Deconstructing Java™

Gilad Bracha

Original Sin: Primitive Types

- Eight types: ***bool***, ***byte***, ***char***, ***short***, ***int***, ***long***, ***float***, ***double***
- ... and ***void*** too!
- Eight special cases in many APIs
- Cannot store in collections

Primitive Types

- A failure of abstraction
- No common supertype

char

- How many characters are there?

char

- How many characters are there?
- In 1995, someone thought there were 64K

char

- How many characters are there?
- In 1995, someone thought there were 64K
- It turned out there were 17M
- So if you work with chars, you may need more than one char per character

int

- How many integers are there?

int

- How many integers are there?
- In 1995, someone thought there were 4G

int

- How many integers are there?
- In 1995, someone thought there were 4G
- BILL GATES CAN'T COUNT HIS MONEY THAT WAY!

int

- How many integers are there?
- In 1996, someone thought there were 4G
- BILL GATES CAN'T COUNT HIS MONEY THAT WAY!
- Poor Bill!

int

- Use COBOL, or BigDecimal
- Can't use operators like +, -, * ...

int

***Integer new(129) == Integer new(129)*?**

int

- 9 integer types: ***byte***, ***short***, ***int***, ***long***, ***Byte***, ***Short***, ***Integer***, ***Long*** and ***BigDecimal***
- Not one of them behaves like an integer

Arrays

```
int i = 91;

long l = i;

int[] ia = new int[1];

long[] la = ia; // oops ..
```

Arrays

Arrays aren't collections

Matrices, Vectors, Complex Numbers …

Matrix M;

Vector v;

I can't write: ***M*v;***

Salvation:
Everything is an Object

int is a just a name for a predefined class

Can be optimized to JVM int by compiler using type info

Salvation:
Everything is an Object

int is a just a name for a predefined class

Can be optimized to JVM int by compiler using type info

... so individual ***int*** operations stay fast

Salvation:
Everything is an Object

int is a just a name for a predefined class

Can be optimized to JVM int by compiler using type info

... so individual ***int*** operations stay fast

and ***ints*** can be stored in collections

Operators aren't Special

Operator names can be used as method names

Precedence the same as today

Operators aren't Special

Operator names can be used as method names

Precedence the same as today

... so coding with ***int*** stays convenient

Operators aren't Special

Operator names can be used as method names

Precedence the same as today

... so coding with ***int*** stays convenient

and so is matrix multiplication, complex arithmetic etc.

Value Types

class Value extends Object ...

All fields must be final

== is a method, which calls ***equal()***

Value Types

class Value extends Object ...

class int extends Value ...

All members must be final

== is a method, which calls ***equal()***

... so ***int*** has the correct identity semantics - you never have two 129s!

Value Types

```
class Complex extends Value {
   double re = 0;
   double im = 0;
   ....
}
```

Compiler can choose to compile *Complex* as a pair of doubles.

Value Types

For extra credit:

```
synchronized(3) {
   thou.shalt.not.have.any.three.but(3);
}
```

What about Arrays?

Object[] oa = int[] ia;

Boxing all the elements of an array is just too expensive

Once identity and subscripting are methods, we can cheat and generate wrapper objects

So primitive arrays behave like object arrays, and arrays can be collections

Hide Representation

Trap overflow/underflow and convert to appropriate representation as needed.

Hide Representation

Trap overflow/underflow and convert to appropriate representation as needed.

... So Bill's money can be counted correctly with ***int***

Hide Representation

Trap overflow/underflow and convert to appropriate representation as needed.

... So Bill's money can be counted correctly with ***int***

and a ***char*** is always a character

Salvation: Everything is an Object

- Language is smaller and simpler
- Platform libraries are too!
- User code as well
- Language is just as efficient, but more expressive

Salvation: Everything is an Object

- **Language is smaller and simpler**
- Platform libraries are too!
- User code as well
- Language is **just as efficient, but more expressive**

The Sin of Public Fields

- Example: **`System.out`**
- Made **`final`** in 1.1 for security reasons
- Major vendor then added native code to force assignment to **`System.out`**
- Q: Can a JIT assume a final variable will not change?

Program to an Interface not an Implementation

At minimum, fields should be private

Program to an Interface not an Implementation

- Better yet, never refer to a field directly, even in your own class!
- This means your code is representation independent

Program to an Interface not an Implementation

- Better yet, never refer to a field directly, even in your own class!
- This means your code is representation independent
- Is it possible? Isn't it too verbose?

Message based Programmming

- Introduced in Self in 1987 (long before Java even started)
- Basis for Newspeak
- More concise than existing practice

Message based Programmming

- More concise and more readable than existing practice
  - Getters: ***a*** instead of ***a()***; applies to all methods without arguments.
  - Setters: ***a:x*** instead of ***a(x)*** or ***a=x***
  - No assignment operation; we can use **=** for equality

The Sin of Static State

- Static state is bad for
  - Distribution
  - Security
  - Reentrancy
  - Startup
  - Memory management

The Sin of Static State

- Static state is bad for
  - Distribution
  - Security
  - Reentrancy
  - Startup
  - Memory management

Security

No Ambient Authority

Static State

- Static state is bad for
  - Distribution
  - Security
  - Reentrancy
  - Startup
  - Memory management

Re-entrancy

- The Java compiler originally had static state
- Not a problem as a batch compiler
- Broke down completely when integrated into IDE

Re-entrancy

Here is a quote from a senior developer:

At the time I wrote it, I couldn't imagine a client talking to more than one server, but now it is needed and the poor guy is desperate - the amount of needed refactoring is enormous.

Re-entrancy

and another:

I once had to debug a huge concurrency problem just to find out a static variable that held a database session. That code wasn't mine, ... in the beginning they didn't believe me it was the cause! Only when the multi-user test suites passed with a patch (and it took days) they looked at me like I was some magician.

The Sin of Static State

- Static state is bad for
  - Distribution
  - Security
  - Reentrancy
  - Startup
  - Memory management

Startup

Did you know Java class initialization can deadlock?

The Sin of Static State

- Static state is bad for
  - Distribution
  - Security
  - Reentrancy
  - Startup
  - Memory management

Memory Management

When do you garbage-collect a class?

Memory Management

When do you garbage-collect a class?

When its loader is collected. Otherwise static state will disappear under the programmers feet.

This was the case in early implementations.

If classes are stateless, they can be loaded an unloaded transparently

Static Methods

- Cannot be described by interfaces
- Special rules: no ***this***, no overriding

Static Methods

- A failure of abstraction
  - No object to abstract over (no ***this***)

The Sin of Constructors

- Remember new Integer(129)?
- Problem stems from public constructor for a value class
- Constructor cannot be used to cache values, or hide implementation
- Constructors cannot be described in interfaces

The Sin of Constructors

- A failure of abstraction
  - No object to abstract over

Crime and Punishment

- By lethal injection
- Dependency Injection Frameworks

Much more

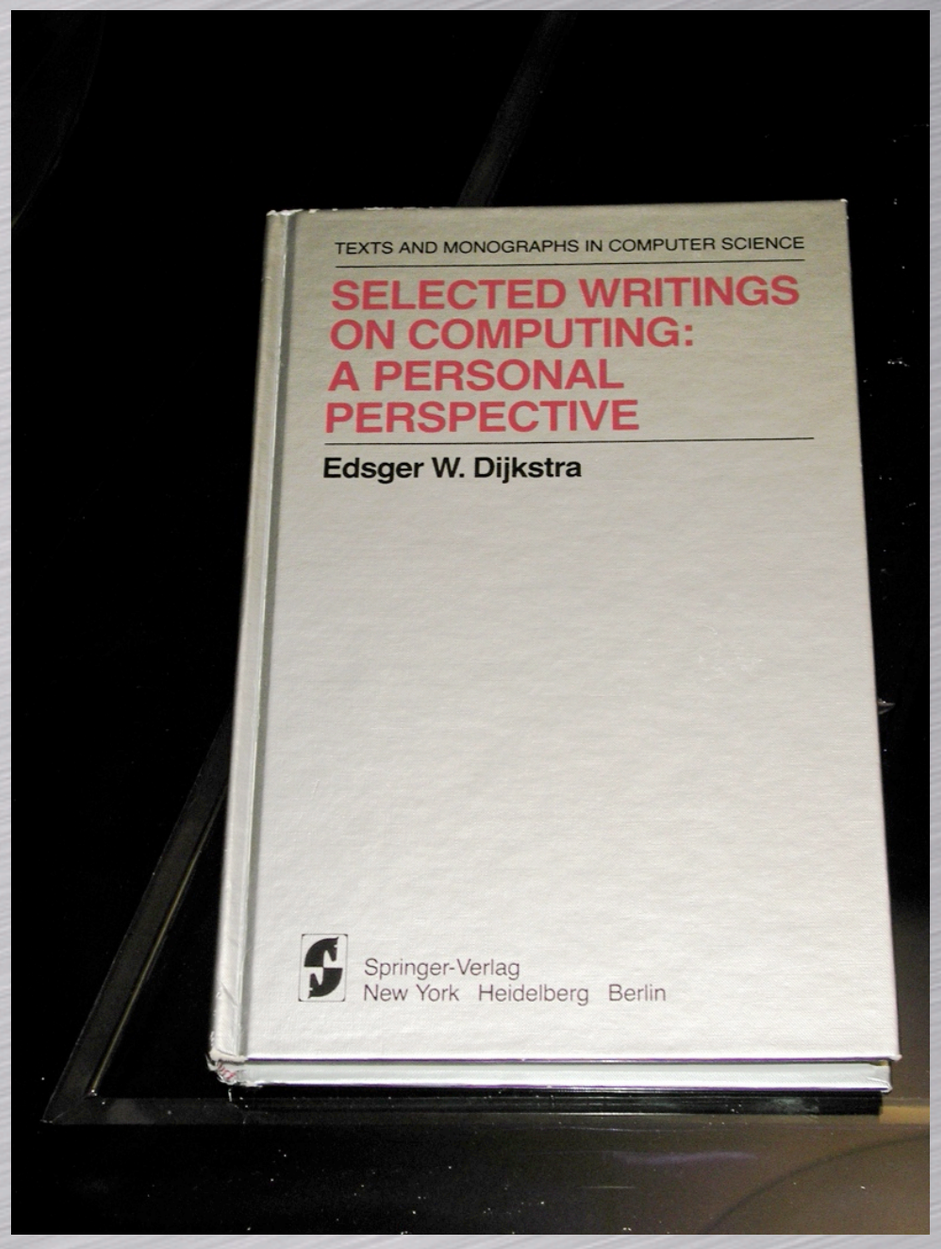
- Reflection
- Class loaders
- Protected access control and security
- The Verifier
- Concurrency
- Nested classes
- Serialization
- I can't remember them all

What does it feel like?

EWD 594: A Parable

Selected Writings on Computing:
A Personal Perspective

Edsger W. Dijkstra



Criticizing is Easy

But can be instructive!

Doing Better is Challenging, but Possible

Addressing all these issues constructively in Newspeak

Interface based Programming as guiding principle