17. Object Reorientation: Classes

If generic functions are the verbs of the object system, classes are the nouns. As I mentioned in the previous chapter, all values in a Common Lisp program are instances of some class. Furthermore, all classes are organized into a single hierarchy rooted at the class **`T`**.

The class hierarchy consists of two major families of classes, built-in and user-defined classes. Classes that represent the data types you've been learning about up until now, classes such as **`INTEGER`**, **`STRING`**, and **`LIST`**, are all built-in. They live in their own section of the class hierarchy, arranged into appropriate sub- and superclass relationships, and are manipulated by the functions I've been discussing for much of the book up until now. You can't subclass these classes, but, as you saw in the previous chapter, you can define methods that specialize on them, effectively extending the behavior of those classes.[1]

But when you want to create new nouns--for instance, the classes used in the previous chapter for representing bank accounts--you need to define your own classes. That's the subject of this chapter.

DEFCLASS

You create user-defined classes with the **`DEFCLASS`** macro. Because behaviors are associated with a class by defining generic functions and methods specialized on the class, **`DEFCLASS`** is responsible only for defining the class as a data type.

The three facets of the class as a data type are its name, its relation to other classes, and the names of the slots that make up instances of the class.[2] The basic form of a **`DEFCLASS`** is quite simple.

```
(defclass name (direct-superclass-name*)
  (slot-specifier*))
```

What Are "User-Defined Classes"?

The term *user-defined classes* isn't a term from the language standard--technically what I'm talking about when I say *user-defined classes* are classes that subclass **`STANDARD-OBJECT`** and whose metaclass is **`STANDARD-CLASS`**. But since I'm not going to talk about the ways you can define classes that don't subclass **`STANDARD-OBJECT`** and whose metaclass isn't **`STANDARD-CLASS`**, you don't really have to worry about that. *User-defined* isn't a perfect term for these classes since the implementation may define certain classes the same way. However, to call them *standard* classes would be even more confusing since the built-in classes, such as **`INTEGER`** and **`STRING`**, are just as standard, if not more so, because they're defined by the language standard but they don't extend **`STANDARD-OBJECT`**. To further complicate matters, it's also

possible for users to define new classes that *don't* subclass **STANDARD-OBJECT**. In particular, the macro **DEFSTRUCT** also defines new classes. But that's largely for backward compatibility--**DEFSTRUCT** predated CLOS and was retrofitted to define classes when CLOS was integrated into the language. But the classes it creates are fairly limited compared to **DEFCLASS**ed classes. So in this chapter I'll be discussing only classes defined with **DEFCLASS** that use the default metaclass of **STANDARD-CLASS**, and I'll refer to them as *user-defined* for lack of a better term.

As with functions and variables, you can use any symbol as the name of a new class.[3] Class names are in a separate namespace from both functions and variables, so you can have a class, function, and variable all with the same name. You'll use the class name as the argument to **MAKE-INSTANCE**, the function that creates new instances of user-defined classes.

The *direct-superclass-names* specify the classes of which the new class is a subclass. If no superclasses are listed, the new class will directly subclass **STANDARD-OBJECT**. Any classes listed must be other user-defined classes, which ensures that each new class is ultimately descended from **STANDARD-OBJECT**. **STANDARD-OBJECT** in turn subclasses **T**, so all user-defined classes are part of the single class hierarchy that also contains all the built-in classes.

Eliding the slot specifiers for a moment, the **DEFCLASS** forms of some of the classes you used in the previous chapter might look like this:

```
(defclass bank-account () ...)

(defclass checking-account (bank-account) ...)

(defclass savings-account (bank-account) ...)
```

I'll discuss in the section "Multiple Inheritance" what it means to list more than one direct superclass in *direct-superclass-names*.

Slot Specifiers

The bulk of a **DEFCLASS** form consists of the list of slot specifiers. Each slot specifier defines a slot that will be part of each instance of the class. Each slot in an instance is a place that can hold a value, which can be accessed using the **SLOT-VALUE** function. **SLOT-VALUE** takes an object and the name of a slot as arguments and returns the value of the named slot in the given object. It can be used with **SETF** to set the value of a slot in an object.

A class also inherits slot specifiers from its superclasses, so the set of slots actually present in any object is the union of all the slots specified in a class's **DEFCLASS** form and those specified in all its superclasses.

At the minimum, a slot specifier names the slot, in which case the slot specifier can be just a name. For instance, you could define a `bank-account` class with two slots, `customer-name` and `balance`, like this:

```
(defclass bank-account ()
  (customer-name
   balance))
```

Each instance of this class will contain two slots, one to hold the name of the customer the account belongs to and another to hold the current balance. With this definition, you can create new `bank-account` objects using **`MAKE-INSTANCE`**.

```
(make-instance 'bank-account) ==> #<BANK-ACCOUNT @ #x724b93ba>
```

The argument to **`MAKE-INSTANCE`** is the name of the class to instantiate, and the value returned is the new object.[4] The printed representation of an object is determined by the generic function **`PRINT-OBJECT`**. In this case, the applicable method will be one provided by the implementation, specialized on **`STANDARD-OBJECT`**. Since not every object can be printed so that it can be read back, the **`STANDARD-OBJECT`** print method uses the #<> syntax, which will cause the reader to signal an error if it tries to read it. The rest of the representation is implementation-defined but will typically be something like the output just shown, including the name of the class and some distinguishing value such as the address of the object in memory. In Chapter 23 you'll see an example of how to define a method on **`PRINT-OBJECT`** to make objects of a certain class be printed in a more informative form.

Using the definition of `bank-account` just given, new objects will be created with their slots *unbound*. Any attempt to get the value of an unbound slot signals an error, so you must set a slot before you can read it.

```
(defparameter *account* (make-instance 'bank-account))  ==> *ACCOUNT*
(setf (slot-value *account* 'customer-name) "John Doe") ==> "John Doe"
(setf (slot-value *account* 'balance) 1000)             ==> 1000
```

Now you can access the value of the slots.

```
(slot-value *account* 'customer-name) ==> "John Doe"
(slot-value *account* 'balance)       ==> 1000
```

Object Initialization

Since you can't do much with an object with unbound slots, it'd be nice to be able to create objects with their slots already initialized. Common Lisp provides three ways to control the initial value of slots. The first two involve adding options to the slot specifier in the **`DEFCLASS`** form: with the `:initarg` option, you can specify a name that can then be used as a keyword parameter to **`MAKE-INSTANCE`** and whose argument will be stored in the slot. A second option, `:initform`, lets you specify a Lisp expression that will be used to compute a value for the slot if no `:initarg` argument is passed to **`MAKE-INSTANCE`**. Finally, for complete control over the initialization, you can define a method on the generic function **`INITIALIZE-INSTANCE`**, which is called by **`MAKE-INSTANCE`**.[5]

A slot specifier that includes options such as `:initarg` or `:initform` is written as a list starting with the name of the slot followed by the options. For example, if you want to modify the definition of `bank-account` to allow callers of **`MAKE-INSTANCE`** to pass the customer name and the initial balance and to provide a default value of zero dollars for the balance, you'd write this:

```
(defclass bank-account ()
  ((customer-name
    :initarg :customer-name)
   (balance
    :initarg :balance
    :initform 0)))
```

Now you can create an account and specify the slot values at the same time.

```
(defparameter *account*
  (make-instance 'bank-account :customer-name "John Doe" :balance 1000))

(slot-value *account* 'customer-name) ==> "John Doe"
(slot-value *account* 'balance)       ==> 1000
```

If you don't supply a `:balance` argument to **`MAKE-INSTANCE`**, the **`SLOT-VALUE`** of `balance` will be computed by evaluating the form specified with the `:initform` option. But if you don't supply a `:customer-name` argument, the `customer-name` slot will be unbound, and an attempt to read it before you set it will signal an error.

```
(slot-value (make-instance 'bank-account) 'balance)       ==> 0
(slot-value (make-instance 'bank-account) 'customer-name) ==> error
```

If you want to ensure that the customer name is supplied when the account is created, you can signal an error in the initform since it will be evaluated only if an initarg isn't supplied. You can also use initforms that generate a different value each time they're evaluated--the initform is evaluated anew for each object. To experiment with these techniques, you can modify the `customer-name` slot specifier and add a new slot, `account-number`, that's initialized with the value of an ever-increasing counter.

```
(defvar *account-numbers* 0)

(defclass bank-account ()
  ((customer-name
    :initarg :customer-name
    :initform (error "Must supply a customer name."))
   (balance
    :initarg :balance
    :initform 0)
   (account-number
    :initform (incf *account-numbers*))))
```

Most of the time the combination of `:initarg` and `:initform` options will be sufficient to properly initialize an object. However, while an initform can be any Lisp expression, it has no access to the object being initialized, so it can't initialize one slot based on the value of another. For that you need to define a method on the generic function **`INITIALIZE-INSTANCE`**.

The primary method on **INITIALIZE-INSTANCE** specialized on **STANDARD-OBJECT** takes care of initializing slots based on their `:initarg` and `:initform` options. Since you don't want to disturb that, the most common way to add custom initialization code is to define an `:after` method specialized on your class.[6] For instance, suppose you want to add a slot `account-type` that needs to be set to one of the values `:gold`, `:silver`, or `:bronze` based on the account's initial balance. You might change your class definition to this, adding the `account-type` slot with no options:

```
(defclass bank-account ()
  ((customer-name
    :initarg :customer-name
    :initform (error "Must supply a customer name."))
   (balance
    :initarg :balance
    :initform 0)
   (account-number
    :initform (incf *account-numbers*))
   account-type))
```

Then you can define an `:after` method on **INITIALIZE-INSTANCE** that sets the `account-type` slot based on the value that has been stored in the `balance` slot.[7]

```
(defmethod initialize-instance :after ((account bank-account) &key)
  (let ((balance (slot-value account 'balance)))
    (setf (slot-value account 'account-type)
          (cond
            ((>= balance 100000) :gold)
            ((>= balance 50000) :silver)
            (t :bronze)))))
```

The **&key** in the parameter list is required to keep the method's parameter list congruent with the generic function's--the parameter list specified for the **INITIALIZE-INSTANCE** generic function includes **&key** in order to allow individual methods to supply their own keyword parameters but doesn't require any particular ones. Thus, every method must specify **&key** even if it doesn't specify any **&key** parameters.

On the other hand, if an **INITIALIZE-INSTANCE** method specialized on a particular class does specify a **&key** parameter, that parameter becomes a legal parameter to **MAKE-INSTANCE** when creating an instance of that class. For instance, if the bank sometimes pays a percentage of the initial balance as a bonus when an account is opened, you could implement that using a method on **INITIALIZE-INSTANCE** that takes a keyword argument to specify the percentage of the bonus like this:

```
(defmethod initialize-instance :after ((account bank-account)
                                       &key opening-bonus-percentage)
  (when opening-bonus-percentage
    (incf (slot-value account 'balance)
          (* (slot-value account 'balance) (/ opening-bonus-percentage 100)))))
```

By defining this **INITIALIZE-INSTANCE** method, you make `:opening-bonus-percentage` a legal argument to **MAKE-INSTANCE** when creating a `bank-account` object.

```
CL-USER> (defparameter *acct* (make-instance
                               'bank-account
                               :customer-name "Sally Sue"
                               :balance 1000
                               :opening-bonus-percentage 5))
*ACCT*
CL-USER> (slot-value *acct* 'balance)
1050
```

Accessor Functions

Between **MAKE-INSTANCE** and **SLOT-VALUE**, you have all the tools you need for creating and manipulating instances of your classes. Everything else you might want to do can be implemented in terms of those two functions. However, as anyone familiar with the principles of good object-oriented programming practices knows, directly accessing the slots (or fields or member variables) of an object can lead to fragile code. The problem is that directly accessing slots ties your code too tightly to the concrete structure of your class. For example, suppose you decide to change the definition of `bank-account` so that, instead of storing the current balance as a number, you store a list of time-stamped withdrawals and deposits. Code that directly accesses the `balance` slot will likely break if you change the class definition to remove the slot or to store the new list in the old slot. On the other hand, if you define a function, `balance`, that accesses the slot, you can redefine it later to preserve its behavior even if the internal representation changes. And code that uses such a function will continue to work without modification.

Another advantage to using accessor functions rather than direct access to slots via **SLOT-VALUE** is that they let you limit the ways outside code can modify a slot.[8] It may be fine for users of the `bank-account` class to get the current balance, but you may want all modifications to the balance to go through other functions you'll provide, such as `deposit` and `withdraw`. If clients know they're supposed to manipulate objects only through the published functional API, you can provide a `balance` function but not make it **SETF**able if you want the balance to be read-only.

Finally, using accessor functions makes your code tidier since it helps you avoid lots of uses of the rather verbose **SLOT-VALUE** function.

It's trivial to define a function that reads the value of the `balance` slot.

```
(defun balance (account)
  (slot-value account 'balance))
```

However, if you know you're going to define subclasses of `bank-account`, it might be a good idea to define `balance` as a generic function. That way, you can provide different methods on `balance` for those subclasses or extend its definition with auxiliary methods. So you might write this instead:

```
(defgeneric balance (account))
```

```
(defmethod balance ((account bank-account))
  (slot-value account 'balance))
```

As I just discussed, you don't want callers to be able to directly set the balance, but for other slots, such as `customer-name`, you may also want to provide a function to set them. The cleanest way to define such a function is as a **`SETF`** function.

A **`SETF`** function is a way to extend **`SETF`**, defining a new kind of place that it knows how to set. The name of a **`SETF`** function is a two-item list whose first element is the symbol `setf` and whose second element is a symbol, typically the name of a function used to access the place the **`SETF`** function will set. A **`SETF`** function can take any number of arguments, but the first argument is always the value to be assigned to the place.[9] You could, for instance, define a **`SETF`** function to set the `customer-name` slot in a `bank-account` like this:

```
(defun (setf customer-name) (name account)
  (setf (slot-value account 'customer-name) name))
```

After evaluating that definition, an expression like the following one:

```
(setf (customer-name my-account) "Sally Sue")
```

will be compiled as a call to the **`SETF`** function you just defined with "Sally Sue" as the first argument and the value of `my-account` as the second argument.

Of course, as with reader functions, you'll probably want your **`SETF`** function to be generic, so you'd actually define it like this:

```
(defgeneric (setf customer-name) (value account))

(defmethod (setf customer-name) (value (account bank-account))
  (setf (slot-value account 'customer-name) value))
```

And of course you'll also want to define a reader function for `customer-name`.

```
(defgeneric customer-name (account))

(defmethod customer-name ((account bank-account))
  (slot-value account 'customer-name))
```

This allows you to write the following:

```
(setf (customer-name *account*) "Sally Sue") ==> "Sally Sue"

(customer-name *account*)                    ==> "Sally Sue"
```

There's nothing hard about writing these accessor functions, but it wouldn't be in keeping with The Lisp Way to have to write them all by hand. Thus, **`DEFCLASS`** supports three slot options that allow you to automatically create reader and writer functions for a specific slot.

The `:reader` option specifies a name to be used as the name of a generic function that accepts an object as its single argument. When the **`DEFCLASS`** is evaluated, the generic function is created, if it doesn't already exist. Then a method specializing its single argument on the new

class and returning the value of the slot is added to the generic function. The name can be anything, but it's typical to name it the same as the slot itself. Thus, instead of explicitly writing the `balance` generic function and method as shown previously, you could change the slot specifier for the `balance` slot in the definition of `bank-account` to this:

```
(balance
 :initarg :balance
 :initform 0
 :reader balance)
```

The `:writer` option is used to create a generic function and method for setting the value of a slot. The function and method created follow the requirements for a **`SETF`** function, taking the new value as the first argument and returning it as the result, so you can define a **`SETF`** function by providing a name such as `(setf customer-name)`. For instance, you could provide reader and writer methods for `customer-name` equivalent to the ones you just wrote by changing the slot specifier to this:

```
(customer-name
 :initarg :customer-name
 :initform (error "Must supply a customer name.")
 :reader customer-name
 :writer (setf customer-name))
```

Since it's quite common to want both reader and writer functions, **`DEFCLASS`** also provides an option, `:accessor`, that creates both a reader function and the corresponding **`SETF`** function. So instead of the slot specifier just shown, you'd typically write this:

```
(customer-name
 :initarg :customer-name
 :initform (error "Must supply a customer name.")
 :accessor customer-name)
```

Finally, one last slot option you should know about is the `:documentation` option, which you can use to provide a string that documents the purpose of the slot. Putting it all together and adding a reader method for the `account-number` and `account-type` slots, the **`DEFCLASS`** form for the `bank-account` class would look like this:

```
(defclass bank-account ()
  ((customer-name
    :initarg :customer-name
    :initform (error "Must supply a customer name.")
    :accessor customer-name
    :documentation "Customer's name")
   (balance
    :initarg :balance
    :initform 0
    :reader balance
    :documentation "Current account balance")
   (account-number
    :initform (incf *account-numbers*)
    :reader account-number
    :documentation "Account number, unique within a bank.")
   (account-type
    :reader account-type
    :documentation "Type of account, one of :gold, :silver, or :bronze.")))
```

WITH-SLOTS and WITH-ACCESSORS

While using accessor functions will make your code easier to maintain, they can still be a bit verbose. And there will be times, when writing methods that implement the low-level behaviors of a class, that you may specifically want to access slots directly to set a slot that has no writer function or to get at the slot value without causing any auxiliary methods defined on the reader function to run.

This is what **SLOT-VALUE** is for; however, it's still quite verbose. To make matters worse, a function or method that accesses the same slot several times can become clogged with calls to accessor functions and **SLOT-VALUE**. For example, even a fairly simple method such as the following, which assesses a penalty on a `bank-account` if its balance falls below a certain minimum, is cluttered with calls to `balance` and **SLOT-VALUE**:

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (when (< (balance account) *minimum-balance*)
    (decf (slot-value account 'balance) (* (balance account) .01))))
```

And if you decide you want to directly access the slot value in order to avoid running auxiliary methods, it gets even more cluttered.

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (when (< (slot-value account 'balance) *minimum-balance*)
    (decf (slot-value account 'balance) (* (slot-value account 'balance) .01))))
```

Two standard macros, **WITH-SLOTS** and **WITH-ACCESSORS**, can help tidy up this clutter. Both macros create a block of code in which simple variable names can be used to refer to slots on a particular object. **WITH-SLOTS** provides direct access to the slots, as if by **SLOT-VALUE**, while **WITH-ACCESSORS** provides a shorthand for accessor methods.

The basic form of **WITH-SLOTS** is as follows:

```
(with-slots (slot*) instance-form
  body-form*)
```

Each element of *slots* can be either the name of a slot, which is also used as a variable name, or a two-item list where the first item is a name to use as a variable and the second is the name of the slot. The *instance-form* is evaluated once to produce the object whose slots will be accessed. Within the body, each occurrence of one of the variable names is translated to a call to **SLOT-VALUE** with the object and the appropriate slot name as arguments.[10] Thus, you can write `assess-low-balance-penalty` like this:

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (with-slots (balance) account
    (when (< balance *minimum-balance*)
      (decf balance (* balance .01)))))
```

or, using the two-item list form, like this:

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (with-slots ((bal balance)) account
    (when (< bal *minimum-balance*)
      (decf bal (* bal .01)))))
```

If you had defined `balance` with an `:accessor` rather than just a `:reader`, then you could also use **`WITH-ACCESSORS`**. The form of **`WITH-ACCESSORS`** is the same as **`WITH-SLOTS`** except each element of the slot list is a two-item list containing a variable name and the name of an accessor function. Within the body of **`WITH-ACCESSORS`**, a reference to one of the variables is equivalent to a call to the corresponding accessor function. If the accessor function is **`SETF`**able, then so is the variable.

```
(defmethod assess-low-balance-penalty ((account bank-account))
  (with-accessors ((balance balance)) account
    (when (< balance *minimum-balance*)
      (decf balance (* balance .01)))))
```

The first `balance` is the name of the variable, and the second is the name of the accessor function; they don't have to be the same. You could, for instance, write a method to merge two accounts using two calls to **`WITH-ACCESSORS`**, one for each account.

```
(defmethod merge-accounts ((account1 bank-account) (account2 bank-account))
  (with-accessors ((balance1 balance)) account1
    (with-accessors ((balance2 balance)) account2
      (incf balance1 balance2)
      (setf balance2 0))))
```

The choice of whether to use **`WITH-SLOTS`** versus **`WITH-ACCESSORS`** is the same as the choice between **`SLOT-VALUE`** and an accessor function: low-level code that provides the basic functionality of a class may use **`SLOT-VALUE`** or **`WITH-SLOTS`** to directly manipulate slots in ways not supported by accessor functions or to explicitly avoid the effects of auxiliary methods that may have been defined on the accessor functions. But you should generally use accessor functions or **`WITH-ACCESSORS`** unless you have a specific reason not to.

Class-Allocated Slots

The last slot option you need to know about is `:allocation`. The value of `:allocation` can be either `:instance` or `:class` and defaults to `:instance` if not specified. When a slot has `:class` allocation, the slot has only a single value, which is stored in the class and shared by all instances.

However, `:class` slots are accessed the same as `:instance` slots--they're accessed with **`SLOT-VALUE`** or an accessor function, which means you can access the slot value only through an instance of the class even though it isn't actually stored in the instance. The `:initform` and `:initarg` options have essentially the same effect except the initform is evaluated once when the class is defined rather than each time an instance is created. On the other hand, passing an initarg to **`MAKE-INSTANCE`** will set the value, affecting all instances of the class.

Because you can't get at a class-allocated slot without an instance of the class, class-allocated slots aren't really equivalent to *static* or *class* fields in languages such as Java, C++, and Python.[11] Rather, class-allocated slots are used primarily to save space; if you're going to create many instances of a class and all instances are going to have a reference to the same object--say, a pool of shared resources--you can save the cost of each instance having its own reference by making the slot class-allocated.

Slots and Inheritance

As I discussed in the previous chapter, classes inherit behavior from their superclasses thanks to the generic function machinery--a method specialized on class `A` is applicable not only to direct instances of `A` but also to instances of `A`'s subclasses. Classes also inherit slots from their superclasses, but the mechanism is slightly different.

In Common Lisp a given object can have only one slot with a particular name. However, it's possible that more than one class in the inheritance hierarchy of a given class will specify a slot with a particular name. This can happen either because a subclass includes a slot specifier with the same name as a slot specified in a superclass or because multiple superclasses specify slots with the same name.

Common Lisp resolves these situations by merging all the specifiers with the same name from the new class and all its superclasses to create a single specifier for each unique slot name. When merging specifiers, different slot options are treated differently. For instance, since a slot can have only a single default value, if multiple classes specify an `:initform`, the new class uses the one from the most specific class. This allows a subclass to specify a different default value than the one it would otherwise inherit.

On the other hand, `:initarg`s needn't be exclusive--each `:initarg` option in a slot specifier creates a keyword parameter that can be used to initialize the slot; multiple parameters don't create a conflict, so the new slot specifier contains all the `:initarg`s. Callers of **`MAKE-INSTANCE`** can use any of the `:initarg`s to initialize the slot. If a caller passes multiple keyword arguments that initialize the same slot, then the leftmost argument in the call to **`MAKE-INSTANCE`** is used.

Inherited `:reader`, `:writer`, and `:accessor` options aren't included in the merged slot specifier since the methods created by the superclass's **`DEFCLASS`** will already apply to the new class. The new class can, however, create its own accessor functions by supplying its own `:reader`, `:writer`, or `:accessor` options.

Finally, the `:allocation` option is, like `:initform`, determined by the most specific class that specifies the slot. Thus, it's possible for all instances of one class to share a `:class` slot while instances of a subclass may each have their own `:instance` slot of the same name. And

a sub-subclass may then redefine it back to `:class` slot, so all instances of *that* class will again share a single slot. In the latter case, the slot shared by instances of the sub-subclass is different than the slot shared by the original superclass.

For instance, suppose you have these classes:

```
(defclass foo ()
  ((a :initarg :a :initform "A" :accessor a)
   (b :initarg :b :initform "B" :accessor b)))

(defclass bar (foo)
  ((a :initform (error "Must supply a value for a"))
   (b :initarg :the-b :accessor the-b :allocation :class)))
```

When instantiating the class `bar`, you can use the inherited initarg, `:a`, to specify a value for the slot `a` and, in fact, must do so to avoid an error, since the `:initform` supplied by `bar` supersedes the one inherited from `foo`. To initialize the `b` slot, you can use either the inherited initarg :`b` or the new initarg `:the-b`. However, because of the `:allocation` option on the `b` slot in `bar`, the value specified will be stored in the slot shared by all instances of `bar`. That same slot can be accessed either with the method on the generic function `b` that specializes on `foo` or with the new method on the generic function `the-b` that specializes directly on `bar`. To access the `a` slot on either a `foo` or a `bar`, you'll continue to use the generic function `a`.

Usually merging slot definitions works quite nicely. However, it's important to be aware when using multiple inheritance that two unrelated slots that happen to have the same name can be merged into a single slot in the new class. Thus, methods specialized on different classes could end up manipulating the same slot when applied to a class that extends those classes. This isn't much of a problem in practice since, as you'll see in Chapter 21, you can use the package system to avoid collisions between names in independently developed pieces of code.

Multiple Inheritance

All the classes you've seen so far have had only a single direct superclass. Common Lisp also supports multiple inheritance--a class can have multiple direct superclasses, inheriting applicable methods and slot specifiers from all of them.

Multiple inheritance doesn't dramatically change any of the mechanisms of inheritance I've discussed so far--every user-defined class already has multiple superclasses since they all extend **`STANDARD-OBJECT`**, which extends **`T`**, and so have at least two superclasses. The wrinkle that multiple inheritance adds is that a class can have more than one *direct* superclass. This complicates the notion of class specificity that's used both when building the effective methods for a generic function and when merging inherited slot specifiers.

That is, if classes could have only a single direct superclass, ordering classes by specificity would be trivial--a class and all its superclasses could be ordered in a straight line starting from the class itself, followed by its single direct superclass, followed by *its* direct superclass, all the

way up to **T**. But when a class has multiple direct superclasses, those superclasses are typically not related to each other--indeed, if one was a subclass of another, you wouldn't need to subclass both directly. In that case, the rule that subclasses are more specific than their superclasses isn't enough to order all the superclasses. So Common Lisp uses a second rule that sorts unrelated superclasses according to the order they're listed in the **DEFCLASS**'s direct superclass list--classes earlier in the list are considered more specific than classes later in the list. This rule is admittedly somewhat arbitrary but does allow every class to have a linear *class precedence list*, which can be used to determine which superclasses should be considered more specific than others. Note, however, there's no global ordering of classes--each class has its own class precedence list, and the same classes can appear in different orders in different classes' class precedence lists.

To see how this works, let's add a class to the banking app: `money-market-account`. A money market account combines the characteristics of a checking account and a savings account: a customer can write checks against it, but it also earns interest. You might define it like this:

```
(defclass money-market-account (checking-account savings-account) ())
```

The class precedence list for `money-market-account` will be as follows:

```
(money-market-account
 checking-account
 savings-account
 bank-account
 standard-object
 t)
```

Note how this list satisfies both rules: every class appears before all its superclasses, and `checking-account` and `savings-account` appear in the order specified in **DEFCLASS**.

This class defines no slots of its own but will inherit slots from both of its direct superclasses, including the slots they inherit from their superclasses. Likewise, any method that's applicable to any class in the class precedence list will be applicable to a `money-market-account` object. Because all slot specifiers for the same slot are merged, it doesn't matter that `money-market-account` inherits the same slot specifiers from `bank-account` twice. [12]

Multiple inheritance is easiest to understand when the different superclasses provide completely independent slots and behaviors. For instance, `money-market-account` will inherit slots and behaviors for dealing with checks from `checking-account` and slots and behaviors for computing interest from `savings-account`. You don't have to worry about the class precedence list for methods and slots inherited from only one superclass or another.

However, it's also possible to inherit different methods for the same generic function from different superclasses. In that case, the class precedence list does come into play. For instance, suppose the banking application defined a generic function `print-statement` used to

generate monthly statements. Presumably there would already be methods for `print-statement` specialized on both `checking-account` and `savings-account`. Both of these methods will be applicable to instances of `money-market-account`, but the one specialized on `checking-account` will be considered more specific than the one on `savings-account` because `checking-account` precedes `savings-account` in `money-market-account`'s class precedence list.

Assuming the inherited methods are all primary methods and you haven't defined any other methods, the method specialized on `checking-account` will be used if you invoke `print-statement` on `money-market-account`. However, that won't necessarily give you the behavior you want since you probably want a money market account's statement to contain elements of both a checking account and a savings account statement.

You can modify the behavior of `print-statement` for `money-market-accounts` in a couple ways. One straightforward way is to define a new primary method specialized on `money-market-account`. This gives you the most control over the new behavior but will probably require more new code than some other options I'll discuss in a moment. The problem is that while you can use **`CALL-NEXT-METHOD`** to call "up" to the next most specific method, namely, the one specialized on `checking-account`, there's no way to invoke a particular less-specific method, such as the one specialized on `savings-account`. Thus, if you want to be able to reuse the code that prints the `savings-account` part of the statement, you'll need to break that code into a separate function, which you can then call directly from both the `money-market-account` and `savings-account` `print-statement` methods.

Another possibility is to write the primary methods of all three classes to call **`CALL-NEXT-METHOD`**. Then the method specialized on `money-market-account` will use **`CALL-NEXT-METHOD`** to invoke the method specialized on `checking-account`. When that method calls **`CALL-NEXT-METHOD`**, it will result in running the `savings-account` method since it will be the next most specific method according to `money-market-account`'s class precedence list.

Of course, if you're going to rely on a coding convention--that every method calls **`CALL-NEXT-METHOD`**--to ensure all the applicable methods run at some point, you should think about using auxiliary methods instead. In this case, instead of defining primary methods on `print-statement` for `checking-account` and `savings-account`, you can define those methods as `:after` methods, defining a single primary method on `bank-account`. Then, `print-statement`, called on a `money-market-account`, will print a basic account statement, output by the primary method specialized on `bank-account`, followed by details output by the `:after` methods specialized on `savings-account` and `checking-account`. And if you want to add details specific to

`money-market-accounts`, you can define an `:after` method specialized on `money-market-account`, which will run last of all.

The advantage of using auxiliary methods is that it makes it quite clear which methods are primarily responsible for implementing the generic function and which ones are only contributing additional bits of functionality. The disadvantage is that you don't get fine-grained control over the order in which the auxiliary methods run--if you wanted the `checking-account` part of the statement to print before the `savings-account` part, you'd have to change the order in which the `money-market-account` subclasses those classes. But that's a fairly dramatic change that could affect other methods and inherited slots. In general, if you find yourself twiddling the order of the direct superclass list as a way of fine-tuning the behavior of specific methods, you probably need to step back and rethink your approach.

On the other hand, if you don't care exactly what the order is but want it to be consistent across several generic functions, then using auxiliary methods may be just the thing. For example, if in addition to `print-statement` you have a `print-detailed-statement` generic function, you can implement both functions using `:after` methods on the various subclasses of `bank-account`, and the order of the parts of both a regular and a detailed statement will be the same.

Good Object-Oriented Design

That's about it for the main features of Common Lisp's object system. If you have lots of experience with object-oriented programming, you can probably see how Common Lisp's features can be used to implement good object-oriented designs. However, if you have less experience with object orientation, you may need to spend some time absorbing the object-oriented way of thinking. Unfortunately, that's a fairly large topic and beyond the scope of this book. Or, as the man page for Perl's object system puts it, "Now you need just to go off and buy a book about object-oriented design methodology and bang your forehead with it for the next six months or so." Or you can wait for some of the practical chapters, later in this book, where you'll see several examples of how these features are used in practice. For now, however, you're ready to take a break from all this theory of object orientation and turn to the rather different topic of how to make good use of Common Lisp's powerful, but sometimes cryptic, **`FORMAT`** function.

---

[1]Defining new methods for an existing class may seem strange to folks used to statically typed languages such as C++ and Java in which all the methods of a class must be defined as part of the class definition. But programmers with experience in dynamically typed object-oriented languages such as Smalltalk and Objective C will find nothing strange about adding new behaviors to existing classes.

[2]In other object-oriented languages, slots might be called *fields*, *member variables*, or *attributes*.

[3]As when naming functions and variables, it's not quite true that you can use *any* symbol as a class name--you can't use names defined by the language standard. You'll see in Chapter 21 how to avoid such name conflicts.

[4]The argument to **MAKE-INSTANCE** can actually be either the name of the class or a class object returned by the function **CLASS-OF** or **FIND-CLASS**.

[5]Another way to affect the values of slots is with the `:default-initargs` option to **DEFCLASS**. This option is used to specify forms that will be evaluated to provide arguments for specific initialization parameters that aren't given a value in a particular call to **MAKE-INSTANCE**. You don't need to worry about `:default-initargs` for now.

[6]Adding an `:after` method to **INITIALIZE-INSTANCE** is the Common Lisp analog to defining a constructor in Java or C++ or an `__init__` method in Python.

[7]One mistake you might make until you get used to using auxiliary methods is to define a method on **INITIALIZE-INSTANCE** but without the `:after` qualifier. If you do that, you'll get a new primary method that shadows the default one. You can remove the unwanted primary method using the functions **REMOVE-METHOD** and **FIND-METHOD**. Certain development environments may provide a graphical user interface to do the same thing.

```
(remove-method #'initialize-instance
  (find-method #'initialize-instance () (list (find-class 'bank-account))))
```

[8]Of course, providing an accessor function doesn't really limit anything since other code can still use **SLOT-VALUE** to get at slots directly. Common Lisp doesn't provide strict encapsulation of slots the way some languages such as C++ and Java do; however, if the author of a class provides accessor functions and you ignore them, using **SLOT-VALUE** instead, you had better know what you're doing. It's also possible to use the package system, which I'll discuss in Chapter 21, to make it even more obvious that certain slots aren't to be accessed directly, by not exporting the names of the slots.

[9]One consequence of defining a **SETF** function--say, `(setf foo)`--is that if you also define the corresponding accessor function, `foo` in this case, you can use all the modify macros built upon **SETF**, such as **INCF**, **DECF**, **PUSH**, and **POP**, on the new kind of place.

[10]The "variable" names provided by **WITH-SLOTS** and **WITH-ACCESSORS** aren't true variables; they're implemented using a special kind of macro, called a *symbol macro*, that allows a simple name to expand into arbitrary code. Symbol macros were introduced into the language to support **WITH-SLOTS** and **WITH-ACCESSORS**, but you can also use them for your own purposes. I'll discuss them in a bit more detail in Chapter 20.

[11]The Meta Object Protocol (MOP), which isn't part of the language standard but is supported by most Common Lisp implementations, provides a function, `class-prototype`, that returns an instance of a class that can be used to access class slots. If you're using an implementation that supports the MOP and happen to be translating some code from another language that makes heavy use of static or class fields, this may give you a way to ease the translation. But it's not all that idiomatic.

[12]In other words, Common Lisp doesn't suffer from the *diamond inheritance* problem the way, say, C++ does. In C++, when one class subclasses two classes that both inherit a member variable from a common superclass, the bottom class inherits the member variable twice, leading to no end of confusion.