**Closures for Java**

**By mr on** [**Nov 24, 2009**](http://blogs.oracle.com/mr/entry/closures)

The free lunch [is over](http://www.gotw.ca/publications/concurrency-ddj.htm).

Multicore processors are not just coming—they’re here.

Leveraging multiple cores requires writing scalable parallel programs, which is incredibly hard.

Tools such as [fork/join frameworks](http://www.ibm.com/developerworks/java/library/j-jtp11137.html) based on [work-stealing](http://en.wikipedia.org/wiki/Cilk#Work-stealing) algorithms make the task easier, but it still takes a fair bit of expertise and tuning.

Bulk-data APIs such as [parallel arrays](http://www.ibm.com/developerworks/java/library/j-jtp03048.html?S_TACT=105AGX01&S_CMP=LP#2.0) allow computations to be expressed in terms of higher-level, SQL-like operations (*e.g.*, ﬁlter, map, and reduce) which can be mapped automatically onto the fork-join paradigm.

Working with parallel arrays in Java, unfortunately, requires [lots of boilerplate code](http://www.ibm.com/developerworks/java/library/j-jtp03048.html?S_TACT=105AGX01&S_CMP=LP#listing2) to solve even simple problems.

[Closures](http://en.wikipedia.org/wiki/Closure_%28computer_science%29) can [eliminate that boilerplate](http://www.ibm.com/developerworks/java/library/j-jtp03048.html?S_TACT=105AGX01&S_CMP=LP#4.0).

It’s time to add them to Java.

**The closures design space** In the last few years three serious proposals for adding closures to Java have been put forward: [BGGA](http://www.javac.info/closures-v05.html), [CICE](http://docs.google.com/Doc.aspx?id=k73_1ggr36h), and [FCM](http://docs.google.com/Doc?id=ddhp95vd_6hg3qhc). These proposals cover a wide range of complexity and expressive power. My view, having studied them all, is that each contains good ideas yet none is entirely appropriate for a “working programmer’s language.”

To support the principal use case of parallel programming we really only need two key features:

* A *literal syntax*, for writing closures, and
* *Function types*, so that closures are ﬁrst-class citizens in the type system.

To integrate closures with the rest of the language and the platform we need two additional features:

* *Closure conversion*, so that a closure of appropriate type can be used where an object of a single-method interface or abstract class is required, and
* *Extension methods*, so that closure-oriented bulk-data methods can be retroﬁtted onto existing libraries, and in particular the [Collections Framework](http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html), without breaking compatibility.

A couple of other integration features worth considering are ﬁrst-class method references and the ability of function types to include exception parameters.

Some of the other features found in the existing proposals carry considerable additional complexity:

* Capture of non-ﬁnal variables,
* Non-local transfer of control, and
* Library-deﬁned control structures (*i.e.*, control abstraction).

At present I see no need to add any of these to Java.

**Let’s be about it** It’s time to learn from the [past debate](http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html) and move forward. Sun will initiate the design and implementation of a simple closures feature, as outlined above, and add it to [JDK 7](http://openjdk.java.net/projects/jdk7) so as to enable broad experimentation. If all goes well then we’ll later submit a language-change JSR which would, in turn, be proposed as a component of the eventual Java SE 7 JSR.

Revising a programming language that’s in active use by millions of developers is no small task. Sun neither can nor should do it alone, so I hereby invite everyone who participated in the earlier closures conversations—as well as anyone else with an informed opinion—to join us.

<http://blogs.oracle.com/mr/entry/closures>

This story appeared on JavaWorld at  
http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html

# Understanding the closures debate

### Does Java need closures? Three proposals compared

By Klaus Kreft and Angelika Langer, JavaWorld.com, 06/17/08

With three proposals vying for inclusion in Java 7, understanding closures and the arguments for and against their inclusion in the Java language is essential. In this article Angelika Langer and Klaus Kreft give us a detailed overview of the three proposals -- BGGA, CICE, and FCM -- discussing the pros and cons of each, where they differ, and how they compare. The authors also explain the arguments against adding closures to Java 7, and conclude with their insight into where this debate will lead in the year ahead.

Closures, also known as blocks, are far from new to computer science. They're a standard feature in older languages such as Scheme, Smalltalk, and Lisp, and have more recently made their way into dynamic languages such as Ruby and Groovy. Closures are a natural fit for statically typed, functional Scala, and recently many have argued that they have a place in the Java language, too.

Many developers argue, conversely, that closures are a bad fit for the Java language. According to this camp, adding closures would be the proverbial "nail in coffin" for the already stuffed syntax of a language that tries to do too much. The end result: a hot debate, which has been making its way toward the JCP, and around the conference circuit and the blogosphere, for more than two years.

While the closures debate may seem best left to Java linguists, the inclusion of closures in Java 7 -- or not -- will ultimately affect anyone who writes Java code. In this article we provide an overview of the new functionality being proposed for the Java language. We start with a simple introduction to closures, then explain how each proposal would integrate them onto the Java language syntax. We also discuss each proposal's cost in terms of complexity. We consider the issue of closure conversion -- essential for backward compatibility with older programs -- and then delve into one of the most controversial aspects of the BGGA proposal: control abstractions.

Throughout the article we compare the pros and cons of the functionality each proposal would bring to the Java language, using code snips to demonstrate their impact on the Java programming experience. We conclude with an overview of the arguments for and against closures, as well as some insight into where these discussions may lead us in the year ahead.

## So, what are closures?

Essentially, a *closure* is a block of code that can be passed as an argument to a function call, which will execute the block immediately or some time later. To some extent, a closure is similar to an unnamed (or anonymous) function. There is more to it, but for the time being, let us stick to this definition.

To better understand the purpose of closures, consider a simple example of a situation where you would want to pass a block of code to another function for repeated execution: a method forEach that takes a sequence of objects is supposed to apply a given functionality to every object in the sequence. How would you pass the functionality to the forEach method? Today in Java, you would have to define an interface, say Block, and pass implementations of that interface to the forEach method, as shown in Listing 1.

#### Listing 1. Java programming without closures

public interface Block<T> {

void invoke(T arg);

}

public class Utils {

public static <T> void forEach(Iterable<T> seq, Block<T> fct) {

for (T elm : seq)

fct.invoke(elm);

}

}

public class Test {

public static void main(String[] args) {

List<Integer> nums = Arrays.asList(1,2,3);

Block<Integer> print = new Block<Integer>() {

public void invoke(Integer arg) {

System.out.println(arg);

}

};

Utils.forEach(nums,print);

}

}

The example is admittedly contrived and extremely simple, but consider it: how often in Java do you find yourself implementing an interface and passing the implementation to a method for execution? We can think of three immediate examples:

* Runnable and Callable, which we pass to threads or thread pools for asynchronous execution.
* Callback interfaces such as ActionListener, which we register for future execution in case a certain event occurs.
* Comparator, which we pass to a TreeMap for maintaining its sorting order.

In all these cases we use an interface, providing some functionality as an implementation of the interface. We then pass the functionality to a method for immediate or delayed, synchronous or asynchronous execution. Closures would simplify this process by allowing a more concise syntax, thereby eliminating some of Java's verbosity. Beyond allowing more concise and readable Java source code, closures would add some functionality completely new to Java, such as custom control structures, which we explain later in the article.

## Closures vs inner classes

In a recent blog post, James Gosling (*aka* Mr. Java himself) discussed [the history of closures in Java](http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html#resources):

*Closures were left out of Java initially more because of time pressures than anything else. In the early days of Java the lack of closures was pretty painful, and so inner classes were born: an uncomfortable compromise that attempted to avoid a number of hard issues. But as is normal in so many design issues, the simplifications didn't really solve any problems, they just moved them.*

Gosling precisely describes our present situation: for eleven years, since the release of JDK 1.1, we've been using inner classes in place of closures. While they have served us well (you might recall how difficult and frustrating it was to implement callback interfaces without them) inner classes have never truly replaced closures. They simply are less powerful and more verbose than they need to be.

So, one thing all of the closures proposals agree to is the need to conceptually replace inner classes with a more powerful programming construct. Adding closures to the Java syntax would not mean losing inner classes; rather, the two constructs would be compatible and interchangeable. The advantage of closures, of course, is that they add flexibility and power not found in inner classes.

In the next section we'll look at the specifics of how each proposal would bring closures to Java 7.

## Closures: Three proposals compared

While the three closures proposals agree in their desire to bring the expressive power of closures to the Java syntax, they disagree about the manner of doing it. For one thing, the proposals differ -- sometimes greatly -- in how much additional power they would bring to the Java language. They also differ in terms of how closures are to be integrated into the existing language. One of the chief arguments against adding closures to the language is, of course, complexity. We'll look at how each of the proposals addresses this issue below, following a brief overview of each one's origin and basic intent.

|  |
| --- |
| **Counting by numbers** |
| To get a feeling for the attention this topic attracts, consider the following figures: with three proposals for closures in Java(not counting what some call the fourth proposal: no closures at all) [Alex Miller's Closures in Java 7 Web page](http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html#resources) lists nearly 150 articles, blog posts, or other resources related to this topic. No other feature in Miller's coverage of proposed additions to Java 7 has received so much attention; most features have 10 entries at best. Even the runner-up, the fairly controversial Java Module System, has just 60 entries to date. |

### BGGA

Issued in August 2006 the [BGGA proposal](http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html#resources) started the "closures in Java" discussion. The acronym *BGGA* comes from the first letters of the last names of its four authors: Gilad Bracha, Neal Gafter, James Gosling, and Peter von der Ahé. Of that group, [Neal Gafter](http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html#resources) has been the most vocal advocate for BGGA, which is undeniably the most ambitious of the three proposals. BGGA adds a lot of power to the Java language, but also a lot of complexity.

### CICE plus ARM

[CICE](http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html#resources) stands for "Concise Instance Creation Expression." Developed by Bob Lee, Doug Lea, and Joshua Bloch, CICE has been strongly championed by Bloch. The CICE proposal is considerably less complex than BGGA and advocates an approach that is best described by its tagline: *closures without complexity*. In CICE, closures are essentially an alternative syntax for inner classes. CICE closures stick close to the existing Java language and do not add much new functionality.

One use case has especially shown the limits of CICE, and that is "resource management with finally," where BGGA outperforms CICE with its functional power. To compensate for this shortcoming Joshua Bloch has proposed an object-oriented resource management approach (an addition to the existing, more functional approach with finally). He calls this proposal [ARM](http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html#resources) (Automatic Resource Management). When most people talk about CICE today, ARM is implicitly included. Mark Mahieu has developed a [prototype implementation for CICE plus ARM](http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html#resources).

### FCM plus JCA

[FCM](http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html#resources), which stands for First Class Methods is the closures proposal developed by Stephen Colebourne and Stefan Schulz. With respect to power and complexity this approach is somewhere between CICE and BGGA, and is much closer to BGGA. It mainly differs from the other two proposals in how it introduces closures into the Java language: not as an isolated novel concept, but embedded into an approach that allows handling of types and references of methods as first-class elements in Java. In FCM, a closure is just a special kind of method: the inner method.

BGGA also outperforms FCM by allowing for the implementation of custom control abstractions, which are static helper methods that receive a closure as a parameter. A separate proposal for [Java control abstractions (JCA)](http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html#resources) addresses the need for custom control abstractions in FCM.

Next, we consider the specific benefits of closures and see how they're handled by the three proposals.

## The closures proposals in practice

One common goal of all the closures proposals is to simplify the process of passing a piece of functionality to a method. Let us see how the various proposals would achieve this goal.

### CICE (Concise Instance Creation Expression)

CICE only introduces minor changes to the way that we currently pass functionality to methods in Java programs. We still need the Block interface, which we use as the type of the functionality when we declare the forEach method. Hence, the forEach method does not change at all. The invocation syntax is more concise and convenient, however, as shown in Listing 2.

#### Listing 2. Declaration and invocation of method forEach using CICE

public interface Block<T> {

void invoke(T arg);

}

public class Utils {

public static <T> void forEach(Iterable<T> seq, Block<T> fct) {

for (T elm : seq)

fct.invoke(elm);

}

}

public class Test {

public static void main(String[] args) {

List<Integer> nums = Arrays.asList(1,2,3);

Block<Integer> print

= **Block<Integer>(Integer arg) { System.out.println(arg); };**

Utils.forEach(nums,print);

}

}

Instead of creating an anonymous class of type Block<Integer> we create a *closure*. In CICE the closure is called a *concise instance creation expression*, hence the name of the proposal. Essentially, the proposal boils down to a more convenient syntax, because we can replace the syntax for creating an anonymous class with the more concise syntax of the closure. The closure syntax is available for all interfaces with exactly one method, such as Runnable, Callable, or ActionListener. CICE meets the goal of simplifying the use of these interfaces.

### BGGA

This proposal introduces a new kind of type so far unknown in Java, namely a *function type*. A function type denotes the type of a closure. With the function type, we no longer need the Block interface in our example. Instead we use the function type {T => void}, which denotes a function with an argument of type T that returns void.

We use this function type when we declare the forEach method's second argument, as shown in Listing 3.

#### Listing 3. Declaration of method forEach with BGGA's function type

class Utils {

public static <T> void forEach(Iterable<T> seq, **{T => void}** fct) {

for (T elm : seq)

fct.invoke(elm);

}

}

All closures implicitly have an invoke method so that we need not change anything else in the forEach method. Basically, you can assume that the compiler translates the function type to a synthetic interface type that looks like our Block interface.

We use another quite similar function type, {Integer => void}, when we create the actual closure that we pass to the call of the forEach method, as shown in Listing 4.

#### Listing 4. Invoking forEach using a BGGA closure

class Test {

public static void main(String[] args) {

List<Integer> nums = Arrays.asList(1,2,3);

**{Integer => void }** print

= **{ Integer arg => System.out.println(arg); }**;

Utils.forEaxh(nums,print);

}

}

The variable print is a function reference of a function type and refers to a closure. It is initialized with a closure that in our example takes an Integer and prints it. The closure is then passed to the forEach method and applied to each element in the sequence.

### FCM

The First Class Methods proposal goes one step further and does not only add function types (here called *method types*) that denote the type of a closure, but the type of a method in general. Closures in this proposal are just a special case of a method, namely an *anonymous inner method*, which is method without a name. Naturally, the syntax is different; our method type looks like #(void(T)) in this proposal and the closure is denoted as #(Integer arg) { System.out.println(arg); }:

#### Listing 5. Declaration and invocation of method forEach using FCM

class Utils {

public static <T> void forEach(Iterable<T> seq, **#(void(T))** fct) {

for (T elm : seq)

fct.invoke(elm);

}

}

class Test {

public static void main(String[] args) {

List<Integer> nums = Arrays.asList(1,2,3);

**#(void(Integer))** print

= **#(Integer arg) { System.out.println(arg); }**;

Utils.forEach(nums,print);

}

}

As you can tell from the examples, the proposals use different syntax to achieve their goal, in this case passing functionality to a method, but the underlying idea is similar. Next we'll look at how each proposal approaches closure conversion and type compatibility.

## Closure conversion

All three proposals allow a certain amount of type compatibility among closures and interfaces types: a closure can be assigned to a variable of a compatible interface type. This is an important feature because it allows use of closures in all places where inner classes are currently used in the Java language. This type compatibility ensures that closures are backward compatible to interfaces and inner classes.

The idea backing this type compatibility is the observation that anonymous inner classes are typically used in situations where an implementation of an interface with just one method is needed. Common examples are callback interfaces such as ActionListener with its actionPerformed method or Runnable with its run method.

For the purpose of closure conversion, an interface type is considered compatible to a closure if it has a single method with a compatible return type and compatible argument types. (Actually this is a slight oversimplification: we are ignoring closures that throw exceptions, in which case the throws clauses must also be compatible.) The compatible interface may have additional methods. As long as the additional methods are inherited from class Object the interface is still considered an interface with a single method. The Comparator interface is an example; it specifies a compare method and an equals method. Because every class inherits a default implementation of equals from Object, the Comparator interface is treated like an interface with a single compare method regarding compatibility to closures.

### An example of closure conversion

Consider an example of closure conversion using the Runnable interface, where we assign a closure to a regular reference variable of the interface type Runnable. Currently in the Java language we would define an anonymous inner class in order to provide an on-the-fly implementation of Runnable:

#### Listing 6. Old-style notation using an anonymous inner class

Runnable r = **new Runnable() { public void run() {** System.out.println("Hello World."); } **};**

new Thread(r).start();

In CICE, we basically do the same, but with a more concise syntax. For this reason, it is not surprising that a CICE closure is compatible with an interface variable.

#### Listing 7. CICE notation

Runnable r = **Runnable() {** System.out.println("Hello World."); **};**

new Thread(r).start();

In BGGA and FCM, the compatibility between a closure and an interface variable is not obvious and requires the implicit closure conversion.

#### Listing 8. BGGA notation

Runnable r = **{ =>** System.out.println("Hello World."); **};**

new Thread(r).start();

#### Listing 9. FCM notation

Runnable r = **# {** System.out.println("Hello World."); **};**

new Thread(r).start();

As you can see, all three proposals permit a substantially more compact notation than old-style Java and the concise notation is considered one of the key reasons for adding closures to the language. In fact, all three allow an even more convenient notation where the closure is defined in-place. This would allow us to eliminate the reference variable entirely and pass the closure literal in lieu of a Runnable variable to the thread constructor, as shown here in BGGA notation:

new Thread({ => System.out.println("Hello World."); }).start();

This is the concise and convenient notation that all three closure proposals strive for and achieve using slightly different rules of syntax.

## Non-local control statements

Naturally, the simple examples above only scratch the surface of what closures are about. Covering all of the issues related to each proposal is beyond the scope of this article. Moreover, in most aspects the proposals are relatively similar in what they propose -- but there is one area where the proposals differ drastically.

In the BGGA proposal it is permissible for a closure to contain *non-local control statements* that affect the control flow of the context in which the closure is defined, not only the control flow within the closure itself. Listing 10 shows an example of a non-local return statement. We use our earlier example, namely the forEach method that applies a closure's functionality to all elements in a sequence. The closure prints the argument it receives and contains a (non-local) return statement that terminates the main method as soon as the argument equals *3*.

#### Listing 10. A non-local return statement in action

class Utils {

public static <T> void for<each(Iterable<T> seq, {T => void } fct) {

for (T elm : seq)

fct.invoke(elm);

}

}

class Test {

public static void main(String[] args) {

List<Integer> nums = Arrays.asList(1,2,3,4,5);

{Integer => void } print = { Integer arg ==> if (arg == 3) return; System.out.println(arg); };

Utils.forEach(nums,print);

}

}

The return statement in the closure does not only return from the closure, but also terminates the main method where the closure was defined. The effect is that with this closure our sample program terminates as soon as a sequence element equal to 3 is encountered; that is, it would print 1 2 and then stop.

The introduction of non-local control statements distinguishes BGGA from the CICE and FCM proposals. In both CICE and FCM a return statement simply means "return from the closure." The effect of corresponding closures in CICE and FCM would be that the closure returns as soon as a sequence element equal to 3 was encountered, but the forEach method as well as the main method would keep going -- so the program would print 1 2 4 5.

BGGA requires non-local return statements because the proposal aims to enable us to factor out common control structures by means of closures -- something we'll demonstrate further below. The idea is that a closure behaves like a block of code that relates to the scope in which it is defined. That is, the closure can both access variables of the enclosing scope and affect the control flow of the enclosing scope, for instance by saying: "return from the enclosing method."

### Lexical binding

*Lexical binding* is the principle behind features like the non-local return statement. BGGA closures apply lexical binding to the following:

* Variables from enclosing scopes
* The meaning of this
* The meaning of break, continue, and return

Lexical binding of variables in a closure means that the closure can access variables from the enclosing scope. That's not a new idea in Java. Local and anonymous classes have a similar feature: they can access final variables of the enclosing scope. In a closure it is similar; just the restriction that the variable must be final goes away. Instead, you may qualify the variable with a @Shared annotation (at least in the BGGA proposal; the other two proposals allow access without final or any other qualification).

Lexical binding of the this keyword is new in Java, however. In anonymous inner classes, any use of the keyword this refers to the instance of the inner class, not to an instance of the enclosing class. The enclosing class's this is referred to using different syntax, namely EnclosingType.this. CICE will stick to this tradition, BGGA and FCM will break with it. In BGGA and FCM, the keyword this will be lexically bound and will automatically refer to an instance of the enclosing type in which the closure is defined.

The fundamental difference between the proposals comes with lexical binding of control statements. In BGGA, not only the return statement but also the control statements break and continue are lexically bound. Only BGGA proposes this feature; CICE and FCM do not have it.

The BGGA proposal needs the lexical binding of control statements in order to ease the factoring out of common code. In contrast, the emphasis of the FCM proposal in not primarily on refactoring, but more on enabling a functional programming style where functions and functionality (in the form of closures) are well supported. As a result, the FCM proposal has lexical binding for variables and this, but not for break, continue, and return -- and for a good reason: non-local return statements have their pitfalls, as we will see shortly.

### Returning a value from a BGGA closure

You might wonder how the compiler distinguishes between a local and a non-local return statement in a BGGA closure. The answer is that there is no local return statement in BGGA. A return statement always returns from the nearest enclosing method or constructor in which the closure is called. If a closure needs to return a value it does so without an explicit return statement. The last expression in a closure's body always serves as an implicit local return statement. For illustration, here is a BGGA closure that returns a value:

#### Listing 11. Returning a value from a BGGA closure

{double => double} f = {double x =>

double res = Math.log(x);

System.out.println("log of "+x+" is "+res);

res **// implicit local return statement**

};

The lexical binding of the control statements break, continue, and return adds quite a bit of complexity to the BGGA proposal. We won't mention all of the complications, but just consider the non-local return statement. What if in our example the closure is not executed in a function that is invoked from main, but in a function that is executed in a different thread? In that situation "return from main" does not make any sense, because the other thread's call stack does not have a main method.

The BGGA proposal addresses this situation by raising an unchecked exception at runtime. In addition, something must be done to catch this kind of problem at compile time. BGGA would add a marker interface or some kind of syntax distinguishing *restricted* from *unrestricted* closures, where restricted closures are those that are not allowed to be executed in a different thread. We won't go into further details, but suffice it to say that the fundamental problem remains: lexical binding of control statements can be error prone.

## Control abstractions with closures

We've mentioned that the BGGA proposal includes lexical binding because the feature is needed for a fine-grained factoring out of common functionality. In fact, closures in BGGA are supposed to enable user-defined control structures, which ideally should be as easy to use as language features. The code to be abstracted in a refactoring might contain a break, continue, or return statement, and such control statements are not supposed to prevent refactoring. For this purpose, BGGA needs lexical binding of control statements.

In fact, non-local control statements enable interesting and powerful closures. Here is a prototypical example of a refactoring that BGGA aims to enable: the use of explicit locks such as java.util.concurrent.Lock. Consider a thread-safe stack abstraction that holds integers. It needs synchronization for its push and pop method and uses explicit locks for it:

#### Listing 12. Explicit locks in Java

public class Stack {

private static final int SIZE = 256;

private int[] arr = new int[SIZE];

private int cnt = 0;

private Lock lock = new ReentrantLock();

public void push(int elm) {

**lock.lock();**

**try {**

arr[cnt++] = elm;

**} finally {**

**lock.unlock();**

**}**

}

public int pop() {

**lock.lock();**

**try {**

return arr[--cnt];

**} finally {**

**lock.unlock();**

**}**

}

}

Obviously, the control structure that acquires and releases the lock is repeated in both the push and the pop method and we might want to factor it out. For this purpose we define a control abstraction withLock, which is a method that takes a closure that represents the critical region.

#### Listing 13. Definition of control abstraction withLock using BGGA

public class Utils {

public static <T> T withLock(Lock lock, **{ => T}** block) {

lock.lock();

try {

return block.invoke();

} finally {

lock.unlock();

}

}

}

Using the withLock method we can refactor the push and the pop method as follows:

#### Listing 14. Refactoring push and pop using withLock

public void push(int elm) {

**Utils.withLock(lock, { =>**

arr[cnt++] = elm;

**});**

}

public int pop() {

**Utils.withLock(lock, { =>**

**return** arr[--cnt];

**});**

}

Regarding readability the BGGA proposal goes even one step further and defines a special *control invocation syntax* that renders it even more readable. To make it perfect we add a static import statement for the withLock method and -- voilá-- the withLock control abstraction looks as pretty as the Java-language-defined synchronized keyword:

#### Listing 15. Control invocation syntax in BGGA

import static Utils.withLock;

public void pushY(int elm) {

**withLock(lock) {**

arr[cnt++] = elm;

**}**

}

public int popY() {

**withLock(lock) {**

**return** arr[--cnt];

**}**

}

Putting syntactic sugar aside (and getting back to the point of lexical binding and non-local return statements) it's evident that we truly need the lexical binding of the return statement here. The closure's return statement is supposed to return from the pop method, not just from the closure.

### Alternate proposals for control abstraction

The withLock control abstraction is a striking example of where the BGGA proposal is headed: it is a key goal of BGGA to allow for reusable abstractions such as withLock that can be used to refactor existing programs and simplify future Java code. Neither CICE nor FCM have such an ambitious goal; instead both strive to simplify specific pieces of the Java syntax.

This said, both CICE and FCM have come up with proposals that address control structures. The [Java Control Abstractions proposal](http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html#resources) suggests an addition to FCM that enables control structure in a way that is similar to what BGGA proposes. The main difference is that JCA deliberately distinguishes between regular use of closures and control constructs that use closures. Writing a control abstraction method such as withLock is more complex than writing a regular method with closures such as forEach. This is because the control abstraction method must take into account how return, continue, and break and exceptions are handled.

Interestingly, BGGA is heading in a similar direction. The latest ideas regarding [restricted vs. unrestricted closures](http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html#resources) also introduce a distinction between closure in control abstraction methods and regular use of closures. Eventually, we might see the two proposals converge.

Meanwhile, [Automatic Resource Management Blocks](http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html#resources) is an addition to CICE that takes an entirely different approach to control abstractions. The observation is that much of the need for control abstractions circles around try-finally constructs and that there is basically a need for automatic resource release. For this reason, the proposal suggests language support for automatic resource disposal at the end of a block and an interface that would be used to make a class eligible for this kind of automatic disposal.

## In conclusion

From a technical viewpoint the three closure proposals along with their respective companion proposals cover similar ground and strive for two main goals:

1. Simplify the syntax for implementing and using certain simple interfaces (with just one method), such as Runnable, ActionListener, and Comparator.
2. Allow common control structures to be refactored into shared utilities by means of closures; that is, enable user-defined control structures that are as easy to use as language features.

All three proposals clearly address the first goal, each in a slightly different way. Strictly speaking, the second goal is met only by BGGA and FCM+JCA. The CICE proposal takes an entirely different approach that does not aim to support custom-control structures, but suggests embedding into the language better support for resource management. Here the goals simply diverge.

In summary: CICE+ARM suggests a simple, easy-to-use feature with minimal modifications of the language; FCM+JCA is more ambitious; and BGGA proposes to add considerably more expressive power and flexibility to the Java language. The flipside of BGGA is quite a bit of complexity and fairly extensive changes to the language.

## The closures debate in essence

Given the intensity with which this topic is occasionally discussed, you might get the feeling that closures are a matter of life or death for Java or your Java career. This argument is used by both people who want closures in Java and those who oppose them -- or who might be willing to settle for the comparably simplistic CICE proposal.

An interesting Java.net [opinion poll](http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html#resources) has been collecting "votes" for the three closures proposals for some time. In the poll, CICE and "no closures" are listed as two separate options, although their supporters want essentially the same thing: to keep the Java language simple. Combined, the "no closures" and CICE options have roughly 42% of the vote, which is roughly equal to the share of BGGA. FCM has nearly 15% of the votes, and less than 1% of voters have voted for "something else."

Those who want closures in Java argue that if Java rejects closures, programmers will [turn to other languages](http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html#resources) that support them well. (Even worse: it could be a language from the Microsoft .NET family.) Those who object to closures in Java argue that their inclusion would complicate the language beyond normal usability: mainstream programmers will turn away from Java, so the argument goes, and move on to simpler programming languages. In essence, those against closures are afraid that Java will turn into a guru language [used only by experts in niche areas](http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html#resources).

The "too complicated" argument is often heard in combination with a reference to generics, which many people also believe are too complex for what they achieve. The sentiment is that the evolution of Java toward ever increasing complexity must stop. As Joshua Bloch put it, we have "used up our complexity budget on generics, and in particular, on wildcards."

Basically, there are two major opposing camps: one in favor of simplicity of use and minimal modification of the language, and the other in favor of increased expressive power at the expense of additional complexity. Which position is preferable is in the eye of the beholder. If we may risk a gaze into our crystal ball we would predict that one side-effect of the heated discussion is that we will not see closures in Java 7. (At the time of this writing, there is not yet a votable JSR for Java SE 7.)

While it may seem to some that the closures debate has already drawn on for long enough, the intensity of the debate, and the seemingly even support for BGGA and CICE/no closures (according to one poll) suggest that the delay is not unreasonable. At the least it will give us all plenty of time for further discussions.

### About the author

Klaus Kreft has 20+ year of experience in the software business and currently works as a software engineer and system architect at SEN (Siemens Enterprise Communications GmbH & Co. KG).

Angelika Langer works worldwide as a freelance instructor, coach, and author with an independent [curriculum](http://www.angelikalanger.com/Courses.html) of C++ and Java courses. Her areas of expertise include advanced C++ and Java programming, concurrent programming, and performance issues. She is the author of the [Java Generics FAQ](http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html), an online resource covering new language features in Java 5.0, a [Java champion](https://java-champions.dev.java.net/content/corechampions.html), and a regular speaker at [Java conferences](http://www.angelikalanger.com/Conferences.html).

<http://www.javaworld.com/javaworld/jw-06-2008/jw-06-closures.html>

# [Neal Gafter's blog](http://gafter.blogspot.com/) Thoughts about the future of the Java Programming Language.

## Friday, August 18, 2006

### [Closures for Java](http://gafter.blogspot.com/2006/08/closures-for-java.html)

*I'm co-author of a draft proposal for adding support for closures to the Java programming language for the Dolphin (JDK 7) release. It was carefully designed to interoperate with the current idiom of one-method interfaces. An abbreviated version of the original proposal is reproduced below. The latest version of the proposal and a prototype can be found at* [*http://www.javac.info/*](http://www.javac.info/)*.*

Gilad Bracha, Neal Gafter, James Gosling, Peter von der Ahé

Modern programming languages provide a mixture of primitives for composing programs. C#, Javascript, Ruby, Scala, and Smalltalk (to name just a few) have direct language support for function types and inline function-valued expression, called closures. A proposal for closures is working its way through the C++ standards committees as well. Function types provide a natural way to express some kinds of abstraction that are currently quite awkward to express in Java. For programming in the small, closures allow one to abstract an algorithm over a piece of code; that is, they allow one to more easily extract the common parts of two almost-identical pieces of code. For programming in the large, closures support APIs that express an algorithm abstracted over some computational aspect of the algorithm. We propose to add function types and closures to Java. We anticipate that the additional expressiveness of the language will simplify the use of existing APIs and enable new kinds of APIs that are currently too awkward to express using the best current idiom: interfaces and anonymous classes.

## Function Types

We introduce a new syntactic form:

Type

Type ( TypeList ) { throws ThrowsTypeList }

ThrowsTypeList

Type

ThrowsTypeList

ThrowsTypeList VBAR Type

VBAR

|

These syntactic forms designate function types. A function type is a kind of reference type. A function type consists of a return type, a list of argument types, and a set of thrown exception types.

Note: the existing syntax for the throws clause in a method declaration uses a comma to separate elements of the ThrowsTypeList. For backward compatibility we continue to allow commas to separate these elements in method and function declarations, but in function types we require the use of the '|' (vertical-bar) character as aseparator to resolve a true ambiguity that would arise when a function type is used in a type list. For uniformity of syntax, we also allow the vertical-bar as a separator in the throws clause of method and function definitions, and as a matter of style we recommend that new code prefer the vertical-bar.

## Local Functions

In addition to function types, we introduce local functions, which are one way to introduce a name with function type:

BlockStatement

LocalFunctionDeclarationStatement

LocalFunctionDeclarationStatement

Type Identifier FormalParameters { throws ThrowsTypeList } Block

A local function declaration has the effect of declaring a final variable of function type. Local functions may not be declared with a variable argument list. Local functions may invoke themselves recursively.

Note: this syntax omits annotations, which should be allowed on local functions.

### Example

Combining these forms, we can write a simple function and assign it to a local function variable:

public static void main(String[] args) {

int plus2(int x) { return x+2; }

int(int) plus2b = plus2;

System.out.println(plus2b(2));

}

## Namespaces and name lookup

The Java programming language currently maintains a strict separation between expression names and method names. These two separate namespaces allow a programmer to use the same name for a variable and for methods. Local functions and closure variables necessarily blur the distinction between these two namespaces: local functions may be used as expression values; contrariwise, variables of function type may be invoked.

A local function declaration introduces the declared name as a variable name. When searching a scope for a method name, if no methods exist with the given name then local functions and variables of the given name that are of function type are considered candidates. If more than one exists (for example, function-typed variable declarations are inherited from separate supertypes), the reference is considered ambiguous; local functions do not overload.

When searching a scope for an expression name, local functions are treated as variables. Function names and values can therefore be used like other values in a program, and can be applied using the existing invocation syntax. In addition, we allow a function to be invoked from an arbitrary (for example, parenthesized) expression:

Primary

Primary Arguments

## Anonymous Functions (Closures)

We also introduce a syntactic form for constructing a function value without declaring a local function (precedence is tentative):

Expression3

Closure

Closure

FormalParameters Block

Closure

FormalParameters : Expression3

### Example

We can rewrite the assignment to plus2b in the previous example using an anonymous function:

int(int) plus2b = (int x) {return x+2; };

Or, using the short form,

int(int) plus2b = (int x) : x+2;

## The type of a closure

The type of a closure is inferred from its form as follows:

The argument types of a closure are the types of the declared arguments.

For the short form of a closure, the return type is the type of the expression following the colon. For a long form of a closure, if the body contains no return statement and the body cannot complete normally, the return type is the type of null. Otherwise if the body contains no return statement or the form of return statements are without a value, the return type is void

Otherwise, consider the set of types appearing in the return statements within the body. These types are combined from left to right using the rules of the conditional operator (JLS3 15.25) to compute a single unique type, which is the return type of the closure.

The set of thrown types of a closure are those checked exception types thrown by the body.

### Example

The following illustrates a closure being assigned to a variable with precisely the type of the closure.

void(int) throws InterruptedException closure =

(int t) { Thread.sleep(t); }

## Subtyping

A function type T is a subtype of function type U iff all of the following hold:

* Either
  + The return type of T is either the same as the return type of U or
  + Both return types are reference types and the return type of T is a subtype of the return type of U, or
  + the return type of U is void.
* T and U have the same number of declared arguments.
* For each corresponding argument position in the argument list of T and U, either both argument types are the same or both are reference types and the type of the argument to U is a subtype of the corresponding argument to T.
* Every exception type in the throws of T is a subtype of some exception type in the throws of U.

## Exception handling

The invocation of a function throws every checked exception type in the function's type.

It is a compile-time error if the body of a function can throw a checked exception type that is not a subtype of some member of the throws clause of the function.

## Reflection

A function type inherits all the non-private methods from Object. The following methods are added to java.lang.Class to support function types:

public final class java.lang.Class<T> ... {

public boolean isFunction();

public java.lang.reflect.FunctionType functionType();

public Object invokeFunction(Object function, Object ... args)

throws IllegalArgumentException | InvocationTargetException;

}

public interface java.lang.reflect.FunctionType {

public Type returnType();

public Type[] argumentTypes();

public Type[] throwsTypes();

}

Note: unlike java.lang.reflect.Method.invoke, Class.invokeFunction cannot throw IllegalAccessException, because there is no access control to enforce; the function value designates either an anonymous or local function, neither of which allows access modifiers in its declaration. Access to function values is controlled at compile-time by their scope, and at runtime by controlling the function value.

## The type of null

We add support for null and the type of null in function types. We introduce a meaning for the keyword null as a type name; it designates the type of the expression null. A class literal for the type of null is null.class. These are necessary to allow reflection, type inference, and closure literals to work for functions that do not return normally. We also add the non-instantiable class java.lang.Null as a placeholder, and its static member field TYPE as a synonym for null.class.

## Referencing names from the enclosing scope

Names that are in scope where a function or closure is defined may be referenced within the closure. We do not propose a rule that requires referenced variables be final, as is currently required for anonymous class instances. The constraint on anonymous class instances is also relaxed to allow them to reference any local variable in scope.

Note: Some who see concurrency constructs being the closure construct's primary use prefer to either require such referenced variables be final, or that such variables be explicitly declared for sharing, perhaps by requiring them be declared volatile. We reject this proposal for a few reasons. First, concurrency has no special role in the need for closures in the Java programming language; the proposal punishes other users of the feature for the convenience of these few. Second, the proposal is non-parallel with the closest existing parallel structure: classes. There is no constraint that a method may only access, for example, volatile fields of itself or other objects or enclosing classes. If compatibility allowed us to add such a rule to Java at this time, such a rule would obviously inconvenience most programmers for very little benefit. Third, marking such variables volatile, with all the semantic meaning implied by volatile, is neither necessary nor sufficient to ensure (and hardly assists!) appropriate use in a multithreaded environment.

## Non-local transfer

One purpose for closures is to allow a programmer to refactor common code into a shared utility, with the difference between the use sites being abstracted into a local function or closure. The code to be abstracted sometimes contains a break, continue, or return statement. This need not be an obstacle to the transformation. A break or continue statement appearing within a closure or local function may transfer to any matching enclosing statement provided the target of the transfer is in the same innermost ClassBody.

Because the return statement within a block of code is given new meaning when transformed by being surrounded by a closure, a different syntactic construct is required to return from an enclosing function or method. The following new form of the return statement may be used within a closure or local function to return from any enclosing (named) local function or method, provided the target of the transfer is in the same innermost *ClassBody*:

NamedReturnStatement

return Identifier : ;

NamedReturnStatement

return Identifier : Expression ;

No syntax is provided to return from a lexically enclosing closure. If such non-local return is required, the code should be rewritten using a local function (i.e. introducing a name) in place of the closure.

If a break statement is executed that would transfer control out of a statement that is no longer executing, or is executing in another thread, the VM throws a new unchecked exception, UnmatchedNonlocalTransfer. (I suspect we can come up with a better name). Similarly, an UnmatchedNonlocalTransfer is thrown when a continue statement attempts to complete a loop iteration that is not executing in the current thread. Finally, an UnmatchedNonlocalTransfer is thrown when a NamedReturnStatement attempts to return from a function or method invocation that is not pending in the current thread.

## Closure conversion

We propose the following closure conversion, to be applied only in those contexts where boxing currently occurs:

There is a closure conversion from every closure of type T to every interface type that has a single method with signature U such that T is a subtype of the function type corresponding to U.

We will want to generalize this rule slightly to allow the conversion when boxing or unboxing of the return type is required, e.g. to allow assigning a closure that returns int to an interface whose method returns Integer or vice versa.

Note: The current Java idiom for capturing a snippet of code requires the use of a one-method interface to represent the function type and an anonymous class instance to represent the closure:

public interface Runnable {

void run();

}

public interface API {

void doRun(Runnable runnable);

}

public class Client {

void doit(API api) {

api.doRun(new Runnable(){

public void run() {

snippetOfCode();

}

});

}

}

Had function types been available when this API was written, it might have been written like this:

public interface API {

void doRun(void() func);

}

And the client like this:

public class Client {

void doit(API api) {

api.doRun(() {snippetOfCode(); });

}

}

Unfortunately, compatibility prevents us from changing existing APIs. One possibility is to introduce a boxing utility method somewhere in the libraries:

Runnable runnable(final void() func) {

return new Runnable() {

public void run() { func(); }

};

}

Allowing the client to write this:

public class Client {

void doit(API api) {

api.doRun(runnable(() {snippetOfCode(); }));

}

}

This may be nearly good enough from the point of view of how concise the usage is, but it has one more serious drawback: every creation of a Runnable this way requires that two objects be allocated instead of one (one for the closure and one for the Runnable), and every invocation of a method constructed this way requires an extra invocation. For some applications -- for example, micro-concurrency -- this overhead may be too high to allow the use of the closure syntax with existing APIs. Moreover, the VM-level optimizations required to generate adequate code for this kind of construct are difficult and unlikely to be widely implemented soon.

The closure conversion is applied only to closures (i.e. function literals), not to arbitrary expressions of function type. This enables javac to allocate only one object, rather than both a closure and an anonymous class instance. The conversion avoids any hidden overhead at runtime. As a practical matter, javac will automatically generate code equivalent to our original client, creating an anonymous class instance in which the body of the lone method corresponds to the body of the closure.

### Example

We can use the existing Executor framework to run a closure in the background:

void sayHello(java.util.concurrent.Executor ex) {

ex.execute((){ System.out.println("hello"); });

}

## Further ideas

We are considering allowing omission of the argument list in a closure when there are no arguments. Further, we could support a sugar for calls to functions whose last argument is a zero argument closure:

void foo(T1 p1, ..., Tn pn, R() pn+1) {...}

could be called as

T1 a1; ... Tn an;

foo(a1, ..., an){...};

where the call is translated to

foo(a1, ..., an, {...});

In the special case where there is only one argument to foo, we also would allow

foo{...}

for example

void sayHello(java.util.concurrent.Executor ex) {

ex.execute { System.out.println("hello"); }

}

We are also experimenting with generalizing this to support an invocation syntax that interleaves parts of the method name and its arguments, which would allow more general user-defined control structures that look like if, if-else, do-while, and so on.

This doesn't play well with the return statement being given a new meaning within a closure; it returns from the closure instead of the enclosing method/function. Perhaps the return from a closure should be given a different syntax:

^ expression;

With this, we probably no longer need the nonlocal return statement.

## Acknowledgments

The authors would like to thank the following people whose discussions and insight have helped us craft, refine, and improve this proposal:

Lars Bak, Cédric Beust, Joshua Bloch, Martin Buchholz, Danny Coward, Erik Ernst, Christian Plesner Hansen, Doug Lea, "crazy" Bob Lee, Martin Odersky, Tim Peierls, John Rose, Ken Russell, Mads Torgersen, Jan Vitek, and Dave Yost.

### 

### Closures

#### By jag on [Jan 31, 2008](http://blogs.oracle.com/jag/entry/closures)

There has been a lot of chatter about the [closures proposal](http://gafter.blogspot.com/2006/08/closures-for-java.html) penned by Neal Gafter. And, in particular, whether or not I support it. I absolutely do. My "Feel of Java" talk of many years ago got rather infamously twisted at JavaPolis a couple of months ago. Java feels alive, not stuck in some chisel marks on stone tablets. Closures were left out of Java initially more because of time pressures than anything else. Closures, as a concept, are tried and true - well past the days of being PhD topics. The arguments are in the details, not the broad concepts. In the early days of Java the lack of closures was pretty painful, and so inner classes were born: an uncomfortable compromise that attempted to avoid a number of hard issues. But as is normal in so many design issues, the simplifications didn't really solve any problems, they just moved them. We should have gone all the way back then. Some have criticized the current proposal as being too complex. If you read through all of what Neal has written, you'll see that there are two sources of this perception: the spec is really detailed and explores all kinds of corner cases that never get touched on in most programming manuals; and the proposal is a collection of features that at first blush seem separate, but in fact are deeply inter-related and push each other into existence.