Closures for Java

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*.

Selection expressions

TODO: Rules are needed to determine the result type of a selection (`?:`) expression when both the second and third operands are function types.

Type inference

TODO: Rules are needed for performing generic type inference involving function types.

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.

Naming conventions

TODO: recommended naming convensions for local functions.

Implementation notes

TODO: discussion of implementation strategy. VM interning of function types. Implementation of subtyping, assignment, and casts. Invocation using `invokedynamic`. Nonlocal control transfers.