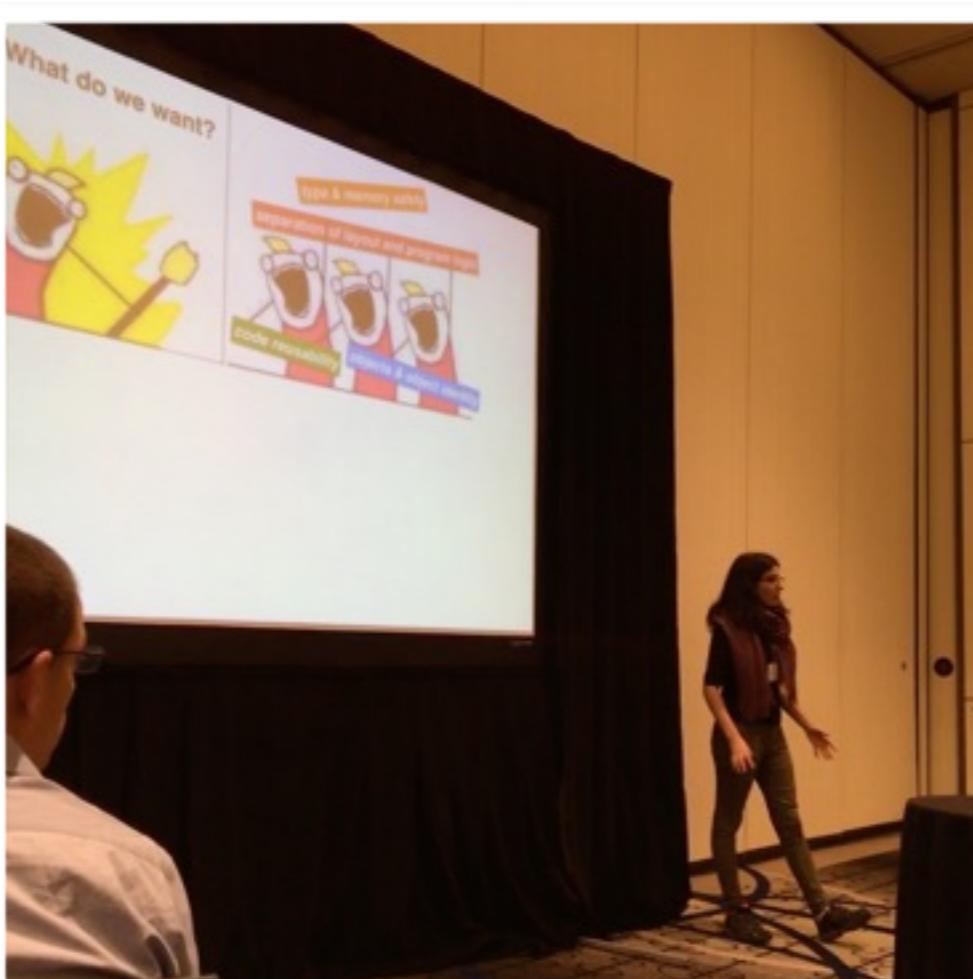




The ACM **SIGPLAN** conference on **Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)** embraces all aspects of software construction and delivery to make it the premier conference at the intersection of programming, languages, and software engineering.

OOPSLA (Object-Oriented Programming, Systems, Languages & Applications) is an annual ACM research conference. **OOPSLA** mainly takes place in the United States, while the sister conference of OOPSLA, **ECOOP**, is typically held in Europe.





SPLASH

-

-

-

-

-

• Achievements within the SPLASH-E-track in SPLASH 2017
• Onward! Performance: an Experience Report within the OCAP 2017-track in OCAP 2017
• Onward! Performance: an Experience Report within the Onward! Papers-track in Onward! 2017
• Languages within the OOPSLA-track in SPLASH 2017





Mastery Learning-Like Teaching with A

TOBIAS WRIGSTAD, Uppsala University
ELIAS CASTEGREN, Uppsala University

This paper describes the design of a second-year, 20 ECTS credit programming. The key design rhetoric is encouraging students to and, to this end, elements from mastery learning and other teaching an achievement-based system where students are put in charge addition to describing the elements of the course design, the paper format, and how the format has evolved over time.

1 INTRODUCTION

Four years ago at Uppsala University, we changed the 2s introduces imperative and object-oriented programming, v the new course design, we hoped to change the way students to assume more responsibility for their studies. This undertaking: it explains the key course design elements from its initial implementation to its current format, lessons qualitative self-evaluation.

First and foremost, it is important to understand the setti a country where university education is "free" in the sens students are given a basic allowance and state-subsidized I Programming Metho

Reference Capabilities for Concurrency & Scalability

ELIAS CASTEGREN, Uppsala University
TOBIAS WRIGSTAD, Uppsala University

In this presentation, we report on our work on Kappa, a We have used variations of Kappa to achieve data-race free and are currently using it to allow safe sharing of data

1 KAPPA: TYPES FOR CONCURRENCY AND

In previous work we introduced Kappa, a capability-[2]. Our design goals was to give static guarantees of sacrificing scalability. Furthermore, we want to encapsulation, code-reuse and subtype g

In Kappa, data-race freedom is achieved through with an object are safe (i.e., free from data-races); a n system makes sure that all aliases may also be safely as operations through its type. Additionally, the means of mode annotation. Kappa defines a taxonomy of modes example, a linear capability is the only reference to an observe mutation of the underlying object, and a local c

An important concept in object-oriented programmi object into different reusable modules, e.g., through inheritance of a capability separately, Kappa allows first defining the then specifying the means of concurrency control (the mod of an object can be specific to the module). In addition, the separation

You Can Have It All

Abstraction and Good Cache Performance

Juliana Franco
Imperial College London
United Kingdom
j.vicente-franco@imperial.ac.uk

Sophia Drossopoulou
Imperial College London
United Kingdom
s.drossopoulou@imperial.ac.uk

Abstract

On current architectures, the optimization of an application's performance often involves data being stored according to access affinity — what is accessed together should be stored together, rather than logical affinity — what belongs together logically stays together. Such low level techniques lead to faster, but more error prone code, and end up tangling the program's logic with low-level data layout details.

Our vision, which we call SHAPES—Safe, High-level, Abstractions for Optimization of memory caches — is that the layout of a data structure should be defined only once, upon instantiation, and the remainder of the code should be layout agnostic. This leads to performance improvements while also simplifying the code.

Martin Hagelin^{*}
Dirac
Sweden

Susan Eisenbach
Imperial College London
United Kingdom
s.eisenbach@imperial.ac.uk

ad hoc polymorphism i representations of diffe change of representati

CCS Concepts: • Software and its engineering → Garbage collection; Concurrent programming lan-

guages; Runtime environments; • Theory of computation → Type structures;

Orca: GC and Type System Co-Design for Actor Languages

SYLVAN CLEBSCH, Microsoft Research Cambridge, United Kingdom
JULIANA FRANCO, Imperial College London, United Kingdom
SOPHIA DROSSOPOULOU, Imperial College London, United Kingdom
ALBERT MINGKUN YANG, Uppsala University, Sweden
TOBIAS WRIGSTAD, Uppsala University, Sweden
JAN VITEK, Northeastern University, United States of America

Orca is a concurrent and parallel garbage collector for actor programs, which does not require any stop-the-world steps, or synchronisation mechanisms, and which has been designed to support zero-copy message passing and sharing of mutable data. Orca is part of the runtime of the actor-based language Pony. Pony's runtime was co-designed with the Pony language. This co-design allowed us to exploit certain language properties in order to optimise performance of garbage collection. Namely, Orca relies on the absence of race conditions in order to avoid read/write barriers, and it leverages actor message passing for synchronisation among actors. This paper describes Pony, its type system, and the Orca garbage collection algorithm. An evaluation of the performance of Orca suggests that it is fast and scalable for idiomatic workloads.

CCS Concepts: • Software and its engineering → Garbage collection; Concurrent programming languages; Runtime environments; • Theory of computation → Type structures;

Additional Key Words and Phrases: actors, messages

ACM Reference Format:

Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. 2017. Orca: GC and Type System Co-Design for Actor Languages. Proc. ACM Program. Lang. 1, OOPSLA, Article 72 (October 2017), 28 pages. <https://doi.org/10.1145/3133896>

1 INTRODUCTION

Pony is an object-oriented programming language designed from the ground up to support low-latency, highly concurrent applications written in the actor model of computation [Hewitt et al. 1973]. The impetus for a new language comes from the authors' experience with the requirements of financial applications, namely a need for i) scalable concurrency, from tens to thousands of concurrent components; ii) performance approaching that of low-level languages; and iii) ease of development and rapid prototyping. Alternatives such as Erlang and Java were considered but performance was felt to be inadequate for the former, and pauses due to garbage collection were a stumbling block for adoption of the latter.

This paper introduces Orca, Pony's concurrent garbage collection algorithm. Orca stands for Counting-based Garbage Collection in the Actor World. It was co-designed with the Pony type system and objects and to reclaim

Introduktion till OO

Tobias Wrigstad
tobias.wrigstad@it.uu.se

Föreläsning 15



What is object-oriented software

Objects are like people. They're living, breathing things that have knowledge inside them about how to do things and have memory inside them so they can remember things. And rather than interacting with them at a very low level, you interact with them at a very high level of abstraction [...]



Ole Johan Dahl

Kristen Nygaard






FLT PICA

©FLT-PICA, Stockholm, Sverige/Sweden. 1998-05-20 Foto Lennart Nygren/FLT-PICA code 30132 ***Arkivbild/File Picture
1978-01-13***
Arbete vid dataterminal i FOA:s datasystem

Jacob Palme



Why Smalltalk?

I made up the term “object-oriented,” and I can tell you I did not have C++ in mind.

– Alan Kay



Alan Kay





Objekt

- Gäst A, gäst B, gäst C...
- Kypare A, kypare B, ...
- Kock A, ...
- Tallrikar med mat
- En massa bord
- En massa stolar
- Dukar, glas, bestick, ...

Objektorienteringens grundkoncept

Aktiva och passiva objekt

Meddelandesändning

Aggregering

Inkapsling

Arv

Polymorfism

Objektorienteringens grundkoncept

Aktiva och passiva objekt

Aktiva: serveringspersonal, gäster, kockar

Passiva: maten, stolarna, borden, etc.

Meddelandesändning

Aggregering

Inkapsling

Arv

Polymorfism

Objektorienteringens grundkoncept

Aktiva och passiva objekt

Meddelandesändning

Mellan aktiva objekt: beställa mat

Aktiva–passiva objekt: dra ut stol, äta, lyfta en gaffel

Aggregering

Inkapsling

Arv

Polymorfism

Objektorienteringens grundkoncept

Aktiva och passiva objekt

Meddelandesändning

Aggregering

Ett bord består av en bordsskiva och fyra ben

Ett middagssällskap består av flera gäster

Inkapsling

Arv

Polymorfism

Objektorienteringens grundkoncept

Aktiva och passiva objekt

Meddelandesändning

Aggregering

Inkapsling

En gäst kan inte interagera direkt med kökspersonalen

Hur maten lagas (Det är inte uppenbart att det är hästkött i lasagnen, jmf. abstraktion)

Arv

Polymorfism

Objektorienteringens grundkoncept

Aktiva och passiva objekt

Meddelandesändning

Aggregering

Inkapsling

Arv

En Gäst är en Person, en Kypare är en Person, en Kock är en Person

Om $\mathcal{P}(\text{Person}) \Rightarrow \mathcal{P}(\text{Gäst}) \wedge \mathcal{P}(\text{Kypare}) \wedge \mathcal{P}(\text{Kock})$

Polymorfism

Objektorienteringens grundkoncept

Aktiva och passiva objekt

Meddelandesändning

Aggregering

Inkapsling

Arv

Polymorfism

Olika objekt kan ha samma gränssnitt

Olika maträtter smakar olika, personer agerar olika, etc.

Kockarna går också på restaurang som gäster, man kan dricka vin ur ölglas

Koncept

Objekt

Klasser – ritningar för objekt

Koncept

Objekt

Världen består av objekt (som består av objekt...) som skickar **meddelanden** till varandra

Objekt ”av samma sort” grupperas i **klasser** som beskriver hur objekten fungerar

Relationer mellan objekt: **aggregering** (objekt har referenser till andra objekt)

Ett objekts ”byggstenar” är inte direkt åtkomliga (**inkapsling**)

Klasser – ritningar för objekt

Koncept

Objekt

Världen består av objekt (som består av objekt...) som skickar **meddelanden** till varandra

Objekt ”av samma sort” grupperas i **klasser** som beskriver hur objekten fungerar

Relationer mellan objekt: **aggregering** (objekt har referenser till andra objekt)

Ett objects ”byggstenar” är inte direkt åtkomliga (**inkapsling**)

Klasser – ritningar för objekt

Beskriver inte bara vad ett objekt innehåller (**tillstånd**) utan också dess **beteende**

Relationer mellan klasser: **arv** (En pudel är en hund är ett djur är ett...)

Hur en klass är uppbygd internt är inte synligt utifrån (**inkapsling**)

Procedurell programmering

$f(x)$ – du bestämmer ”nu skall jag göra f på datat x ”

Objektorienterad programmering

$x.f()$ – du ber ”snälla objekt x , utför f ”



Statisk bindning i C

$f(x)$ – gcc väljer f beroende på x :s typ vid kompilering

Dynamisk bindning i Java

$x.f()$ – VM:en väljer f beroende på vad som finns i x under körning!



Introduktion till OOP

med Java

Tobias Wrigstad
tobias.wrigstad@it.uu.se



Objekt

- En samling data (tillstånd) samt operationer som opererar på datat
- Man kan skicka meddelanden till ett objekt

Objektet väljer själv vad som skall utföras som svar på ett meddelande

- Objekt-orienterad design är data-driven design

Vilka aktörer finns det?

Hur är de relaterade med **arv**, **aggregering**, **användning**, etc. (mer senare)

Klass (finns i nästan alla OO-språk)

- En klass är en ”ritning” från vilken man kan bygga oändligt många objekt
- Medlemmar

Instansvariabler (även fält)

Metoder

- En klass är ung. som en strukt + alla funktioner som opererar på strukten
- Saker vi skall prata om senare

Relationer mellan klasser

Åtkomstmodifikatorer

Arv

- **Instansiering:** att skapa ett objekt från en klass

Java

- Utvecklades av Sun (James Gosling) under 90-talets början, släpptes 1995
- Några designprinciper för Java

Enkelt, objektorienterat och familjärt

Robust och säkert

Arkitekturoberoende och portabelt

Snabbt

Tydligt

Klassen Foo måste ligga i Foo.java

Ett första Java-program

```
/**  
 * @author Tobias Wrigstad (tobias.wrigstad@it.uu.se)  
 * @date 2013-10-01  
 */  
public class Hello {  
    String who = null;  
    public Hello (String who) {  
        this.who = who;  
    }  
    public void greet() {  
        System.out.println("Hello " + who);  
    }  
    public static void main(String args[]) {  
        if (args.length > 0) {  
            Hello hello = new Hello(args[0]);  
            hello.greet();  
        } else {  
            System.out.println("Usage: java Hello <who>");  
        }  
    }  
}
```

Koncept

- Klass
- Objekt
- Instansvariabel
- Konstruktor
- Metod
- Main-metod
- Arrayer är objekt
- Instantiering
- Metodanrop
- Åtkomstmodifierare
- En vettig sträng-typ

```
/**  
 * @author Tobias Wrigstad (tobias.wrigstad@it.uu.se)  
 * @date 2013-10-01  
 */  
public class Hello {  
    String who = null;  
    public Hello (String who) {  
        this.who = who;  
    }  
    public void greet() {  
        System.out.println("Hello " + who);  
    }  
    public static void main(String args[]) {  
        if (args.length > 0) {  
            Hello hello = new Hello(args[0]);  
            hello.greet();  
        } else {  
            System.out.println("Usage: java Hello <who>");  
        }  
    }  
}
```

Kompilera och köra...

- Kompilatorn ”**javac**”

Förstår beroenden

Kompilerar till ”**Java-bytekod**”

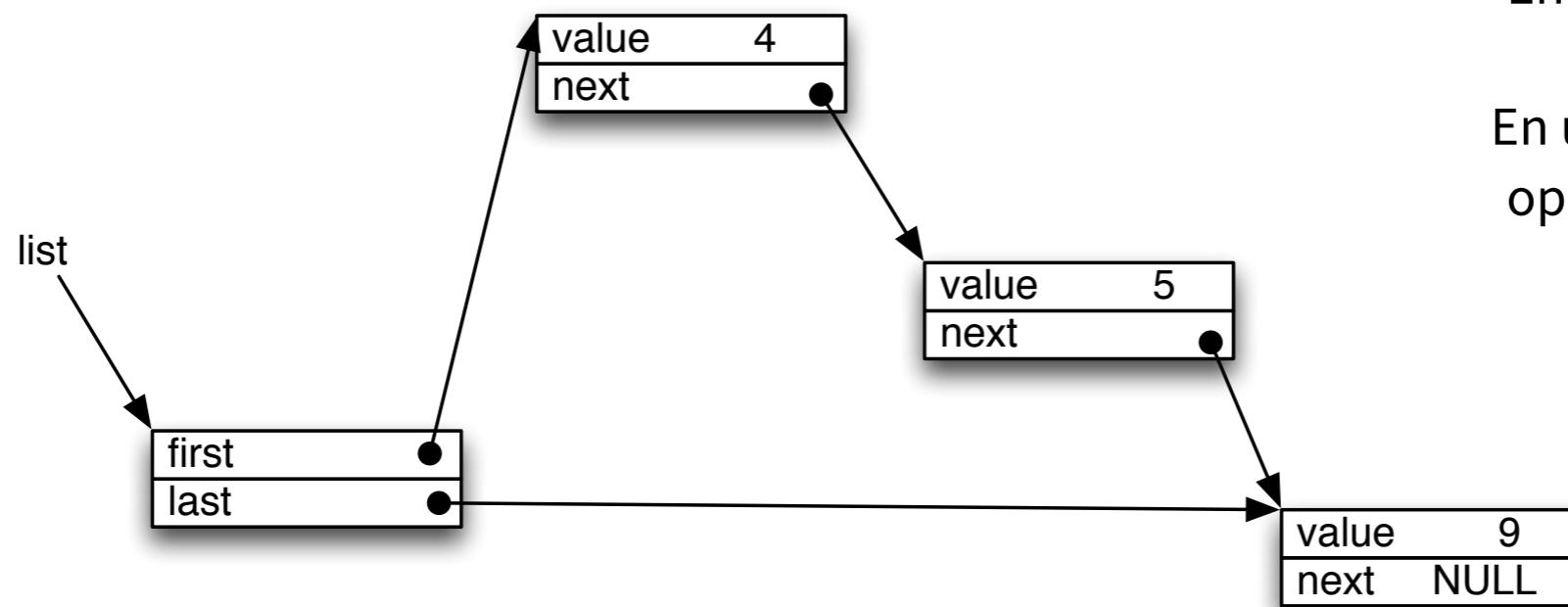
- Programmet måste köras i den virtuella maskinen

Programmet ”**java**”

Tar som argument namnet på en klass med en main-metod

```
$ javac Hello.java
$ ls
Hello.java
Hello.class
$ java Hello
Usage: java Hello <name>
$ java Hello "Tobias"
Hello Tobias!
$
```

Länkad lista i C

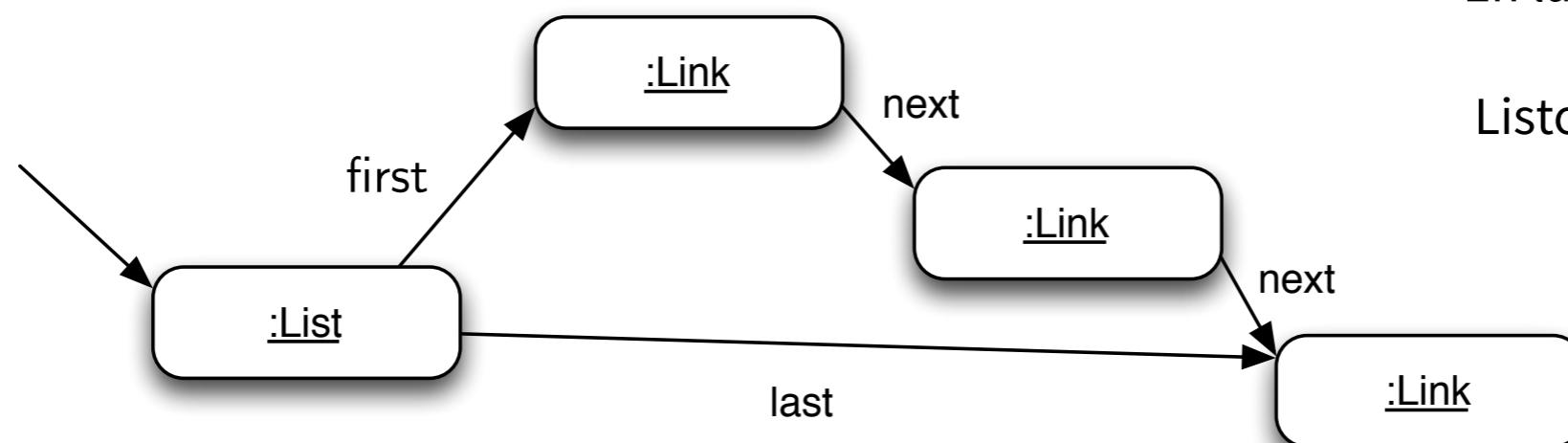


Varje länk är en allokerad struct

En länk *pekar* ut sin nästa länk

En uppsättning funktioner som
opererar på alla delar av listan

Länkad lista i Java



Varje länk är ett objekt

En länk *refererar* sin nästa länk

Listobjekten och länkobjekten
har separat tillstånd och
definierar ett eget
beteende



Demo: länkad lista i Java

- Ett program som alla borde känna igen från C

(kod kommer på GitHub)

Observationer

- Två klasser: List och Link

List-objekt aggregerar länk-objekt

Rekursion och iteration som i C (märk att det rekursiva anropet växlar mottagare!)

- Privat och publik åtkomst

Kräver set- och get-metoder i Link för privata medlemsvariabler

Referenser ≠ pekare

- En referens är ett handtag till ett objekt – det är inte en adress till en plats i minnet

Alla valida pekare i C pekar inte på det de skall (eller något alls)

Alla referenser i Java pekar alltid på det de skall och på någonting!

- Referensen null är inte adressen 0
- Den är inte heller ett booleskt värde
- Referenser möjliggör automatisk minneshantering (GC)

Automatisk minneshantering

- Java hanterar minnet automatiskt

`new ClassName(...)` allokerar automatiskt nog med minne på heapen

När sista referensen till ett objekt tas bort är objektet skräp

När minnet blir fullt städas skräp bort automatiskt för att lämna plats för nya objekt

- Alltså:

Ingen `malloc` (`new` allokerar alltid på heapen, åtminstone vad du vet!)

Ingen `free`

Förrädiskt likt C

- Syntaxen vald för att göra det enkelt för C och C++-programmerare att programmera Java
- Många konceptuella skillnader (Java är mer likt Smalltalk än C++)
- **Men:** vi kan ta med oss mycket från C!

While, for, if, switch, variabeldeklarationer, funktionsyntax, primitiva typer, etc....

I stort sett vet ni redan hur man programmerar Java, bara *inte hur man programmerar objektorienterat i Java!*

- Tag er i akt så ni inte programmerar C i Java!