State of the Lambda

September 2013

Java SE 8 Edition

This is an informal overview of the enhancements to the Java programming language specified by JSR 335 and implemented in the OpenJDK Lambda Project. It refines the previous iteration posted in December 2011. A formal description of some of the language changes may be found in the Early Draft Specification for the JSR; an OpenJDK Developer Preview is also available. Additional historical design documents can be found at the OpenJDK project page. There is also a companion document, State of the Lambda, Libraries Edition, describing the library enhancements added as part of JSR 335.

The high-level goal of Project Lambda is to enable programming patterns that require modeling code as data to be convenient and idiomatic in Java. The principal new language features include:

- Lambda expressions (informally, "closures" or "anonymous methods")
- Method and constructor references
- Expanded target typing and type inference
- Default and static methods in interfaces

These are described and illustrated below.

1. Background

Java is, primarily, an object-oriented programming language. In both object-oriented and functional languages, basic values can dynamically encapsulate program behavior: object-oriented languages have objects with methods, and functional languages have functions. This similarity may not be obvious, however, because Java objects tend to be relatively heavyweight: instantiations of separately-declared classes wrapping a handful of fields and many methods.

Yet it is common for some objects to essentially encode nothing more than a function. In a typical use case, a Java API defines an interface, sometimes described as a "callback interface," expecting the user to provide an instance of the interface when invoking the API. For example:

```
public interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```

Rather than declaring a class that implements **`ActionListener`** for the sole purpose of allocating it once at an invocation site, a user typically instantiates the implementing class inline, anonymously:

```
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    ui.dazzle(e.getModifiers());
  }
});
```

Many useful libraries rely on this pattern. It is particularly important for parallel APIs, in which the code to execute must be expressed independently of the thread in which it will run. The parallel-programming domain is of special interest, because as Moore's Law continues to give us more cores but not faster cores, serial APIs are limited to a shrinking fraction of available processing power.

Given the increasing relevance of callbacks and other functional-style idioms, it is important that modeling code as data in Java be as lightweight as possible. In this respect, anonymous inner classes are imperfect for a number of reasons, primarily:

1. Bulky syntax
2. Confusion surrounding the meaning of names and **`this`**
3. Inflexible class-loading and instance-creation semantics
4. Inability to capture non-final local variables
5. Inability to abstract over control flow

This project addresses many of these issues. It eliminates (1) and (2) by introducing new, much more concise expression forms with local scoping rules, sidesteps (3) by defining the semantics of the new expressions in a more flexible, optimization-friendly manner, and ameliorates (4) by allowing the compiler to infer finality (allowing capture of *effectively final* local variables).

However, it is *not* a goal of this project to address all the problems of inner classes. Neither arbitrary capture of mutable variables (4) nor nonlocal control flow (5) are within this project's scope (though such features may be revisited in a future iteration of the language.)

2. Functional interfaces

The anonymous inner class approach, despite its limitations, has the nice property of fitting very cleanly into Java's type system: a function value with an interface type. This is convenient for a number of reasons: interfaces are already an intrinsic part of the type system; they naturally have a runtime representation; and they carry with them informal contracts expressed by Javadoc comments, such as an assertion that an operation is commutative.

The interface **`ActionListener`**, used above, has just one method. Many common callback interfaces have this property, such as **`Runnable`** and **`Comparator`**. We'll give all interfaces that have just one method a name: *functional interfaces*. (These were previously called *SAM Types*, which stood for "Single Abstract Method".)

Nothing special needs to be done to declare an interface as functional; the compiler identifies it as such based on its structure. (This identification process is a little more than just counting method declarations; an interface might redundantly declare a method that is automatically provided by the class **`Object`**, such as **`toString()`**, or might declare static or default methods, none of which count against the one-method limit.) However, API authors may additionally capture the *design intent* that an interface be functional (as opposed to accidentally having only one method) with the **`@FunctionalInterface`** annotation, in which case the compiler will validate that the interface meets the structural requirements to be a functional interface.

An alternative (or complementary) approach to function types, suggested by some early proposals, would have been to introduce a new, *structural* function type, sometimes called *arrow types*. A type like "function from a **`String`** and an **`Object`** to an **`int`**" might be expressed as **`(String,Object)->int`**. This idea was considered and rejected, at least for now, due to several disadvantages:

- It would add complexity to the type system and further mix structural and nominal types (Java is almost entirely nominally typed).
- It would lead to a divergence of library styles -- some libraries would continue to use callback interfaces, while others would use structural function types.
- The syntax could be unwieldy, especially when checked exceptions were included.
- It is unlikely that there would be a runtime representation for each distinct function type, meaning developers would be further exposed to and limited by erasure. For example, it would not be possible (perhaps surprisingly) to overload methods **`m(T->U)`** and **`m(X->Y)`**.

So, we have instead followed the path of "use what you know" -- since existing libraries use functional interfaces extensively, we codify and leverage this pattern. This enables *existing* libraries to be used with lambda expressions.

To illustrate, here is a sampling of some of the functional interfaces already in Java SE 7 that are well-suited for being used with the new language features; the examples that follow illustrate the use of a few of them.

- **`java.lang.Runnable`**
- **`java.util.concurrent.Callable`**
- **`java.security.PrivilegedAction`**
- **`java.util.Comparator`**
- **`java.io.FileFilter`**
- **`java.beans.PropertyChangeListener`**

In addition, Java SE 8 adds a new package, **`java.util.function`**, which contains functional interfaces that are expected to be commonly used, such as:

- **`Predicate<T>`** -- a boolean-valued property of an object
- **`Consumer<T>`** -- an action to be performed on an object
- **`Function<T,R>`** -- a function transforming a T to a R
- **`Supplier<T>`** -- provide an instance of a T (such as a factory)
- **`UnaryOperator<T>`** -- a function from T to T
- **`BinaryOperator<T>`** -- a function from (T, T) to T

In addition to these basic "shapes", there are also primitive specializations such as **`IntSupplier`** or **`LongBinaryOperator`**. (Rather than provide the full complement of primitive specializations, we provide only specializations for **`int`**, **`long`**, and **`double`**; the other primitive types can be accomodated through conversions.) Similarly, there are some specializations for multiple arities, such as **`BiFunction<T,U,R>`**, which represents a function from **`(T,U)`** to **`R`**.

3. Lambda expressions

The biggest pain point for anonymous classes is bulkiness. They have what we might call a "vertical problem": the **`ActionListener`** instance from section 1 uses five lines of source code to encapsulate a single aspect of behavior.

Lambda expressions are anonymous methods, aimed at addressing the "vertical problem" by replacing the machinery of anonymous inner classes with a lighter-weight mechanism.

Here are some examples of lambda expressions:

```
(int x, int y) -> x + y

() -> 42

(String s) -> { System.out.println(s); }
```

The first expression takes two integer arguments, named **x** and **y**, and returns their sum. The second takes *no* arguments and returns the integer **42**. The third takes a string and prints it to the console, returning nothing.

The general syntax consists of an argument list, the arrow token **->**, and a body. The body can either be a single expression, or a statement block. In the expression form, the body is simply evaluated and returned. In the block form, the body is evaluated like a method body -- a **return** statement returns control to the caller of the anonymous method; **break** and **continue** are illegal at the top level, but are of course permitted within loops; and if the body produces a result, every control path must return something or throw an exception.

The syntax is optimized for the common case in which a lambda expression is quite small, as illustrated above. For example, the expression-body form eliminates the need for a **return** keyword, which could otherwise represent a substantial syntactic overhead relative to the size of the expression.

It is also expected that lambda expressions will frequently appear in nested contexts, such as the argument to a method invocation or the result of *another* lambda expression. To minimize noise in these cases, unnecessary delimiters are avoided. However, for situations in which it is useful to set the entire expression apart, it can be surrounded with parentheses, just like any other expression.

Here are some examples of lambda expressions appearing in statements:

```
FileFilter java = (File f) -> f.getName().endsWith(".java");

String user = doPrivileged(() -> System.getProperty("user.name"));

new Thread(() -> {
  connectToService();
  sendNotification();
}).start();
```

4. Target typing

Note that the name of a functional interface is *not* part of the lambda expression syntax. So what kind of object does a lambda expression represent? Its type is inferred from the surrounding context. For example, the following lambda expression is an **ActionListener**:

```
ActionListener l = (ActionEvent e) -> ui.dazzle(e.getModifiers());
```

An implication of this approach is that the same lambda expression can have different types in different contexts:

```
Callable<String> c = () -> "done";

PrivilegedAction<String> a = () -> "done";
```

In the first case, the lambda expression **() -> "done"** represents an instance of **Callable**. In the second case, the same expression represents an instance of **PrivilegedAction**.

The compiler is responsible for inferring the type of each lambda expression. It uses the type expected in the context in which the expression appears; this type is called the *target type*. A lambda expression can only appear in a context whose target type is a functional interface.

Of course, no lambda expression will be compatible with *every* possible target type. The compiler checks that the types used by the lambda expression are consistent with the target type's method signature. That is, a lambda expression can be assigned to a target type **T** if all of the following conditions hold:

- **T** is a functional interface type
- The lambda expression has the same number of parameters as **T**'s method, and those parameters' types are the same
- Each expression returned by the lambda body is compatible with **T**'s method's return type
- Each exception thrown by the lambda body is allowed by **T**'s method's **throws** clause

Since a functional interface target type already "knows" what types the lambda expression's formal parameters should have, it is often unnecessary to repeat them. The use of target typing enables the lambda parameters' types to be inferred:

```
Comparator<String> c = (s1, s2) -> s1.compareToIgnoreCase(s2);
```

Here, the compiler infers that the type of **s1** and **s2** is **String**. In addition, when there is just one parameter whose type is inferred (a very common case), the parentheses surrounding a single parameter name are optional:

```
FileFilter java = f -> f.getName().endsWith(".java");

button.addActionListener(e -> ui.dazzle(e.getModifiers()));
```

These enhancements further a desirable design goal: "Don't turn a vertical problem into a horizontal problem." We want the reader of the code to have to wade through as little syntax as possible before arriving at the "meat" of the lambda expression.

Lambda expressions are not the first Java expressions to have context-dependent types: generic method invocations and "diamond" constructor invocations, for example, are similarly type-checked based on an assignment's target type.

```
List<String> ls = Collections.emptyList();
List<Integer> li = Collections.emptyList();

Map<String,Integer> m1 = new HashMap<>();
Map<Integer,String> m2 = new HashMap<>();
```

5. Contexts for target typing

We stated earlier that lambda expressions can only appear in contexts that have target types. The following contexts have target types:

- Variable declarations
- Assignments
- Return statements
- Array initializers
- Method or constructor arguments
- Lambda expression bodies
- Conditional expressions (`?:`)
- Cast expressions

In the first three cases, the target type is simply the type being assigned to or returned.

```
Comparator<String> c;
c = (String s1, String s2) -> s1.compareToIgnoreCase(s2);

public Runnable toDoLater() {
  return () -> {
    System.out.println("later");
  };
}
```

Array initializer contexts are like assignments, except that the "variable" is an array component and its type is derived from the array's type.

```
filterFiles(new FileFilter[] {
               f -> f.exists(), f -> f.canRead(), f -> f.getName().startsWith("q")
            });
```

In the method argument case, things are more complicated: target type determination interacts with two other language features, *overload resolution* and *type argument inference*.

Overload resolution involves finding the best method declaration for a particular method invocation. Since different declarations have different signatures, this can impact the target type of a lambda expression used as an argument. The compiler will use what it knows about the lambda expression to make this choice. If a lambda expression is *explicitly typed* (specifies the types of its parameters), the compiler will know not only the parameter types but also the type of all return expressions in its body. If the lambda is *implicitly typed* (inferred parameter types), overload resolution will ignore the lambda body and only use the number of lambda parameters.

If the choice of a best method declaration is ambiguous, casts or explicit lambdas can provide additional type information for the compiler to disambiguate. If the return type targeted by a lambda expression depends on type argument inference, then the lambda body may provide information to the compiler to help infer the type arguments.

```
List<Person> ps = ...
String<String> names = ps.stream().map(p -> p.getName());
```

Here, `ps` is a `List<Person>`, so `ps.stream()` is a `Stream<Person>`. The `map()` method is generic in `R`, where the parameter of `map()` is a `Function<T,R>`, where `T` is the stream element type. (`T` is known to be `Person` at this point.) Once the overload is selected and the lambda's target type is known, we need to infer `R`; we do this by type-checking the lambda body, and discovering that its return type is `String`, and hence `R` is `String`, and therefore the `map()` expression has a type of `Stream<String>`. Most of the time, the compiler just figures this all out, but if it gets stuck, we can provide additional type information via an explicit lambda (give the argument `p` an explicit type), casting the lambda to an explicit target type such as `Function<Person,String>`, or providing an explicit type witness for the generic parameter `R` (`.<String>map(p -> p.getName())`).

Lambda expressions themselves provide target types for their bodies, in this case by deriving that type from the outer target type. This makes it convenient to write functions that return other functions:

```
Supplier<Runnable> c = () -> () -> { System.out.println("hi"); };
```

Similarly, conditional expressions can "pass down" a target type from the surrounding context:

```
Callable<Integer> c = flag ? (() -> 23) : (() -> 42);
```

Finally, cast expressions provide a mechanism to explicitly provide a lambda expression's type if none can be conveniently inferred from context:

```
// Illegal: Object o = () -> { System.out.println("hi"); };
Object o = (Runnable) () -> { System.out.println("hi"); };
```

Casts are also useful to help resolve ambiguity when a method declaration is overloaded with unrelated functional interface types.

The expanded role of target typing in the compiler is not limited to lambda expressions: generic method invocations and "diamond" constructor invocations can also take advantage of target types wherever they are available. The following declarations are illegal in Java SE 7 but valid in Java SE 8:

```
List<String> ls =
  Collections.checkedList(new ArrayList<>(), String.class);

Set<Integer> si = flag ? Collections.singleton(23)
                       : Collections.emptySet();
```

6. Lexical scoping

Determining the meaning of names (and **`this`**) in inner classes is significantly more difficult and error-prone than when classes are limited to the top level. Inherited members -- including methods of class **`Object`** -- can accidentally shadow outer declarations, and unqualified references to **`this`** always refer to the inner class itself.

Lambda expressions are much simpler: they do not inherit any names from a supertype, nor do they introduce a new level of scoping. Instead, they are lexically scoped, meaning names in the body are interpreted just as they are in the enclosing environment (with the addition of new names for the lambda expression's formal parameters). As a natural extension, the **`this`** keyword and references to its members have the same meaning as they would immediately outside the lambda expression.

To illustrate, the following program prints **`"Hello, world!"`** twice to the console:

```
public class Hello {
  Runnable r1 = () -> { System.out.println(this); }
  Runnable r2 = () -> { System.out.println(toString()); }

  public String toString() { return "Hello, world!"; }

  public static void main(String... args) {
    new Hello().r1.run();
    new Hello().r2.run();
  }
}
```

The equivalent using anonymous inner classes would instead, perhaps to the programmer's surprise, print something like **`Hello$1@5b89a773`** and **`Hello$2@537a7706`**.

Consistent with the lexical-scoping approach, and following the pattern set by other local parameterized constructs like **`for`** loops and **`catch`** clauses, the parameters of a lambda expression must not shadow any local variables in the enclosing context.

7. Variable capture

The compiler check for references to local variables of enclosing contexts in inner classes (*captured* variables) is quite restrictive in Java SE 7: an error occurs if the captured variable is not declared **`final`**. We relax this restriction -- for both lambda expressions and inner classes -- by also allowing the capture of *effectively final* local variables.

Informally, a local variable is effectively final if its initial value is never changed -- in other words, declaring it **`final`** would not cause a compilation failure.

```
Callable<String> helloCallable(String name) {
  String hello = "Hello";
  return () -> (hello + ", " + name);
}
```

References to **`this`** -- including implicit references through unqualified field references or method invocations -- are,

essentially, references to a `final` local variable. Lambda bodies that contain such references capture the appropriate instance of `this`. In other cases, no reference to `this` is retained by the object.

This has a beneficial implication for memory management: while inner class instances always hold a strong reference to their enclosing instance, lambdas that do not capture members from the enclosing instance do *not* hold a reference to it. This characteristic of inner class instances can often be a source of memory leaks.

While we relax the syntactic restrictions on captured values, we still prohibit capture of mutable local variables. The reason is that idioms like this:

```
int sum = 0;
list.forEach(e -> { sum += e.size(); }); // ERROR
```

are fundamentally serial; it is quite difficult to write lambda bodies like this that do not have race conditions. Unless we are willing to enforce -- preferably at compile time -- that such a function cannot escape its capturing thread, this feature may well cause more trouble than it solves. Lambda expressions close over *values*, not *variables*.

Another reason to not support capture of mutable variables is that there's a better way to address accumulation problems without mutation, and instead treat this problem as a *reduction*. The `java.util.stream` package provides both general and specialized (such as sum, min, and max) reductions on collections and other data structures. For example, instead of using `forEach` and mutation, we could do a reduction which is safe both sequentially or in parallel:

```
int sum = list.stream()
               .mapToInt(e -> e.size())
               .sum();
```

The `sum()` method is provided for convenience, but is equivalent to the more general form of reduction:

```
int sum = list.stream()
               .mapToInt(e -> e.size())
               .reduce(0, (x,y) -> x+y);
```

Reduction takes a base value (in case the input is empty) and an operator (here, addition), and computes the following expression:

```
0 + list[0] + list[1] + list[2] + ...
```

Reduction can be done with other operations as well, such as minimum, maximum, product, etc, and if the operator is associative, is easily and safely parallelized. So, rather than supporting an idiom that is fundamentally sequential and prone to data races (mutable accumulators), we instead choose to provide library support to express accumulations in a more parallelizable and less error-prone way.

8. Method references

Lambda expressions allow us to define an anonymous method and treat it as an instance of a functional interface. It is often desirable to do the same with an *existing* method.

Method references are expressions which have the same treatment as lambda expressions (i.e., they require a target type and encode functional interface instances), but instead of providing a method body, they refer an existing method by name.

For example, consider a `Person` class that can be sorted by name or by age.

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

    public int getAge() { return age; }
    public String getName() { return name; }
   ...
}

Person[] people = ...
Comparator<Person> byName = Comparator.comparing(p -> p.getName());
Arrays.sort(people, byName);
```

We can rewrite this to use a method reference to `Person.getName()` instead:

```
Comparator<Person> byName = Comparator.comparing(Person::getName);
```

Here, the expression `Person::getName` can be considered shorthand for a lambda expression which simply invokes the named method with its arguments, and returns the result. While the method reference may not (in this case) be any more syntactically compact, it is clearer -- the method that we want to call has a name, and so we can refer to it directly by name.

Because the functional interface method's parameter types act as arguments in an implicit method invocation, the referenced method signature is allowed to manipulate the parameters -- via widening, boxing, grouping as a variable-

arity array, etc. -- just like a method invocation.

```
Consumer<Integer> b1 = System::exit;   // void exit(int status)
Consumer<String[]> b2 = Arrays::sort;  // void sort(Object[] a)
Consumer<String> b3 = MyProgram::main; // void main(String... args)
Runnable r = MyProgram::main;          // void main(String... args)
```

9. Kinds of method references

There are several different kinds of method references, each with slightly different syntax:

- A static method (**`ClassName::methName`**)
- An instance method of a particular object (**`instanceRef::methName`**)
- A **`super`** method of a particular object (**`super::methName`**)
- An instance method of an arbitrary object of a particular type (**`ClassName::methName`**)
- A class constructor reference (**`ClassName::new`**)
- An array constructor reference (**`TypeName[]::new`**)

For a static method reference, the class to which the method belongs precedes the `::` delimiter, such as in **`Integer::sum`**.

For a reference to an instance method of a particular object, an expression evaluating to an object reference precedes the delimiter:

```
Set<String> knownNames = ...
Predicate<String> isKnown = knownNames::contains;
```

Here, the implicit lambda expression would capture the **`String`** object referred to by **`knownNames`**, and the body would invoke **`Set.contains`** using that object as the receiver.

The ability to reference the method of a specific object provides a convenient way to convert between different functional interface types:

```
Callable<Path> c = ...
PrivilegedAction<Path> a = c::call;
```

For a reference to an instance method of an arbitrary object, the type to which the method belongs precedes the delimiter, and the invocation's receiver is the first parameter of the functional interface method:

```
Function<String, String> upperfier = String::toUpperCase;
```

Here, the implicit lambda expression has one parameter, the string to be converted to upper case, which becomes the receiver of the invocation of the **`toUpperCase()`** method.

If the class of the instance method is generic, its type parameters can be provided before the `::` delimiter or, in most cases, inferred from the target type.

Note that the syntax for a static method reference might also be interpreted as a reference to an instance method of a class. The compiler determines which is intended by attempting to identify an applicable method of each kind (noting that the instance method has one less argument).

For all forms of method references, method type arguments are inferred as necessary, or they can be explicitly provided following the `::` delimiter.

Constructors can be referenced in much the same was as static methods by using the name **`new`**:

```
SocketImplFactory factory = MySocketImpl::new;
```

If a class has multiple constructors, the target type's method signature is used to select the best match in the same way that a constructor invocation is resolved.

For inner classes, no syntax supports explicitly providing an enclosing instance parameter at the site of the constructor reference.

If the class to instantiate is generic, type arguments can be provided after the class name, or they are inferred as for a "diamond" constructor invocation.

There is a special syntactic form of constructor references for arrays, which treats arrays as if they had a constructor that accepts an **`int`** parameter. For example:

```
IntFunction<int[]> arrayMaker = int[]::new;
int[] array = arrayMaker.apply(10);  // creates an int[10]
```

10. Default and static interface methods

Lambda expressions and method references add a lot of expressiveness to the Java language, but the key to really achieving our goal of making code-as-data patterns convenient and idiomatic is to complement these new features

with libraries tailored to take advantage of them.

Adding new functionality to existing libraries is somewhat difficult in Java SE 7. In particular, interfaces are essentially set in stone once they are published; unless one can update all possible implementations of an interface simultaneously, adding a new method to an interface can cause existing implementations to break. The purpose of *default methods* (previously referred to as *virtual extension methods* or *defender methods*) is to enable interfaces to be evolved in a compatible manner after their initial publication.

To illustrate, the standard collections API obviously ought to provide new lambda-friendly operations. For example, the **`removeAll`** method could be generalized to remove any of a collection's elements for which an arbitrary property held, where the property was expressed as an instance of a functional interface **`Predicate`**. But where would this new method be defined? We can't add an abstract method to the **`Collection`** interface -- many existing implementations wouldn't know about the change. We could make it a static method in the **`Collections`** utility class, but that would relegate these new operations to a sort of second-class status.

Default methods provide a more object-oriented way to add concrete behavior to an interface. These are a new kind of method: interface method can either be *abstract* or *default*. Default methods have an implementation that is inherited by classes that do not override it (see the next section for the details). Default methods in a functional interface don't count against its limit of one abstract method. For example, we could have (though did not) add a **`skip`** method to **`Iterator`**, as follows:

```
interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();

    default void skip(int i) {
        for (; i > 0 && hasNext(); i--) next();
    }
}
```

Given the above definition of **`Iterator`**, all classes that implement **`Iterator`** would inherit a **`skip`** method. From a client's perspective, **`skip`** is just another virtual method provided by the interface. Invoking **`skip`** on an instance of a subclass of **`Iterator`** that does not provide a body for **`skip`** has the effect of invoking the default implementation: calling **`hasNext`** and **`next`** up to a certain number of times. If a class wants to override **`skip`** with a better implementation -- by advancing a private cursor directly, for example, or incorporating an atomicity guarantee -- it is free to do so.

When one interface extends another, it can add a default to an inherited abstract method, provide a new default for an inherited default method, or reabstract a default method by redeclaring the method as abstract.

In addition to allowing code in interfaces in the form of default methods, Java SE 8 also introduces the ability to place *static* methods in interfaces as well. This allows helper methods that are specific to an interface to live with the interface, rather than in a side class (which is often named for the plural of the interface). For example, **`Comparator`** acquired static helper methods for making comparators, which takes a function that extracts a **`Comparable`** sort key and produces a **`Comparator`**:

```
public static <T, U extends Comparable<? super U>>
Comparator<T> comparing(Function<T, U> keyExtractor) {
    return (c1, c2) -> keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

11. Inheritance of default methods

Default methods are inherited just like other methods; in most cases, the behavior is just as one would expect. However, when a class's or interface's supertypes provide multiple methods with the same signature, the inheritance rules attempt to resolve the conflict. Two basic principles drive these rules:

- Class method declarations are preferred to interface defaults. This is true whether the class method is concrete or abstract. (Hence the **`default`** keyword: default methods are a fallback if the class hierarchy doesn't say anything.)
- Methods that are already overridden by other candidates are ignored. This circumstance can arise when supertypes share a common ancestor.

As an example of how the second rule comes into play, say the **`Collection`** and **`List`** interfaces provided different defaults for **`removeAll`**, and **`Queue`** inherits the default method from **`Collection`**; in the following **`implements`** clause, the **`List`** declaration would have priority over the **`Collection`** declaration inherited by **`Queue`**:

```
class LinkedList<E> implements List<E>, Queue<E> { ... }
```

In the event that two independently-defined defaults conflict, or a default method conflicts with an abstract method, it is a compilation error. In this case, the programmer must explicitly override the supertype methods. Often, this

amounts to picking the preferred default, and declaring a body that invokes the preferred default. An enhanced syntax for **super** supports the invocation of a particular superinterface's default implementation:

```
interface Robot implements Artist, Gun {
    default void draw() { Artist.super.draw(); }
}
```

The name preceding **super** must refer to a direct superinterface that defines or inherits a default for the invoked method. This form of method invocation is not restricted to simple disambiguation -- it can be used just like any other invocation, in both classes and interfaces.

In no case does the order in which interfaces are declared in an **inherits** or **extends** clause, or which interface was implemented "first" or "more recently", affect inheritance.

12. Putting it together

The language and library features for Project Lambda were designed to work together. To illustrate, we'll consider the task of sorting a list of people by last name.

Today we write:

```
List<Person> people = ...
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

This is a very verbose way to write "sort people by last name"!

With lambda expressions, we can make this expression more concise:

```
Collections.sort(people,
                 (Person x, Person y) -> x.getLastName().compareTo(y.getLastName()));
```

However, while more concise, it is not any more abstract; it still burdens the programmer with the need to do the actual comparison (which is even worse when the sort key is a primitive). Small changes to the libraries can help here, such the static **comparing** method added to **Comparator**:

```
Collections.sort(people, Comparator.comparing((Person p) -> p.getLastName()));
```

This can be shortened by allowing the compiler to infer the type of the lambda parameter, and importing the **comparing** method via a static import:

```
Collections.sort(people, comparing(p -> p.getLastName()));
```

The lambda in the above expression is simply a forwarder for the existing method **getLastName**. We can use method references to reuse the existing method in place of the lambda expression:

```
Collections.sort(people, comparing(Person::getLastName));
```

Finally, the use of an ancillary method like **Collections.sort** is undesirable for many reasons: it is more verbose; it can't be specialized for each data structure that implements **List**; and it undermines the value of the **List** interface since users can't easily discover the static **sort** method when inspecting the documentation for **List**.

Default methods provide a more object-oriented solution for this problem, where we've added a **sort()** method to **List**:

```
people.sort(comparing(Person::getLastName));
```

Which also reads much more like to the problem statement in the first place: sort the **people** list by last name.

If we add a default method **reversed()** to **Comparator**, which produces a **Comparator** that uses the same sort key but in reverse order, we can just as easily express a descending sort:

```
people.sort(comparing(Person::getLastName).reversed());
```

13. Summary

Java SE 8 adds a relatively small number of new language features -- lambda expressions, method references, default and static methods in interfaces, and more widespread use of type inference. Taken together, though, they enable programmers to express their intent more clearly and concisely with less boilerplate, and enable the development of more powerful, parallel-friendly libraries.