4. Types and Classes

4.1 Introduction

A *type* is a (possibly infinite) set of *objects*. An *object* can belong to more than one *type*. *Types* are never explicitly represented as *objects* by Common Lisp. Instead, they are referred to indirectly by the use of *type specifiers*, which are *objects* that denote *types*.

New *types* can be defined using **deftype**, **defstruct**, **defclass**, and **define-condition**.

The *function* **typep**, a set membership test, is used to determine whether a given *object* is of a given *type*. The function **subtypep**, a subset test, is used to determine whether a given *type* is a *subtype* of another given *type*. The function **type-of** returns a particular *type* to which a given *object* belongs, even though that *object* must belong to one or more other *types* as well. (For example, every *object* is of *type* **t**, but **type-of** always returns a *type specifier* for a *type* more specific than **t**.)

Objects, not *variables*, have *types*. Normally, any *variable* can have any *object* as its *value*. It is possible to declare that a *variable* takes on only values of a given *type* by making an explicit *type declaration*. *Types* are arranged in a directed acyclic graph, except for the presence of equivalences.

Declarations can be made about *types* using **declare**, **proclaim**, **declaim**, or **the**. For more information about *declarations*, see Section 3.3 (Declarations).

Among the fundamental *objects* of the object system are *classes*. A *class* determines the structure and behavior of a set of other *objects*, which are called its *instances*. Every *object* is a *direct instance* of a *class*. The *class* of an *object* determines the set of operations that can be performed on the *object*. For more information, see Section 4.3 (Classes).

It is possible to write *functions* that have behavior *specialized* to the class of the *objects* which are their *arguments*. For more information, see Section 7.6 (Generic Functions and Methods).

4.2 Types

4.2.1 Data Type Definition

Information about *type* usage is located in the sections specified in Figure 4-1. Figure 4-7 lists some *classes* that are particularly relevant to the object system. Figure 9-1 lists the defined *condition types*.

```
Section                                       Data Type
----------
```

Figure 4-1. Cross-References to Data Type Information

4.2.2 Type Relationships

* The *types* **cons**, **symbol**, **array**, **number**, **character**, **hash-table**, **function**, **readtable**, **package**, **pathname**, **stream**, **random-state**, **condition**, **restart**, and any single other *type* created by **defstruct**, **define-condition**, or **defclass** are *pairwise disjoint*, except for type relations explicitly established by specifying *superclasses* in **defclass** or **define-condition** or the `:include` option of **destruct**.
* Any two *types* created by **defstruct** are *disjoint* unless one is a *supertype* of the other by virtue of the **defstruct** `:include` option.

* Any two *distinct classes* created by **defclass** or **define-condition** are *disjoint* unless they have a common *subclass* or one *class* is a *subclass* of the other.
* An implementation may be extended to add other *subtype* relationships between the specified *types*, as long as they do not violate the type relationships and disjointness requirements specified here. An implementation may define additional *types* that are *subtypes* or *supertypes* of any specified *types*, as long as each additional *type* is a *subtype* of *type* **t** and a *supertype* of *type* **nil** and the disjointness requirements are not violated.

At the discretion of the implementation, either **standard-object** or **structure-object** might appear in any class precedence list for a *system class* that does not already specify either **standard-object** or **structure-object**. If it does, it must precede the *class* **t** and follow all other *standardized classes*.

4.2.3 Type Specifiers

Type specifiers can be *symbols*, *classes*, or *lists*. Figure 4-2 lists *symbols* that are *standardized atomic type specifiers*, and Figure 4-3 lists *standardized compound type specifier names*. For syntax information, see the dictionary entry for the corresponding *type specifier*. It is possible to define new *type specifiers* using **defclass**, **define-condition**, **defstruct**, or **deftype**.

```
arithmetic-error                   function             simple-condition
array                              generic-function     simple-error
atom                               hash-table           simple-string
base-char                          integer              simple-type-error
base-string                        keyword              simple-vector
bignum                             list                 simple-warning
bit                                logical-pathname     single-float
bit-vector                         long-float           standard-char
broadcast-stream                   method               standard-class
built-in-class                     method-combination   standard-generic-function
cell-error                         nil                  standard-method
character                          null                 standard-object
class                              number               storage-condition
compiled-function                  package              stream
complex                            package-error        stream-error
concatenated-stream                parse-error          string
condition                          pathname             string-stream
cons                               print-not-readable   structure-class
control-error                      program-error        structure-object
division-by-zero                   random-state         style-warning
double-float                       ratio                symbol
echo-stream                        rational             synonym-stream
end-of-file                        reader-error         t
error                              readtable            two-way-stream
extended-char                      real                 type-error
file-error                         restart              unbound-slot
file-stream                        sequence             unbound-variable
fixnum                             serious-condition    undefined-function
float                              short-float          unsigned-byte
floating-point-inexact             signed-byte          vector
floating-point-invalid-operation   simple-array         warning
floating-point-overflow            simple-base-string
floating-point-underflow           simple-bit-vector
```

Figure 4-2. Standardized Atomic Type Specifiers

If a *type specifier* is a *list*, the *car* of the *list* is a *symbol*, and the rest of the *list* is subsidiary *type* information. Such a *type specifier* is called a *compound type specifier*. Except as explicitly stated otherwise, the subsidiary items can be unspecified. The unspecified subsidiary items are indicated by writing `*`. For example, to completely specify a *vector*, the *type* of the elements and the length of the *vector* must be present.

```
(vector double-float 100)
```

The following leaves the length unspecified:

```
(vector double-float *)
```

The following leaves the element type unspecified:

```
(vector * 100)
```

Suppose that two *type specifiers* are the same except that the first has a `*` where the second has a more explicit specification. Then the second denotes a *subtype* of the *type* denoted by the first.

If a *list* has one or more unspecified items at the end, those items can be dropped. If dropping all occurrences of `*` results in a *singleton list*, then the parentheses can be dropped as well (the list can be replaced by the *symbol* in its *car*). For example, `(vector double-float *)` can be abbreviated to `(vector double-float)`, and `(vector * *)` can be abbreviated to `(vector)` and then to `vector`.

```
and           long-float    simple-base-string
array         member        simple-bit-vector
base-string   mod           simple-string
bit-vector    not           simple-vector
complex       or            single-float
cons          rational      string
double-float  real          unsigned-byte
eql           satisfies     values
float         short-float   vector
function      signed-byte
integer       simple-array
```

Figure 4-3. Standardized Compound Type Specifier Names

The next figure show the *defined names* that can be used as *compound type specifier names* but that cannot be used as *atomic type specifiers*.

```
and     mod  satisfies
eql     not  values
member  or
```

Figure 4-4. Standardized Compound-Only Type Specifier Names

New *type specifiers* can come into existence in two ways.

* Defining a structure by using **defstruct** without using the `:type` specifier or defining a *class* by using **defclass** or **define-condition** automatically causes the name of the structure or class to be a new *type specifier symbol*.
* **deftype** can be used to define *derived type specifiers*, which act as ‘abbreviations’ for other *type specifiers*.

A *class object* can be used as a *type specifier*. When used this way, it denotes the set of all members of that *class*.

The next figure shows some *defined names* relating to *types* and *declarations*.

```
coerce            defstruct  subtypep
declaim           deftype    the
declare           ftype      type
defclass          locally    type-of
define-condition  proclaim   typep
```

Figure 4-5. Defined names relating to types and declarations.

The next figure shows all *defined names* that are *type specifier names*, whether for *atomic type specifiers* or *compound type specifiers*; this list is the union of the lists in Figure 4-2 and Figure 4-3.

`and`	`function`	`simple-array`
`arithmetic-error`	`generic-function`	`simple-base-string`
`array`	`hash-table`	`simple-bit-vector`
`atom`	`integer`	`simple-condition`
`base-char`	`keyword`	`simple-error`
`base-string`	`list`	`simple-string`
`bignum`	`logical-pathname`	`simple-type-error`
`bit`	`long-float`	`simple-vector`
`bit-vector`	`member`	`simple-warning`
`broadcast-stream`	`method`	`single-float`
`built-in-class`	`method-combination`	`standard-char`
`cell-error`	`mod`	`standard-class`
`character`	`nil`	`standard-generic-function`
`class`	`not`	`standard-method`
`compiled-function`	`null`	`standard-object`
`complex`	`number`	`storage-condition`
`concatenated-stream`	`or`	`stream`
`condition`	`package`	`stream-error`
`cons`	`package-error`	`string`
`control-error`	`parse-error`	`string-stream`
`division-by-zero`	`pathname`	`structure-class`
`double-float`	`print-not-readable`	`structure-object`
`echo-stream`	`program-error`	`style-warning`
`end-of-file`	`random-state`	`symbol`
`eql`	`ratio`	`synonym-stream`
`error`	`rational`	`t`
`extended-char`	`reader-error`	`two-way-stream`
`file-error`	`readtable`	`type-error`
`file-stream`	`real`	`unbound-slot`
`fixnum`	`restart`	`unbound-variable`
`float`	`satisfies`	`undefined-function`
`floating-point-inexact`	`sequence`	`unsigned-byte`
`floating-point-invalid-operation`	`serious-condition`	`values`
`floating-point-overflow`	`short-float`	`vector`
`floating-point-underflow`	`signed-byte`	`warning`

Figure 4-6. Standardized Type Specifier Names

4.3 Classes

While the object system is general enough to describe all *standardized classes* (including, for example, **number**, **hash-table**, and **symbol**), the next figure contains a list of *classes* that are especially relevant to understanding the object system.

`built-in-class`	`method-combination`	`standard-object`
`class`	`standard-class`	`structure-class`
`generic-function`	`standard-generic-function`	`structure-object`
`method`	`standard-method`	

Figure 4-7. Object System Classes

4.3.1 Introduction to Classes

A *class* is an *object* that determines the structure and behavior of a set of other *objects*, which are called its *instances*.

A *class* can inherit structure and behavior from other *classes*. A *class* whose definition refers to other *classes* for the purpose of inheriting from them is said to be a *subclass* of each of those *classes*. The *classes* that are designated for purposes of inheritance are said to be *superclasses* of the inheriting *class*.

A *class* can have a *name*. The *function* **class-name** takes a *class object* and returns its *name*. The *name* of an anonymous *class* is **nil**. A *symbol* can *name* a *class*. The *function* **find-class** takes a *symbol* and returns the *class* that the *symbol* names. A *class* has a *proper name* if the *name* is a *symbol* and if the *name* of the *class* names that *class*. That is, a *class* C has the *proper name* S if S= `(class-name C)` and C= `(find-class S)`. Notice that it is possible for `(find-class S1)` = `(find-class S2)` and S1/=S2. If C= `(find-class S)`, we say that C is the *class named* S.

A *class* C1 is a *direct superclass* of a *class* C2 if C2 explicitly designates C1 as a *superclass* in its definition. In this case C2 is a *direct subclass* of C1. A *class* Cn is a *superclass* of a *class* C1 if there exists a series of *classes* C2,...,Cn-1 such that Ci+1 is a *direct superclass* of Ci for 1 <=i<n. In this case, C1 is a *subclass* of Cn. A *class* is considered neither a *superclass* nor a *subclass* of itself. That is, if C1 is a *superclass* of C2, then C1 /=C2. The set of *classes* consisting of some given *class* C along with all of its *superclasses* is called "C and its superclasses."

Each *class* has a *class precedence list*, which is a total ordering on the set of the given *class* and its *superclasses*. The total ordering is expressed as a list ordered from most specific to least specific. The *class precedence list* is used in several ways. In general, more specific *classes* can *shadow*[1] features that would otherwise be inherited from less specific *classes*. The *method* selection and combination process uses the *class precedence list* to order *methods* from most specific to least specific.

When a *class* is defined, the order in which its direct *superclasses* are mentioned in the defining form is important. Each *class* has a *local precedence order*, which is a *list* consisting of the *class* followed by its *direct superclasses* in the order mentioned in the defining *form*.

A *class precedence list* is always consistent with the *local precedence order* of each *class* in the list. The *classes* in each *local precedence order* appear within the *class precedence list* in the same order. If the *local precedence orders* are inconsistent with each other, no *class precedence list* can be constructed, and an error is signaled. The *class precedence list* and its computation is discussed in Section 4.3.5 (Determining the Class Precedence List).

classes are organized into a directed acyclic graph. There are two distinguished *classes*, named **t** and **standard-object**. The *class* named **t** has no *superclasses*. It is a *superclass* of every *class* except itself. The *class* named **standard-object** is an *instance* of the *class* **standard-class** and is a *superclass* of every *class* that is an *instance* of the *class* **standard-class** except itself.

There is a mapping from the object system *class* space into the *type* space. Many of the standard *types* specified in this document have a corresponding *class* that has the same *name* as the *type*. Some *types* do not have a corresponding *class*. The integration of the *type* and *class* systems is discussed in Section 4.3.7 (Integrating Types and Classes).

Classes are represented by *objects* that are themselves *instances* of *classes*. The *class* of the *class* of an *object* is termed the *metaclass* of that *object*. When no misinterpretation is possible, the term *metaclass* is used to refer to a *class* that has *instances* that are themselves *classes*. The *metaclass* determines the form of inheritance used by the *classes* that are its *instances* and the representation of the *instances* of those *classes*. The object system provides a default *metaclass*, **standard-class**, that is appropriate for most programs.

Except where otherwise specified, all *classes* mentioned in this standard are *instances* of the *class* **standard-class**, all *generic functions* are *instances* of the *class* **standard-generic-function**, and all *methods* are *instances* of the *class* **standard-method**.

4.3.1.1 Standard Metaclasses

The object system provides a number of predefined *metaclasses*. These include the *classes* **standard-class**, **built-in-class**, and **structure-class**:

* The *class* **standard-class** is the default *class* of *classes* defined by **defclass**.
* The *class* **built-in-class** is the *class* whose *instances* are *classes* that have special implementations with restricted capabilities. Any *class* that corresponds to a standard *type* might be an *instance* of **built-in-class**. The predefined *type* specifiers that are required to have corresponding *classes* are listed in Figure 4-8. It is *implementation-dependent* whether each of these *classes* is implemented as a *built-in class*.
* All *classes* defined by means of **defstruct** are *instances* of the *class* **structure-class**.

4.3.2 Defining Classes

The macro **defclass** is used to define a new named *class*.

The definition of a *class* includes:

* The *name* of the new *class*. For newly-defined *classes* this *name* is a *proper name*.
* The list of the direct *superclasses* of the new *class*.
* A set of *slot specifiers*. Each *slot specifier* includes the *name* of the *slot* and zero or more *slot* options. A *slot* option pertains only to a single *slot*. If a *class* definition contains two *slot specifiers* with the same *name*, an error is signaled.
* A set of *class* options. Each *class* option pertains to the *class* as a whole.

The *slot* options and *class* options of the **defclass** form provide mechanisms for the following:

* Supplying a default initial value *form* for a given *slot*.
* Requesting that *methods* for *generic functions* be automatically generated for reading or writing *slots*.
* Controlling whether a given *slot* is shared by all *instances* of the *class* or whether each *instance* of the *class* has its own *slot*.
* Supplying a set of initialization arguments and initialization argument defaults to be used in *instance* creation.
* Indicating that the *metaclass* is to be other than the default. The `:metaclass` option is reserved for future use; an implementation can be extended to make use of the `:metaclass` option.
* Indicating the expected *type* for the value stored in the *slot*.
* Indicating the *documentation string* for the *slot*.

4.3.3 Creating Instances of Classes

The generic function **make-instance** creates and returns a new *instance* of a *class*. The object system provides several mechanisms for specifying how a new *instance* is to be initialized. For example, it is possible to specify the initial values for *slots* in newly created *instances* either by giving arguments to **make-instance** or by providing default initial values. Further initialization activities can be performed by *methods* written for *generic functions* that are part of the initialization protocol. The complete initialization protocol is described in Section 7.1 (Object Creation and Initialization).

4.3.4 Inheritance

A *class* can inherit *methods*, *slots*, and some **defclass** options from its *superclasses*. Other sections describe the inheritance of *methods*, the inheritance of *slots* and *slot* options, and the inheritance of *class* options.

4.3.4.1 Examples of Inheritance

```
(defclass C1 ()
    ((S1 :initform 5.4 :type number)
     (S2 :allocation :class)))

(defclass C2 (C1)
    ((S1 :initform 5 :type integer)
     (S2 :allocation :instance)
     (S3 :accessor C2-S3)))
```

Instances of the class `C1` have a *local slot* named `S1`, whose default initial value is 5.4 and whose *value* should always be a *number*. The class `C1` also has a *shared slot* named `S2`.

There is a *local slot* named `S1` in *instances* of `C2`. The default initial value of `S1` is 5. The value of `S1` should always be of type `(and integer number)`. There are also *local slots* named `S2` and `S3` in *instances* of `C2`. The class `C2` has a *method* for `C2-S3` for reading the value of slot `S3`; there is also a *method* for `(setf C2-S3)` that writes the value of `S3`.

4.3.4.2 Inheritance of Class Options

The `:default-initargs` class option is inherited. The set of defaulted initialization arguments for a *class* is the union of the sets of initialization arguments supplied in the `:default-initargs` class options of the *class* and its *superclasses*. When more than one default initial value *form* is supplied for a given initialization argument, the default initial value *form* that is used is the one supplied by the *class* that is most specific according to the *class precedence list*.

If a given `:default-initargs` class option specifies an initialization argument of the same *name* more than once, an error of *type* **program-error** is signaled.

4.3.5 Determining the Class Precedence List

The **defclass** form for a *class* provides a total ordering on that *class* and its direct *superclasses*. This ordering is called the *local precedence order*. It is an ordered list of the *class* and its direct *superclasses*. The *class precedence list* for a class C is a total ordering on C and its *superclasses* that is consistent with the *local precedence orders* for each of C and its *superclasses*.

A *class* precedes its direct *superclasses*, and a direct *superclass* precedes all other direct *superclasses* specified to its right in the *superclasses* list of the **defclass** form. For every class C, define

RC={(C,C1),(C1,C2),...,(Cn-1,Cn)}

where C1,...,Cn are the direct *superclasses* of C in the order in which they are mentioned in the **defclass** form. These ordered pairs generate the total ordering on the class C and its direct *superclasses*.

Let SC be the set of C and its *superclasses*. Let R be

R=Uc<ELEMENT-OF>SCRc

.

The set R might or might not generate a partial ordering, depending on whether the Rc, c<ELEMENT-OF>SC, are consistent; it is assumed that they are consistent and that R generates a partial ordering. When the Rc are not consistent, it is said that R is inconsistent.

To compute the *class precedence list* for C, topologically sort the elements of SC with respect to the partial ordering generated by R. When the topological sort must select a *class* from a set of two or more *classes*, none of which are preceded by other *classes* with respect to R, the *class* selected is chosen deterministically, as described below.

If R is inconsistent, an error is signaled.

4.3.5.1 Topological Sorting

Topological sorting proceeds by finding a class C in SC such that no other *class* precedes that element according to the elements in R. The class C is placed first in the result. Remove C from SC, and remove all pairs of the form (C,D), D<ELEMENT-OF>SC, from R. Repeat the process, adding *classes* with no predecessors to the end of the result. Stop when no element can be found that has no predecessor.

If SC is not empty and the process has stopped, the set R is inconsistent. If every *class* in the finite set of *classes* is preceded by another, then R contains a loop. That is, there is a chain of classes C1,...,Cn such that Ci precedes Ci+1, 1<=i<n, and Cn precedes C1.

Sometimes there are several *classes* from SC with no predecessors. In this case select the one that has a direct *subclass* rightmost in the *class precedence list* computed so far. (If there is no such candidate *class*, R does not generate a partial ordering---the Rc, c<ELEMENT-OF>SC, are inconsistent.)

In more precise terms, let {N1,...,Nm}, m>=2, be the *classes* from SC with no predecessors. Let (C1...Cn), n>=1, be the *class precedence list* constructed so far. C1 is the most specific *class*, and Cn is the least specific. Let 1<=j<=n be the largest number such that there exists an i where 1<=i<=m and Ni is a direct *superclass* of Cj; Ni is placed next.

The effect of this rule for selecting from a set of *classes* with no predecessors is that the *classes* in a simple *superclass* chain are adjacent in the *class precedence list* and that *classes* in each relatively separated subgraph are adjacent in the *class precedence list*. For example, let T1 and T2 be subgraphs whose only element in common is the class J. Suppose that no superclass of J appears in either T1 or T2, and that J is in the superclass chain of every class in both T1 and T2. Let C1 be the bottom of T1; and let C2 be the bottom of T2. Suppose C is a *class* whose direct *superclasses* are C1 and C2 in that order, then the *class precedence list* for C starts with C and is followed by all *classes* in T1 except J. All the *classes* of T2 are next. The *class* J and its *superclasses* appear last.

4.3.5.2 Examples of Class Precedence List Determination

This example determines a *class precedence list* for the class `pie`. The following *classes* are defined:

```
(defclass pie (apple cinnamon) ())

(defclass apple (fruit) ())

(defclass cinnamon (spice) ())

(defclass fruit (food) ())

(defclass spice (food) ())

(defclass food () ())
```

The set Spie = {`pie, apple, cinnamon, fruit, spice, food, standard-object, t`}. The set R = {`(pie, apple), (apple, cinnamon), (apple, fruit), (cinnamon, spice), (fruit, food), (spice, food), (food, standard-object), (standard-object, t)`}.

The class `pie` is not preceded by anything, so it comes first; the result so far is `(pie)`. Remove `pie` from S and pairs mentioning `pie` from R to get S = {`apple, cinnamon, fruit, spice, food, standard-object, t`} and R = {`(apple, cinnamon), (apple, fruit), (cinnamon, spice), (fruit, food), (spice, food), (food, standard-object), (standard-object, t)`}.

The class `apple` is not preceded by anything, so it is next; the result is `(pie apple)`. Removing `apple` and the relevant pairs results in S = {`cinnamon, fruit, spice, food, standard-object, t`} and R = {`(cinnamon, spice), (fruit, food), (spice, food), (food, standard-object), (standard-object, t)`}.

The classes `cinnamon` and `fruit` are not preceded by anything, so the one with a direct *subclass* rightmost in the *class precedence list* computed so far goes next. The class `apple` is a direct *subclass* of `fruit`, and the class `pie` is a direct *subclass* of `cinnamon`. Because `apple` appears to the right of `pie` in the *class precedence list*, `fruit` goes next, and the result so far is `(pie apple fruit)`. S = {`cinnamon, spice, food, standard-object, t`}; R = {`(cinnamon, spice), (spice, food), (food, standard-object), (standard-object, t)`}.

The class `cinnamon` is next, giving the result so far as `(pie apple fruit cinnamon)`. At this point S = {`spice, food, standard-object, t`}; R = {`(spice, food), (food, standard-object), (standard-object, t)`}.

The classes `spice`, `food`, **standard-object**, and **t** are added in that order, and the *class precedence list* is `(pie apple fruit cinnamon spice food standard-object t)`.

It is possible to write a set of *class* definitions that cannot be ordered. For example:

```
(defclass new-class (fruit apple) ())

(defclass apple (fruit) ())
```

The class `fruit` must precede `apple` because the local ordering of *superclasses* must be preserved. The class `apple` must precede `fruit` because a *class* always precedes its own *superclasses*. When this situation occurs, an error is signaled, as happens here when the system tries to compute the *class precedence list* of `new-class`.

The following might appear to be a conflicting set of definitions:

```
(defclass pie (apple cinnamon) ())

(defclass pastry (cinnamon apple) ())

(defclass apple () ())

(defclass cinnamon () ())
```

The *class precedence list* for `pie` is `(pie apple cinnamon standard-object t)`.

The *class precedence list* for `pastry` is `(pastry cinnamon apple standard-object t)`.

It is not a problem for `apple` to precede `cinnamon` in the ordering of the *superclasses* of `pie` but not in the ordering for `pastry`. However, it is not possible to build a new *class* that has both `pie` and `pastry` as *superclasses*.

4.3.6 Redefining Classes

A *class* that is a *direct instance* of **standard-class** can be redefined if the new *class* is also a *direct instance* of **standard-class**. Redefining a *class* modifies the existing *class object* to reflect the new *class* definition; it does not create a new *class object* for the *class*. Any *method object* created by a `:reader`, `:writer`, or `:accessor` option specified by the old **defclass** form is removed from the corresponding *generic function*. *Methods* specified by the new **defclass** form are added.

When the class C is redefined, changes are propagated to its *instances* and to *instances* of any of its *subclasses*. Updating such an *instance* occurs at an *implementation-dependent* time, but no later than the next time a *slot* of that *instance* is read or written. Updating an *instance* does not change its identity as defined by the *function* **eq**. The updating process may change the *slots* of that particular *instance*, but it does not create a new *instance*. Whether updating an *instance* consumes storage is *implementation-dependent*.

Note that redefining a *class* may cause *slots* to be added or deleted. If a *class* is redefined in a way that changes the set of *local slots accessible* in *instances*, the *instances* are updated. It is *implementation-dependent* whether *instances* are updated if a *class* is redefined in a way that does not change the set of *local slots accessible* in *instances*.

The value of a *slot* that is specified as shared both in the old *class* and in the new *class* is retained. If such a *shared slot* was unbound in the old *class*, it is unbound in the new *class*. *Slots* that were local in the old *class* and that are shared in the new *class* are initialized. Newly added *shared slots* are initialized.

Each newly added *shared slot* is set to the result of evaluating the *captured initialization form* for the *slot* that was specified in the **defclass** *form* for the new *class*. If there was no *initialization form*, the *slot* is unbound.

If a *class* is redefined in such a way that the set of *local slots accessible* in an *instance* of the *class* is changed, a two-step process of updating the *instances* of the *class* takes place. The process may be explicitly started by invoking the generic function **make-instances-obsolete**. This two-step process can happen in other circumstances in some implementations. For example, in some implementations this two-step process is triggered if the order of *slots* in storage is changed.

The first step modifies the structure of the *instance* by adding new *local slots* and discarding *local slots* that are not defined in the new version of the *class*. The second step initializes the newly-added *local slots* and performs any other user-defined actions. These two steps are further specified in the next two sections.

4.3.6.1 Modifying the Structure of Instances

The first step modifies the structure of *instances* of the redefined *class* to conform to its new *class* definition. *Local slots* specified by the new *class* definition that are not specified as either local or shared by the old *class* are added, and *slots* not specified as either local or shared by the new *class* definition that are specified as local by the old *class* are discarded. The *names* of these added and discarded *slots* are passed as arguments to **update-instance-for-redefined-class** as described in the next section.

The values of *local slots* specified by both the new and old *classes* are retained. If such a *local slot* was unbound, it remains unbound.

The value of a *slot* that is specified as shared in the old *class* and as local in the new *class* is retained. If such a *shared slot* was unbound, the *local slot* is unbound.

4.3.6.2 Initializing Newly Added Local Slots

The second step initializes the newly added *local slots* and performs any other user-defined actions. This step is implemented by the generic function **update-instance-for-redefined-class**, which is called after completion of the first step of modifying the structure of the *instance*.

The generic function **update-instance-for-redefined-class** takes four required arguments: the *instance* being updated after it has undergone the first step, a list of the names of *local slots* that were added, a list of the names of *local slots* that were discarded, and a property list containing the *slot* names and values of *slots* that were discarded and had values. Included among the discarded *slots* are *slots* that were local in the old *class* and that are shared in the new *class*.

The generic function **update-instance-for-redefined-class** also takes any number of initialization arguments. When it is called by the system to update an *instance* whose *class* has been redefined, no initialization arguments are provided.

There is a system-supplied primary *method* for **update-instance-for-redefined-class** whose *parameter specializer* for its *instance* argument is the *class* **standard-object**. First this *method* checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared as valid. (For more information, see Section 7.1.2 (Declaring the Validity of Initialization Arguments).) Then it calls the generic function **shared-initialize** with the following arguments: the *instance*, the list of *names* of the newly added *slots*, and the initialization arguments it received.

4.3.6.3 Customizing Class Redefinition

Methods for **update-instance-for-redefined-class** may be defined to specify actions to be taken when an *instance* is updated. If only *after methods* for **update-instance-for-redefined-class** are defined, they will be run after the system-supplied primary *method* for initialization and therefore will not interfere with the default behavior of **update-instance-for-redefined-class**. Because no initialization arguments are passed to **update-instance-for-redefined-class** when it is called by the system, the *initialization forms* for *slots* that are filled by *before methods* for **update-instance-for-redefined-class** will not be evaluated by **shared-initialize**.

Methods for **shared-initialize** may be defined to customize *class* redefinition. For more information, see Section 7.1.5 (Shared-Initialize).

4.3.7 Integrating Types and Classes

The object system maps the space of *classes* into the space of *types*. Every *class* that has a proper name has a corresponding *type* with the same *name*.

The proper name of every *class* is a valid *type specifier*. In addition, every *class object* is a valid *type specifier*. Thus the expression `(typep` *object class*`)` evaluates to *true* if the *class* of *object* is *class* itself or a *subclass* of *class*. The evaluation of the expression `(subtypep class1 class2)` returns the values *true* and *true* if `class1` is a subclass of `class2` or if they are the same *class*; otherwise it returns the values *false* and *true*. If I is an *instance* of some *class* C named S and C is an *instance* of **standard-class**, the evaluation of the expression `(type-of I)` returns S if S is the *proper name* of C; otherwise, it returns C.

Because the names of *classes* and *class objects* are *type specifiers*, they may be used in the special form **the** and in type declarations.

Many but not all of the predefined *type specifiers* have a corresponding *class* with the same proper name as the *type*. These type specifiers are listed in Figure 4-8. For example, the *type* **array** has a corresponding *class* named **array**. No *type specifier* that is a list, such as `(vector double-float 100)`, has a corresponding *class*. The *operator* **deftype** does not create any *classes*.

Each *class* that corresponds to a predefined *type specifier* can be implemented in one of three ways, at the discretion of each implementation. It can be a *standard class*, a *structure class*, or a *system class*.

A *built-in class* is one whose *generalized instances* have restricted capabilities or special representations. Attempting to use **defclass** to define *subclasses* of a **built-in-class** signals an error. Calling **make-instance** to create a *generalized instance* of a *built-in class* signals an error. Calling **slot-value** on a *generalized instance* of a *built-in class* signals an error. Redefining a *built-in class* or using **change-class** to change the *class* of an *object* to or from a *built-in class* signals an error. However, *built-in classes* can be used as *parameter specializers* in *methods*.

It is possible to determine whether a *class* is a *built-in class* by checking the *metaclass*. A *standard class* is an *instance* of the *class* **standard-class**, a *built-in class* is an *instance* of the *class* **built-in-class**, and a *structure class* is an *instance* of the *class* **structure-class**.

Each *structure type* created by **defstruct** without using the `:type` option has a corresponding *class*. This *class* is a *generalized instance* of the *class* **structure-class**. The `:include` option of **defstruct** creates a direct *subclass* of the *class* that corresponds to the included *structure type*.

It is *implementation-dependent* whether *slots* are involved in the operation of *functions* defined in this specification on *instances* of *classes* defined in this specification, except when *slots* are explicitly defined by this specification.

If in a particular *implementation* a *class* defined in this specification has *slots* that are not defined by this specfication, the names of these *slots* must not be *external symbols* of *packages* defined in this specification nor otherwise *accessible* in the `CL-USER` package.

The purpose of specifying that many of the standard *type specifiers* have a corresponding *class* is to enable users to write *methods* that discriminate on these *types*. *Method* selection requires that a *class precedence list* can be determined for each *class*.

The hierarchical relationships among the *type specifiers* are mirrored by relationships among the *classes* corresponding to those *types*.

Figure 4-8 lists the set of *classes* that correspond to predefined *type specifiers*.

arithmetic-error	generic-function	simple-error
array	hash-table	simple-type-error
bit-vector	integer	simple-warning
broadcast-stream	list	standard-class
built-in-class	logical-pathname	standard-generic-function
cell-error	method	standard-method
character	method-combination	standard-object
class	null	storage-condition
complex	number	stream
concatenated-stream	package	stream-error
condition	package-error	string
cons	parse-error	string-stream
control-error	pathname	structure-class
division-by-zero	print-not-readable	structure-object
echo-stream	program-error	style-warning
end-of-file	random-state	symbol
error	ratio	synonym-stream
file-error	rational	t
file-stream	reader-error	two-way-stream
float	readtable	type-error
floating-point-inexact	real	unbound-slot
floating-point-invalid-operation	restart	unbound-variable
floating-point-overflow	sequence	undefined-function
floating-point-underflow	serious-condition	vector
function	simple-condition	warning

Figure 4-8. Classes that correspond to pre-defined type specifiers

The *class precedence list* information specified in the entries for each of these *classes* are those that are required by the object system.

Individual implementations may be extended to define other type specifiers to have a corresponding *class*. Individual implementations may be extended to add other *subclass* relationships and to add other *elements* to the *class precedence lists* as long as they do not violate the type relationships and disjointness requirements specified by this standard. A standard *class* defined with no direct *superclasses* is guaranteed to be disjoint from all of the *classes* in the table, except for the class named **t**.