

## The dark side of JAVA

as a general-purpose object-oriented programming language for major applications.

Some concerns about:

- ▶ the language itself
- ▶ the Java community and Java's future



APCU October 28

Conrad Weisert, Information Disciplines, Inc., Chicago

cweisert@acm.org

www.idinews.com

Java is becoming  
a worldwide *de facto* standard.

*For what?*

*Java bien,  
j'espère?*



## Bias

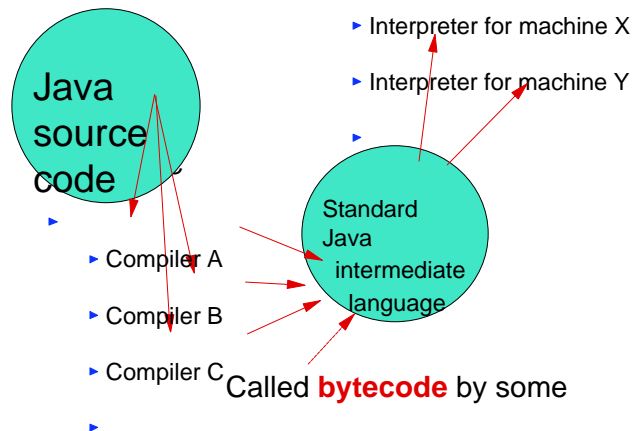
- This will not be a balanced assessment of the merits of Java!
- Java has many merits. Indeed if you attend a typical presentation by a Java guru you will learn that:
  - Java is the perfect solution for just about every conceivable problem in computer programming.*
- We're just going to restore the balance a bit.



## Background and assumptions -- We know that:

- Java is a general-purpose programming language that:
  - supports the **object-oriented** paradigm
  - originated in Sun (Gosling et al) as a language for programming **embedded systems**.
  - was extended to support programming of secure **web applets**.
- Java compilers and interpreters:
  - compiler generates code for a Java **pseudo-machine**, called "bytecode" by Java insiders.

## Two-stage processing



## Where's the Java interpreter? 3 choices:

- An independent program  
*This is what we'll use at first.*
- Integrated with an Internet Web browser, such as Netscape  
*This is the "applet" gimmick.*
- Integrated with an operating system, such as OS/2 v.4

## Unimportant Concerns

- ▶ Early negative reactions focused on the Java environment:
  - ▶ **Inefficiency** of interpretive execution
  - ▶ Potential **security breach** in web applets
- ▶ Neither of these is particularly important to us:
  - ▶ They're not major obstacles to using Java even now
  - ▶ They're sure to get better with improving technology
  - ▶ They have no impact on Java *as a language*.

## Important Concerns

- ▶ Concerns about the language itself:
  - ▶ Difficulties in teaching / learning
  - ▶ Obstacles to maintenance / reuse
  - ▶ Barriers to software quality
- ▶ Concerns about Java's promoters, who:
  - ▶ present major omissions as virtues,
  - ▶ justify restrictions as protections against misuse,
  - ▶ explain inconsistencies as contributions to simplicity,
  - ▶ assert that anything Java won't let you do can't be worth doing!
- ▶ *Does the "Java community" really understand large-scale design and programming?*

## Criteria -- context for concerns

- Many features of today's tools demand new ways of thinking about solutions.
- But time-tested measures of **program quality** (Myers\*, et al) still apply.

\* Glenford Meyers: *Composite Structured Design*, 1978, Van Nostrand

## What measures of program quality?

- *Understandability*, readability
- *Localization* (minimal repetition) of:
  - ▶ values
  - ▶ properties
  - ▶ processes
- *Modularity*
  - ▶ Minimum, well-defined *coupling* (interfaces)
  - ▶ Maximum functional *cohesion*
  - ▶ Maximum *reuse* potential

*Any tool for serious work must strongly support these vitally important criteria.*

## Where does Java undermine software quality?

- Java demands more **repetition** of knowledge throughout a program than many other languages
- Java imposes obstacles to **re-use**
- Java forces unnatural distinctions between primitive data and **reference data**.
- Much Java syntax is hard to read

*We shall examine these issues*

## Java data types

- Java distinguishes between:
  - **Primitive** (built-in elementary, C-like) data types (*int, short, long, float, double, char, ...*)
  - **Reference** data
    - ▶ objects: instances of programmer-defined classes
    - ▶ 1-dimensional arrays
    - ▶ character strings

## Primitive and reference data types

- C++ tries (or allows the programmer) to treat all data items in a consistent way (as *similar* as possible).
- But Java makes the programmer treat primitive and reference data differently in almost every context! (as *different* as possible!)

## Example:

- Suppose we declare some amounts of money as primitive data items:

```
double unitPrice;  
double amtDue;
```

and then manipulate these items in the program in various conventional ways.
- Suppose we later develop or find a **Money class** and change those declarations to:

```
Money unitPrice;  
Money amtDue;
```

  - ▶ Q: Which references to **unitPrice** and **amtDue** will we have to change?
  - A: Almost all of them!

## How are primitive types and reference types different?

- Objects must be *created* (**new** operator) as well as declared.
- Operators have different syntax and semantics (if they exist at all):
  - ▶ Assignment
  - ▶ Equality, comparison
  - ▶ Arithmetic
- Function parameters:
  - ▶ Primitive data always passed by **value**
  - ▶ Reference data always passed by **address**

## Declaration, creation, and initialization of numeric types

- Primitive data (both Java and C++):

```
double amt1;  
double amt2 = 49.95;
```
  - Objects (C++):

```
Money amt1;  
Money amt2 = 49.95;
```
  - Objects (Java):

```
Money amt1;  
Money amt2 = new Money(49.95);
```
- Just the same*
-

## Pointers -- the bane of C++ programmers

- C++ uses pointer (address) data extensively (some say excessively) for:
  - ▶ memory management
  - ▶ array manipulation
  - ▶ polymorphic function invocation
- This is C++'s most clumsy and error-prone feature.
- Java claims to liberate the programmer from explicit pointer manipulation. But does it really? Is the cure worse than the disease?

## Java and pointers

- Java has no pointer data types.
- So you { don't have to / can't } manipulate them.
- Java takes care of memory management (via garbage collection). *Good news*
- Pointers are still there, and the programmer must still be very much aware of them as *references*. *Bad news*

## Comparison operators

- To compare 2 primitive data items (Java and C/C++)

```
if (amt1 > 0) . . .
if (amt1 == amt2) . . .
```
- To compare 2 objects: (Java)
  - ▶ `if (amt1 > 0) . . . // illegal`
  - ▶ `if (amt1 == amt2) . . .`  
legal, but true if and only if `amt1` and `amt2` *point to the same object*. If they refer to different objects, even if they have the *same value*, the result is `false`!
  - ▶ To compare *values* we must code something like:

```
if (amt1.greaterThan(0)) . . .
if (amt1.equals(amt2)) . . .
```

## Arithmetic operators work only for *primitive* data types!

- ". . . the language designers decided (after much debate) that **overloaded operators** were a neat idea, but that code that relied on them became hard to read and understand." (Flanagan, p. 35)
- So instead of this (**C++**):

```
total = unitPrice * quantity + shipChg;
```
- We get (at best) this (**Java**):

```
total.setValue(
    shipChg.add(unitPrice.mpy(quantity)));
```

*A matter of understandability?*

## Even Assignment is different

- The semantics of *assignment* conflict not only with primitive types but also with what programmers normally expect:

```
amt1.setValue(49.95);
```

```
amt2 = amt1; ← Assignment sets reference, not value.
```

```
amt1.setValue(0);
```

```
amt2.print(); ← Value is now zero!
```

- If we want to change just the *value* of `amt2`, we must code something like:

```
amt2.setValue(amt1);
```

or worse

```
amt2.setValue(amt1.getValue());
```

## Three unsatisfactory solutions suggested by some Java gurus

1. Never use primitive types for application-domain data.
  - ▶ Use special library classes that wrap primitive data representations in reference semantics.
2. Always use primitive types for numeric and other elementary data. (Forgo OOP!)
  - ▶ Give up type safety
  - ▶ Give up flexibility of hidden internal representation
3. Keep converting back and forth between objects and a publicly known external representation
  - ▶ At best inefficient and hard to understand
  - ▶ At worst not possible (see [www.idinews.com/quasiClass.pdf](http://www.idinews.com/quasiClass.pdf))

## Localization problems occur even with only primitive types

- Consider the following code (from a `Calendar` class):

```
short daysPrYr    = 365;
short daysPr4Yrs  = daysPrYr    * 4 + 1;
int   daysPrCent  = daysPr4Yrs  * 25 - 1;
int   daysPr4Cents= daysPrCent  * 4 + 1;
```

- One of the 4 declarations is illegal. Which one? How can the programmer fix it?

## Repetition requirement even with primitive types

```
short daysPrYr    = 365;
short daysPr4Yrs  = daysPrYr    * 4 + 1; ←
int   daysPrCent  = daysPr4Yrs  * 25 - 1;
int   daysPr4Cents= daysPrCent  * 4 + 1;
```

- The second one is illegal! (Why?)

*Why is that bad?*

*What's the impact on program structure?*

## Repetition requirement even with primitive types

```
■ short daysPrYr    = 365;
▶ short daysPr4Yrs = daysPrYr * 4 + 1;
  int  daysPrCent  = daysPr4Yrs * 25 - 1;
  int  daysPr4Cents= daysPrCent * 4 + 1;
```

- To get the code to compile we must code:

```
short daysPr4Yrs
= (short) (daysPrYr * 4 + 1);
```

*Explicit type cast*

- Which quality criterion does this violate?  
What's the impact on maintainability?

*How can we circumvent this?*

## Java's Casts of Thousands

- Virtually every other high-level programming language separates data declaration from procedural code.
  - ▶ That way, you only have to make one program change when you change your mind about a data item's type.
  - ▶ But in Java you have to hunt through the whole program for repeated occurrences of type-specific casting operators.

## Can we get around the casting problem?

- "It's tedious but at least I can get around the repetition with a typedef, i.e.

```
typedef short dayCtr;
.
.
dayCtr daysPr4Yrs
= (dayCtr) (daysPrYr * 4 + 1); "
```

- a C++ programmer

- "No you can't. Java doesn't allow typedef, because it can be confusing."  
- the Java Guru

## Can't we get around the casting problem?

- "Well, OK, we'll make it a preprocessor macro:  
#define dayCtr short;  
.  
.  
dayCtr daysPr4Yrs  
= (dayCtr) (daysPrYr \* 4 + 1); "

- a C++ programmer

- "Wrong again. Java doesn't allow macros, because they're often misused."  
- the Java Guru

### Agree or disagree?

- "Java is a *pure* object-oriented language.  
In java everything is an object.

C++, on the other hand, is a hybrid language, since it permits data and functions outside any class. A C++ programmer is free to ignore OOP altogether."

### Agree or disagree?

- "Java is a *pure* object-oriented language.  
In java everything is an object.

C++, on the other hand, is a hybrid language, since it permits data and functions outside any class. A C++ programmer is free to ignore OOP altogether."

*Balderdash!*

### Pseudo classes

- Java is no more *purely* O.O. than C++.
- Consider the frequently used library classes **Math** and **System**. Do we ever:
  - ▶ Instantiate objects of those classes?
  - ▶ Derive subclasses from them?
  - ▶ Involve them in any way in polymorphism or in any other aspect of OOP?
- Java *pseudo classes* are simply packaging artifices for code that's type independent.
- This is, in fact, another confusing *complication* for students.

### Standard library

- Like C/C++, Java's power comes less from the language than from *standard library* classes.
- Some of Java's exhibit atrociously bad design
- Java has redone many of the most fundamental library classes several times, e.g. the so-called "swing" library.

## Standard library (continued)

- Consider `Date` (JDK 1.0)
  - ▶ Java's *only* **elementary numeric** class, but an important one that most applications need.
  - ▶ Hideous interface (e.g. What date does `Date(104,2,15)` yield?)
  - ▶ Violates (at least in spirit) encapsulation (e.g. everyone knows the magic origin date in 1970!)
  - ▶ Fails to provide essential functionality, esp. arithmetic (e.g. can't add a day count to a date or take the difference between two dates).

## Java Date arithmetic

- To compute the difference between two dates:  
`(d1.getTime()-d2.getTime())/86400000`  
Number of milliseconds since January 1, 1970!
- To add a number of days to a date:  
`d1.setDate(d1.getDate() + n)`  
Not the date, just the day of month! January 25 + 10 days = January 35 (unnormalized), which Java considers the same as February 4.

## Why should we worry about `Date`?

- Not because it makes date handling awkward
  - ▶ If we don't like the standard version we can still write our own. *Already done*
  - ▶ An improved version may come along eventually. *← in JDK 1.1, but not much better!*
- but rather for what it reveals about the language designers we're depending on.
  - ▶ Someone designing a mainstream tool must have actually thought this was a reasonable design.
  - ▶ Others in the Java community present an obvious abomination unapologetically.

## Case study on reuse: developing our own `Date` class

- First we want to design a better `Date` class (or more accurately a package of essential calendar manipulation facilities).
- In the course of developing date and duration classes, we discover a *pattern* that has potential use elsewhere (i.e. the *point-extent* pattern, [www.idinews.com/pointext.pdf](http://www.idinews.com/pointext.pdf)).
- We'd like to abstract the essence of that pattern and package it for reuse.

## An well-intended afterthought: wrapper classes

- For each primitive type the Java library provides a corresponding class in order to help unify the two kinds of data. For example:
  - ▶ `Double`, `Long`, (but not `Int`!)
- Each class supports:
  - ▶ A constructor that takes the corresponding primitive type as parameter
  - ▶ An accessor `doubleValue()`, etc.
  - ▶ `toString()`
  - ▶ A string parser `parseShort()`
- But no ordering or arithmetic operations!  
*How do you get around that omission?*

## Wrapper class summary

- Each built-in primitive numeric type has a corresponding *wrapper* class:
  - ▶ Each derived from `Number`
- Purposes:
  - To allow a function to have **side effects** on its numeric arguments.
  - To package related **static functions** and **conversion methods**.
  - But, surprisingly, **not**:
    - ▶ To facilitate polymorphic treatment of numeric objects
    - ▶ To serve as base classes for numeric types (e.g. `Money`).

<code>byte</code>	<code>Byte</code>
<code>int</code>	<code>Integer</code>
<code>short</code>	<code>Short</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>

## Wrapper-class inconsistencies

- Java's "wrapper" classes exhibit careless and amateurish design, in their lack of symmetry and incomplete functionality.
  - Relational operator functions:
    - ▶ equality (equals) is defined, ordering (lessThan) is not.
  - Arithmetic operator functions:
    - ▶ none provided
  - Methods and method naming:
    - ▶ Unnecessary and undesirable repetition of the class name, e.g. `parseLong`, `getLong`.
    - ▶

## Getting numeric information from a string using wrapper classes (JDK 1.1)

<code>int</code>	<code>Integer.parseInt(s)</code>
<code>long</code>	<code>Long.parseLong(s)</code>
<code>short</code>	<code>Short.parseShort(s)</code>
<code>byte</code>	<code>Byte.parseByte(s)</code>
<code>float</code>	<code>Float.valueOf(s).value()</code>
<code>double</code>	<code>Double.valueOf(s).value()</code>

*Another triumph of consistent  
and elegant language design?*

## Obstacles to learning

- Java enforces disciplines that make it hard to teach/learn the language incrementally. We have to confront the whole language at once!
- The instructor has to tell beginning students to copy certain constructs on faith, without understanding what they do or why they're needed.

▶ example: main function header

```
public static void main (String args[])  
    throws IOException
```

*What's this for?*

*What's does this do?*

## The bottom line: Conclusions about Java

- on re-use
  - on quality and maintainability
  - on teaching and learning
- 

## The bottom line on reuse

- Java provides excellent support for reuse of:
  - complete classes
  - distributed components ("Beans")
- But it's very poor in support for lower-level building blocks (worse even than Cobol!)
  - no macros or preprocessor (not even `#include`)
  - no templates or generic functions
- So-called "interface specifications" are just a tool for enforcing discipline, not a way of packaging components for reuse.

## The bottom line on quality

- Mediocre programmers will create poor programs in any language.
- Superior programmers encounter obstacles in Java to important techniques in program structure and coding style. It's hard or impossible to do some things in Java that are easy in:
  - ▶ C++
  - ▶ PL/I
  - ▶ Ada
  - ▶ even modern versions of Cobol!
- Large Java programs, therefore, will be more costly to maintain than equivalent *high-quality* programs in those languages.

*So when should we  
use Java?*

### **The bottom line on teaching/ learning the O.O. paradigm**

- Java is somewhat easier to learn than C++ (or Smalltalk) in a *short* course for programmers of limited experience or casual interest, e.g. in a 2nd course in programming.
  - ▶ Avoids C++ pointers and memory management
  - ▶ But imposes a lot of confusing complications that partly offset that advantage.
- C++ provides a sounder foundation than Java in a longer course for professional software developers.
  - ▶ Provides more opportunity to demonstrate good programming practices, re-use, etc.
  - ▶ But takes longer to get started, mainly due to the need to get comfortable with pointers.

### **Ease of learning for students who already know C++**

- Programmers who have a good command of C++ pick up Java quickly.
  - ▶ They recognize Java's strengths
  - ▶ They have a basis for comparison in understanding Java's weaknesses, and know from their C++ experience what they should be trying to accomplish.
- For programmers who already have a good command of C (procedural programming) we can cover both languages in a full-semester (16 weeks) course, C++ in depth plus a good introduction to Java