IBM

developerWorks.

Java.next: Common ground in Groovy, Scala, and Clojure, Part 2

Learn how the Java.next languages reduce boilerplate and complexity

Neal Ford
Director / Software Architect / Meme Wrangler
ThoughtWorks Inc.

Skill Level: Intermediate

Date: 16 Apr 2013

Common complaints about the Java™ language concern excessive ceremony for simple tasks and defaults that are sometimes confusing. All three of the Java.next languages take more sensible approaches in those areas. This installment of *Java.next* shows how Groovy, Scala, and Clojure smooth out the Java language's rough edges.

View more content in this series

About this series

The Java legacy will be the platform, not the language. More than 200 languages run on the JVM, each bringing interesting new capabilities beyond those of the Java language. This series explores three next-generation JVM languages — Groovy, Scala, and Clojure — comparing and contrasting new capabilities and paradigms. The series aims to give Java developers a glimpse into their own near future — and help them make educated choices about the time they devote to new-language learning.

The Java programming language emerged under constraints that are different from the conditions that developers face today. In particular, the primitive types in the Java language exist because of the performance and memory constraints of mid-1990s hardware. Since then the Java language evolved to autobox much of the pain away, but the Java.next languages — Groovy, Scala, and Clojure — go a step further toward removing inconsistencies and friction throughout each language.

In this installment, I show how the Java.next languages do away with some common Java limitations, both in syntax and in default behavior. The first limitation to go is the presence of primitive data types.

The death of primitives

The Java language began with eight pairs of primitives and corresponding type-wrapper classes — originally to address performance and memory constraints — and gradually blurred the distinction with autoboxing. The Java.next languages go much further, making it seem to developers as if a difference no longer exists.

Groovy completely hides the existence of the primitive types from you. For example, `int` always stands in for `Integer`, and Groovy automatically handles upconverting numeric types to prevent numeric-overflow errors. For example, see the Groovy shell interaction in Listing 1:

Listing 1. Groovy's automatic treatment of primitives

```
groovy:000> 1.class
===> class java.lang.Integer
groovy:000> 1e12.class
===> class java.math.BigDecimal
```

In Listing 1, the Groovy shell shows that even constants are represented by underlying classes. Because all numbers (and other ostensible primitives) are really classes, you can use metaprogramming techniques. The techniques include adding methods to numbers — often used in building domain-specific languages (DSLs), allowing expressions like `3.cm`. I'll cover this capability more fully in an upcoming installment about extensibility.

As in Groovy, Clojure automatically masks the difference between primitives and wrappers, allowing method invocations against all types and handling type conversions for capacity automatically. Clojure encapsulates a huge number of optimizations underneath, which is well-covered in the language documentation (see Resources). In many cases, you can provide type *hints* to enable the compiler to generate faster code. For example, rather than define a method with `(defn sum[x] ... )`, you can add a type hint such as `(defn sum[^float x] ... )`, which generates more-efficient code for critical sections.

Scala also masks the difference, generally using primitives underneath for time-critical parts of code. It also allows method invocations on constants, as in `2.toString`. With its ability to mix and match primitives and wrappers like `Integer`, Scala is more transparent than Java autoboxing. For example, the `==` operator in Scala works correctly (comparing the values, not the references) across primitive and object references, unlike the Java version of the same operator. Scala also includes an `eq` method (with a symmetrical `ne` method) that always compares equality (or inequality) of the underlying reference type. Basically, Scala intelligently switched the default behavior. In the Java language, `==` compares the references, which you almost never need, whereas the less intuitive `equals()` compares values. In Scala, `==` does the correct thing (compares values) regardless of underlying implementation, and provides a method for the less-common reference-equality check.

This feature of Scala illustrates that one key benefit of the Java.next languages lies in offloading low-level details to the language and run time, freeing developers to think more about higher-level problems.

Simplifying defaults

With near-universal consensus, most Java developers agree that common operations require too much syntax in the Java language. For example, property definitions and other boilerplate code clutter class definitions, obscuring the important methods. All the Java.next languages provide ways to streamline creation and access.

Classes and case classes in Scala

Scala already simplifies class definitions by automatically creating accessors, mutators, and constructors for you. For example, consider the Java class in Listing 2:

Listing 2. Simple `Person` class in Java

```
class Person {
    private String name;
    private int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return name + " is " + age + " years old.";
    }
}
```

The only nonboilerplate code in Listing 2 is the overridden `toString()` method. The constructor and all methods were generated by the IDE. Producing code quickly isn't as valuable as understanding it easily later. Needless syntax adds to the volume of code that you must use before you understand the underlying meaning.

Scala `Person` class

Stunningly, the simple three-line definition in Scala in Listing 3 creates an equivalent class:

Listing 3. Equivalent class in Scala

```
class Person(val name: String, var age: Int) {
  override def toString = name + " is " + age + " years old."
}
```

The `Person` class in Listing 3 boils down to a mutable `age` property, an immutable `name` property, and a two-parameter constructor, plus my overridden `toString()` method. You can easily see what's unique about this class because the interesting parts aren't drowning in syntax.

Scala's design emphasizes the ability to create code with minimal excess syntax, and it makes much syntax optional. The simple class in Listing 4 illustrates a verbose class that changes a string to uppercase:

Listing 4. Verbose class

```
class UpperVerbose {
  def upper(strings: String*) : Seq[String] = {
    strings.map((s:String) => s.toUpperCase())
  }
}
```

Much of the code in Listing 4 is optional. Listing 5 shows the same code, now an `object` rather than a `class`:

Listing 5. A simpler to-uppercase object

```
object Up {
  def upper(strings: String*) = strings.map(_.toUpperCase())
}
```

For the Scala equivalent to Java static methods, you create an `object` — Scala's built-in equivalent of a singleton instance — rather than a class. The method's return type, the braces that delimit the single-line method body, and the needless `s` parameter from Listing 4 all disappear in Listing 5. This "collapsible syntax" is both a blessing and curse in Scala. With collapsible syntax, you can write in extraordinarily idiomatic ways, which can make your code difficult for the uninitiated to read.

Case classes

Simple classes that act as data holders are common in object-oriented systems, particularly ones that must communicate with dissimilar systems. The prevalence of this type of class encouraged the Scala project to go even further and create *case classes*. Case classes automatically provide several syntactic conveniences:

- You can create a factory method that is based on the name of the class. For example, you can construct a new instance without bothering with the `new` keyword: `val bob = Person("Bob", 42)`.
- All arguments in the class's parameter list are automatically `val`s, meaning they are maintained as immutable internal fields.

- The compiler generates reasonable default `equals()`, `hashCode()`, and `toString()` methods for your class.
- The compiler adds a `copy()` method to your class so you can make mutating changes by returning a new copy.

The Java.next languages don't merely fix syntactic warts but also manifest a better understanding of how modern software works, molding their facilities toward it.

Groovy's autogenerated properties

Of the Java.next languages, Groovy adheres most closely to Java syntax, providing syntactic-sugar code generation for common cases. Consider the simple `Person` class in Groovy in Listing 6:

Listing 6. Groovy `Person` class

```
class Person {
  private name
  def age

  def getName() {
    name
  }

  @Override
  String toString() {
    "${name} is ${age} years old."
  }
}

def bob = new Person(name: "Bob", age:42)

println(bob.name)
```

In the Groovy code in Listing 6, defining a field `def` yields both an accessor and mutator. If you prefer only one or the other, you can define it yourself, as I do with the `name` property. Even though the method is named `getName()`, I can still access it through the more intuitive `bob.name` syntax.

If you want Groovy to generate the `equals()` and `hashCode()` pair of methods for you automatically, add the `@EqualsAndHashCode` annotation to your class. This annotation uses Groovy's *Abstract Syntax Tree (AST) Transformations* to generate methods that are based on your properties (see Resources). By default, this annotation takes only properties (not fields) into account; it considers fields too if you add the `includeFields=true` modifier.

Clojure's map-like records

You can create the same `Person` class in Clojure just as you can in the other languages, but that wouldn't be idiomatic. Traditionally, languages like Clojure rely on *map* (name-value pair) data structures to hold this type of information, with functions that operate against that structure. Although you still can model structured data in maps, the more common case now uses *records*. A record is Clojure's more formal encapsulation of a type name with properties (often nested), all with the same semantic meaning per instance. (Records in Clojure are like `struct`s in C-like languages.)

For example, consider the following person definition:

```
(def mario {:fname "Mario"
            :age "18"})
```

Given this structure, I can access `age` through `(get mario :age)`. Simple access is a common operation on maps. With Clojure, I can use the syntactically sugary fact that *keys act as accessor functions upon their maps* to use the better `(:age mario)` shorthand. Clojure expects operations with maps, so it has lots of syntactic sugar that makes it easy.

Clojure also has syntactic sugar for accessing nested map elements, as in Listing 7:

Listing 7. Clojure's shorthand access

```
(def hal {:fname "hal"
          :age "17"
          :address {:street "Enfield Tennis Academy"
                    :city "Boston"
                    :state "MA"}})

(println (:fname hal))
(println (:city (:address hal)))
(println (-> hal :address :city))
```

In Listing 7, I define a nested data structure named `hal`. Access to the outer elements is as expected (`(:fname hal)`). As the next-to-last line in Listing 7 shows, Lisp syntax does "inside out" evaluation. First I must get the `address` record from `hal`, then access the `city` field. Because "inside out" evaluation is a common usage, Clojure has a special operator — the `->` *thread* operator — that inverts expressions to make them more naturally readable: `(-> hal :address :city)`.

You can create the equivalent structure with records, as in Listing 8:

Listing 8. Creating a structure with records

```
(defrecord Person [fname lname address])
(defrecord Address [street city state])
(def don (Person. "Don" "Gately"
           (Address. "Ennet House" "Boston", "MA")))

(println (:fname don))
(println (-> don :address :city))
```

In Listing 8, I create the same structure with `defrecord`, yielding a more traditional class structure. With Clojure, I get the same convenient access within the record structure through familiar map operations and idioms.

In many situations, records are preferable to maps and plain structures. First, `defrecord` creates a Java class, making it easier to use in multimethod definitions. Second, `defrecord` does more tasks, enabling field validation and other niceties as you define your record. Third, records are much faster, especially in situations in which you have a fixed set of well-known keys.

Clojure uses records and *protocols* together to structure code. I'll explore that relationship in a future installment.

Conclusion

All three of the Java.next languages offer syntactic conveniences when compared to the Java language. Both Groovy and Scala make building classes and common cases easier, while Clojure makes interoperability of maps, records, and classes seamless. A common theme across all the Java.next languages is the removal of needless boilerplate. In the next installment, I continue with that theme and discuss exceptions.

Resources

Learn

- Java.next: Common ground in Groovy, Scala, and Clojure, Part 1 (Neal Ford, developerWorks, March 2013): Address the inability to overload operators in the Java language with the Java.next languages: Groovy, Scala, and Clojure).
- Java.next: The Java.next languages (Neal Ford, developerWorks, January 2013): Explore the similarities and differences of three next-generation JVM languages (Groovy, Scala, and Clojure) in this overview of the Java.next languages and their benefits.
- Scala: Scala is a modern, functional language on the JVM.
- Clojure: Clojure is a modern, functional Lisp that runs on the JVM. Read the Clojure primitive optimizations and interactions documentation.
- Groovy: Groovy is a dynamic language for the JVM. Read the Groovy AST Transformations documentation.
- Explore alternative languages for the Java platform: Follow this knowledge path to investigate developerWorks content about various alternative JVM languages.
- *Language designer's notebook*: In this developerWorks series, Java Language Architect Brian Goetz explores some language design issues that present challenges for the evolution of the Java language in Java SE 7, Java SE 8, and beyond.
- *Functional thinking*: Explore functional programming in Neal Ford's column series on developerWorks.
- More articles by this author (Neal Ford, developerWorks, June 2005-current): Learn about Groovy, Scala, Clojure, functional programming, architecture, design, Ruby, Eclipse, and other Java-related technologies.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Download IBM product evaluation versions or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Neal Ford

Neal Ford is Director, Software Architect, and Meme Wrangler at **Thought**Works, a global IT consultancy. He is also the designer and developer of applications, instructional materials, magazine articles, courseware, and video/DVD presentations, and he is the author or editor of books spanning a variety of technologies, including the most recent *Presentation Patterns*. He focuses on designing and building large-scale enterprise applications. He is also an internationally acclaimed speaker at developer conferences worldwide. Check out his website.