Object-oriented programming: challenges for the next fifty years

Prof. Andrew P. Black
Portland State University,
Portland, Oregon, USA.

Portland State UNIVERSITY

UiO's 200th birthday is celebrated with a big concert, and the entire university comes together for a unique shared experience at Blindern!

Bigbang gives us an exclusive concert for our birthday! They invite a number of musical friends to play their own and each others' songs.

Food and drink on sale at Frederikkeplassen

19:00 Two options:

1. Birthday Party: Ole-Johan Dahl's House (IFI2)

UiO's newest venue Ole-Johan Dahl's house is inaugurated with a birthday party on three floors! There will be concerts, comedy, long tables and Oslo's longest bar.

2. Classic club: Georg Sverdrup's House

The foyer of Georg Sverdrup's house is transformed into classic club! A dream team of classical performers and UiO's own choirs and orchestra make for a festive evening. Artists: Arve Tellefsen, Elizabeth Norberg-Schulz and more

22.00 Afterparty, Chateau Neuf

The party rounds up with a packed club night at Betong and the rest of Chateau Neuf. Here you can dance the night away or keep the conversations going late into the night.

Hel
Ser
Luc
Ot
For
ær
Jub
Ær
pris
Off
Dal
Fes
Uni
åpr
Uni
jubi

Just suppose ...

Just suppose ...

- You have been “drafted”

Just suppose ...

- You have been "drafted"
- Your assignment:

Just suppose ...

- You have been "drafted"
- Your assignment:

    design your country's first nuclear reactor

Just suppose ...

- You have been “drafted”
- Your assignment:

    design your country’s first nuclear reactor

- What would *you* do?

Just suppose ...

- You have been "drafted"
- Your assignment:

  design your country's first nuclear reactor

- What would *you* do?
- What did Kristen Nygaard do?

A Little History

1948: Nygaard conscripted into the Norwegian Defense Research Establishment

1949–1950: Resonance absorption calculations related to the construction of Norway's first nuclear reactor. Introduced "Monte Carlo" simulation methods

1950-1952: Head of the "computing office"

1960: Moved to the Norwegian Computing Centre. "Many of the civilian tasks turned out to present the same kind of methodological problems: the necessity of using simulation, the need of concepts and a language for system description, lack of tools for generating simulation programs." [Nygaar1981]

1961: Started designing a simulation language

[Nygaar1981] K. Nygaard and O.-J. Dahl. The development of the SIMULA languages. In R. L. Wexelblat, editor, History of programming languages I, chapter IX, pages 439–480. ACM, New York, NY, USA, 1981.

Nygaard's Famous Letter: 5th January 1962

"*The status of the Simulation Language* (*Monte Carlo Compiler*) *is that I have rather clear ideas on how to describe queueing systems, and have developed concepts which I feel allow a reasonably easy description of large classes of situations. I believe that these results have some interest even isolated from the compiler, since the presently used ways of describing such systems are not very satisfactory. … The work on the compiler could not start before the language was fairly well developed, but this stage seems now to have been reached. The* expert programmer *who is interested in this part of the job will meet me tomorrow. He has been rather optimistic during our previous meetings*." [Nygaar1981]

Ole-Johan Dahl

The “Expert Programmer”

Ole-Johan Dahl

1931–2002

Norway's foremost computer scientist

With Kristen Nygaard, produced initial ideas for Object-oriented programming

Ole-Johan Dahl

Honours:

Royal Norwegian Order of St. Olav (2000)

ACM Turing Award (2001)

IEEE von Neumann Medal (2002)

ACM Turing Award Citation

"… to Ole-Johan Dahl and Kristen Nygaard of Norway for their role in the invention of object-oriented programming, the most widely used programming model today.

… the core concepts embodied in their object-oriented methods were designed for both system description and programming … ”

Today's Talk:

- What are those "core concepts"?
- How they have evolved over the last 50 years.
- How they might adapt to the future.

General Program for the Centennial Celebration Massachusetts Institute of Technology, Cambridge April 7, 8, and 9, 1961

Saturday April 8

PANEL, 10:00 A.M., ROCKWELL CAGE. How Has Science in the Last Century Changed Man's View of Himself? JEROME S. BRUNER, ALDOUS HUXLEY, J. ROBERT OPPENHEIMER, and PAUL J. TILLICH.

PANEL, 10:00 A.M., KRESGE AUDITORIUM. The Future of the Arts in a World of Science. LUKAS FOSS, HOWARD MUMFORD JONES, LOUIS I. KAHN, and RICHARD LIPPOLD.

PANEL, 10:00 A.M., COMPTON LECTURE HALL. The Future in the Physical Sciences. SIR JOHN D. COCKCROFT, RICHARD P. FEYNMAN, RUDOLF PEIERLS, and CHEN NING YANG.

PANEL, 2:30 P.M., ROCKWELL CAGE. Arms Control. PAUL M. DOTY, HERMAN H. KAHN, RICHARD S. LEGHORN, and THE RIGHT HONORABLE PHILIP J. NOEL-BAKER.

PANEL, 2:30 P.M., KRESGE AUDITORIUM. The Life of Man in Industry. WILLIAM O. BAKER, EDWIN H. LAND, FRANK PACE, JR., and WILLIAM H. WHYTE.

PANEL, 2:30 P.M., COMPTON LECTURE HALL. The Future in the Life Sciences. GEORGE W. BEADLE, PETER B. MEDAWAR, HERMANN J. MULLER, and DR. JONAS E. SALK.

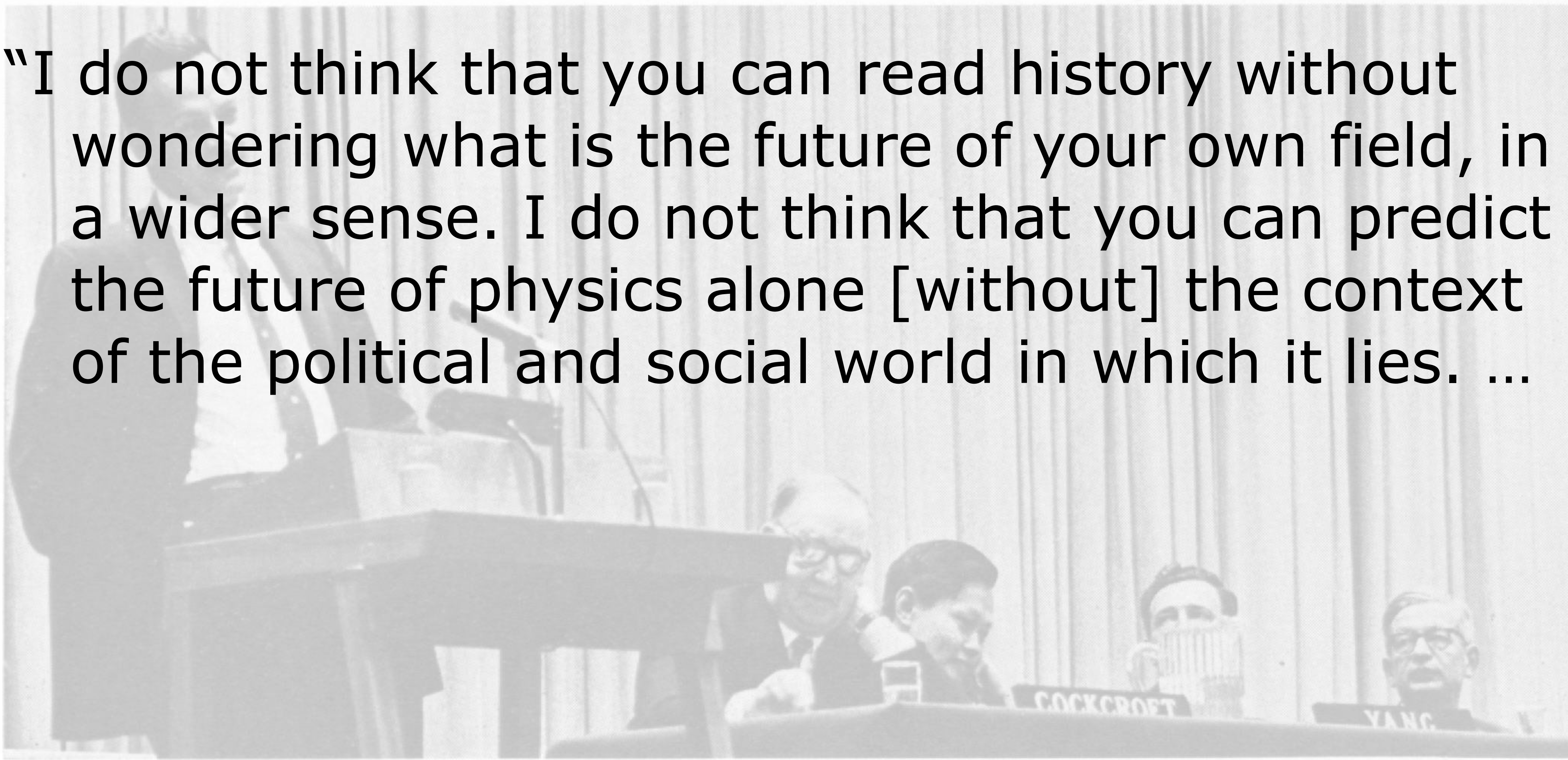
Feynman's speech:

Richard Feynman, '39, speaking with (from left) Sir John Cockcroft, Chen Ning Yang, Francis Low, and R. E. Peierls.

[Feynman1962] Transcript of Speech from the Feynman Archives at CalTech

Feynman's speech:

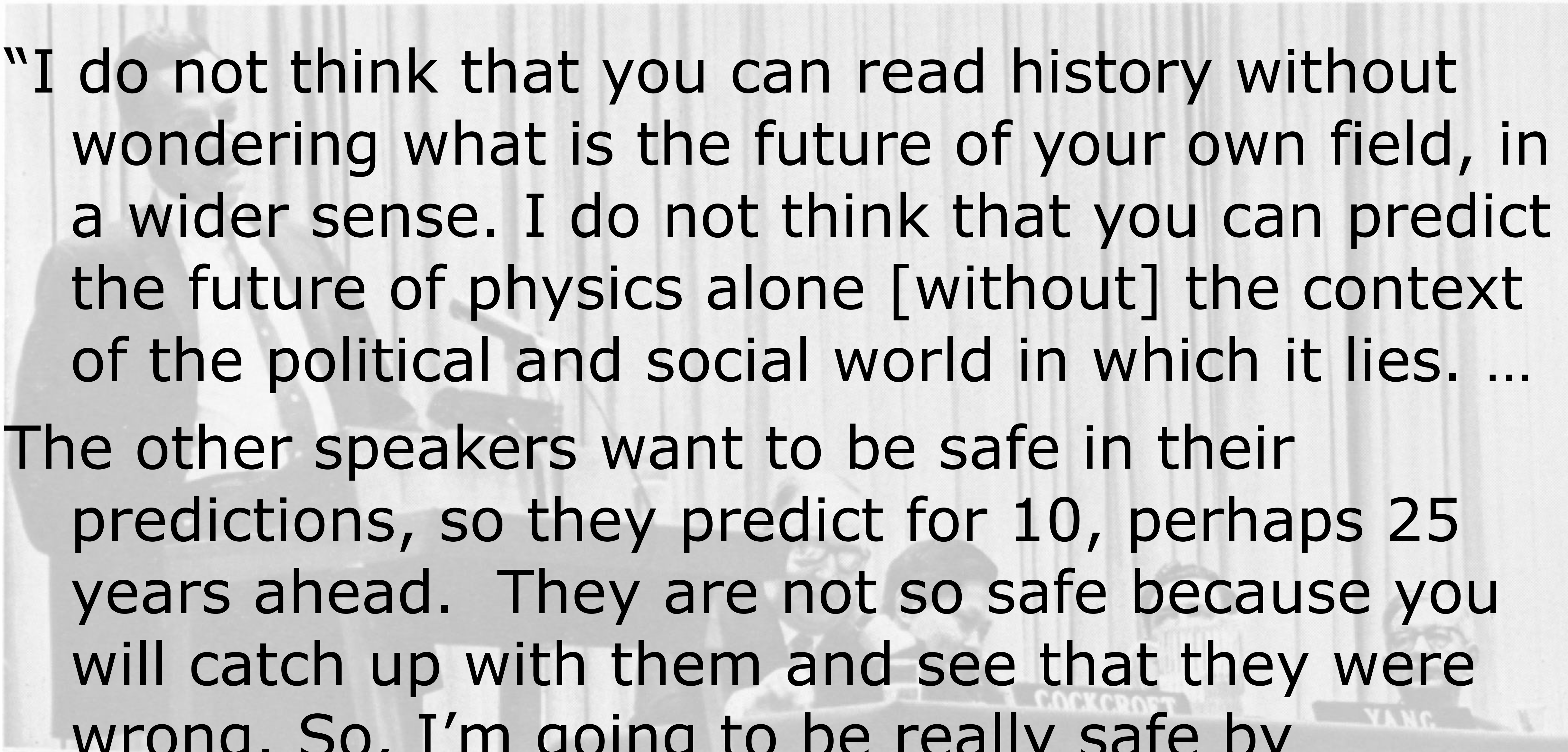
"I do not think that you can read history without wondering what is the future of your own field, in a wider sense. I do not think that you can predict the future of physics alone [without] the context of the political and social world in which it lies. …



Richard Feynman, '39, speaking with (from left) Sir John Cockcroft, Chen Ning Yang, Francis Low, and R. E. Peierls.

[Feynman1962] Transcript of Speech from the Feynman Archives at CalTech

Feynman's speech:



"I do not think that you can read history without wondering what is the future of your own field, in a wider sense. I do not think that you can predict the future of physics alone [without] the context of the political and social world in which it lies. …

The other speakers want to be safe in their predictions, so they predict for 10, perhaps 25 years ahead. They are not so safe because you will catch up with them and see that they were wrong. So, I'm going to be really safe by predicting 1000 years ahead." [Feynman1962]

Richard Feynman, speaking with (from left) Sir John Cockcroft, Chen Ning Yang, Francis Low, and R. E. Peierls.

[Feynman1962] Transcript of Speech from the Feynman Archives at CalTech

Political and Social Context

Political and Social Context

1. Simula was designed as process description language as well as a programming language.

Political and Social Context

1. Simula was designed as process description language as well as a programming language.

   When SIMULA I was put to practical work it turned out that to a large extent it was used as a system description language. A common attitude among its simulation users seemed to be: sometimes actual simulation runs on the computer provided useful information. The writing of the SIMULA program was almost always useful, since … it resulted in a better understanding of the system. [Nygaar1981]

Political and Social Context

[Ungar2011] David Ungar, Personal Communication

Political and Social Context

2. Nygaard had been using simulations to design Nuclear reactors.

[Ungar2011] David Ungar, Personal Communication

Political and Social Context

2. Nygaard had been using simulations to design Nuclear reactors.

   He did not want to be responsible for the first nuclear accident on the continent of Europe. [Ungar2011]

[Ungar2011] David Ungar, Personal Communication

Core Ideas of SIMULA

According to Nygaard:

1. Modelling

   The actions and interactions of the objects created by the program model the actions and interactions of the real-world objects that they are designed to simulate.

2. Security

   The behavior of a program can be understood and explained entirely in terms of the semantics of the programming language in which it is written.

Core Ideas of SIMULA

According to Dahl:
[Dahl1981]

1. Record structures

2. Procedural data abstraction

3. Processes
4. Prefixing (inheritance)
5. Modules

[Dahl1981] O.-J. Dahl. Transcript of discussant's remarks. In R. L. Wexelblat, editor, History of programming languages I, chapter IX, pages 488–490. ACM, New York, NY, USA, 1981.

Core Ideas of SIMULA

According to Dahl: all came from the Algol 60 block [Dahl1981]

1. Record structures (block with variable declarations but no statements)
2. Procedural data abstraction (block with variable and procedure declarations)
3. Processes (detached blocks)
4. Prefixing (inheritance) (prefix blocks)
5. Modules (nested blocks)

The SIMULA **class** construct

All these ideas were realized as special cases of a single general construct: the **class**.

But object-oriented programming is *not* class-oriented programming!

> Dahl wrote: "I know that SIMULA has been criticized for perhaps having put too many things into that single basket of class. Maybe that is correct; I'm not sure myself. But it was certainly great fun during the development of the language to see how the block concept could be remodeled in all these ways" [Dahl1981]

The Origin of the Core Ideas

[Nygaar1981a]

[Nygaar1981a] K. Nygaard. Transcript of presentation. In R. L. Wexelblat, editor, History of Programming Languages I, chapter IX, pages 480–488. ACM, New York, NY, USA, 1981.

The Origin of the Core Ideas

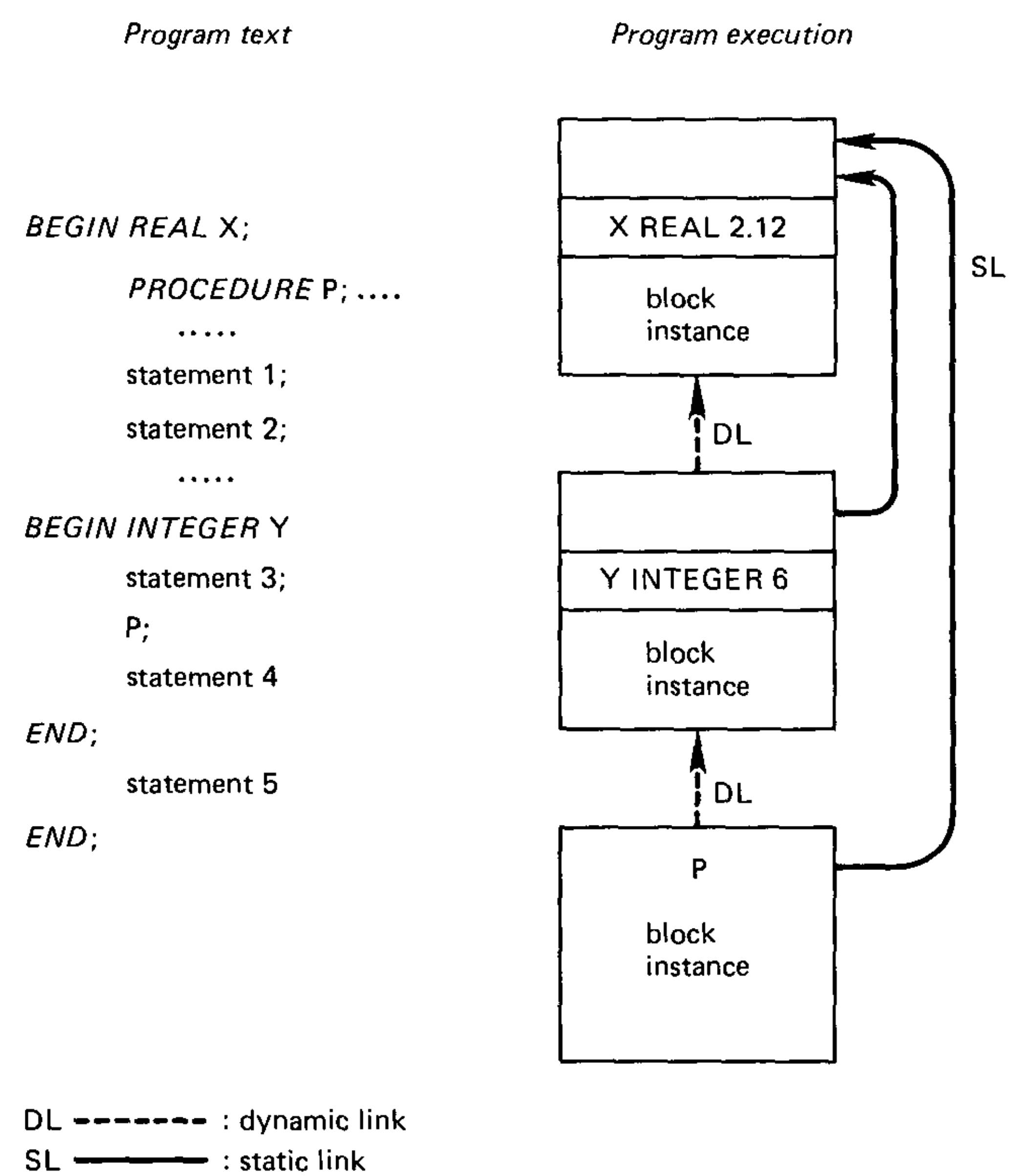
Dahl was inspired by visualizing the *runtime representation* of an Algol 60 program.

[Nygaar1981a]

[Nygaar1981a] K. Nygaard. Transcript of presentation. In R. L. Wexelblat, editor, History of Programming Languages I, chapter IX, pages 480–488. ACM, New York, NY, USA, 1981.

The Origin of the Core Ideas

Dahl was inspired by visualizing the *runtime representation* of an Algol 60 program.



Frame 5

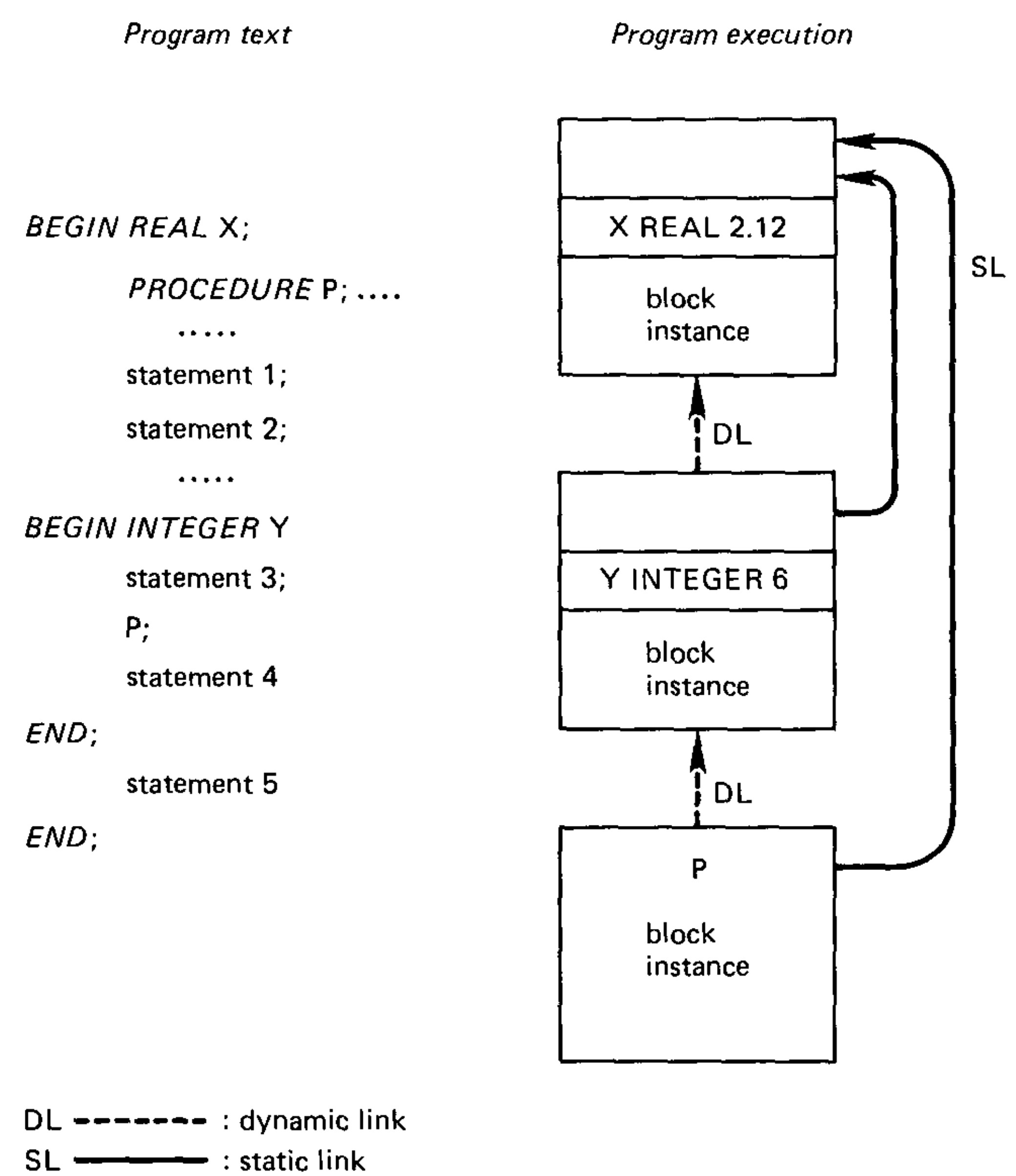
[Nygaar1981a]

[Nygaar1981a] K. Nygaard. Transcript of presentation. In R. L. Wexelblat, editor, History of Programming Languages I, chapter IX, pages 480–488. ACM, New York, NY, USA, 1981.

The Origin of the Core Ideas

Dahl was inspired by visualizing the *runtime representation* of an Algol 60 program.

Objects were already in existence inside every executing Algol program — they just needed to be freed from the "stack discipline"



Frame 5

[Nygaar1981a]

[Nygaar1981a] K. Nygaard. Transcript of presentation. In R. L. Wexelblat, editor, History of Programming Languages I, chapter IX, pages 480–488. ACM, New York, NY, USA, 1981.

Algol 60's "Stack discipline"

[Dahl1972] O.-J. Dahl and C. Hoare. Hierarchical program structures.
In Structured Programming, pages 175–220. Academic Press, 1972.

Algol 60's "Stack discipline"

"In ALGOL 60, the rules of the language have been carefully designed to ensure that the lifetimes of block instances are nested, in the sense that those instances that are latest activated are the first to go out of existence. It is this feature that permits an ALGOL 60 implementation to take advantage of a stack as a method of dynamic storage allocation and relinquishment. But it has the disadvantage that a program which creates a new block instance can never interact with it as an object which exists and has attributes, since it has disappeared by the time the calling program regains control. Thus the calling program can observe only the results of the actions of the procedures it calls. Consequently, the operational aspects of a block are overemphasised; and algorithms (for example, matrix multiplication) are the only concepts that can be modelled." [Dahl1972]

[Dahl1972] O.-J. Dahl and C. Hoare. Hierarchical program structures. In Structured Programming, pages 175–220. Academic Press, 1972.

Two simple changes:

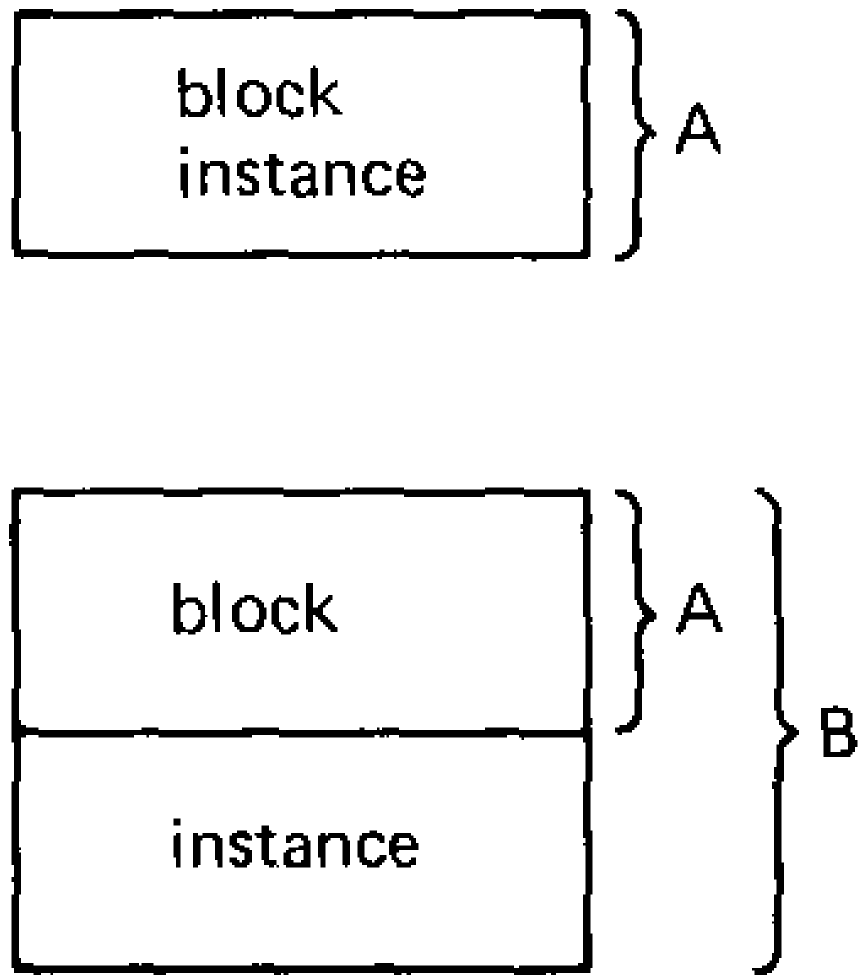
"In SIMULA 67, a block instance is permitted to outlive its calling statement, and to remain in existence for as long as the program needs to refer to it." [Dahl1972]

A way of referring to "it": object references as data

Simula Class Prefixing

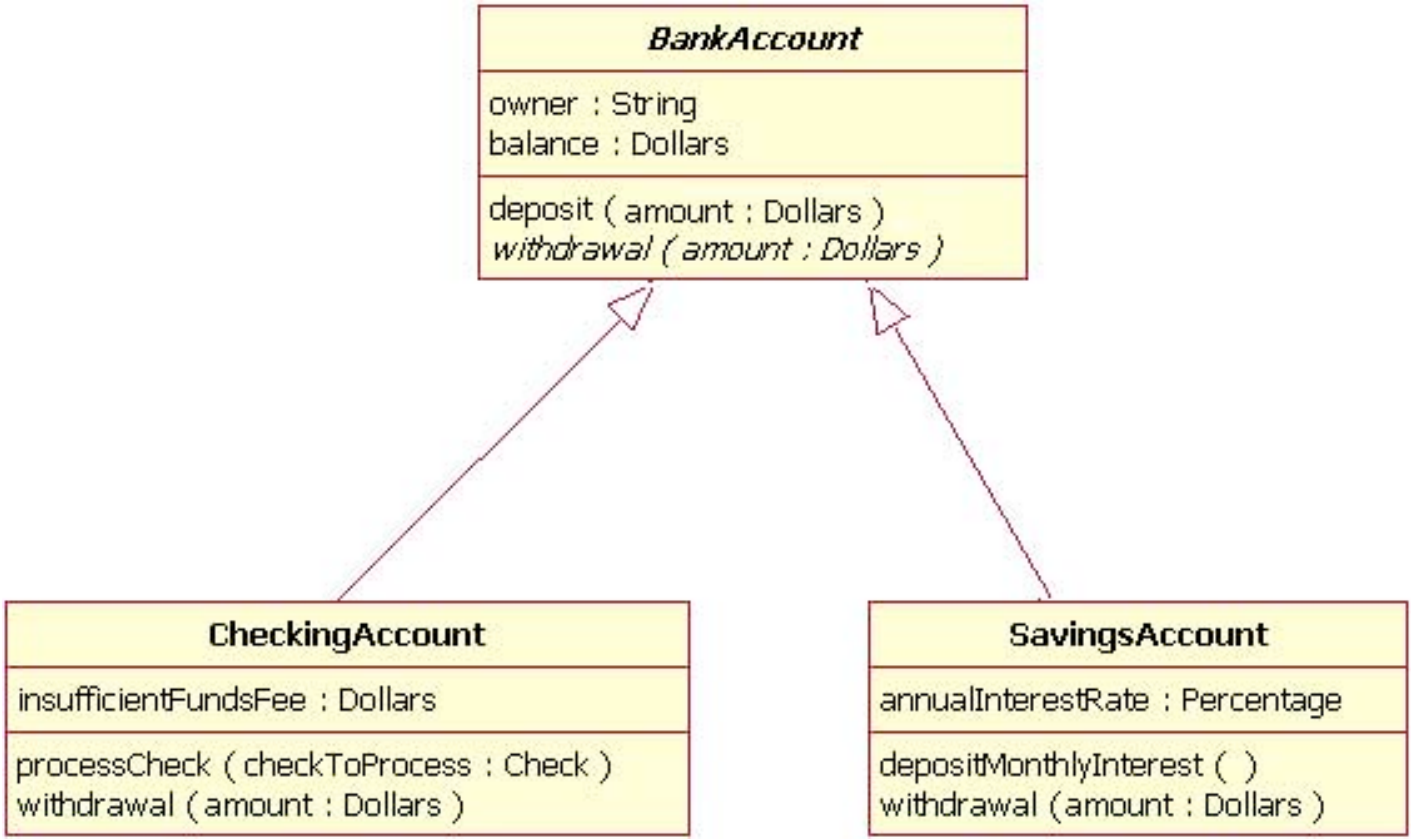
prefixing

```
CLASS A; ...
REF (A) X;
.....

X: -NEW A

A CLASS B; ...
REF (B) Y;
....

Y: -NEW B
```



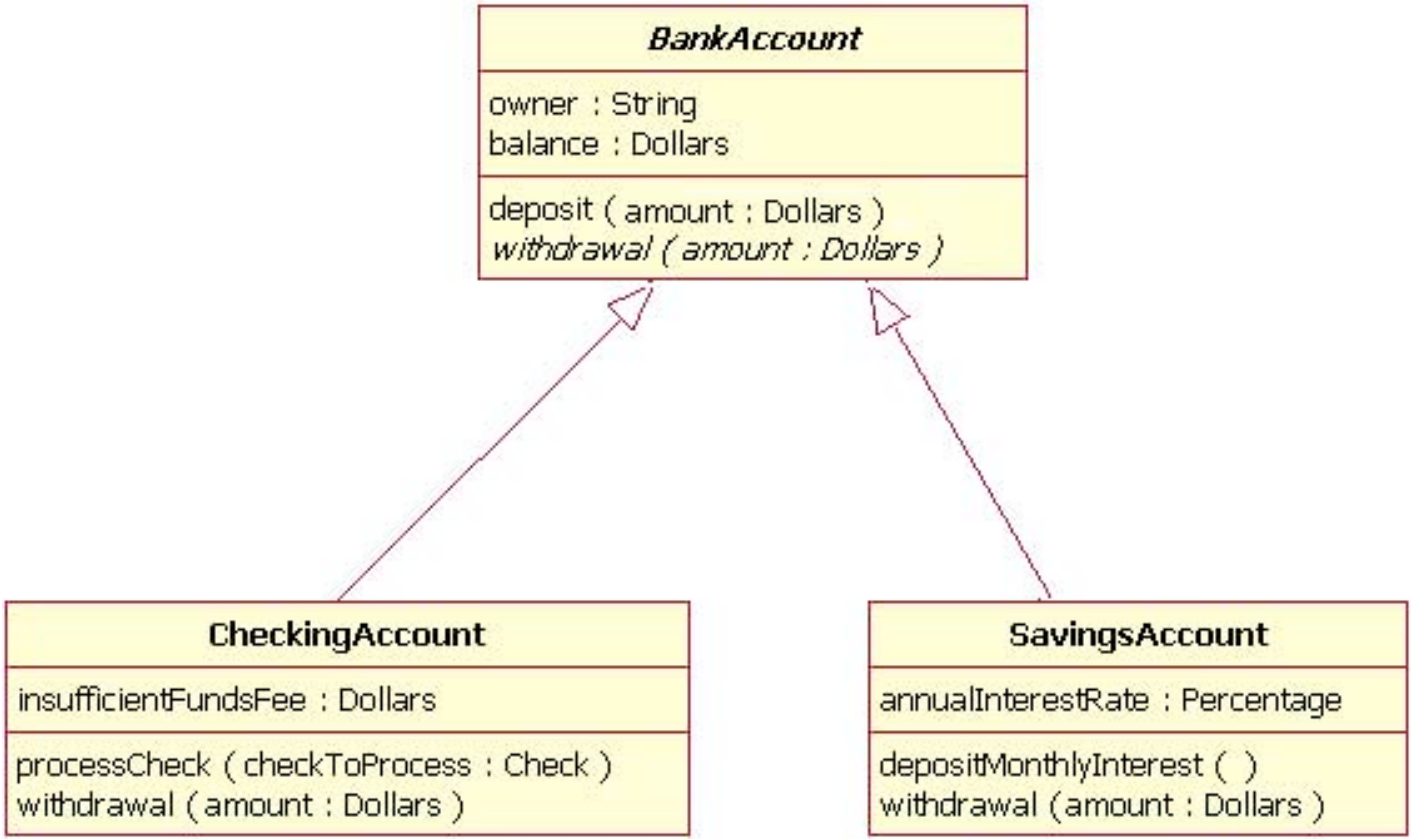
Frame 8
[Nygaar1981a]

Modern Class Prefixing



diagrams from IBM developerworks

Modern Class ~~Prefixing~~ Inheritance



diagrams from IBM developerworks

The Importance of Inheritance

Since 1989, thanks to William Cook, we have known that inheritance can be translated into fixpoints of generators of self-referential functions. [Cook1989a]

So much for the theory.

Are functions parameterized by functions as good as inheritance?

In theory: yes.

[Cook1989a] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In Conference on Object-oriented programming systems, languages and applications, pages 433–443, New Orleans, LA USA, 1989. ACM Press.

The Importance of Inheritance

Since 1989, thanks to William Cook, we have known that inheritance can be translated into fixpoints of generators of self-referential functions. [Cook1989a]

So much for the theory.

Are functions parameterized by functions as good as inheritance?

In theory: yes.

In practice: no.

[Cook1989a] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In Conference on Object-oriented programming systems, languages and applications, pages 433–443, New Orleans, LA USA, 1989. ACM Press.

Parameterized functions instead of Inheritance?

Parameterized functions instead of Inheritance?

When you parameterize a function, you have to *plan ahead* and make parameters out of every part that could possibly change.

functional programmers call this “abstraction”

Parameterized functions instead of Inheritance?

When you parameterize a function, you have to *plan ahead* and make parameters out of every part that could possibly change.

functional programmers call this "abstraction"

Two problems:

1. Life is uncertain
2. Most people think better about the concrete than the abstract

The value of Inheritance

When you inherit from a class or an object, you still have to *plan ahead* and make methods out of every part that could possibly change.

o-o programmers call this "writing short methods"

Two benefits:

1. You don't have to get it right
2. The short methods are *concretions*, not abstractions

Inheritance Example

```
Rectangle extends Object
        def bounds          — my bounding box
        def inset           — space around me
Rectangle » drawOn(aCanvas)
        self drawFrameOn(aCanvas)
        self fillRegionOf(aCanvas)
Rectangle » drawFrameOn(aCanvas)
        aCanvas strokeRectangle(bounds+inset)
Rectangle » fillRegionOf(aCanvas)
        aCanvas fillRectangle(bounds)
```

Inheritance Example

```
Circle extends Rectangle
      def radius        — my radius

Circle » fillRegionOf(aCanvas)
      aCanvas
            fillCircleWithCenterAndRadius
                  (bounds center, radius)
```

People Learn from Examples

Inheritance provides a concrete example,and then generalizes from it.

For example:

1. Solve the problem for n = 4
2. Then make the changes necessary for 4 to approach infinity

Object-oriented Frameworks

Object-oriented Frameworks

In my view, one of the most significant contributions of SIMULA 67:

Object-oriented Frameworks

In my view, one of the most significant contributions of SIMULA 67:

Allowing SIMULA to be expressed as a framework within SIMULA 67

Object-oriented Frameworks

In my view, one of the most significant contributions of SIMULA 67:

Allowing SIMULA to be expressed as a framework within SIMULA 67

SIMULA begin … end

Object-oriented Frameworks

In my view, one of the most significant contributions of SIMULA 67:

Allowing SIMULA to be expressed as a framework within SIMULA 67

SIMULA begin … end

SIMULATION begin … end

Object-oriented Frameworks

In my view, one of the most significant contributions of SIMULA 67:

Allowing SIMULA to be expressed as a framework within SIMULA 67

SIMULA begin … end

SIMULATION begin … end

class SIMSET, and
SIMSET class SIMULATION

What is an O-O Framework?

Generalization of a subroutine library:

- client calls subroutines in a library, which always return to the caller
- in a Framework:
  - client method calls methods in the framework
  - framework methods call methods in the client
- e.g., a simulation framework might tell objects representing reactor control rods or industrial saws to *perform*
- *perform* methods might ask the framework about environmental conditions

Smalltalk

Smalltalk-72 was clearly inspired by Simula

- It took:

  Classes, Objects, Inheritance, Object References

- It refined and explored:

  Objects as *little computers*: "a recursion on the notion of computer itself" [Kay1993]

  Objects combining data and the operations on that data

- It dropped:

  Objects as processes, classes as packages

[Kay1993] A. C. Kay. The early history of Smalltalk. In The second ACM SIGPLAN conference on History of programming languages, HOPL-II, chapter XI, pages 511–598. ACM, New York, NY, USA, 1993.

From Snyder: The Essence of Objects [Snyder1991]

Warning:

Unlike Simula and Smalltalk, this is a descriptive work, not a prescriptive one

[Synder1991] A. Snyder. The essence of objects: Common concepts and terminology. Technical Report HPL-91-50, June 1991.

From Snyder: The Essence of Objects [Snyder1991]

The essential concepts

- An object embodies an **abstraction** characterized by **services.**
- Clients request services from objects.

  Clients issue **requests.**

  Objects are **encapsulated.**

  Requests identify **operations.**

  Requests can identify objects.
- New objects can be **created.**
- Operations can be **generic.**
- Objects can be **classified** in terms of their services (**interface hierarchy**).
- Objects can share implementations.

  Objects can share a common implementation (multiple **instances**).

  Objects can share partial implementations
  (**implementation inheritance** or **delegation**).

US & Scandinavian Objects

Feature	Simula 67	Smalltalk 80	Snyder (1991)
Abstraction	"Modelling": attributes exposed	attributes encapsulated	Objects characterized by offered services
Active Objects	Yes	No	"Associated Concept"
Dynamic Objects	Yes	Yes	Yes
Classes	Yes	Yes	"Shared implementations"
Subclassing	Yes	Yes	"shared partial implementations"
Overriding	under control of superclass	under control of subclass	optional; delegation permitted
Classes as packages	Yes	No	No

US & Scandinavian Objects

Feature	Simula 67	Smalltalk 80	Snyder (1991)
Abstraction	"Modelling": attributes exposed	attributes encapsulated	Objects characterized by offered services
Active Objects	Yes	No	"Associated Concept"
Dynamic Objects	Yes	Yes	Yes
Classes	Yes	Yes	"Shared implementations"
Subclassing	Yes	Yes	"shared partial implementations"
Overriding	under control of superclass	under control of subclass	optional; delegation permitted
Classes as packages	Yes	No	No

US & Scandinavian Objects

Feature	Simula 67	Smalltalk 80	Snyder (1991)
Abstraction	"Modelling": attributes exposed	attributes encapsulated	Objects characterized by offered services
Active Objects	Yes	No	"Associated Concept"
Dynamic Objects	Yes	Yes	Yes
Classes	Yes	Yes	"Shared implementations"
Subclassing	Yes	Yes	"shared partial implementations"
Overriding	under control of superclass	under control of subclass	optional; delegation permitted
Classes as packages	Yes	No	No

US & Scandinavian Objects

Feature	Simula 67	Smalltalk 80	Snyder (1991)
Abstraction	"Modelling": attributes exposed	attributes encapsulated	Objects characterized by offered services
Active Objects	Yes	No	"Associated Concept"
Dynamic Objects	Yes	Yes	Yes
Classes	Yes	Yes	"Shared implementations"
Subclassing	Yes	Yes	"shared partial implementations"
Overriding	under control of superclass	under control of subclass	optional; delegation permitted
Classes as packages	Yes	No	No

US & Scandinavian Objects

Feature	Simula 67	Smalltalk 80	Snyder (1991)
Abstraction	"Modelling": attributes exposed	attributes encapsulated	Objects characterized by offered services
Active Objects	Yes	No	"Associated Concept"
Dynamic Objects	Yes	Yes	Yes
Classes	Yes	Yes	"Shared implementations"
Subclassing	Yes	Yes	"shared partial implementations"
Overriding	under control of superclass	under control of subclass	optional; delegation permitted
Classes as packages	Yes	No	No

US & Scandinavian Objects

Feature	Simula 67	Smalltalk 80	Snyder (1991)
Abstraction	"Modelling": attributes exposed	attributes encapsulated	Objects characterized by offered services
Active Objects	Yes	No	"Associated Concept"
Dynamic Objects	Yes	Yes	Yes
Classes	Yes	Yes	"Shared implementations"
Subclassing	Yes	Yes	"shared partial implementations"
Overriding	under control of superclass	under control of subclass	optional; delegation permitted
Classes as packages	Yes	No	No

US & Scandinavian Objects

Feature	Simula 67	Smalltalk 80	Snyder (1991)
Abstraction	"Modelling": attributes exposed	attributes encapsulated	Objects characterized by offered services
Active Objects	Yes	No	"Associated Concept"
Dynamic Objects	Yes	Yes	Yes
Classes	Yes	Yes	"Shared implementations"
Subclassing	Yes	Yes	"shared partial implementations"
Overriding	under control of superclass	under control of subclass	optional; delegation permitted
Classes as packages	Yes	No	No

US & Scandinavian Objects

Feature	Simula 67	Smalltalk 80	Snyder (1991)
Abstraction	"Modelling": attributes exposed	attributes encapsulated	Objects characterized by offered services
Active Objects	Yes	No	"Associated Concept"
Dynamic Objects	Yes	Yes	Yes
Classes	Yes	Yes	"Shared implementations"
Subclassing	Yes	Yes	"shared partial implementations"
Overriding	under control of superclass	under control of subclass	optional; delegation permitted
Classes as packages	Yes	No	No

US & Scandinavian Objects

Feature	Simula 67	Smalltalk 80	Snyder (1991)
Abstraction	"Modelling": attributes exposed	attributes encapsulated	Objects characterized by offered services
Active Objects	Yes	No	"Associated Concept"
Dynamic Objects	Yes	Yes	Yes
Classes	Yes	Yes	"Shared implementations"
Subclassing	Yes	Yes	"shared partial implementations"
Overriding	under control of superclass	under control of subclass	optional; delegation permitted
Classes as packages	Yes	No	No

Abstraction

Abstraction: key idea of O-O

Simula doesn't mention abstraction specifically. It speaks of "modelling"

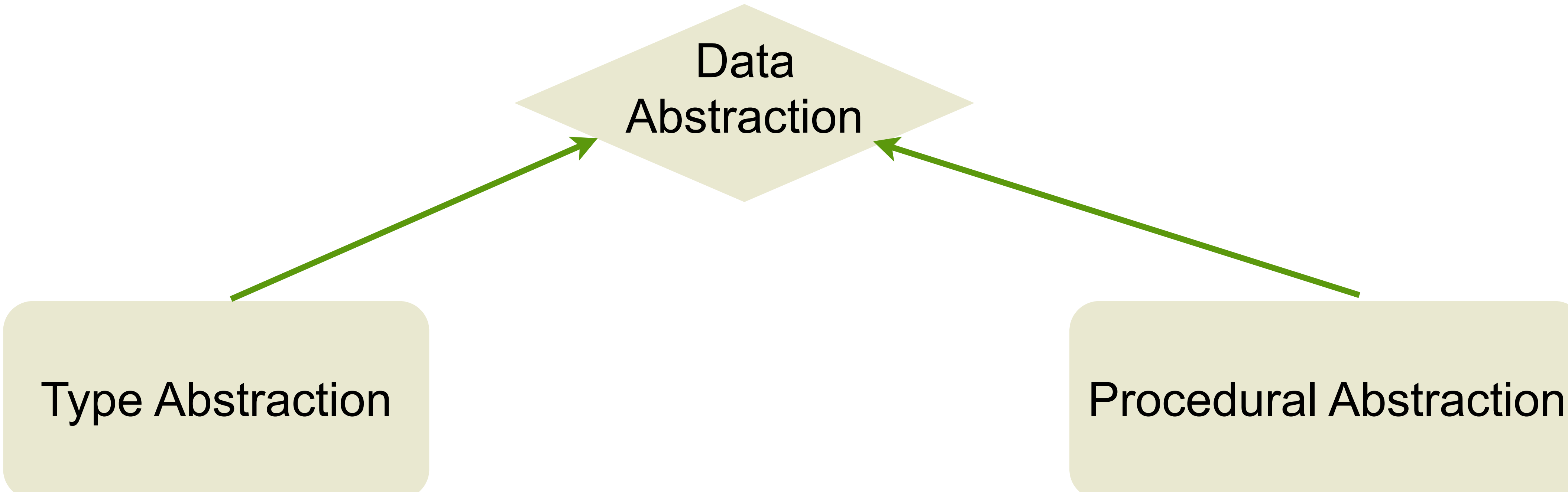
a model: an abstraction with a mission

The idea of separating the internal (concrete) and external (abstract) views of data was yet to mature.

- Hoare 1972 — Proof of Correctness of Data Representations
- Parnas 1972 — Decomposing Systems into Modules
- CLU — 1974–5 — **rep**, **up**, **down** and **cvt**

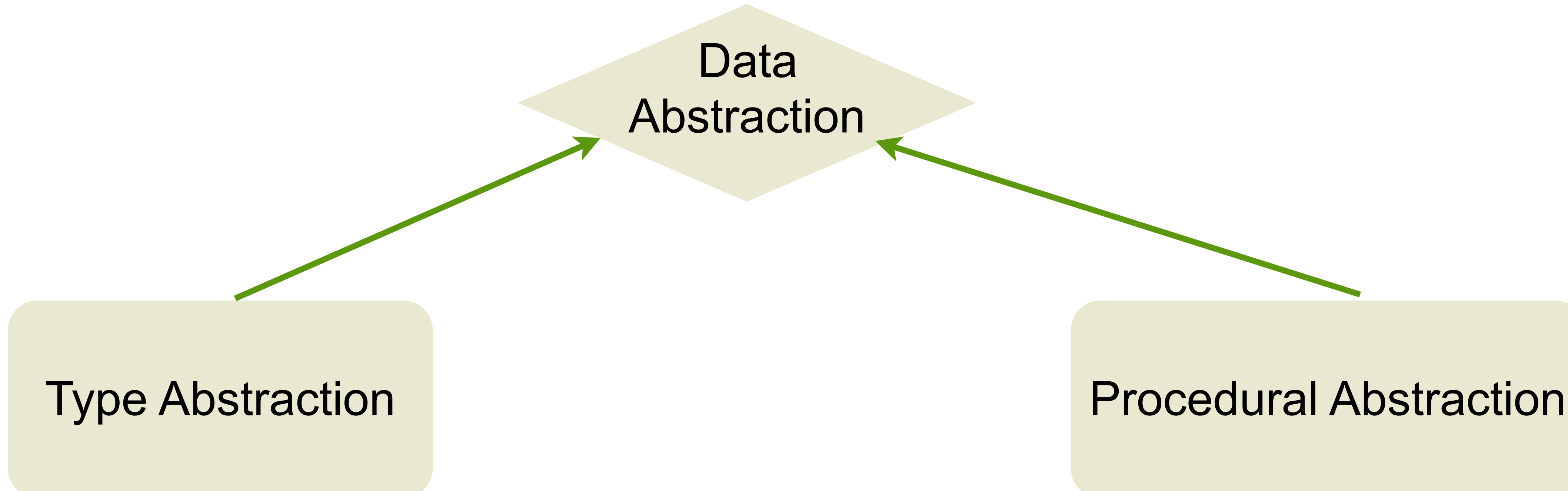
Portland State UNIVERSITY

Type Abstraction ≠ Procedural Abstraction



[Reynol1975] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Conference on New Directions in Algorithmic Languages, Munich, Germany, August 1975. IFIP Working Group 2.1.

Type Abstraction ≠ Procedural Abstraction



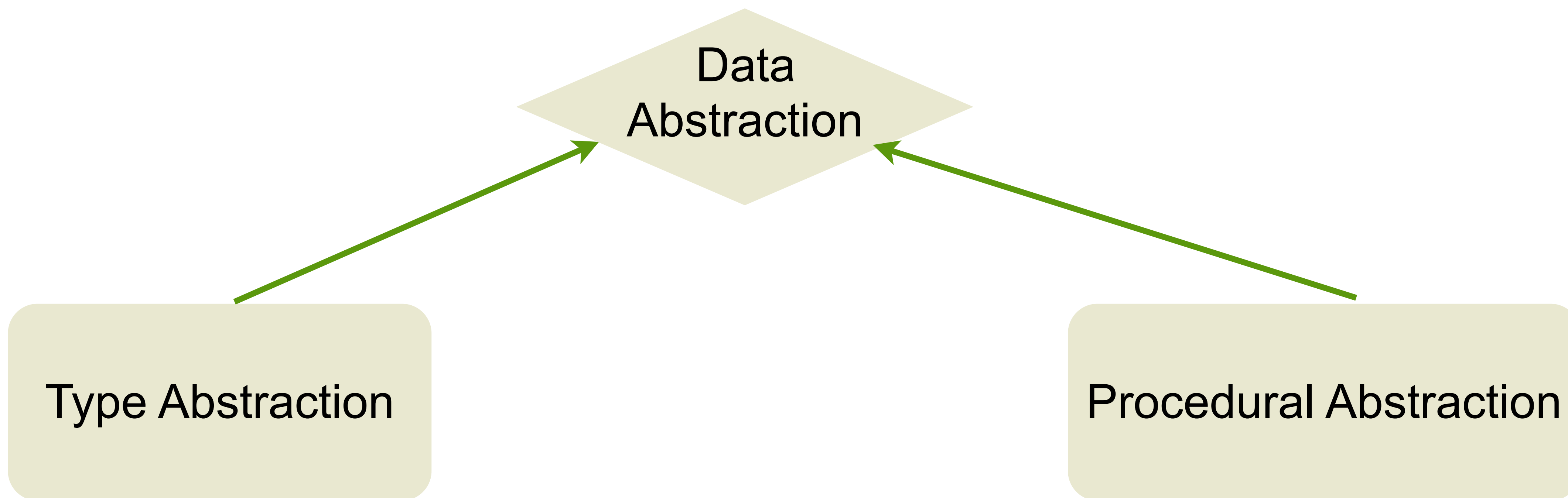
Don’t need types

multiple implementations can co-exist

Autognostic

[Reynol1975] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Conference on New Directions in Algorithmic Languages, Munich, Germany, August 1975. IFIP Working Group 2.1.

Type Abstraction ≠ Procedural Abstraction



Type Abstraction	Procedural Abstraction
Types are essential	Don’t need types
exactly one implementation	multiple implement-ations can co-exist
Pasignostic	Autognostic

[Reynol1975] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Conference on New Directions in Algorithmic Languages, Munich, Germany, August 1975. IFIP Working Group 2.1.

Type Abstraction ≠ Procedural Abstraction

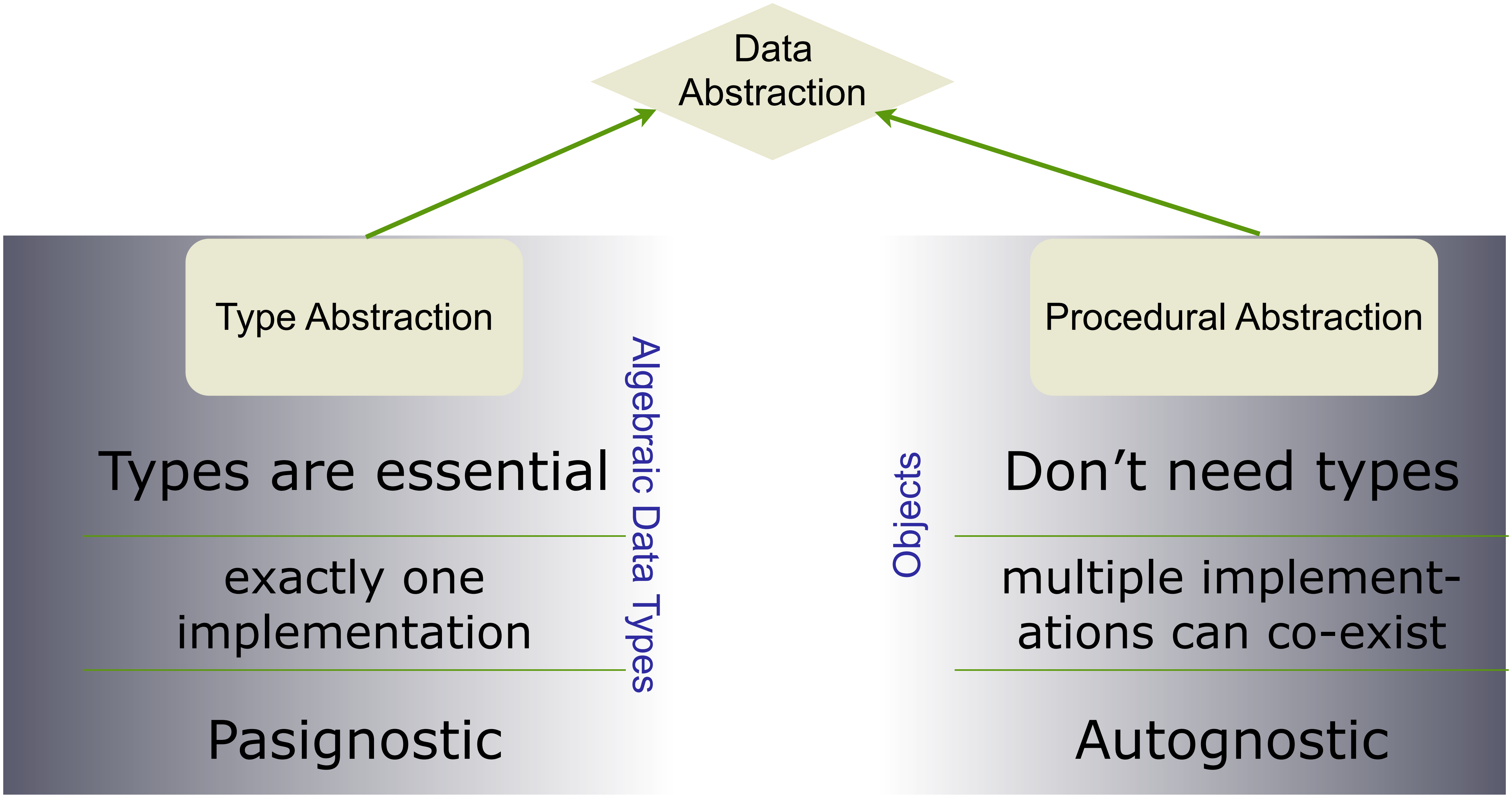
Data Abstraction

Type Abstraction	Procedural Abstraction
Types are essential	Don't need types
exactly one implementation	multiple implementations can co-exist
Pasignostic	Autognostic

Objects

[Reynol1975] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Conference on New Directions in Algorithmic Languages, Munich, Germany, August 1975. IFIP Working Group 2.1.

Type Abstraction ≠ Procedural Abstraction



[Reynol1975] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Conference on New Directions in Algorithmic Languages, Munich, Germany, August 1975. IFIP Working Group 2.1.

Type Abstraction ≠ Procedural Abstraction

Data Abstraction

Type Abstraction

Procedural Abstraction

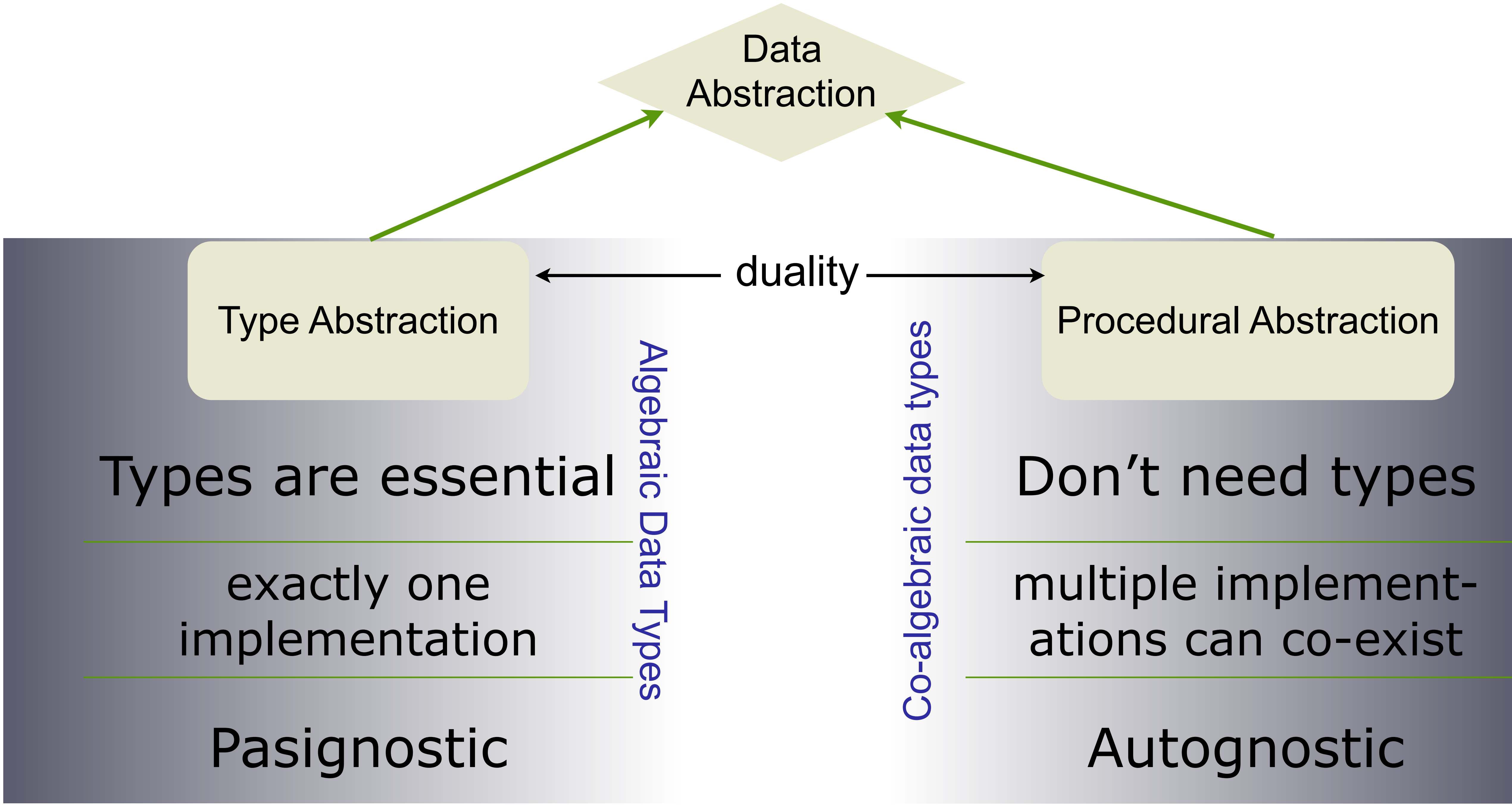
Algebraic Data Types

Co-algebraic data types

Type Abstraction	Procedural Abstraction
Types are essential	Don't need types
exactly one implementation	multiple implement-ations can co-exist
Pasignostic	Autognostic

[Reynol1975] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Conference on New Directions in Algorithmic Languages, Munich, Germany, August 1975. IFIP Working Group 2.1.

Type Abstraction ≠ Procedural Abstraction



[Reynol1975] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Conference on New Directions in Algorithmic Languages, Munich, Germany, August 1975. IFIP Working Group 2.1.

Type Abstraction ≠ Procedural Abstraction

CLU provides ADTs: fundamentally different from objects!

- Did Liskov and the CLU team realize this?

Simula's class construct can be used to generate both records (unprotected, or protected by type abstraction) and objects (protected by procedural abstraction)

C++ can also be used to program data abstractions as well as objects

Active Objects

Active Objects

Active objects is an idea that has become lost to the object-oriented community.

Activity was an *important* part of Simula

- “quasi-parallelism” was a sweet-spot in 1961
- Hewitt’s Actor model [1973] built on this idea
- Emerald used it [Black1986]

But activity has gone from “mainstream” O-O

[Hewitt1973] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In IJCAI, pages 235–245, August 1973.

[Black1986] A. P. Black, N. Hutchinson, E. Jul, and H. M. Levy. Object structure in the Emerald system. In OOPSLA, pages 78–86, 1986.

Why are Smalltalk Objects passive?

I don't *know*

- *Perhaps:* Kay and Ingalls had a philosophical objection to combining what they saw as separate ideas

Or

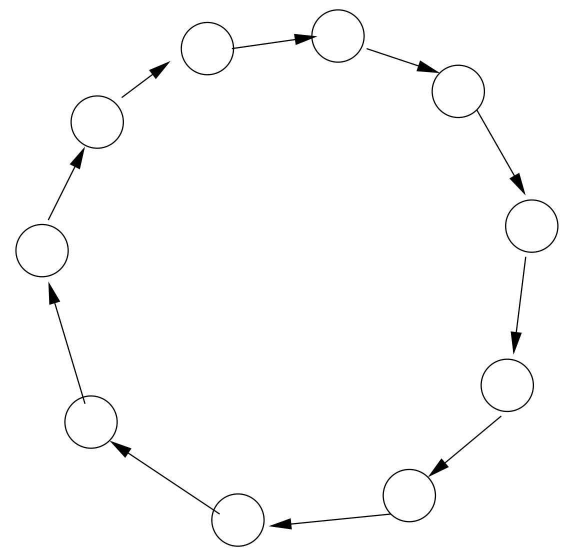
- *Perhaps:* The realities of programming on the Alto set limits as to what was possible

Or

- *Perhaps:* They wanted real processes, not co-routines

Erlang Process Challenge

Put N processes in a ring:



Send a simple message round the ring M times.

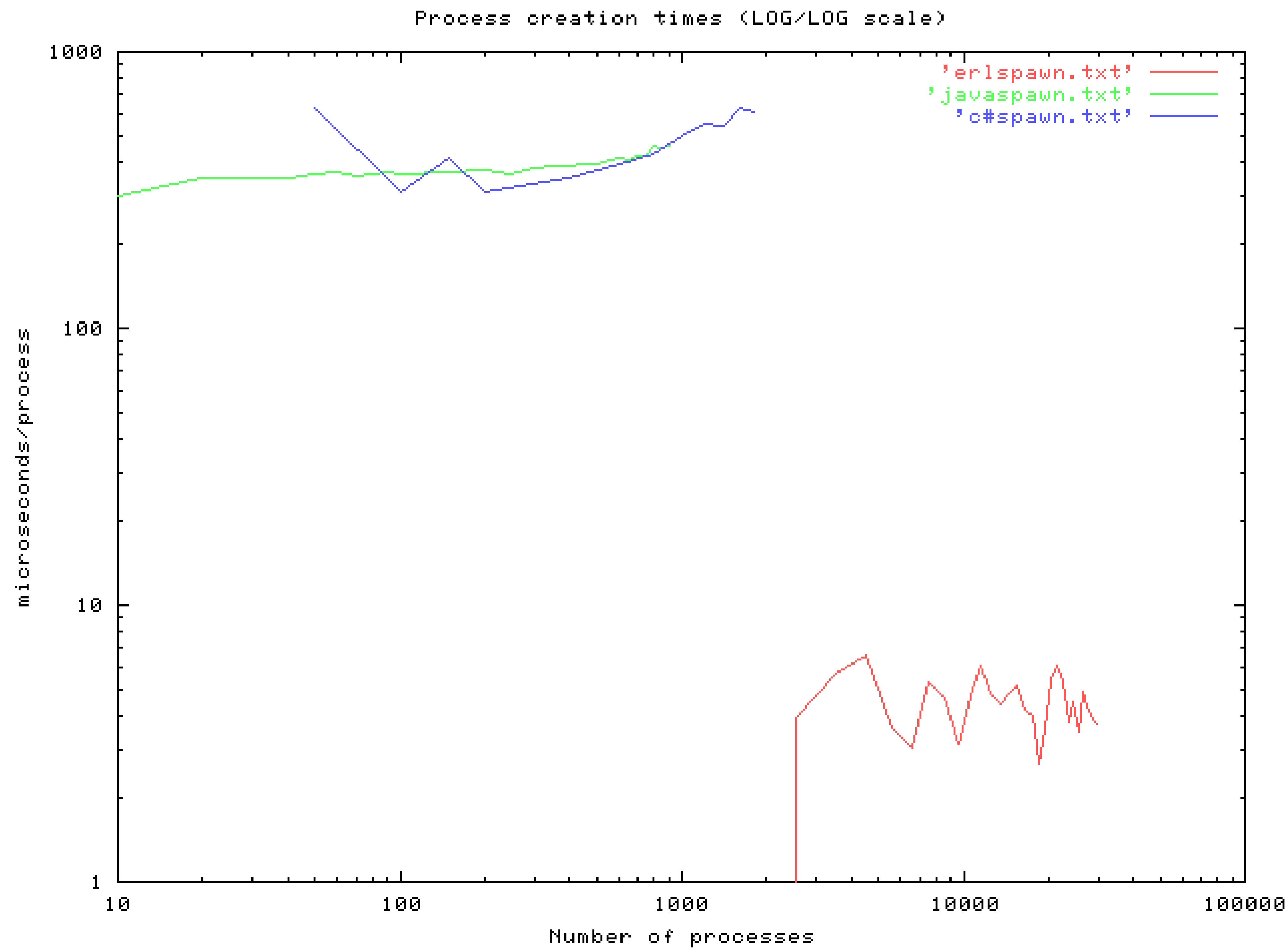
Increase N until the system crashes.

How long did it take to start the ring?

How long did it take to send a message?

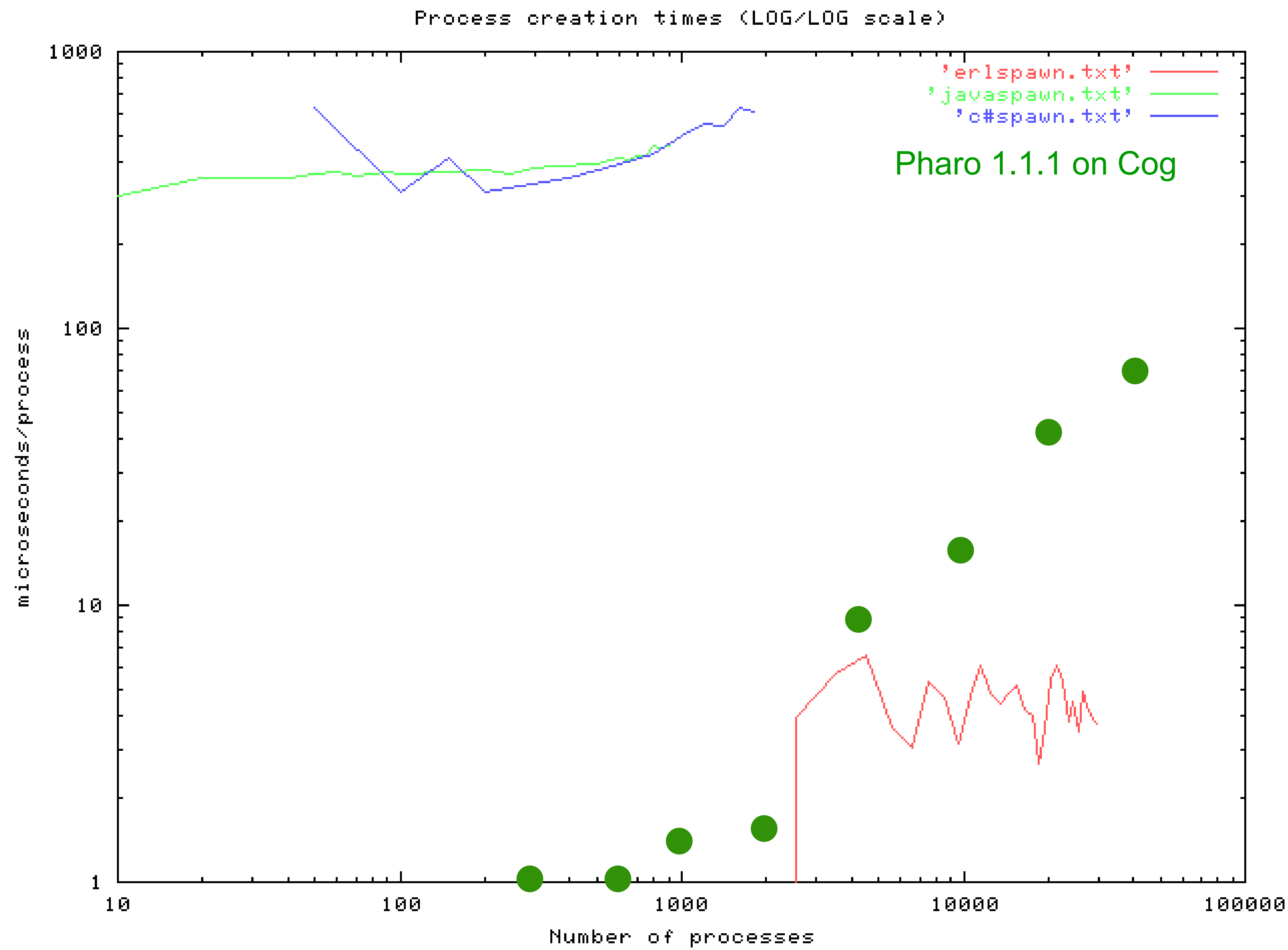
When did it crash?

Process-creation Times



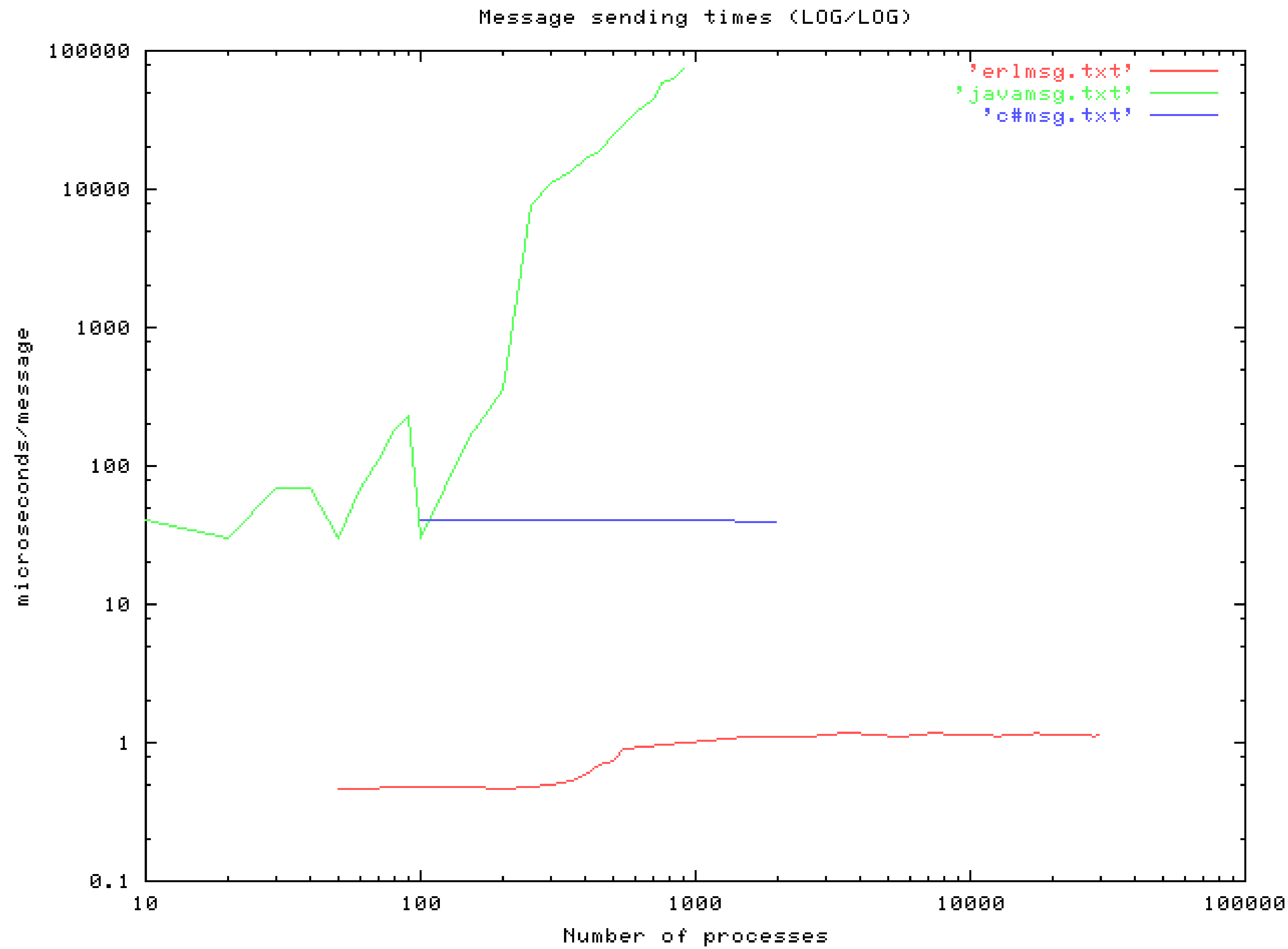
Portland State UNIVERSITY

Process-creation Times



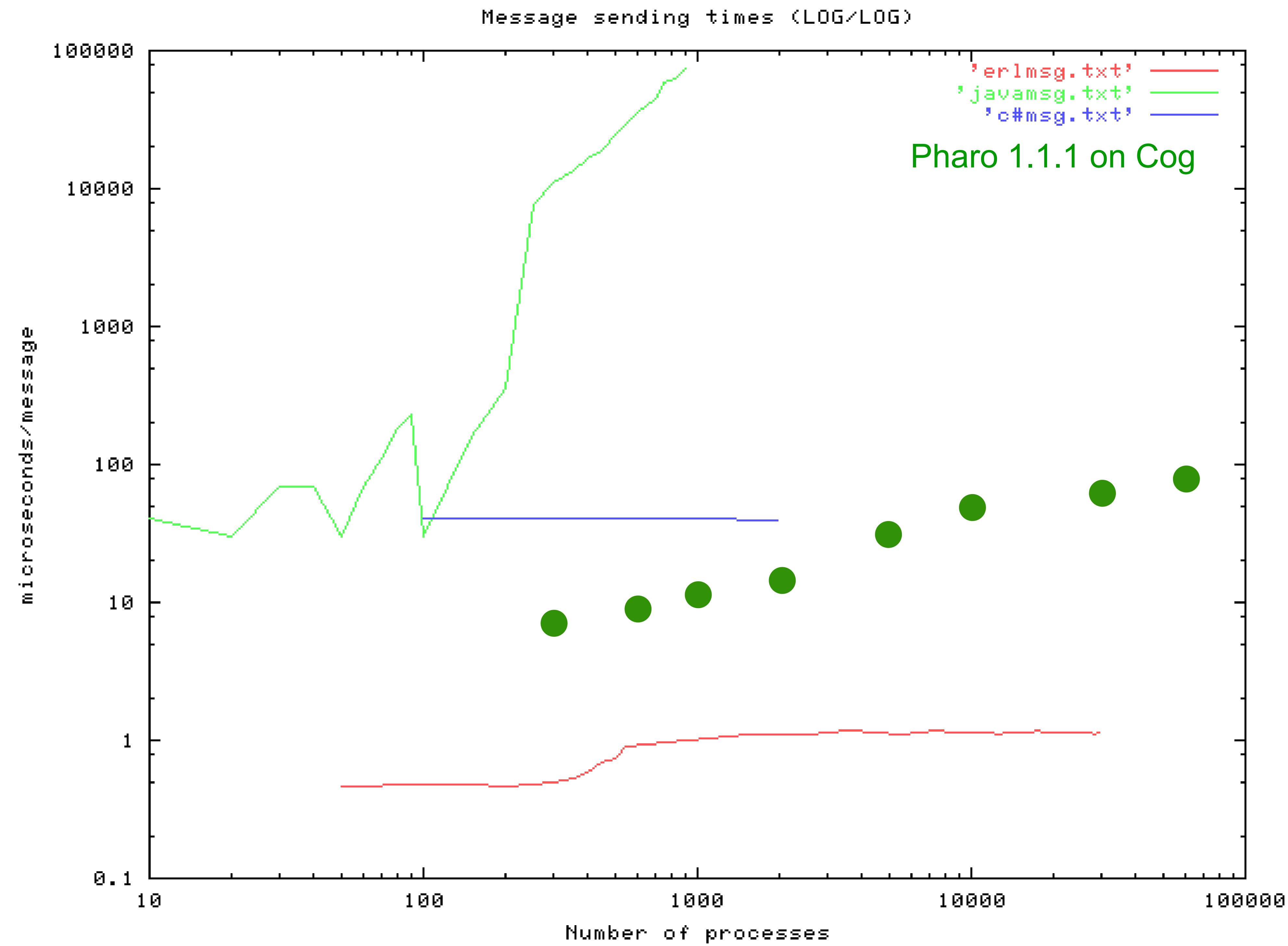
Portland State University

Message-passing times



Portland State University

Message-passing times



Classes & Objects

Does Object-O mean Class-O?

Historically, most of the ideas in o-o came from the class concept

But it's the dynamic objects, not the classes, that form the system model

Classes are interesting only as a way of creating the dynamic system model of interacting objects

- They are a great tool if you want hundreds of similar objects

But what if you want just one object?

Classes are Meta

Classes are meta, and meta isn't always betta!

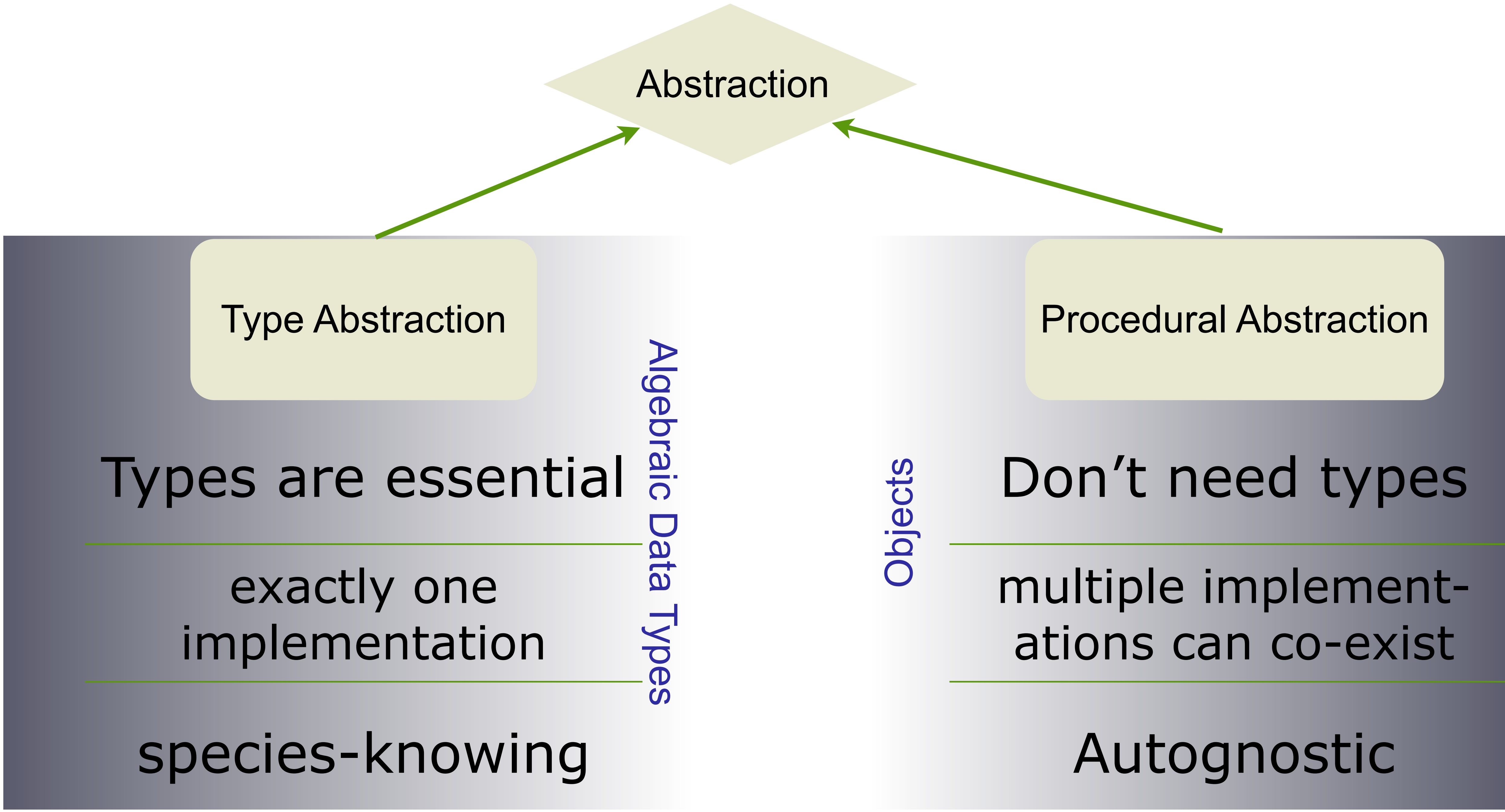
- if you need one or two objects, it's simpler and more direct to describe them in the program directly …
- rather than to describe a factory (class) to make them, and then use it once or twice.

This is the idea behind Self, Emerald, NewtonScript and JavaScript

- Classes (object factories) can in any case be defined as objects

Types

Type Abstraction ≠ Procedural Abstraction



Types for Objects are Optional

- Algebraic data types need types to get encapsulation
- Objects don't: they enjoy procedural encapsulation.

Object-oriented abstraction can exist without types.

Types are Optional

- Why would one want to add type annotations to an object-oriented program?
  - To add *redundancy*
  - Type annotations are assertions
    - just like assert s.notEmpty
- Redundancy is a “good thing”:
  - It provides more information for readers
  - It means that more errors can be detected sooner

Claim: types can be harmful!

Claim: types can be harmful!

Question:

- If types add redundancy, and redundancy is good, how can types be harmful?

Claim: types can be harmful!

Question:

If types add redundancy, and redundancy is good, how can types be harmful?

Answer:

Because types are too much of an invitation to mess up your language design!

Two approaches to type-checking

The “laissez faire”, or George W. Bush interpretation:

Do what you want, we won’t try to stop you. If you mess up, the PDIC will bail you out.

Two approaches to type-checking

The "laissez faire", or George W. Bush interpretation:

- Do what you want, we won't try to stop you. If you mess up, the PDIC will bail you out.

Program debugger and interactive checker

Two approaches to type-checking

The "laissez faire", or George W. Bush interpretation:

- Do what you want, we won't try to stop you. If you mess up, the PDIC will bail you out.

Portland State UNIVERSITY

Two approaches to type-checking

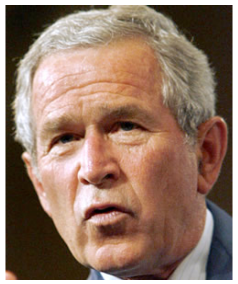
The “laissez faire”, or George W. Bush interpretation:

Do what you want, we won’t try to stop you. If you mess up, the PDIC will bail you out.

The “nanny state” or Harold Wilson interpretation.

We will look after you. If it is even remotely possible that something may go wrong, we won’t let you try.

A *third* interpretation is useful:

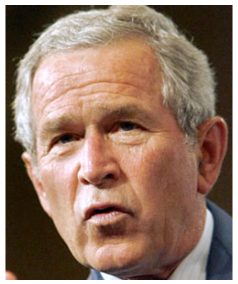


The “laissez faire”, or
George W. Bush interpretation

The “nanny state”, or
Harold Wilson interpretation

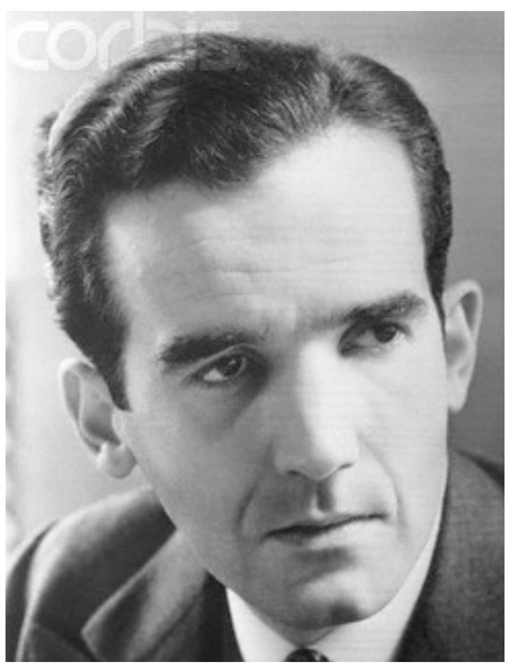
Portland State UNIVERSITY

A *third* interpretation is useful:



The "laissez faire", or
George W. Bush interpretation

The "nanny state", or
Harold Wilson interpretation



The "Proceed with caution", or
Edward R. Murrow, interpretation

The checker has been unable to prove that there are no type errors in your program. It may work; it may give you a run-time error. *Good night, and good luck.*

Three interpretations

Under all three interpretations, an error-free program has the same meaning.

- Under Wilson: conventional static typing
  - An erroneous program will result in a static error, and won't be permitted to run.
  - Some error-free programs won't be permitted to run

Three interpretations

- Under all three interpretations, an error-free program has the same meaning.
  - Under Bush: conventional dynamic typing
    - all checks will be performed at runtime
    - Even those that are guaranteed to fail
      - a counter-example is more useful than a type-error message

Three interpretations

Under all three interpretations, an error-free program has the same meaning

- Under Wilson, you are not permitted to run a program that *might* have a type-error
- Under Bush, any program can be run, but you will get no static warnings.
- Under Murrow interpretation, you will get a mix of compile-time warnings and run-time checks.

I’m for Murrow!

I believe that the Murrow interpretation of types is the most useful for programmers

Wilson’s “Nanny Statism” is an invitation to mess up your *language design*!

- language designers don’t want to include any construct that can’t be statically checked

SIMULA was for Murrow too!

Core Ideas of SIMULA

According to Nygaard:

1. Modelling

   The actions and interactions of the objects created by the program model the actions and interactions of the real-world objects that they are designed to simulate.

2. Security

   The behavior of a program can be understood and explained entirely in terms of the semantics of the programming language in which it is written.

SIMULA was for Murrow too!

- Modelling came first!
- SIMULA did not compromise its modelling ability to achieve security
- It compromised its run-time performance
  incorporating explicit checks where necessary when a construct necessary for modelling was not statically provable as safe

The “Wilson obsession”

Results in:

- type systems of overwhelming complexity
- languages that are
  - larger
  - less regular
  - less expressive

Example: parametric superclasses

```
class Dictionary extends Hashtable {
    method findIndex (predicate) overrides { … }
    method at (key) put (value) adds { … }
    …
```

This is fine so long as Hashtable is a globally known class

But suppose that I want to let the *client* choose the actual class that I'm extending?

Example: parametric superclasses

```
class Dictionary (ht) extends ht {
   method findIndex (predicate) overrides { … }
   method at (key) put (value) adds { … }
   …
```

This is not so fine:

we need a new notion of "heir types" so we can statically check that arguments to Dictionary have the right properties

Example: parametric superclasses

This is not so fine:

we need a new notion of "heir types" so we can statically check that arguments to Dictionary have the right properties

Or:

we need a new function & parameter mechanism for classes

Or:

we ban parametric superclasses, add global variables, add open classes, and still decrease usability

Or:

Virtual Classes

- Virtual classes, as found in BETA, are another approach to this problem
  - They feature co-variant methods
    - methods whose arguments are specialized along with their results
  - Not statically guaranteed to be safe
  - Nevertheless, useful for modelling real systems

Example: type parameters

Types need parameters, e.g.,

Set.of (Informatician)

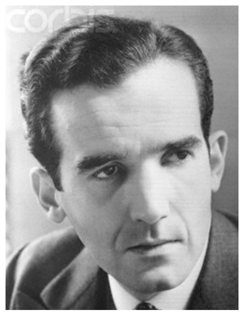
where Informatician is another type

Obvious solution:

Represent types as Objects, and use the normal method & parameter mechanism.

Bad news: type checking is no longer decidable

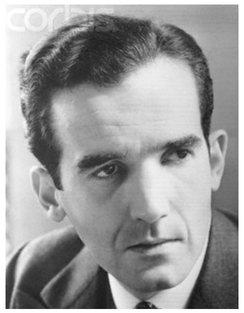
Type-checking is not decidable!



Murrow Reaction:

So what? Interesting programs will need some run-time checking anyway.

Type-checking is not decidable!



Murrow Reaction:

So what? Interesting programs will need some run-time checking anyway.

Wilson Reaction:

Shock! Horror! We can’t do that!

- Invent a new parameter passing mechanism for types, with new syntax, and new semantics, and a bunch of restrictions to ensure decidability
- Some programs will *still* be untypeable (Gödel)
- Result: language becomes larger, expressiveness is reduced.

The Future of Objects (according to Black)

Current Trends in Computing

- *Multicore → Manycore*
- *Energy-Efficiency*
- *Mobility and “the cloud”*
- *Reliability*
- *Distributed development teams*

Multicore and Manycore

What do objects have to offer us in the era of manycore?

Processes interacting through messages!

A Cost Model for Manycore

- Most computing models treat *computation* as the expensive resource
  - it was so when those models were developed!
    - e.g. moving an operation out of a loop is an "optimization"
- Today: computation is free
  - it happens "in the cracks" between data fetch and data store
- Data movement is expensive both in time and energy

A Problem:

Today's computing models can't even express the thing that needs to be carefully controlled on a manycore chip:

Data movement

A Cost Model for Manycore

Spatial arrangement of Small Objects

- small method suite as well as small data

local operations are free

- optimization means reducing the *size* of the object, not the *number* of computations

message-passing is costly

- cost of message = (amount of data) x (distance)
- the “message.byte.nm” model

Mobility and the Cloud

- Fundamentally relies on replication and caching for performance and availability
  - Do Objects help?
- Best model for distributed access seems to be (distributed) version control
    - svn, Hg, git
  - Can we adapt objects to live in a versioned world?
- Object identity is problematic

Object references in a Versioned world

Learn from Erlang:

- Erlang messages can be sent to a prodessId, or to a processName (“controllerForArea503”)

Perhaps: we should be able to reference objects *either* by a descriptor

- e.g. “Most recent version of the Oslo talk”

or

- by an Object id?
  - Object16x45d023f

Reliability

Failures are always partial

What's the unit of failure in the object model?

- Is it the object, or is there some other unit?
- Whatever it is must "leak failure"

How to mask failure:

- replication in space
- replication in time

Distributed Development Teams

What's this to do with objects?

Packaging!

- Collaborating in loosely-knot teams demands better tools for packaging code
- All modules are parameterized by other modules
  - No global namespace?
  - URLs as the global namespace?
- Versioned objects as the basis of a shared programming environment?

Algol 60 and Simula:

- Dahl recognized that *the runtime structures of Algol 60* already contained the mechanisms that were necessary for simulation.
- It is the *runtime behavior* of a simulation system that models the real world, *not* the program text.

Agile Design:

- Agile software development is a methodology in which a program is developed in small increments, in close consultation with the customer.
  - The code runs at the end of the first week, and new functions are added every week.
- How could this possibly work! Isn't it important to design the program?
  - Yes! Design is important. It's *so* important, we don't do it only when we know nothing about the program. We design every day. The program is continuously re-designed as the programmers learn from the behavior of the running system.

Insight:

- The program's run-time behavior is what matters!
  - This is obvious if programs exist to control computers; less so, if programs are system descriptions
- Program *behavior*, not programs, model real-world systems
  - The program-text is a meta-description of the program behavior.
  - It's not always easy to infer the behavior from the meta-description

Observation:

- I know that I have succeeded as a teacher when students anthropomorphize their objects
- This happens more often and more quickly when I teach with Smalltalk than when I teach with Java
- Smalltalk programmers talk about objects, Java programmers talk about classes

Why is this?

The Value of Dynamism:

- Smalltalk is a "Dynamic Language"
  - Many features of the language and the programing environment help the programmer to interact with objects, rather than with code
- Proposed definition: a "Dynamic Programming Language" is one designed to facilitate the programmer learning from the run-time behavior of the program.

Portland State UNIVERSITY

Summary

- What are the major concepts of object-orientation?
  - it depends on the social and political context!
- After 50 years, there are still ideas in SIMULA to be mined to solve 21st century problems.
- 1000 years from now, there may not be any programming,
- but I'm willing to wager that Dahl's ideas will still, in some form, be familiar to programmers in 2061.