On Understanding Data Abstraction

...

Revisited

William R. Cook
The University
of Texas at Austin

Dedicated to P. Wegner

Objects

????

Abstract Data Types

Ignore
non-essentials:

“Objects Model the Real World”

Inheritance

Mutable State

Subtyping

...these are

not essential

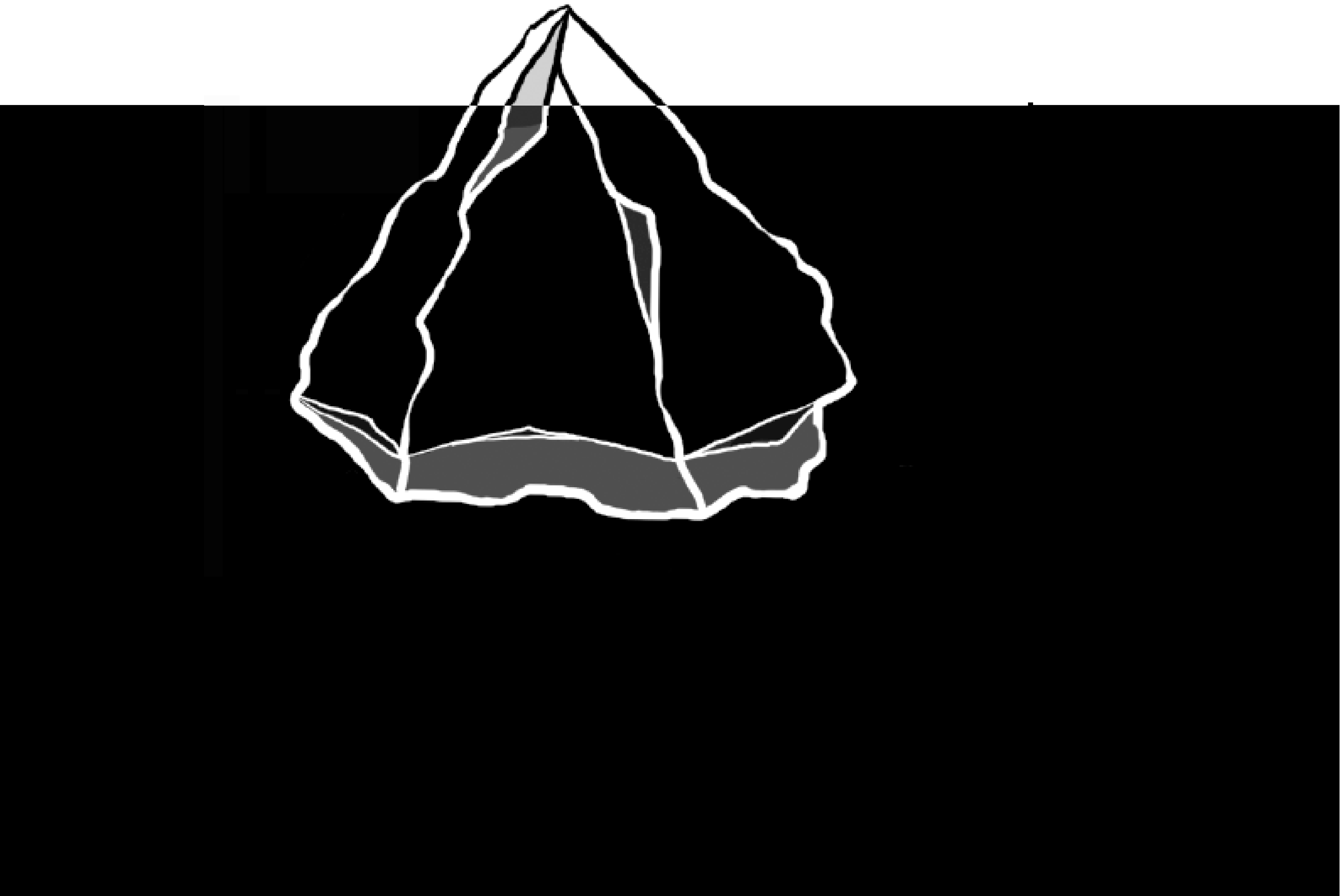
for OOP

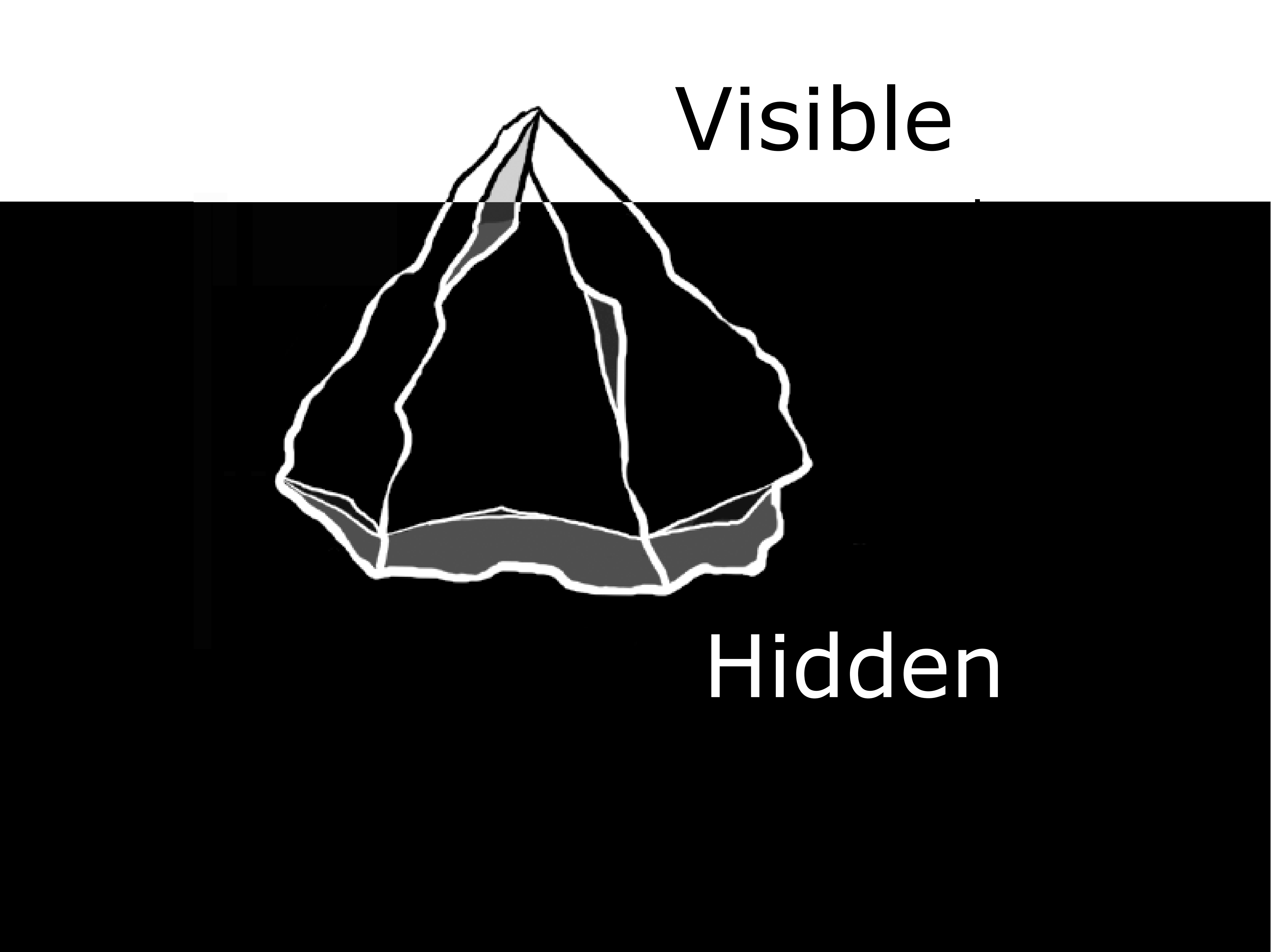
(very nice but not essential)

Essential: Interfaces as types

discuss
inheritance
later

Abstraction




Visible
Hidden

Procedural Abstraction

```
bool f(int x) { … }
```

Procedural Abstraction

$\text{int} \rightarrow \text{bool}$

(one kind of)

Type Abstraction

class Set<T>

(one kind of)

Type Abstraction

$\forall T.Set[T]$

Abstract Data Type

signature Set

empty	: Set
insert	: Set, Int $\rightarrow$ Set
isEmpty	: Set $\rightarrow$ Bool
contains	: Set, Int $\rightarrow$ Bool

Abstract Data Type

Abstract

signature <u>Set</u>

empty : Set
insert : Set, Int → Set
isEmpty : Set → Bool
contains : Set, Int → Bool

Type

+

Operations

ADT Implementation

abstype Set = List of Int

  empty = []

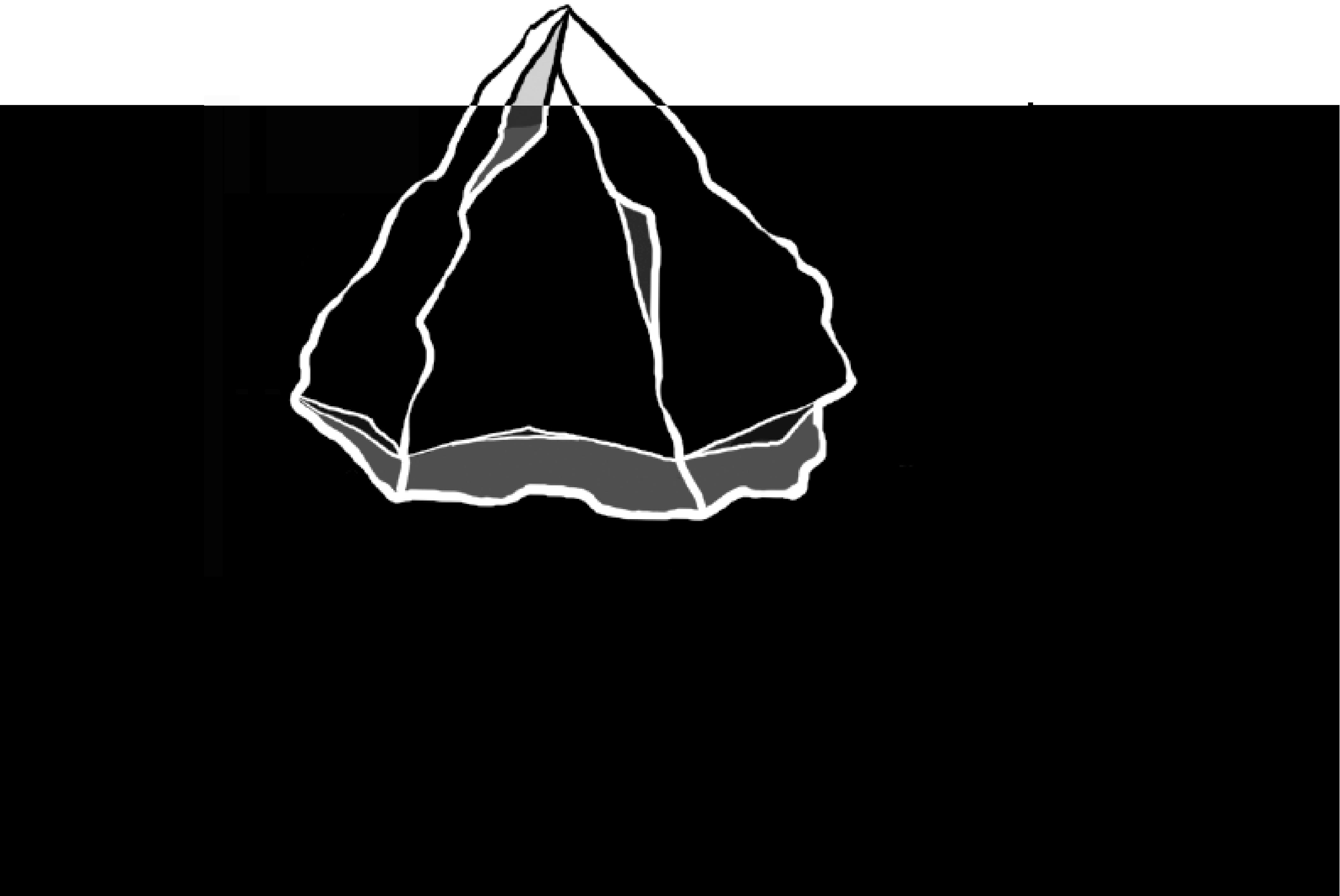
  insert(s, n) = (n : s)

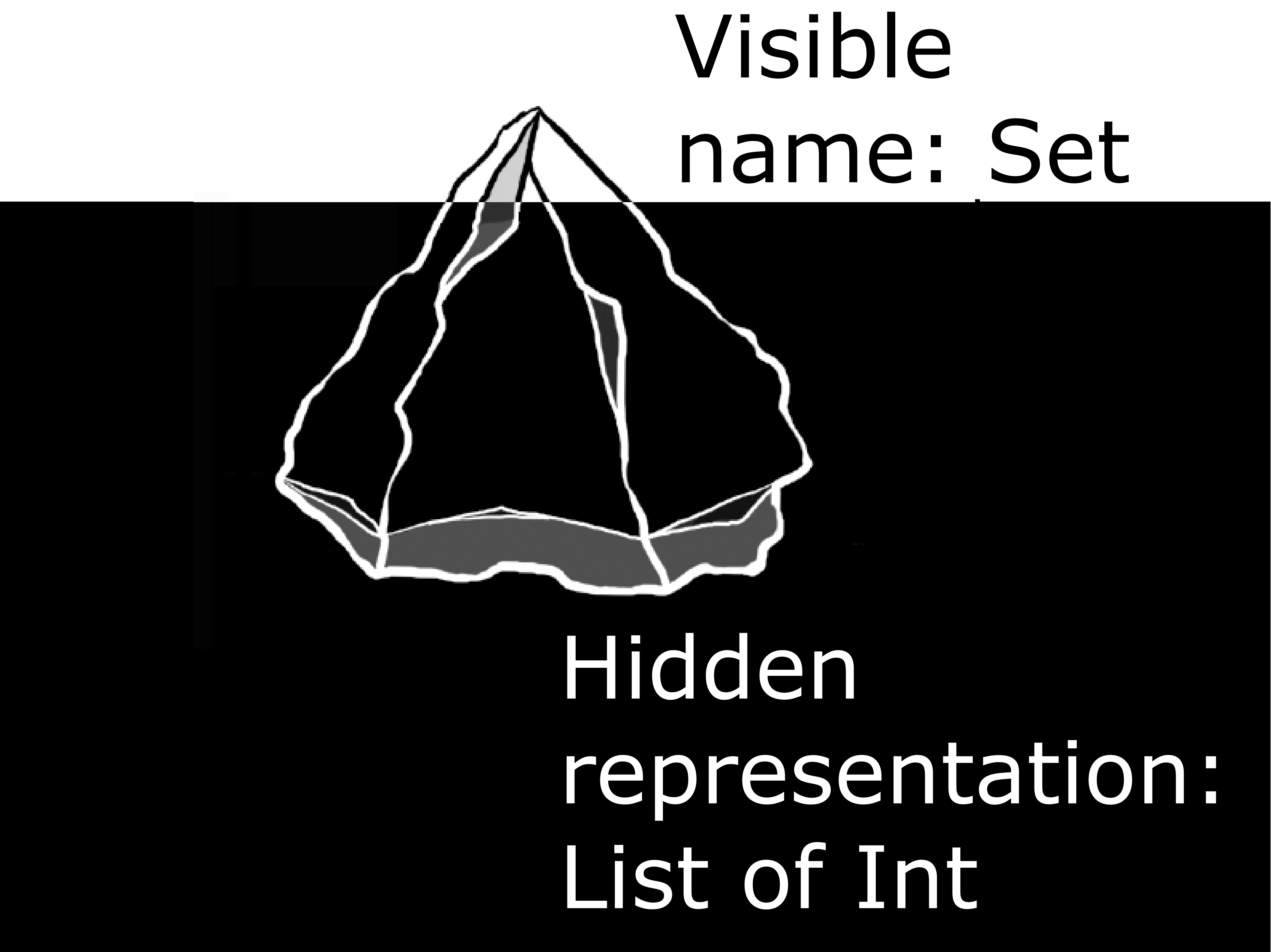
  isEmpty(s) = (s == [])

  contains(s, n) = ($n \in s$)

Using ADT values

```
Set x = empty
Set y = insert(x, 3)
Set z = insert(y, 5)
print( contains(z, 2) )
==> false
```




Visible name: Set
Hidden representation: List of Int

$$
\begin{aligned}
&\mathrm{ISetModule} = \exists \mathrm{Set}.\{\\
&\quad \mathrm{empty} &&: \mathrm{Set}\\
&\quad \mathrm{insert} &&: \mathrm{Set}, \mathrm{Int} \to \mathrm{Set}\\
&\\
&\quad \mathrm{isEmpty} &&: \mathrm{Set} \to \mathrm{Bool}\\
&\quad \mathrm{contains} &&: \mathrm{Set}, \mathrm{Int} \to \mathrm{Bool}\\
&\}
\end{aligned}
$$

Natural!

just like

built-in types

Mathematical

Abstract Algebra

Type Theory

$\exists x.P$

(existential types)

Abstract Data Type

=

Data Abstraction

Right?

$$S = \{ 1, 3, 5, 7, 9 \}$$

Another way

$$P(n) = \mathrm{even}(n)\ \&\ 1 \leq n \leq 9$$

$$S = \{ 1, 3, 5, 7, 9 \}$$

$$P(n) = \text{even}(n) \;\&\; 1 \leq n \leq 9$$

Sets as characteristic functions

type Set =

$$\text{Int} \rightarrow \text{Bool}$$

Empty =

λn. false

$$\mathrm{Insert}(s, m) =$$

$$\lambda n.\ (n == m) \text{ or } s(n)$$

Using them is easy

```
Set x = Empty
Set y = Insert(x, 3)
Set z = Insert(y, 5)
print( z(2) )
==> false
```

So What?

Flexibility

set of all

even numbers

Set ADT:

Not Allowed!

or…

break open ADT

& change

representation

set of

even numbers

as a

function?

Even =

$\lambda n.\ (n \% 2 == 0)$

Even interoperates

```
Set x = Even
Set y = Insert(x, 3)
Set z = Insert(y, 5)
print( z(2) )
==> true
```

Sets-as-functions

are

objects!

No type abstraction

$$\text{type Set} = \text{Int} \rightarrow \text{Bool}$$

multiple

methods?

sure...

```
interface Set {
  contains: Int → Bool
  isEmpty: Bool
}
```

What about

Empty and Insert?

(they are classes)

```
class Empty {
  contains(n) { return false;}
  isEmpty()   { return true;}
}
```

```
class Insert(s, m) {
  contains(n) { return (n==m)
                  or s.contains(n); }
  isEmpty() { return false; }
}
```

Using Classes

```
Set x = Empty()
Set y = Insert(x, 3)
Set z = Insert(y, 5)
print( z.contains(2) )
==> false
```

An object

is the

set of observations

that can be

made upon it

Including
more methods

```
interface Set {
  contains: Int → Bool
  isEmpty: Bool
  insert    : Int → Set
}
```

```
interface Set {
  contains: Int → Bool
  isEmpty: Bool
  insert    : Int → Set
}
```

Type
Recursion

```
class Empty {
  contains(n) { return false;}
  isEmpty()   { return true;}
  insert(n)  { return
                  Insert(this, n);}
}
```

```
class Empty {
  contains(n) { return false;}
  isEmpty()   { return true;}
  insert(n)  { return
                   Insert(this, n);}
}
```

Value
Recursion

Using objects

```
Set x = Empty
Set y = x.insert(3)
Set z = y.insert(5)
print( z.contains(2) )
==> false
```

Autognosis

Autognosis

An object can only access other objects through public interfaces operations

on

multiple objects?

union

of

two sets

```
class Union(a, b) {
  contains(n) { a.contains(n)
                or b.contains(n); }
  isEmpty()   { a.isEmpty(n)
              and b.isEmpty(n); }
  ...
}
```

```
interface Set {
  contains: Int → Bool
  isEmpty: Bool
  insert    : Int → Set
  union     : Set → Set
}
```

Complex Operation

(binary)

<u>intersection</u>

of

two sets

??

```
class Intersection(a, b) {
  contains(n) { a.contains(n)
                and b.contains(n); }

  isEmpty()   { ???no way!??? }
  ...
}
```

Autognosis:

complicates some

operations

(complex ops)

Autognosis: complicates some optimizations (complex ops)

Inspecting two representations & optimization is easy in ADT

Objects are fundamentally different from ADTs

Object Interface
(recursive types)

```
Set = {
  isEmpty  : Bool
  contains : Int → Bool
  insert   : Int → Set
  union    : Set → Set
}
Empty  : Set
Insert : Set, Int → Set
Union  : Set, Set → Set
```

ADT
(existential types)

```
SetImpl = ∃ Set . {
  empty    : Set
  isEmpty  : Set → Bool
  contains : Set, Int → Bool
  insert   : Set, Int → Set
  union    : Set, Set → Set
}
```

Operations/Observations

	s	
	Empty	Insert(s', m)
isEmpty(s)	true	false
contains(s, n)	false	n=m | contains(s', n)
insert(s, n)	Insert(s, n)	Insert(s, n)
union(s, s'')	s''	Union(s, s'')

ADT Organization

	s	
	Empty	Insert(s', m)
isEmpty(s)	true	false
contains(s, n)	false	n=m | contains(s', n)
insert(s, n)	Insert(s, n)	Insert(s, n)
union(s, s'')	s''	Union(s, s'')

OO Organization

	s	
	Empty	Insert(s', m)
isEmpty(s)	true	false
contains(s, n)	false	n=m | contains(s', n)
insert(s, n)	Insert(s, n)	Insert(s, n)
union(s, s'')	s''	Union(s, s'')

Objects are fundamental (too)

Mathematical

functional

representation

of data

Type Theory

μX.P

(recursive types)

ADTs require

a

static type system

Objects work great

with

dynamic typing

“Binary” Operations?

Stack, Socket, Window,
Service, DOM,
Enterprise Data, ...

Objects are

<u>very</u>

higher-order

(functions passed as data and returned as results)

Verification

ADTs: construction

Objects: observation

ADTs: induction

Objects: coinduction

complicated by:

callbacks, state

Objects are designed to be as difficult as possible to verify

Simulation

One object can simulate another!

(identity is bad)

Java

What is a type?

Declare variables

Classify values

Class as type

=> representation

Class as type

=> ADT

Interfaces as type

=> behavior

pure objects

Harmful!

instanceof *Class*

(*Class*) exp

Class x;

Object-Oriented subset of Java: class name is <u>only</u> after “new”

Its not an accident that "int" is an ADT in Java

Smalltalk

```
class True
  ifTrue: a ifFalse: b
      ^a value

class False
  ifTrue: a ifFalse: b
      ^b value
```

True =

λ a . λ b .

a

False =

λ a . λ b .

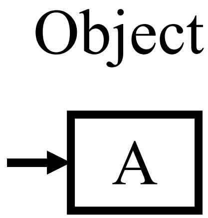
b

Inheritance

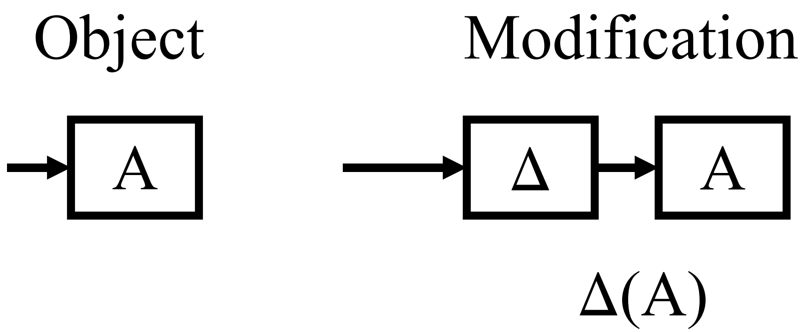
(in one slide)

(animated)

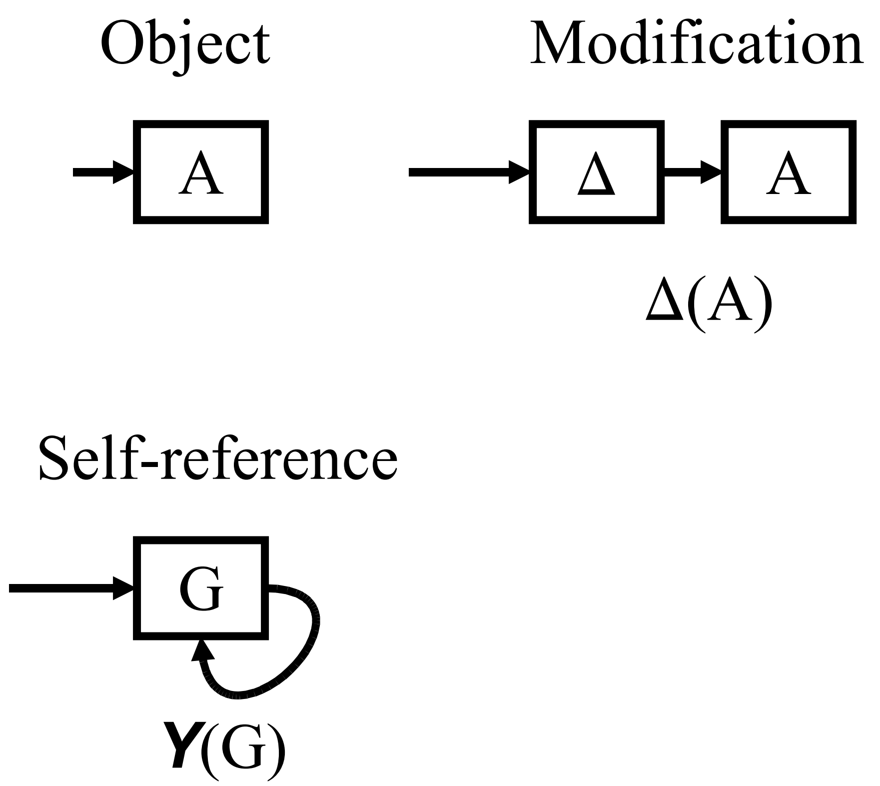
Inheritance



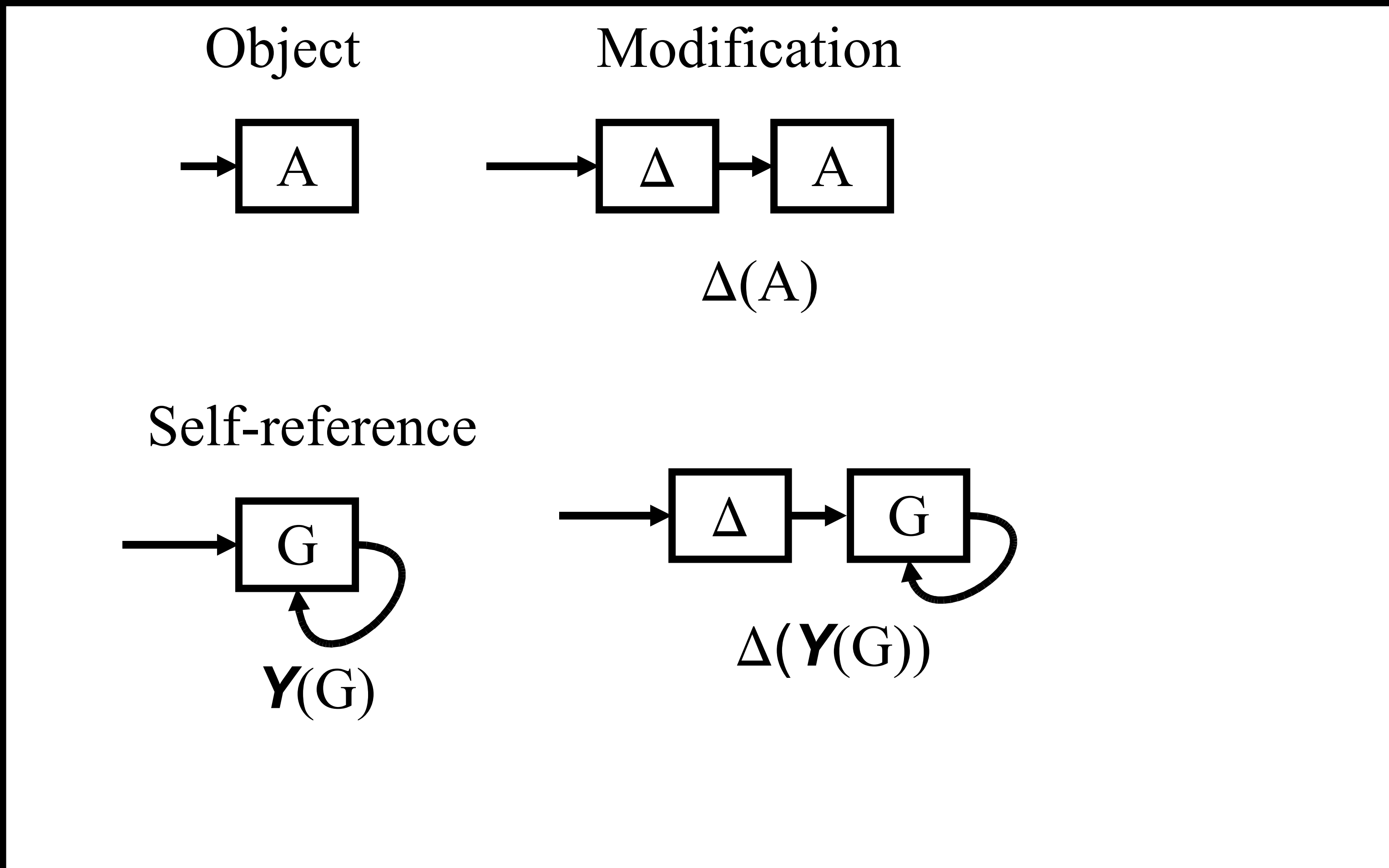
Inheritance



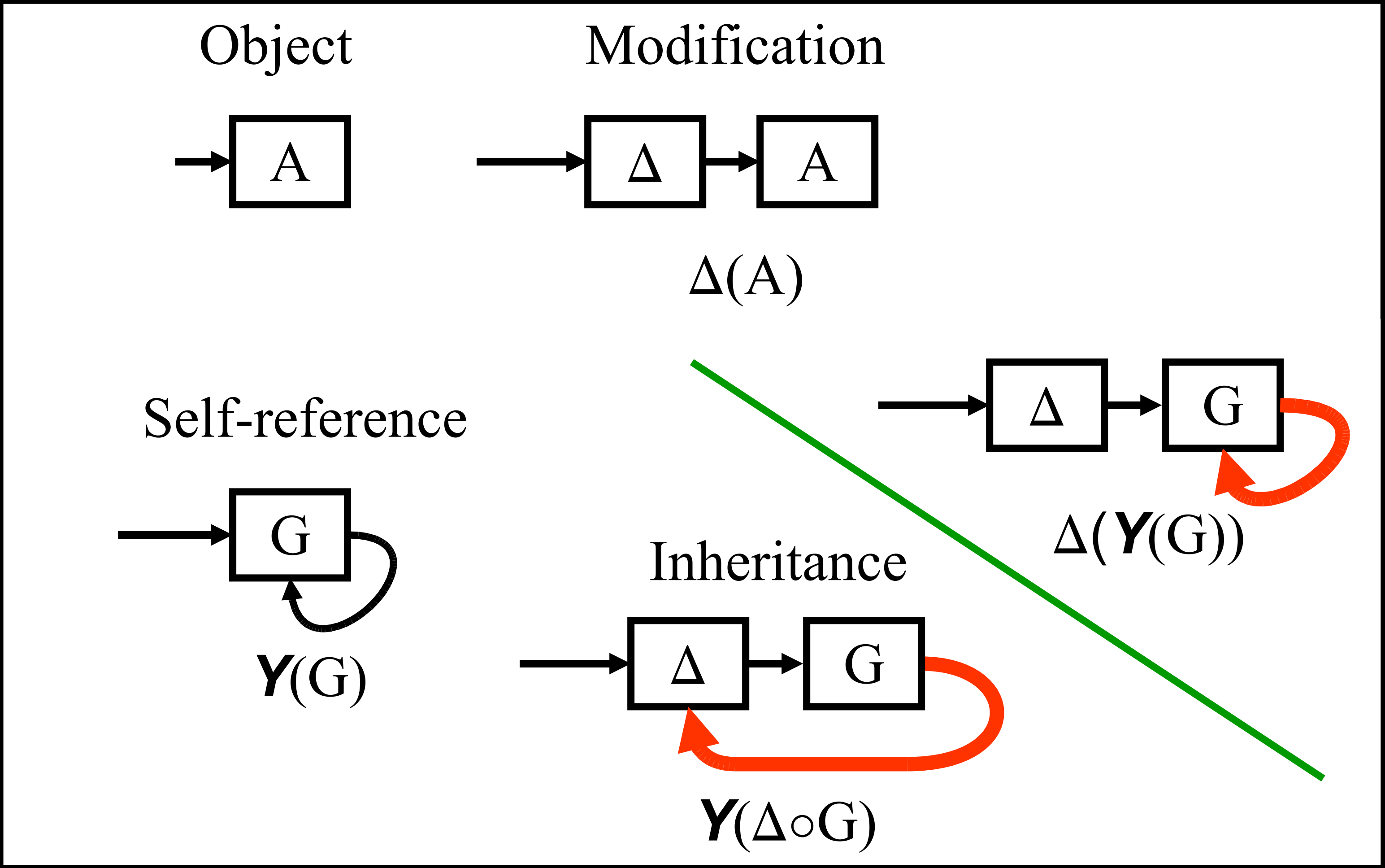
Inheritance



Inheritance



Inheritance



History

User-defined types
and
procedural data structures
as
complementary approaches
to
data abstraction

by J. C. Reynolds
New Advances in Algorithmic Languages
INRIA, 1975

Abstract data types

~~User-defined types~~

and

objects

~~procedural data structures~~

as

complementary approaches

to

data abstraction

by J. C. Reynolds

New Advances in Algorithmic Languages

INRIA, 1975

“[an object with two methods] is more a tour de force than a specimen of clear programming.”

- J. Reynolds

Extensibility Problem

(aka Expression Problem)

1975 Discovered by J. Reynolds

1990 Elaborated by W. Cook

1998 Renamed by P. Wadler

2005 Solved by M. Odersky (?)

2025 Widely understood (?)

Summary

It is possible to do Object-Oriented programming in Java

Lambda-calculus was the first object-oriented language (1941)

Data Abstraction

/ \

ADT Objects