extends Animal {
}

class Cat extends Animal {
}

public class Test {
    public static void main(String[] args) {
        Animal animals[] = new Animal[5];
        Dog dogs[] = new Dog[5];

        animals = dogs;
        animals[0] = new Dog();
        animals[1] = new Cat();
        // -> java.lang.ArrayStoreException: Cat
    }
}
```

→ broken also in C# (außer bei `System.ValueType`-Typen)!

”Varianz” in Java und C# – 2

- Java
  - Methoden
    - Rückgabetyp kovariant!
    - Parametertyp kontravariant!
- C#
  - Methoden: wie in C++
  - Delegates
    - Rückgabetyp kovariant
    - Parametertyp kontravariant

”Varianz” in C#: Vererbung

```
using System;

class Animal {}

class Dog : Animal {}

class AnimalBreeder {
    public virtual Animal make_animal() { return new Animal(); }
    //public virtual void say_hi(Animal a) { a.make_noise(); }
}

class DogBreeder : AnimalBreeder {
  //public override Dog make_animal() { return new Dog(); }
  // -> 'DogBreeder.make_animal()': return type must be 'Animal' to
  //     match overridden member 'AnimalBreeder.make_animal()'
  //public override void say_hi(Dog a) { d.make_noise(); }
  // ->  'DogBreeder.say_hi(Dog)': no suitable method found to overrid
}
```

”Varianz” in C#: delegates

```
public class Test {
  static Dog make_dog() { return new Dog(); }
  static void make_noise1(Dog d) { Console.WriteLine(”bow-bow”); }
  static void make_noise2(Animal d) { Console.WriteLine(”beep beep”);

  public static void Main() {
      Func<Animal> func = make_dog;
      Action<Dog> action1 = make_noise1;
      Action<Dog> action2 = make_noise2;
      //Action<Animal> action3 = make_noise1;
  }
}
```

Begriffe – 2

- Vorbedingung, engl. pre-condition, kurz *pre*
  - Bedingung, die am Beginn der Operation wahr sein muss, damit Operation wie spezifiziert funktioniert.
- Nachbedingung, engl. post-condition, kurz *post*
  - Bedingung, die am Ende der Operation wahr ist, wenn Vorbedingungen wahr sind.
- Invariante, engl. invariant, kurz *inv*
  - Bedingung, die während der gesamten Ausführung der Operation wahr ist. Z.B., dass Werte nicht geändert werden

Vor- und Nachbedingungen in C++23

- `expects`, `ensures`, `assert`
- modifier
  - `default`: runtime checking klein
  - `audit`: runtime checking groß
  - `axiom`: keine runtime checking

```
double sqrt(double x)
  [[ expects: x >= 0 ]]
  [[ ensures ret: ret * ret = x ]] {
    double res{0};
    while (1) {
        /* calculate something */
        [[ assert audit : res >= 0 ]];
        /* calculate something, too */
        /* break out when ready */
    }
    return res;
}
```

Implementierungsvererbung

- keine konzeptionelle Beziehung zwischen Ober- und Unterklasse wird vorausgesetzt.
- Vererbung von Eigenschaften (properties) steht im Vordergrund
- Motivation: code-sharing
- Alternative: Aggregation und Delegation
  - oder: private Vererbung in C++
- Beispiel: Ellipse als Unterklasse von Kreis
- Synonyme: Codeverbung, nicht-strikte Vererbung, willkürliche Vererbung

Spezialisierungsvererbung

- Taxonomische Beziehung (hierarchische Klassifikation) zwischen Ober- und Unterbegriffen
  - Wissensrepräsentation, semantische Datenmodellierung
- U is-a O: U Instanzen sind spezielle O Instanzen
  - $\rightarrow$ extensionale Ebene
    - Extension: Gesamtheit der Dinge über die sich Begriff erstreckt
    - d.h. Menge der Instanzen von U ist Teilmenge von O
- Beispiele: Integer is-a Rational, Kreis is-a Ellipse
- Bedingungen
  - $\forall o : pre(U :: op) \rightarrow pre(O :: op)$
    - speziell für Typen: PO $\geq$ PU (kovariant)
  - $\forall o : post(U :: op) \rightarrow post(O :: op)$
    - speziell für Typen: RO $\geq$ RU (kovariant)
- Synonym: is-a Beziehung (Achtung: Homonym!)

Spezifikationsvererbung

- Substitutionsprinzip
  - U muss alle öffentlichen Operationen von O anbieten
    - durch erben oder überschreiben
  - Die in U überschriebenen Operationen müssen in allen Situationen aufrufbar sein, in denen die Operation von O aufrufbar sind und kompatible Ergebnisse liefern
- in jeder Phase der SW Entwicklung!!!

Spezifikationsvererbung – 2

- Bedingungen
  - $\forall o : pre(O :: op) \rightarrow pre(U :: op)$
    - d.h. $U :: op$ darf keine strengeren Vorbedingungen voraussetzen als $O :: op$
    - speziell für Typen: PU $\geq$ PO (kontravariant)
  - $\forall o : post(U :: op) \rightarrow post(O :: op)$
    - d.h. $U :: op$ muss zumindest das erreichen, das auch $O :: op$ erreicht
    - speziell für Typen: RO $\geq$ RU (kovariant)
- Synonyme: Subtypbeziehung, strikte Vererbung, essentielle Vererbung

Liskov’sches Substitutionsprinzip

- Formulierung der Spezifikationsvererbung
- Liskov’sches Substitutionsprinzip
  - Barbara Liskov, 1988
  - Es muss gewährleistet sein, dass ein Exemplar eines Subtyps überall dort stehen kann, wo ein Exemplar des Supertyps erlaubt ist!
- Eine Methode im Basistyp darf nie durch eine Methode im Subtyp ersetzt werden, die
  - einen Parameter nicht ”verträgt”, den die Supertypmethode verträgt (kontravariant),
  - deklariert, abrupt mit einer Ausnahme enden zu können, mit der nicht auch die Supertyp-Methode hätte terminieren können (kovariant),
  - oder einen Rückgabewert liefert, den nicht auch die Supertyp-Methode hätte liefern können (kovariant).

Kreis-Ellipse Problem

- Kreis ist eine Ellipse $\rightarrow$ `Circle` von `Ellipse` ableiten
  - $\rightarrow$ Spezialisierungsvererbung
  - Methoden `stretch_x` und `stretch_y` in `Ellipse`?!
    - `Circle` erbt diese Methoden $\rightarrow$ dann kein Kreis mehr!
    - `Circle` überschreibt Methoden $\rightarrow$ Verstoß gegen Liskov'sches Substitutionsprinzip, da sich ein Objekt von `Kreis` nicht verhält wie man es sich von einer Ellipse erwaret

Kreis-Ellipse Problem – 2

- Ellipse hat eine Achse mehr als ein Kreis $\rightarrow$ `Ellipse` von `Circle` ableiten
  - $\rightarrow$ Implementierungsvererbung
  - aber widersinnig: Ellipse ist kein Subtyp von Kreis!
    - Methode `radius` von `Circle` wird an `Ellipse` vererbt
    - verstößt klarerweise ebenfalls gegen Liskov'sches Substitutionsprinzip

Kreis-Ellipse Problem – 3

- Keine Vererbungsbeziehung zwischen `Ellipse` und `Circle`!
  - u.U. gemeinsame Überklasse `GraphicElement`
  - u.U. gemeinsame Überklasse `CircleOrEllipse`
    - beinhaltet gemeinsame Funktionalität
  - u.U. `Circle::as_ellipse()` liefert veränderbare Instanz von `Ellipse`
- Nur Klasse `Ellipse` mit Methode `is_circle()`
- Klassen `Circle` und `Ellipse` *immutable* implementiert
  - $\rightarrow$ `Circle::stretch_x` liefert neues Objekt zurück
  - d.h. `Circle` kann von `Ellipse` abgeleitet sein

Binden – binding

- Zuordnung einer Nachricht (Methodenname) zum Code (Implementierung einer Methode)
- statisches Binden (static binding) ist Binden zur Übersetzungszeit
- dynamisches Binden (dynamic binding, late binding) ist Binden zur Laufzeit
- C++, C#: statisches und dynamisches Binden
- Java, Python, JavaScript: nur dynamisches Binden

Typgebundenheit – typing

- Typ gebunden an Objekte / Variablen
  - statisch getypt (statically typed)
    - Typ ist an Variable gebunden
    - liegt daher zur Übersetzungszeit fest
    - Beispiele: Java, C++, C#
  - dynamisch getypt (dynamically typed)
    - Typ ist an Objekt gebunden
    - ist daher erst zur Laufzeit bekannt
    - Beispiele: Python, JavaScript, Ruby, C#
    - Duck Typing

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

– James Whitcomb Riley (1849 - 1916, amerikanischer Schriftsteller)

Duck Typing in Python

```
class Duck:
    def quack(self):
        return ”Quaaaack!”

class Person:
    def quack(self):
        # glaubt eine Ente zu sein! ;-)
        return ”Quack!”

o = Duck()
print(o.quack())
o = Person()
print(o.quack())
```

Duck Typing – 2

Es geht eigentlich nicht darum was etwas *ist* sondern was etwas *kann*:

> *In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with.*
>
> *– Alex Martelli (Senior Staff Engineer, Google)*

C# ab Version 4.0

```
class Duck {
    void quack() {
        return ”Quaaaack!”;
    }
}

class Person {
    void quack() {
        return ”Quack!”;
    }
}

dynamic o = new Duck();
System.Console.WriteLine(o.quack());
o = new Person();
System.Console.WriteLine(o.quack());
```

Polymorphie – polymorphism

- Fähigkeit verschiedene Gestalt anzunehmen
- Polymorphe Operation kann auf Objekten verschiedener Klassen ausgeführt werden und jeweils eine andere Semantik haben.
  - wird erreicht durch:
    - Vererben, Überschreiben und dynamischem Binden von Methoden
    - Überladen (z.B. Methoden, Funktionen, Operatoren)
- Polymorphe Variable
  - kann im Laufe der Ausführung eines Programmes auf Instanzen verschiedener Klassen referenzieren.
  - hat eine
    - statische Klasse: wird bei der Deklaration spezifiziert und ist zur Übersetzungszeit bekannt (fix!)
    - dynamische Klasse: jeweils die Klasse des Objektes, das die Variable zur Laufzeit referenziert (variabel!)

First class object

In einer Programmiersprache ist ein Konstrukt (z.B. Funktion oder Klasse) ein first class object, wenn es

- in Variablen und Datenstrukturen gespeichert werden kann
- als Parameter übergeben werden kann
- als Return-Wert zurückgegeben werden kann
- zur Laufzeit erzeugt werden kann
- eine eigene Identität hat

First class object – 2

```
def add(a, b):
    return a + b

def sum_up(seq, f):
    acc = 0
    for x in seq:
        acc = f(acc, x)
    return acc

print(sum_up(range(1, 11), add))  # -> 55
print(id(add))  # -> z.B.: 3069449532
```

Typgebundenheit – 2

Implizite vs. explizite Typkonversion bei Variablen

- schwach getypt (weakly typed): PHP, JavaScript, Perl
  Typ wird implizit in beliebigen Typ gewandelt (implicitly type coercion),
  - z.B. in PHP:

```
$a = 9; $b = ”9”; $c = $a + $b; // -> 18
```

Typgebundenheit – 2

Implizite vs. explizite Typkonversion bei Variablen

- schwach getypt (weakly typed): PHP, JavaScript, Perl
  Typ wird implizit in beliebigen Typ gewandelt (implicitly type coercion),
  - z.B. in PHP:

```
$a = 9; $b = ”9”; $c = $a + $b; // -> 18
$zahl = 6 + ”7.7 Maxi und Minis”; // -> 13.7!!!
$zahl = 8 + ”Maxi und Minis-9”;  // -> 8!!!
```

  - z.B. in Javascript

```
a = 4; b = ”2”; c = a + b;  // -> ”42”
```

Typgebundenheit – 3

In Java, everything is an object.
In Closure, everything is a list.
In Javascript, everything is a terrible mistake.

– unbekannt

Typgebundenheit – 4

- stark getypt (strongly typed): Java, Python, C#
  Typ wird nicht implizit, sondern muss explizit konvertiert (explicitly type coercion), z.B.:

```
a=9; b="9"; c=a + int(b); # Python -> 18
```

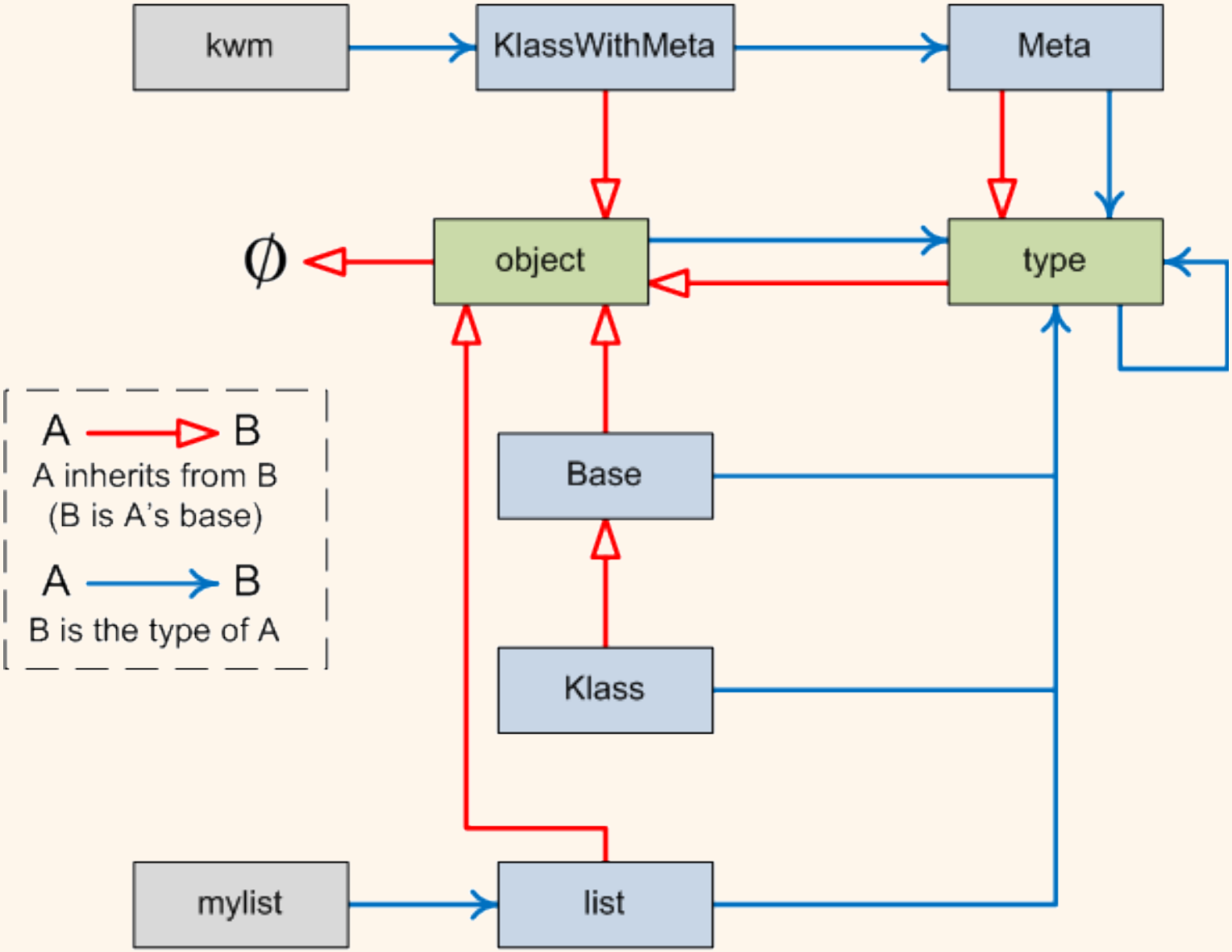
  Aber in Java:

```
out.println(a + " + " + b + " = " + a + b);
// -> 1 + 2 = 12
```

Metaklassen

```
class Base:
    pass

class Klass(Base):
    pass

class Meta(type):
    pass

class KlassWithMeta(metaclass=Meta):
    pass

kwm = KlassWithMeta()
```

Metaklassen – 2



Metaklassen – 3

```
>>> issubclass(Klass, Base) and issubclass(Base, object)
True
>>> issubclass(KlassWithMeta, object)
True
>>> kwm = KlassWithMeta()
>>> isinstance(kwm, KlassWithMeta)
True
>>> type(kwm)
<class '__main__.KlassWithMeta'>
>>> type(KlassWithMeta)
<class '__main__.Meta'>
>>> type(Meta)
<class 'type'>
>>> issubclass(Meta, type)
True
>>> type(object)
<class 'type'>
>>> type(type)
<class 'type'>
>>> issubclass(type, object)
True
```