10 reasons why Java now rocks more than ever

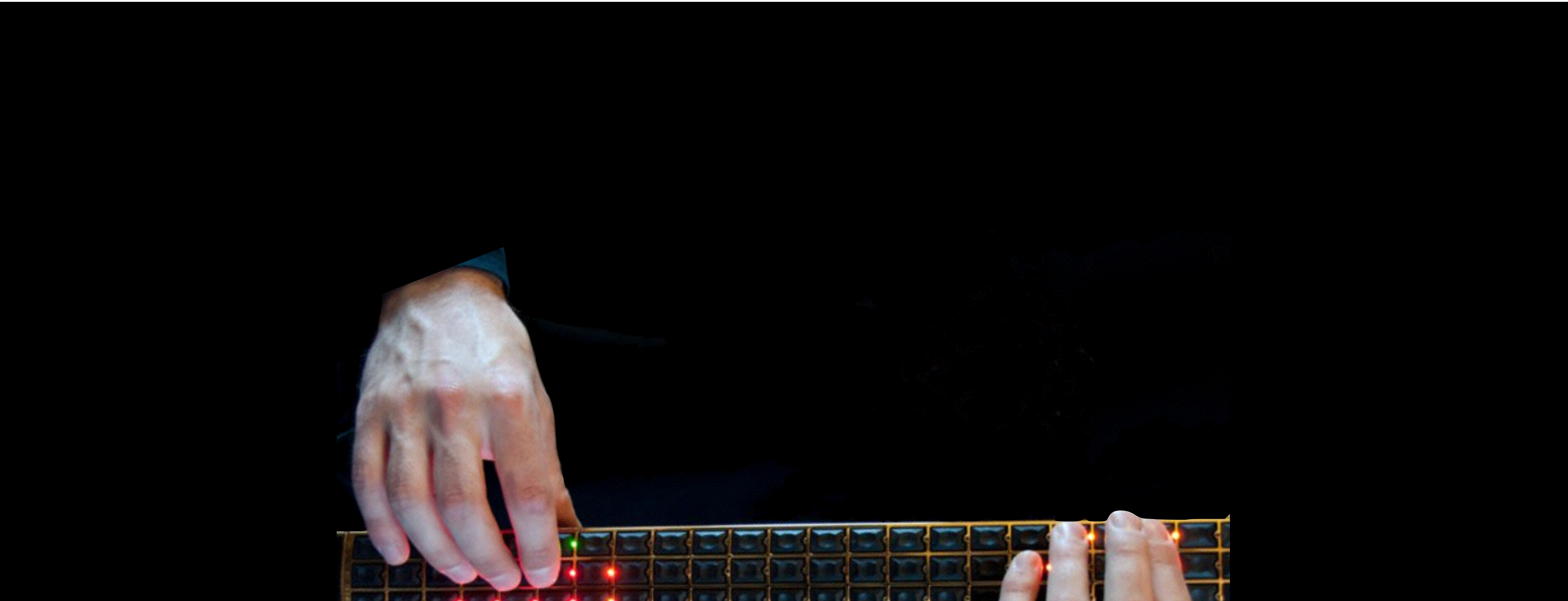
Geert Bevin - XRebel Product Manager

ZEROTURNAROUND

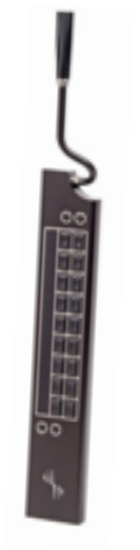
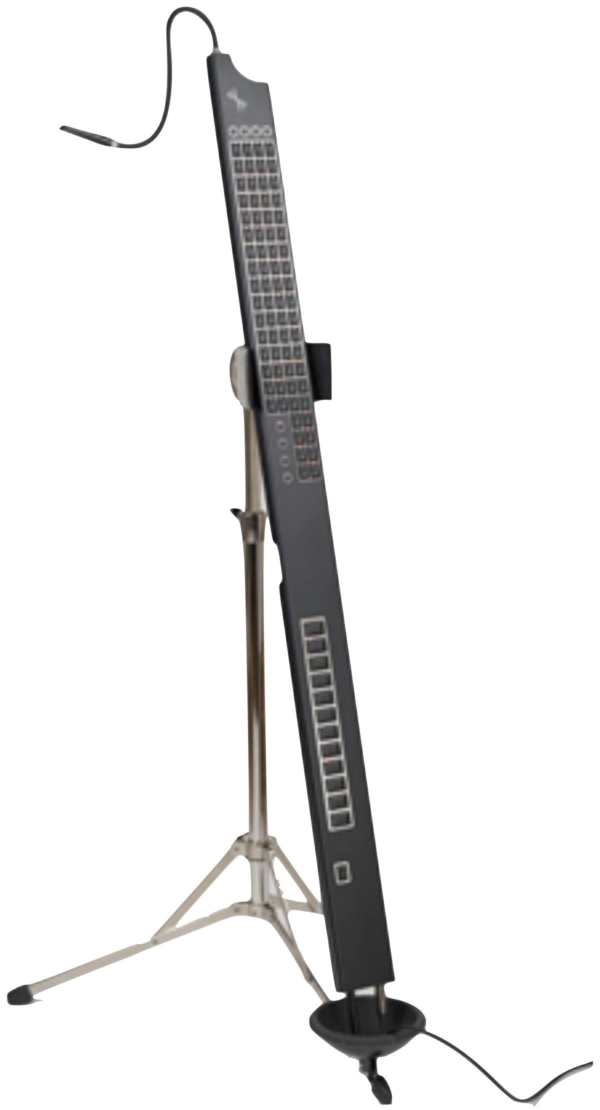
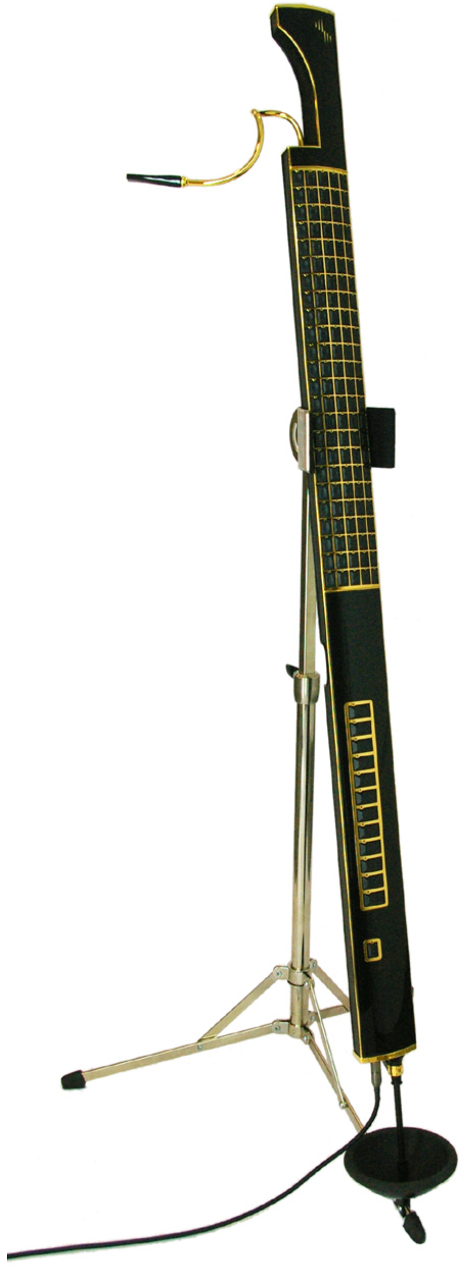
Who am I?

- Geert Bevin
- XRebel Product Manager at ZeroTurnaround
- Java Champion
- Creator of RIFE framework, pioneering native Java continuations
- Many other open-source projects

My previous three years



Software for Eigenharp instruments

Away from Java

- Cross-platform **real-time audio** software
- Massive throughput, **minimal and constant latency**
- **C++** (with Juce library) and **CPython**
- Designed for **musical performance** and total **customization**

Frustrating development

- Started **missing Java** more and more
- Constantly stumbled into things I had **taken for granted**
- So here are ...

10 features we take for granted, making Java rock more than ever

#1 The Java Compiler

The Java Compiler

- Compiles to bytecode, allowing for JIT at runtime
- Very few compiler-specific semantics to understand
- Everything is dynamically linked and loaded
- Useful error messages

☹ Straight to native compilation

- Native platform bleeds through in your high-level code
- Decide optimization levels up-front with aggressive levels potentially causing problems
- Distribution and maintenance of different platform binaries

☹ C++ error message example

```
- pic::lckvector_t<piw::data_nb_t>::nbtype last_audio_;
+ pic::lckvector_t<piw::data_t>::nbtype last_audio_;
```

```
tmp/obj/eigend-gpl/piw/src/piw_gain.os (g++-4.2)
eigend-gpl/piw/src/piw_gain.cpp:In member function ‘long long unsigned
int<unnamed>::gainbase_t::input_audio(unsigned int, const piw::data_nb_t&)’:
eigend-gpl/piw/src/piw_gain.cpp:51:error: no match for ‘operator=’ in
‘((<unnamed>::gainbase_t*)this)-><unnamed>::gainbase_t::last_audio_.
std::vector<_Tp, _Alloc>::operator[] [with _Tp = piw::data_t, _Alloc =
pic::stlnballocator_t<piw::data_t>](((long unsigned int)index)) = d’
eigend-gpl/piw/piw_data.h:247:note: candidates are: piw::data_t&
piw::data_t::operator=(const piw::data_t&)
```

☹ G++ != Clang != ICC != VSCC

- Each compiler on each platform compiles differently
- Each compiler has different features, capabilities and arguments
- Hundreds of options to wade through, requiring sometimes very low-level knowledge of the hardware

☹ Linking and libraries

- Native linking is slow, especially when link-time optimization is enabled
- Constant trade-off between dynamic libraries and static libraries
- DLL version management
- DLL visibility management (symbols export and import)

#2 The Core API

The Core Java SDK

- Official SDK is very complete and sufficient for writing apps
- Used consistently throughout Java projects, people don't avoid it
- Even when wrapped, it's still underneath other libraries
- Free to use and deploy with a liberal license
- Drive towards standardization on the platform, not just language
- Stand on shoulders of everyone, vibrant open-source community

☹ Others have limited core SDK

> C++ has the STL but it’s very limited and not sufficient

> Mixing libraries without foundation turns into a nightmare

> Typical projects have 3 different String classes, sometimes more

> Different handling of endianness, encodings, threading, …

> Relying on existing solutions often pulls in additional platforms

> Python’s SDK is more complete, documentation is lacking

#3 Open-Source

Open-Source

- Pervasive open-source mindset with vibrant communities
- Spirit of collaboration and curiousness
- Genuinely useful and active open-source projects
- Everything about Java as a platform and language is open
- Many professional open-source companies
- Massive collection of working code examples

☹ Islands and fragmentation

> Lack of core API splits open-source efforts into isolated silos
> Open-source is mostly academic or lone individuals
> Expected music software world to be all about sharing, total opposite
> Slightest piece of functionality is closely guarded in secrecy
> Stuck into standards from 1980, nobody works together to innovate

#4 The Java Memory Model

☺ The Java Memory Model

- Bullet-proof specification of multi-threading interactions
- Low-level JSR-133 set of ‘happens-before’ rules
- Predictable visibility across threads
- Predictable ordering across threads
- Allows for higher level concurrency constructs like actors
- 1st platform to provide this, perfected and stabilized over years

☹ Concurrency is afterthought

- Taken until 2011 for C/C++ to standardize memory model
- Threads are difficult, imagine using and combining different implementations
- Python has GIL, threads can’t concurrently access state
- Some advocate resorting even to multi processes, what about state there?
- On your own when using other concurrency constructs

#5 High-Performance VM

High-Performance VM

- Built-in garbage collection that is under constant improvement
- Truly multi-platform and actively supported on each OS
- Very tunable with startup options
- Out-of-the-box runtime monitoring and management
- Runs on everything from embedded devices to super-servers
- Standard JMX spec available for managing non-VM resources

☹ Manual memory management

> High-throughput & high-performance needs manual heap pools
> No visibility, profiling, monitoring
> Code is dictated by how memory is managed: RAII, smart pointers, reference counting, ownership transfer
> Now imagine managing memory across C++ and CPython
> Without a stable memory model!

☹ Work for target architecture

- Different tests for each architecture
- Different build process for each architecture
- Different compiler and language variant for each architecture
- Different build tools for each architecture
- Different packaging and installing for each architecture
- Different development tools on each architecture

#6 Bytecode

Java

```java
public class HelloWorld {
  public HelloWorld() {
  }

  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

javap -p -c HelloWorld

```
public class HelloWorld {
  public HelloWorld();
    Code:
       0: aload_0
       1: invokespecial #1  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3  // String Hello World!
       5: invokevirtual #4  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

Bytecode

> Understandable for Java developers
> Offers indispensable genuine 'glue' for different languages and tools
> Can be generated and modified at runtime
> Enabled alternative languages and frameworks
> Inspectable through JDK tools and IDEs

Bytecode changed Java

- FindBugs for static code analysis
- Groovy, Scala, Clojure generate bytecode from different source
- ORM tools instrument your code for database operations
- DI frameworks seamlessly weave application lifecycle together
- Augment Java by modifying classes that javac generated
- JRebel instruments for instant reloads
- XRebel instruments to find performance problems

Your imagination is the limit

- Write your own compiler for a language you come up with
- Write a static transformer that pre-processes existing classes
- Write an instrumentation agent that plugs right into the JVM and performs bytecode manipulation on-the-fly
- Write custom classloaders but that's nowadays strongly discouraged

#7 Intelligent IDEs

Intelligent IDEs

- Project-wide understanding of the Abstract Syntax Tree
- Refactor with confidence
- On-the-fly error highlighting
- Local and remote debugging with great visibility
- Native support for most frameworks, languages and libraries
- Integration with run-time platforms

#8 Profiling Tools

Profiling Tools

> Profiling tools used during production and attached dynamically
> CPU, threading, memory, exception, GC, ... profiling
> Snapshotting with deep inspection and offline analysis
> IDE integration
> Heap dump facility built into JVM and available to anyone
> All the tools your need to perform root cause analysis

#9 Backwards Compatibility

Backwards Compatibility

- Applications from more than a decade ago are still supported
- Leverage modern JVM improvements
- Drop-in replacement of new JVMs with existing applications
- Older source code still compiles even though the Java language evolved
- Standing on each-other's shoulders for 18 years!

Forwards Compatibility

- Current applications will continue to run on newer JVMs
- Reap benefits of hardware improvements without knowing low-level semantics
- Shield yourself from platform obsolescence
- Examples: garbage collector, management and monitoring, real-time profiling, HotSpot adaptive optimization, escape analysis for lock elision, lock coarsening, class data sharing, ...

Sideways Compatibility

- Comfort to change your environment to different OSs
- You can rely on your own software components even if the entire industry jumped ship
- Your current investments will remain useful for decades

#10 Maturity With Innovation

Maturity With Innovation

- Platform is very mature and used for mission-critical operations
- Maturity allows for teamwork
- Language and VM are under steady research and innovation
- Great balance between steady pace to allow adoption and introduction of new features
- J8: lambdas, default methods, streams, compact profiles, type annotations, G1 garbage collector, Nashorn, JavaFX, ...

The most well-balanced compromise between writing productivity, reading intuitiveness, execution performance, project maintainability, technology evolution, application stability and runtime visibility

productivity

intuitiveness

performance **maintainability**

evolution

stability **visibility**

Read further at

http://zeroturnaround.com/rebellabs

ZEROTURNAROUND