the Bad, and the Ugly
Java: The Good Parts

Joshua Bloch
Google
Preliminaries

- This talk is highly opinionated
  - All opinions are my own, not my employer’s or Miss Piggy’s
- This talk contains some criticism
  - All of it is intended to be constructive
  - I still ♥ Java
Agenda

• This talk is limited to the platform as *initially released* (1.02)
  – The best and worst features in the language
  – A more thorough critique of the libraries—every non-exception class!
  – What made the platform succeed, and what limited its success
  – On Friday, I’ll give a talk about subsequent language changes

• JDK 1.02 is 16 years old, and memories are starting to fade
  – A shout-out to Doug Lea, Jesse Wilson, and Frank Yellin for contributing their recollections
Language

The Good Parts
Attributes Critical to Java’s Success—Fundamentals

• Safe language / managed runtime
  – No segfaults, scribble bugs, etc.

• Tightly specified primitive types, expression evaluation order, etc.
  – Greatly facilitates program portability
  – A natural accompaniment to a managed runtime

• Dynamic linking
  – In bad old days, changing a library required recompile of all clients

• Superficial similarity to C/C++
  – Appealed immediately to lots of C and C++ programmers
  – Many Lisp and Smalltalk advantages, but no syntactic culture shock
Attributes Critical to Java’s Success—Type System

• Object Oriented language
  – Encapsulation is necessary to prove components correct in isolation
  – Inheritance was, if nothing else, a marketing necessity

• Multiple interface inheritance
  – Enables easy pluggability of components
  – Avoids the pain of multiple implementation inheritance

• Static typing
  – Detects bugs at compile time, increasing program reliability
  – Enables IDE to help programmer write correct code quickly (autocompletion, code generation)
  – Enables VM to generate high performance code
Attributes Critical to Java’s Success—Features

• **Threads**
  – Twilight of uniprocessor era; concurrency was increasingly important
  – Threading must be designed into the language; it can't be tacked on
    • Boehm, Hans-J., *Threads Cannot be Implemented as a Library*, PLDI 2005
    • Buhr, Peter A., *Are Safe Concurrency Libraries Possible?*, CACM, 38(2):117-120, 2/95

• **Garbage Collection**
  – Eliminates pain and bugs that go with manual memory management

• **Exceptions**
  – Error codes are error prone
What You Leave Out Can Be as Important as What You Put In

• I believe these omissions were critical to Java’s success
• Lexical macros
  – Made all Java programs look similar
  – Enabled programmer portability
  – Enabled world-class toolability
• Multiple implementation inheritance
• Operator overloading
A Potent Pair of Design Decisions That is Often Overlooked

• (1) Java omitted support for header files (external interface declarations)
• (2) Javadoc largely eliminated external API documentation
• Declaration, documentation, and implementation collocated!
  – Greatly eases the task of keeping everything in sync
• Good API documentation became part of culture from start
Language
The Bad and The Ugly Parts
The Bad and The Ugly—
Expression Evaluation

• Silent “widening” conversions from int to float and long to double are lossy

• Compound assignment operators cause a silent narrowing cast

  // Infinite loop! See Java Puzzler 31, Ghost of Looper
  short i = -1;
  while (i != 0)
    i >>= 1;  // i = (short) (i >> 1);

• Operators == and != do reference comparisons even if equals is overridden

• Constant variables are inlined where they are used
  – Violates dynamic linking invariant ("chink in the armor")
  – Conditional compilation was insufficient justification
The Bad and The Ugly—Constructors

• Default constructors should not exist
  – And they certainly shouldn’t be `public`
  – Lead to unintentional instantiability, and sloppy API documentation

• Invoking overridden method from constructor should be illegal
  – It has no valid uses, and leads to subtle bugs
  – See Puzzler “That Sinking Feeling”
The Bad and The Ugly—Concurrency

• All objects have a lock associated with them
• The lock associated with an object is publicly visible
  – So, for example, any applet can do this:
    ```java
    synchronized(Thread.class) {
      // Prevents thread creation
      Thread.sleep(Integer.MAX_VALUE);
    }
    ```
• All locks have exactly one associated “condition variable”
  – Results in unnecessary context switches
  – Under some circumstances, $O(n^2)$ instead of $O(n)$
• Lack of unsigned `int` and `long` types
  – Worse, `byte` is signed!

• Switch statement is not structured

• Arrays should have overridden `toString` to be informative
  – Every CS101 student stumbles over this

• Exceptions obliterate pending exceptions

• Guaranteed `String` constant interning
  – Should have been a suggestion to implementers
  – Any program that depends on it for correctness is broken
  – But interning and left-to-right evaluation decided at same meeting!
  – Frank Yellin takes credit for both ("They were stolen from Common Lisp")
  – Hindsight is 20-20!
### Only 7 Packages—4 Core, 3 AWT

<table>
<thead>
<tr>
<th>Package</th>
<th>Classes</th>
<th>Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>java.util</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>java.io</td>
<td>23</td>
<td>3</td>
</tr>
<tr>
<td>java.net</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>65</strong></td>
<td><strong>10</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Package</th>
<th>Classes</th>
<th>Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.awt</td>
<td>42</td>
<td>2</td>
</tr>
<tr>
<td>java.awt.image</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>java.awt.peer</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>java.applet</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>52</strong></td>
<td><strong>8</strong></td>
</tr>
</tbody>
</table>

**GRAND TOTAL** 117 classes 18 interfaces
Object - The root class
String - A string
StringBuffer - A builder for strings
Integer, Long, Float, Double - Numerical primitive wrappers
Number - Superclass for numerical wrappers
Boolean, Character - Wrappers for other primitives
Math - Static utility class for numerical computation

Throwable - The root of the exception hierarchy
Thread - A thread of control
ThreadGroup - A hierarchical group of threads
Class - A class (rudimentary reflection)
Process - A process on the underlying OS
System, Runtime - Access to VM and environs
Runnable - A closure, per Guy Steele’s definition
Cloneable - “Mixin” indicating cloneability

ClassLoader - Flexible dynamic class loader
SecurityManager - Pluggable security policy
Compiler - Static class to control JIT (if any)
Object - Methods `equals` and `hashCode` enabled global equivalence relation, hash-based collections

String - Immutability very ballsy for the time, a huge win
- Parameterless and copy constructors were misguided
- Hash values specified down to the bit (in JLS!)—very bad idea

StringBuffer - Mutable Companion Class / Builder
- Overall structure is great
- But should never have been synchronized
**java.lang Critique (2/6)**

**Integer, Long, Float, Double, Character, Boolean**

- Immutable is beautiful!
- `equals` semantics differ from primitive `==`. Necessary but confusing
- Hash values specified down to the bit
- `Boolean(booleanValue)` - should have been a static factory
- `MIN_VALUE` - Inconsistent meaning in floating point and integral types
- Hideous hierarchy inversion - `Boolean.getBoolean(String)` returns `true` if named system property exists and is equal to "true"
- WTF happened to `Byte` and `Short`?

**Number**

- Failed attempt to abstract over wrapped numerical primitives
- Impossible task; should have been omitted
Math

- Reusing well-known C names and semantics a wise choice
- Providing only floating point `random` was a bad idea
- `abs(Integer.MIN_VALUE)` maybe should have thrown exception

Throwable - Exception hierarchy is broken

- Unchecked (and checked) throwables don’t have a single root
- Checked exceptions heavily overused
  - `InterruptedException`, `IOException` particularly painful
- Little consistency in organizing exceptions
- Many botches, e.g. `NoSuchMethodError`, `NoSuchMethodException`
- “I don’t believe much thought was given to the exception hierarchy.” – Frank Yellin
Thread

- Extending Runnable violates LSP, but made sense at the time
- stop, suspend, resume - deprecated in 1.2; should’ve been omitted
- State transitions ill-defined; it took years to get this right (1.5!)
- Thread priorities are complex and not portable but necessary

ThreadGroup - Pretty much useless and filled with errors

- Originally thought to have security uses—it didn’t
- “Putting your cooperating threads into a hierarchy seemed like a good idea at the time” – Frank Yellin
java.lang Critique (5/6)

Class - `newInstance` limited and broke exception safety

Process - Extremely useful but completely nonportable
- With one rubicon crossed, should have included other such features!
- Serious usability issues concerning output, error streams (Puzzler 82)

System, Runtime - should have been combined
- `in`, `out`, `err` are public fields we soon came to regret!
  - “This came straight from C/Unix.” – Frank Yellin
  - Final fields, but we added methods to change them in 1.1 (!)
  - Memory model (1.5) needed special language to cover this mess
- `arraycopy` violates naming conventions (arrayCopy), type safety

Runnable - `void return` a bit sad, but correct for its time
Cloneable - utterly broken, should have been omitted
- It lacks a clone method
- Worse, creating objects without invoking constructors is evil
- Sent us down the wrong path for serialization

ClassLoader - Very powerful but ill-understood
- Left huge bug/complexity tail in its wake (e.g., “shadow type system”)

SecurityManager - designed to enable foreign code
- Clever idea for its time; it seemed that security was designed in
- Never made good on promise—shared memory security is intractable
  - To this day! (Search web for: site:oracle.com applet security vulnerability)
- Security is a cross-cutting concern; made Java code made uglier

Compiler - initially forward-looking, then obsolete
java.util Summary

Vector - Growable array
Hashtable - Map from keys to values
Enumeration - Stream of values
Date - Point in time
Random - Pseudorandom number generator
BitSet - Growable bit vector
Properties - Persistable String to String map
Dictionary - Abstract map from keys to values
Stack - LIFO stack
StringTokenizer - Rudimentary String scanner
Observable - Superclass for observable objects
Observer - Callback to monitor Observable
Vector, Hashtable - Fantastic! Most useful data structures

- C++ STL takes a book to describe and still no a hash table (2011)
- Synchronizing everything was probably a mistake, but defensible
- Minor botches, e.g., `Hashtable.contains(Object value)`, `Hashtable -> HashTable`
- Implementation dependent tuning parameters hinder evolution

Enumeration - did the job

- External iteration was correct for its day
- Names too long for such commonly used functionality

Date - Should have been immutable!

- “Java’s worst botch” – Doug Lea
Random - Decent, but should have been an interface
  – “Interfaces were second-class citizens. Concept of programming to interfaces never occurred to anyone.” – Frank Yellin
  – Like hash codes, it was a mistake to specify results bit-for-bit

BitSet - Not clear that it paid for itself, but not bad
  – Many important operations missing; added in 1.2
  – Size abstraction ill-defined; “fixed” in 1.2

Properties - A huge mess
  – Should not be a subclass of hashable (is-a / has-a confusion)
  – Hierarchical defaults are complexity for its own sake
  – On disk format is so poor that it’s not amenable to a BNF...
  – And it’s with us forever because of system properties
  – API says keys, values must be strings; impl permits arbitrary objects
Dictionary
– Should have been interface not abstract class
– Blessing in disguise: In 1.2, it was clear that Map was the real deal
– Should have been excluded (YAGNI)

Stack - Hideously broken toy
– Extreme is-a / has-a confusion (extends Vector)
– A great piece of real estate wasted forever
StringTokenizer - A broken toy
  – Unrelated StreamTokenizer class was a bad sign
  – Multiple ways of doing things have multiplied...
  – But other languages are even more prone to this fault

Observable/Observer - Another broken toy
  – Interface is overly complex (hasChanged, clearChanged)
  – To a first approximation, no one ever used it.
  – Not a big problem, but arguably a lost opportunity (pub/sub service)
java.io Summary

InputStream - Abstract superclasses for byte-oriented IO streams
FileInputStream - Concrete file impls
ByteArrayInputStream - Concrete byte[] impl
StringBufferInputStream - Concrete StringBuffer impl
PipedInputStream - Concrete stream pair
BufferedInputStream - Buffering decorators
DataInputStream - Primitive typing “decorator”
PushbackInputStream - Pushback “decorator”
LineNumberInputStream - Line numbering “decorator” (deprecated in 1.1)
SequenceInputStream - Sequencing “decorator”
FilterInputStream - Forwarding wrappers

PrintStream - Typed, string-converting output stream (e.g., System.out)
StreamTokenizer - Tokenizer that reads from an InputStream
RandomAccessFile - Positionable, streaming, primitive typed read/write file

File - A file name and metadata; no read/write functionality
FileDescriptor - An open file or socket; similar to Unix FD

DataInputStream - Mixin interfaces for primitive typing
FilenameFilter - A predicate on (dir, filename)
Looked decent, but didn’t stand up to serious use
Inconvenient to program against
  – Buffering requires wrapping in `Buffered{Input,Output}Stream`
  – All exceptions checked (they inherit from `IOException`)
  – `InputStream.skip` treats its argument as a hint!
    – Using it safely requires a huge amount of boilerplate
    – See Puzzler “Mind the Gap” for details

Lacked orthogonality, e.g. `RandomAccessFile`

Performance was inherently poor
Did not scale well
  – `java.nio` was intended to address this in 1.4
  – It didn’t (see Paul Tyma’s blog post http://goo.gl/IFT6k)
java.net Summary

Network APIs

- **DatagramPacket** - Container to send and receive `byte[]` data
- **DatagramSocket** - Socket on which to send and receive packets
- **Socket** - Stream socket to communicate via `{Input, Output}`

Stream APIs

- **ServerSocket** - Socket-factory for servers to accept stream socket connections
- **InetAddress** - An IP address

- **SocketImpl** - (abstract superclass for `Socket`)
- **SocketImplFactory** - “Plugin” for socket creation

Web APIs

- **URL** - a WWW URL
- **URLConnection** - Communication link between application and URL
- **URLEncoder** - Single static method to format strings for use in URLs

- **URLStreamHandler** - Generates `URLConnection` from `URL` for a protocol type
- **ContentHandler** - Reads an object from a `URLConnection`
- **ContentHandlerFactory** - Translates MIME types to `ContentHandler`
- **URLStreamHandlerFactory** - Translates protocol names to `URLStreamHandler`
“Java’s socket API was big breath of fresh air after C/Unix; Students had to suffer through a week of wasted lecture time to learn how to use the C API. In Java, ten minutes and you’re ready to go. Sure, it’s a little bit broken; for economy they conflate the notion of socket and socket factory (ServerSocket). But that’s a minor quibble.” — Doug Lea

A fine abstraction of BSD networking
All still commonly used today
Few TCP options supported; fixed by SocketOptions in 1.1
Extensibility turned out to be largely unnecessary
And the edifice wasn’t terribly well designed

- `URL.hashCode` and `equals` are blocking operations and violated their general contracts - `URI` replaced `URL` in 1.4
“Duke could dance. He could juggle. I could write a chess program. AWT was amazing for its time. Remember that it was designed for applets.” — Frank Yellin

“Unless your app was a single-line text entry field with buttons and labels, you couldn’t do anything useful. If you wanted to do any reasonable programming in ’97, you needed a table / list view control. With AWT if you needed a component and you didn’t have it, you were just #$%^.” — Jesse Wilson

“The huge challenge was doing layouts—it was thoroughly painful. It wasn’t until IDE tools that you had a chance.” — Doug Lea
• No one understood the AWT threading model in those days
  – Maybe there wasn’t one
• All published AWT/Swing examples through 2000 were broken
  – In those days it didn’t matter much
• java.awt.EventQueue.invokeLater not added till 1.2
My Take on AWT, for What It’s Worth

• Best viewed as a proof of concept
• Impressive for how quickly it was developed
• Showed the power of Java in the browser
  – Essential for Java’s success (at the time)
• Never did much more than demo applets
  – But that’s understandable
• No good reason for AWT’s zombie to exist in Swing
  – Should have veneered over it
Overall Libraries Critique

- Small and manageable
  - A programmer could easily understand the entire platform
- Far simpler than C++ and far more powerful than C
- It was clear that many APIs hadn’t been seriously used
  - Some toys should have been left out
  - Some APIs should have been simplified
  - There were many small API flaws
- Oversynchronization was common
  - Design philosophy: synchronize every method, don’t worry about threads
  - “I guess that philosophy is morally bankrupt.” – Tim Lindholm, ca. 2000
- On balance, libraries were good enough
  - Astonishing considering time constraints
Conclusion

• The good parts: key design decisions
• The bad and the ugly: largely confined to details
• Market window was open in 1995, and Java leapt through it
• Java’s success wasn’t a result of hype and marketing
• Gosling and Company made the right design decisions at the right time
the Bad, and the Ugly

Java: The Good Parts

Joshua Bloch
Google