16. Object Reorientation: Generic Functions

Because the invention of Lisp predated the rise of object-oriented programming by a couple decades,[1] new Lispers are sometimes surprised to discover what a thoroughly object-oriented language Common Lisp is. Common Lisp's immediate predecessors were developed at a time when object orientation was an exciting new idea and there were many experiments with ways to incorporate the ideas of object orientation, especially as manifested in Smalltalk, into Lisp. As part of the Common Lisp standardization, a synthesis of several of these experiments emerged under the name Common Lisp Object System, or CLOS. The ANSI standard incorporated CLOS into the language, so it no longer really makes sense to speak of CLOS as a separate entity.

The features CLOS contributed to Common Lisp range from those that can hardly be avoided to relatively esoteric manifestations of Lisp's language-as-language-building-tool philosophy. Complete coverage of all these features is beyond the scope of this book, but in this chapter and the next I'll describe the bread-and-butter features and give an overview of Common Lisp's approach to objects.

You should note at the outset that Common Lisp's object system offers a fairly different embodiment of the principles of object orientation than many other languages. If you have a deep understanding of the fundamental ideas behind object orientation, you'll likely appreciate the particularly powerful and general way Common Lisp manifests those ideas. On the other hand, if your experience with object orientation has been largely with a single language, you may find Common Lisp's approach somewhat foreign; you should try to avoid assuming that there's only one way for a language to support object orientation.[2] If you have little object-oriented programming experience, you should have no trouble understanding the explanations here, though it may help to ignore the occasional comparisons to the way other languages do things.

Generic Functions and Classes

The fundamental idea of object orientation is that a powerful way to organize a program is to define data types and then associate operations with those data types. In particular, you want to be able to invoke an operation and have the exact behavior determined by the type of the object or objects on which the operation was invoked. The classic example used, seemingly by all introductions to object orientation, is an operation `draw` that can be applied to objects representing various geometric shapes. Different implementations of the `draw` operation can be provided for drawing circles, triangles, and squares, and a call to `draw` will actually result in drawing a circle, triangle, or square, depending on the type of the object to which the `draw` operation is applied. The different implementations of `draw` are defined separately, and new versions can be defined that draw other shapes without having to change the code of either the caller or any of the other `draw` implementations. This feature of object orientation goes by the fancy Greek name *polymorphism*, meaning "many forms," because a single conceptual operation, such as drawing an object, can take many different concrete forms.

Common Lisp, like most object-oriented languages today, is class-based; all objects are *instances* of a particular class.[3] The class of an object determines its representation--built-in classes such as **`NUMBER`** and **`STRING`** have opaque representations accessible only via the standard functions for manipulating those types, while instances of user-defined classes, as you'll see in the next chapter, consist of named parts called *slots*.

Classes are arranged in a hierarchy, a taxonomy for all objects. A class can be defined as a *subclass* of other classes, called its *superclasses*. A class *inherits* part of its definition from its superclasses and instances of a class are also considered instances of the superclasses. In Common Lisp, the hierarchy of classes has a single root, the class **`T`**, which is a direct or indirect superclass of every other class. Thus, every datum in Common Lisp is an instance of **`T`**.[4] Common Lisp also supports *multiple inheritance*--a single class can have multiple direct superclasses.

Outside the Lisp family, almost all object-oriented languages follow the basic pattern established by Simula of having behavior associated with classes through *methods* or *member functions* that belong to a particular class. In these languages, a method is invoked on a particular object, and the class of that object determines what code runs. This model of method invocation is called--after the Smalltalk terminology--*message passing*. Conceptually, method invocation in a message-passing system starts by sending a *message* containing the name of the method to run and any arguments to the object on which the method is being invoked. The object then uses its class to look up the method associated with the name in the message and runs it. Because each class can have its own method for a given name, the same message, sent to different objects, can invoke different methods.

Early Lisp object systems worked in a similar way, providing a special function `SEND` that could be used to send a message to a particular object. However, this wasn't entirely satisfactory, as it made method invocations different from normal function calls. Syntactically method invocations were written like this:

```
(send object 'foo)
```

rather than like this:

```
(foo object)
```

More significantly, because methods weren't functions, they couldn't be passed as arguments to higher-order functions such as **MAPCAR**; if one wanted to call a method on all the elements of a list with **MAPCAR**, one had to write this:

```
(mapcar #'(lambda (object) (send object 'foo)) objects)
```

rather than this:

```
(mapcar #'foo objects)
```

Eventually the folks working on Lisp object systems unified methods with functions by creating a new kind of function called a *generic function*. In addition to solving the problems just described, generic functions opened up new possibilities for the object system, including many features that simply don't make sense in a message-passing object system.

Generic functions are the heart of Common Lisp's object system and the topic of the rest of this chapter. While I can't talk about generic functions without some mention of classes, for now I'll focus on how to define and use generic functions. In the next chapter I'll show you how to define your own classes.

Generic Functions and Methods

A generic function defines an abstract operation, specifying its name and a parameter list but no implementation. Here, for example, is how you might define a generic function, `draw`, that will be used to draw different kinds of shapes on the screen:

```
(defgeneric draw (shape)
  (:documentation "Draw the given shape on the screen."))
```

I'll discuss the syntax of **DEFGENERIC** in the next section; for now just note that this definition doesn't contain any actual code.

A generic function is generic in the sense that it can--at least in theory--accept any objects as arguments.[5] However, by itself a generic function can't actually do anything; if you just define a generic function, no matter what arguments you call it with, it will signal an error. The actual implementation of a generic function is provided by *methods*. Each method provides an implementation of the generic function for particular classes of arguments. Perhaps the biggest difference between a generic function-based system and a message-passing system is that methods don't belong to classes; they belong to the generic function, which is responsible for determining what method or methods to run in response to a particular invocation.

Methods indicate what kinds of arguments they can handle by *specializing* the required parameters defined by the generic function. For instance, on the generic function `draw`, you might define one method that specializes the `shape` parameter for objects that are instances of the class `circle` while another method specializes `shape` for objects that are instances of the class `triangle`. They would look like this, eliding the actual drawing code:

```
(defmethod draw ((shape circle))
  ...)

(defmethod draw ((shape triangle))
  ...)
```

When a generic function is invoked, it compares the actual arguments it was passed with the specializers of each of its methods to find the *applicable* methods--those methods whose specializers are compatible with the actual arguments. If you invoke `draw`, passing an instance of `circle`, the method that specialized `shape` on the class `circle` is applicable, while if you pass it a `triangle`, then the method that specializes `shape` on the class `triangle` applies. In simple cases, only one method will be applicable, and it will handle the invocation. In more complex cases, there may be multiple methods that apply; they're then combined, as I'll discuss in the section "Method Combination," into a single *effective method* that handles the invocation.

You can specialize a parameter in two ways--usually you'll specify a class that the argument must be an instance of. Because instances of a class are also considered instances of that class's superclasses, a method with a parameter specialized on a particular class can be applicable whenever the corresponding argument is a direct instance of the specializing class or of any of its subclasses. The other kind of specializer is a so-called **`EQL`** specializer, which specifies a particular object to which the method applies.

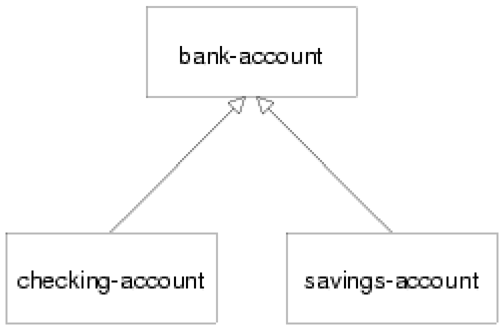
When a generic function has only methods specialized on a single parameter and all the specializers are class specializers, the result of invoking a generic function is quite similar to the result of invoking a method in a message-passing system--the combination of the name of the operation and the class of the object on which it's invoked determines what method to run.

However, reversing the order of lookup opens up possibilities not found in message-passing systems. Generic functions support methods that specialize on multiple parameters, provide a framework that makes multiple inheritance much more manageable, and let you use declarative constructs to control how methods are combined into an effective method, supporting several common usage patterns without a lot of boilerplate code. I'll discuss those topics in a moment. But first you need to look at the basics of the two macros used to define the generic functions **`DEFGENERIC`** and **`DEFMETHOD`**.

DEFGENERIC

To give you a feel for these macros and the various facilities they support, I'll show you some code you might write as part of a banking application--or, rather, a toy banking application; the point is to look at a few language features, not to learn how to really write banking software. For instance, this code doesn't even pretend to deal with such issues as multiple currencies let alone audit trails and transactional integrity.

Because I'm not going to discuss how to define new classes until the next chapter, for now you can just assume that certain classes already exist: for starters, assume there's a class `bank-account` and that it has two subclasses, `checking-account` and `savings-account`. The class hierarchy looks like this:



The first generic function will be `withdraw`, which decreases the account balance by a specified amount. If the balance is less than the amount, it should signal an error and leave the balance unchanged. You can start by defining the generic function with **`DEFGENERIC`**.

The basic form of **`DEFGENERIC`** is similar to **`DEFUN`** except with no body. The parameter list of **`DEFGENERIC`** specifies the parameters that must be accepted by all the methods that will be defined on the generic function. In the place of the body, a **`DEFGENERIC`** can contain various options. One option you should always include is `:documentation`, which you use to provide a string describing the purpose of the generic function. Because a generic function is purely abstract, it's important to be clear to both users and implementers what it's for. Thus, you might define `withdraw` like this:

```
(defgeneric withdraw (account amount)
  (:documentation "Withdraw the specified amount from the account.
Signal an error if the current balance is less than amount."))
```

DEFMETHOD

Now you're ready to use **`DEFMETHOD`** to define methods that implement `withdraw`.[6]

A method's parameter list must be *congruent* with its generic function's. In this case, that means all methods defined on `withdraw` must have exactly two required parameters. More generally, methods must have the same number of required and optional parameters and must be capable of accepting any arguments corresponding to any **`&rest`** or **`&key`** parameters specified by the generic function.[7]

Since the basics of withdrawing are the same for all accounts, you can define a method that specializes the `account` parameter on the `bank-account` class. You can assume the function `balance` returns the current balance of the account and can be used with **`SETF`**--and thus with **`DECF`**--to set the balance. The function **`ERROR`** is a standard function used to signal an error, which I'll discuss in greater detail in Chapter 19. Using those two functions, you can define a basic `withdraw` method that looks like this:

```
(defmethod withdraw ((account bank-account) amount)
  (when (< (balance account) amount)
    (error "Account overdrawn."))
  (decf (balance account) amount))
```

As this code suggests, the form of **DEFMETHOD** is even more like that of **DEFUN** than **DEFGENERIC**'s is. The only difference is that the required parameters can be specialized by replacing the parameter name with a two-element list. The first element is the name of the parameter, and the second element is the specializer, either the name of a class or an **EQL** specializer, the form of which I'll discuss in a moment. The parameter name can be anything--it doesn't have to match the name used in the generic function, though it often will.

This method will apply whenever the first argument to `withdraw` is an instance of `bank-account`. The second parameter, `amount`, is implicitly specialized on **T**, and since all objects are instances of **T**, it doesn't affect the applicability of the method.

Now suppose all checking accounts have overdraft protection. That is, each checking account is linked to another bank account that's drawn upon when the balance of the checking account itself can't cover a withdrawal. You can assume that the function `overdraft-account` takes a `checking-account` object and returns a `bank-account` object representing the linked account.

Thus, withdrawing from a `checking-account` object requires a few extra steps compared to withdrawing from a standard `bank-account` object. You must first check whether the amount being withdrawn is greater than the account's current balance and, if it is, transfer the difference from the overdraft account. Then you can proceed as with a standard `bank-account` object.

So what you'd like to do is define a method on `withdraw` that specializes on `checking-account` to handle the transfer and then lets the method specialized on `bank-account` take control. Such a method might look like this:

```
(defmethod withdraw ((account checking-account) amount)
  (let ((overdraft (- amount (balance account))))
    (when (plusp overdraft)
      (withdraw (overdraft-account account) overdraft)
      (incf (balance account) overdraft)))
  (call-next-method))
```

The function **CALL-NEXT-METHOD** is part of the generic function machinery used to combine applicable methods. It indicates that control should be passed from this method to the method specialized on `bank-account`.[8] When it's called with no arguments, as it is here, the next method is invoked with whatever arguments were originally passed to the generic function. It can also be called with arguments, which will then be passed onto the next method.

You aren't required to invoke **CALL-NEXT-METHOD** in every method. However, if you don't, the new method is then responsible for completely implementing the desired behavior of the generic function. For example, if you had a subclass of `bank-account`, `proxy-account`, that didn't actually keep track of its own balance but instead delegated withdrawals to another account, you might write a method like this (assuming a function, `proxied-account`, that returns the proxied account):

```
(defmethod withdraw ((proxy proxy-account) amount)
  (withdraw (proxied-account proxy) amount))
```

Finally, **`DEFMETHOD`** also allows you to create methods specialized on a particular object with an **`EQL`** specializer. For example, suppose the banking app is going to be deployed in a particularly corrupt bank. Suppose the variable `*account-of-bank-president*` holds a reference to a particular bank account that belongs--as the name suggests--to the bank's president. Further suppose the variable `*bank*` represents the bank as a whole, and the function `embezzle` steals money from the bank. The bank president might ask you to "fix" `withdraw` to handle his account specially.

```
(defmethod withdraw ((account (eql *account-of-bank-president*)) amount)
  (let ((overdraft (- amount (balance account))))
    (when (plusp overdraft)
      (incf (balance account) (embezzle *bank* overdraft)))
  (call-next-method)))
```

Note, however, that the form in the **`EQL`** specializer that provides the object to specialize on--`*account-of-bank-president*` in this case--is evaluated once, when the **`DEFMETHOD`** is evaluated. This method will be specialized on the value of `*account-of-bank-president*` at the time the method is defined; changing the variable later won't change the method.

Method Combination

Outside the body of a method, **`CALL-NEXT-METHOD`** has no meaning. Within a method, it's given a meaning by the generic function machinery that builds an *effective method* each time the generic function is invoked using all the methods applicable to that particular invocation. This notion of building an effective method by combining applicable methods is the heart of the generic function concept and is the thing that allows generic functions to support facilities not found in message-passing systems. So it's worth taking a closer look at what's really happening. Folks with the message-passing model deeply ingrained in their consciousness should pay particular attention because generic functions turn method dispatching inside out compared to message passing, making the generic function, rather than the class, the prime mover.

Conceptually, the effective method is built in three steps: First, the generic function builds a list of applicable methods based on the actual arguments it was passed. Second, the list of applicable methods is sorted according to the *specificity* of their parameter specializers. Finally, methods are taken in order from the sorted list and their code combined to produce the effective method.[9]

To find applicable methods, the generic function compares the actual arguments with the corresponding parameter specializers in each of its methods. A method is applicable if, and only if, all the specializers are compatible with the corresponding arguments.

When the specializer is the name of a class, it's compatible if it names the actual class of the argument or one of its superclasses. (Recall that parameters without explicit specializers are implicitly specialized on the class **`T`** so will be compatible with any argument.) An **`EQL`** specializer is compatible only when the argument is the same object as was specified in the specializer.

Because *all* the arguments are checked against the corresponding specializers, they all affect whether a method is applicable. Methods that explicitly specialize more than one parameter are called *multimethods*; I'll discuss them in the section "Multimethods."

After the applicable methods have been found, the generic function machinery needs to sort them before it can combine them into an effective method. To order two applicable methods, the generic function compares their parameter specializers from left to right,[10] and the first specializer that's different between the two methods determines their ordering, with the method with the more specific specializer coming first.

Because only applicable methods are being sorted, you know all class specializers will name classes that the corresponding argument is actually an instance of. In the typical case, if two class specializers differ, one will be a subclass of the other. In that case, the specializer naming the subclass is considered more specific. This is why the method that specialized `account` on `checking-account` was considered more specific than the method that specialized it on `bank-account`.

Multiple inheritance slightly complicates the notion of specificity since the actual argument may be an instance of two classes, neither of which is a subclass of the other. If such classes are used as parameter specializers, the generic function can't order them using only the rule that subclasses are more specific than their superclasses. In the next chapter I'll discuss how the notion of specificity is extended to deal with multiple inheritance. For now, suffice it to say that there's a deterministic algorithm for ordering class specializers.

Finally, an **`EQL`** specializer is always more specific than any class specializer, and because only applicable methods are being considered, if more than one method has an **`EQL`** specializer for a particular parameter, they must all have the same **`EQL`** specializer. The comparison of those methods will thus be decided based on other parameters.

The Standard Method Combination

Now that you understand how the applicable methods are found and sorted, you're ready to take a closer look at the last step--how the sorted list of methods is combined into a single effective method. By default, generic functions use what's called the *standard method combination*. The standard method combination combines methods so that **`CALL-NEXT-METHOD`** works as you've already seen--the most specific method runs first, and each method can pass control to the next most specific method via **`CALL-NEXT-METHOD`**.

However, there's a bit more to it than that. The methods I've been discussing so far are called *primary methods*. Primary methods, as their name suggests, are responsible for providing the primary implementation of a generic function. The standard method combination also supports three kinds of *auxiliary* methods: `:before`, `:after`, and `:around` methods. An auxiliary method definition is written with **`DEFMETHOD`** like a primary method but with a *method qualifier,* which names the type of method, between the name of the method and the parameter list. For instance, a `:before` method on `withdraw` that specializes the `account` parameter on the class `bank-account` would start like this:

```
(defmethod withdraw :before ((account bank-account) amount) ...)
```

Each kind of auxiliary method is combined into the effective method in a different way. All the applicable `:before` methods--not just the most specific--are run as part of the effective method. They run, as their name suggests, before the most specific primary method and are run in most-specific-first order. Thus, `:before` methods can be used to do any preparation needed to ensure that the primary method can run. For instance, you could've used a `:before` method specialized on `checking-account` to implement the overdraft protection on checking accounts like this:

```
(defmethod withdraw :before ((account checking-account) amount)
  (let ((overdraft (- amount (balance account))))
    (when (plusp overdraft)
      (withdraw (overdraft-account account) overdraft)
      (incf (balance account) overdraft))))
```

This `:before` method has three advantages over a primary method. One is that it makes it immediately obvious how the method changes the overall behavior of the `withdraw` function--it's not going to interfere with the main behavior or change the result returned.

The next advantage is that a primary method specialized on a class more specific than `checking-account` won't interfere with this `:before` method, making it easier for an author of a subclass of `checking-account` to extend the behavior of `withdraw` while keeping part of the old behavior.

Lastly, since a `:before` method doesn't have to call **`CALL-NEXT-METHOD`** to pass control to the remaining methods, it's impossible to introduce a bug by forgetting to.

The other auxiliary methods also fit into the effective method in ways suggested by their names. All the `:after` methods run after the primary methods in most-specific-last order, that is, the reverse of the `:before` methods. Thus, the `:before` and `:after` methods combine to create a sort of nested wrapping around the core functionality provided by the primary methods--each more-specific `:before` method will get a chance to set things up so the less-specific `:before` methods and primary methods can run successfully, and each more-specific `:after` method will get a chance to clean up after all the primary methods and less-specific `:after` methods.

Finally, `:around` methods are combined much like primary methods except they're run "around" all the other methods. That is, the code from the most specific `:around` method is run before anything else. Within the body of an `:around` method, **CALL-NEXT-METHOD** will lead to the code of the next most specific `:around` method or, in the least specific `:around` method, to the complex of `:before`, primary, and `:after` methods. Almost all `:around` methods will contain such a call to **CALL-NEXT-METHOD** because an `:around` method that doesn't will completely hijack the implementation of the generic function from all the methods except for more-specific `:around` methods.

Occasionally that kind of hijacking is called for, but typically `:around` methods are used to establish some dynamic context in which the rest of the methods will run--to bind a dynamic variable, for example, or to establish an error handler (as I'll discuss in Chapter 19). About the only time it's appropriate for an `:around` method to not call **CALL-NEXT-METHOD** is when it returns a result cached from a previous call to **CALL-NEXT-METHOD**. At any rate, an `:around` method that doesn't call **CALL-NEXT-METHOD** is responsible for correctly implementing the semantics of the generic function for all classes of arguments to which the method may apply, including future subclasses.

Auxiliary methods are just a convenient way to express certain common patterns more concisely and concretely. They don't actually allow you to do anything you couldn't do by combining primary methods with diligent adherence to a few coding conventions and some extra typing. Perhaps their biggest benefit is that they provide a uniform framework for extending generic functions. Often a library will define a generic function and provide a default primary method, allowing users of the library to customize its behavior by defining appropriate auxiliary methods.

Other Method Combinations

In addition to the standard method combination, the language specifies nine other built-in method combinations known as the *simple* built-in method combinations. You can also define custom method combinations, though that's a fairly esoteric feature and beyond the scope of this book. I'll briefly cover how to use the simple built-in combinations to give you a sense of the possibilities.

All the simple combinations follow the same pattern: instead of invoking the most specific primary method and letting it invoke less-specific primary methods via **CALL-NEXT-METHOD**, the simple method combinations produce an effective method that contains the code of all the primary methods, one after another, all wrapped in a call to the function, macro, or special operator that gives the method combination its name. The nine combinations are named for the operators: **+**, **AND**, **OR**, **LIST**, **APPEND**, **NCONC**, **MIN**, **MAX**, and **PROGN**. The simple combinations also support only two kinds of methods, primary methods, which are combined as just described, and `:around` methods, which work like `:around` methods in the standard method combination.

For example, a generic function that uses the **+** method combination will return the sum of all the results returned by its primary methods. Note that the **AND** and **OR** method combinations won't necessarily run all the primary methods because of those macros' short-circuiting behavior--a generic function using the **AND** combination will return **NIL** as soon as one of the methods does and will return the value of the last method otherwise. Similarly, the **OR** combination will return the first non-**NIL** value returned by any of the methods.

To define a generic function that uses a particular method combination, you include a `:method-combination` option in the **DEFGENERIC** form. The value supplied with this option is the name of the method combination you want to use. For example, to define a generic function, `priority`, that returns the sum of values returned by individual methods using the **+** method combination, you might write this:

```
(defgeneric priority (job)
  (:documentation "Return the priority at which the job should be run.")
  (:method-combination +))
```

By default all these method combinations combine the primary methods in most-specific-first order. However, you can reverse the order by including the keyword `:most-specific-last` after the name of the method combination in the **DEFGENERIC** form. The order probably doesn't matter if you're using the **+** combination unless the methods have side effects, but for demonstration purposes you can change `priority` to use most-specific-last order like this:

```
(defgeneric priority (job)
  (:documentation "Return the priority at which the job should be run.")
  (:method-combination + :most-specific-last))
```

The primary methods on a generic function that uses one of these combinations must be qualified with the name of the method combination. Thus, a primary method defined on `priority` might look like this:

```
(defmethod priority + ((job express-job)) 10)
```

This makes it obvious when you see a method definition that it's part of a particular kind of generic function.

All the simple built-in method combinations also support `:around` methods that work like `:around` methods in the standard method combination: the most specific `:around` method runs before any other methods, and **CALL-NEXT-METHOD** is used to pass control to less-and-less-specific `:around` methods until it reaches the combined primary methods. The `:most-specific-last` option doesn't affect the order of `:around` methods. And, as I mentioned before, the built-in method combinations don't support `:before` or `:after` methods.

Like the standard method combination, these method combinations don't allow you to do anything you couldn't do "by hand." Rather, they allow you to express *what* you want and let the language take care of wiring everything together for you, making your code both more concise and more expressive.

That said, probably 99 percent of the time, the standard method combination will be exactly what you want. Of the remaining 1 percent, probably 99 percent of them will be handled by one of the simple built-in method combinations. If you run into one of the 1 percent of 1 percent of cases where none of the built-in combinations suffices, you can look up **`DEFINE-METHOD-COMBINATION`** in your favorite Common Lisp reference.

Multimethods

Methods that explicitly specialize more than one of the generic function's required parameters are called *multimethods*. Multimethods are where generic functions and message passing really part ways. Multimethods don't fit into message-passing languages because they don't belong to a particular class; instead, each multimethod defines a part of the implementations of a given generic function that applies when the generic function is invoked with arguments that match *all* the method's specialized parameters.

Multimethods vs. Method Overloading

Programmers used to statically typed message-passing languages such as Java and C++ may think multimethods sound similar to a feature of those languages called *method overloading*. However, these two language features are actually quite different since overloaded methods are chosen at compile time, based on the compile-time type of the arguments, not at runtime. To see how this works, consider the following two Java classes:

```
public class A {
  public void foo(A a) { System.out.println("A/A"); }
  public void foo(B b) { System.out.println("A/B"); }
}

public class B extends A {
  public void foo(A a) { System.out.println("B/A"); }
  public void foo(B b) { System.out.println("B/B"); }
}
```

Now consider what happens when you run the `main` method from this class.

```
public class Main {
  public static void main(String[] argv) {
    A obj = argv[0].equals("A") ? new A() : new B();
    obj.foo(obj);
  }
}
```

When you tell `Main` to instantiate an `A`, it prints "A/A" as you'd probably expect.

```
bash$ java com.gigamonkeys.Main A
A/A
```

However, if you tell `Main` to instantiate a `B`, then the true type of `obj` is taken into account for only half the dispatching.

```
bash$ java com.gigamonkeys.Main B
B/A
```

If overloaded methods worked like Common Lisp's multimethods, then that would print "B/B" instead. It is possible to implement multiple dispatch by hand in message-passing languages, but this runs against the grain of the message-passing model since the code in a multiply dispatched method doesn't belong to any one class.

Multimethods are perfect for all those situations where, in a message-passing language, you struggle to decide to which class a certain behavior ought to belong. Is the sound a drum makes when it's hit with a drumstick a function of what kind of drum it is or what kind of stick you use to hit it? Both, of course. To model this situation in Common Lisp, you simply define a generic function `beat` that takes two arguments.

```
(defgeneric beat (drum stick)
  (:documentation
   "Produce a sound by hitting the given drum with the given stick."))
```

Then you can define various multimethods to implement `beat` for the combinations you care about. For example:

```
(defmethod beat ((drum snare-drum) (stick wooden-drumstick)) ...)
(defmethod beat ((drum snare-drum) (stick brush)) ...)
(defmethod beat ((drum snare-drum) (stick soft-mallet)) ...)
(defmethod beat ((drum tom-tom) (stick wooden-drumstick)) ...)
(defmethod beat ((drum tom-tom) (stick brush)) ...)
(defmethod beat ((drum tom-tom) (stick soft-mallet)) ...)
```

Multimethods don't help with the combinatorial explosion--if you need to model five kinds of drums and six kinds of sticks, and every combination makes a different sound, there's no way around it; you need thirty different methods to implement all the combinations, with or without multimethods. What multimethods do save you from is having to write a bunch of dispatching code by letting you use the same built-in polymorphic dispatching that's so useful when dealing with methods specialized on a single parameter.[11]

Multimethods also save you from having to tightly couple one set of classes with the other. In the drum/stick example, nothing requires the implementation of the drum classes to know about the various classes of drumstick, and nothing requires the drumstick classes to know anything about the various classes of drum. The multimethods connect the otherwise independent classes to describe their joint behavior without requiring any cooperation from the classes themselves.

To Be Continued . . .

I've covered the basics--and a bit beyond--of generic functions, the verbs of Common Lisp's object system. In the next chapter I'll show you how to define your own classes.

---

[1]The language now generally considered the first object-oriented language, Simula, was invented in the early 1960s, only a few years after McCarthy's first Lisp. However, object orientation didn't really take off until the 1980s when the first widely available version of Smalltalk was released, followed by the release of C++ a few years later. Smalltalk took quite a bit of inspiration from

Lisp and combined it with ideas from Simula to produce a dynamic object-oriented language, while C++ combined Simula with C, another fairly static language, to yield a static object-oriented language. This early split has led to much confusion in the definition of object orientation. Folks who come from the C++ tradition tend to consider certain aspects of C++, such as strict data encapsulation, to be key characteristics of object orientation. Folks from the Smalltalk tradition, however, consider many features of C++ to be just that, features of C++, and not core to object orientation. Indeed, Alan Kay, the inventor of Smalltalk, is reported to have said, "I invented the term *object oriented*, and I can tell you that C++ wasn't what I had in mind."

[2]There are those who reject the notion that Common Lisp is in fact object oriented at all. In particular, folks who consider strict data encapsulation a key characteristic of object orientation--usually advocates of relatively static languages such as C++, Eiffel, or Java--don't consider Common Lisp to be properly object oriented. Of course, by that definition, Smalltalk, arguably one of the original and purest object-oriented languages, isn't object oriented either. On the other hand, folks who consider message passing to be the key to object orientation will also not be happy with the claim that Common Lisp is object oriented since Common Lisp's generic function orientation provides degrees of freedom not offered by pure message passing.

[3]Prototype-based languages are the other style of object-oriented language. In these languages, JavaScript being perhaps the most famous example, objects are created by cloning a prototypical object. The clone can then be modified and used as a prototype for other objects.

[4]**`T`** the constant value and **`T`** the class have no particular relationship except they happen to have the same name. **`T`** the value is a direct instance of the class **`SYMBOL`** and only indirectly an instance of **`T`** the class.

[5]Here, as elsewhere, *object* means any Lisp datum--Common Lisp doesn't distinguish, as some languages do, between objects and "primitive" data types; all data in Common Lisp are objects, and every object is an instance of a class.

[6]Technically you could skip the **`DEFGENERIC`** altogether--if you define a method with **`DEFMETHOD`** and no such generic function has been defined, one is automatically created. But it's good form to define generic functions explicitly, if only because it gives you a good place to document the intended behavior.

[7]A method can "accept" **`&key`** and **`&rest`** arguments defined in its generic function by having a **`&rest`** parameter, by having the same **`&key`** parameters, or by specifying **`&allow-other-keys`** along with **`&key`**. A method can also specify **`&key`** parameters not found in the generic function's parameter list--when the generic function is called, any **`&key`** parameter specified by the generic function or any applicable method will be accepted.

[8]**`CALL-NEXT-METHOD`** is roughly analogous to invoking a method on `super` in Java or using an explicitly class-qualified method or function name in Python or C++.

[9]While building the effective method sounds time-consuming, quite a bit of the effort in developing fast Common Lisp implementations has gone into making it efficient. One strategy is to cache the effective method so future calls with the same argument types will be able to proceed directly.

[10]Actually, the order in which specializers are compared is customizable via the **`DEFGENERIC`** option `:argument-precedence-order`, though that option is rarely used.

[11]In languages without multimethods, you must write dispatching code yourself to implement behavior that depends on the class of more than one object. The purpose of the popular Visitor design pattern is to structure a series of singly dispatched method calls so as to provide multiple dispatch. However, it requires one set of classes to know about the other. The Visitor pattern also quickly bogs down in a combinatorial explosion of dispatching methods if it's used to dispatch on more than two objects.