CHAPTER 9

Interfaces

My apple trees will never get across
And eat the cones under his pines, I tell him.
He only says "Good Fences Make Good Neighbors."
—Robert Frost, *Mending Wall* (1914)

AN interface declaration introduces a new reference type whose members are classes, interfaces, constants and abstract methods. This type has no implementation, but otherwise unrelated classes can implement it by providing implementations for its abstract methods.

A *nested interface* is any interface whose declaration occurs within the body of another class or interface. A *top-level interface* is an interface that is not a nested interface.

We distinguish between two kinds of interfaces - normal interfaces and annotation types.

This chapter discusses the common semantics of all interfaces—normal interfaces and annotation types (§9.6), top-level (§7.6) and nested (§8.5, §9.5). Details that are specific to particular kinds of interfaces are discussed in the sections dedicated to these constructs.

Programs can use interfaces to make it unnecessary for related classes to share a common abstract superclass or to add methods to `Object`.

An interface may be declared to be a *direct extension* of one or more other interfaces, meaning that it implicitly specifies all the member types, abstract methods and constants of the interfaces it extends, except for any member types and constants that it may hide.

A class may be declared to *directly implement* one or more interfaces, meaning that any instance of the class implements all the abstract methods specified by the interface or interfaces. A class necessarily implements all the interfaces that its direct superclasses and direct superinterfaces do. This (multiple) interface inheritance allows objects to support (multiple) common behaviors without sharing any implementation.

A variable whose declared type is an interface type may have as its value a reference to any instance of a class which implements the specified interface. It is not sufficient that the class happen to implement all the abstract methods of the interface; the class or one of its superclasses must actually be declared to implement the interface, or else the class is not considered to implement the interface.

9.1 Interface Declarations

An *interface declaration* specifies a new named reference type. There are two kinds of interface declarations - *normal interface declarations* and *annotation type declarations*:

InterfaceDeclaration:
NormalInterfaceDeclaration
AnnotationTypeDeclaration

NormalInterfaceDeclaration:
InterfaceModifiers$_{opt}$ `interface` *Identifier* *TypeParameters*$_{opt}$ *ExtendsInterfaces*$_{opt}$ *InterfaceBody*

The *Identifier* in an interface declaration specifies the name of the interface. A compile-time error occurs if an interface has the same simple name as any of its enclosing classes or interfaces.

9.1.1 Interface Modifiers

An interface declaration may include *interface modifiers*:

InterfaceModifiers:
InterfaceModifier
InterfaceModifiers *InterfaceModifier*

InterfaceModifier: one of
Annotation `public` `protected` `private`
`abstract` `static` `strictfp`

The access modifier `public` is discussed in §6.6. Not all modifiers are applicable to all kinds of interface declarations. The access modifiers `protected` and `private` pertain only to member interfaces within a directly enclosing class declaration (§8.5) and are discussed in §8.5.1. The access modifier `static` pertains only to member interfaces (§8.5, §9.5). A compile-time error occurs if the same modifier appears more than once in an interface declaration. If an annotation *a* on an interface declaration corresponds to an annotation type *T*, and *T* has a (meta-)annotation *m* that corresponds to `annotation.Target`, then *m* must have an ele-

ment whose value is `annotation.ElementType.TYPE`, or a compile-time error occurs. Annotation modifiers are described further in (§9.7).

9.1.1.1 `abstract` *Interfaces*

Every interface is implicitly `abstract`. This modifier is obsolete and should not be used in new programs.

9.1.1.2 `strictfp` *Interfaces*

The effect of the `strictfp` modifier is to make all `float` or `double` expressions within the interface declaration be explicitly FP-strict (§15.4).

This implies that all nested types declared in the interface are implicitly `strictfp`.

9.1.2 Generic Interfaces and Type Parameters

An interface is *generic* if it declares one or more type variables (§4.4). These type variables are known as the *type parameters* of the interface. The type parameter section follows the interface name and is delimited by angle brackets. It defines one or more type variables that act as parameters. A parameterized interface declaration defines a set of types, one for each possible invocation of the type parameter section. All parameterized types share the same interface at runtime.

The scope of an interface's type parameter is the entire declaration of the interface including the type parameter section itself. Therefore, type parameters can appear as parts of their own bounds, or as bounds of other type parameters declared in the same section.

It is a compile-time error to refer to a type parameter of an interface *I* anywhere in the declaration of a field or type member of *I*.

9.1.3 Superinterfaces and Subinterfaces

If an `extends` clause is provided, then the interface being declared extends each of the other named interfaces and therefore inherits the member types, methods, and constants of each of the other named interfaces. These other named interfaces are the *direct superinterfaces* of the interface being declared. Any class that `implements` the declared interface is also considered to implement all the interfaces that this interface `extends`.

ExtendsInterfaces:
`extends` *InterfaceType*
ExtendsInterfaces , *InterfaceType*

The following is repeated from §4.2 to make the presentation here clearer:

InterfaceType:
TypeName

Given a (possibly generic) interface declaration for `I<A1,...,An>`, $n \geq 0$, the *direct superinterfaces* of the interface type (§4.5) `I<`A_1`,...,`A_n`>` are the types given in the extends clause of the declaration of `I` if an extends clause is present.

Let `I<A1,...,An>`, $n > 0$, be a generic interface declaration. The direct superinterfaces of the parameterized interface type `I<`T_1`,...,`T_n`>`, where T_i, `1 <= i <= n`, is a type, are all types `J<`U_1 `theta , ..., `U_k `theta>`, where `J<`U_1`,...,`U_k`>` is a direct superinterface of `I<`A_1`,...,`A_n`>`, and theta is the substitution [A_1 := T_1, ..., A_n := T_n].

Each *InterfaceType* in the `extends` clause of an interface declaration must name an accessible interface type; otherwise a compile-time error occurs.

An interface `I` *directly depends* on a type `T` if `T` is mentioned in the `extends` clause of `I` either as a superinterface or as a qualifier within a superinterface name. An interface `I` *depends* on a reference type `T` if any of the following conditions hold:

- `I` directly depends on `T`.
- `I` directly depends on a class `C` that depends (§8.1.3) on `T`.
- `I` directly depends on an interface `J` that depends on `T` (using this definition recursively).

A compile-time error occurs if an interface depends on itself.

While every class is an extension of class `Object`, there is no single interface of which all interfaces are extensions.

The *superinterface* relationship is the transitive closure of the direct superinterface relationship. An interface `K` is a superinterface of interface `I` if either of the following is true:

- `K` is a direct superinterface of `I`.
- There exists an interface `J` such that `K` is a superinterface of `J`, and `J` is a superinterface of `I`, applying this definition recursively.

Interface `I` is said to be a *subinterface* of interface `K` whenever `K` is a superinterface of `I`.

9.1.4 Interface Body and Member Declarations

The body of an interface may declare members of the interface:

InterfaceBody:
`{` *InterfaceMemberDeclarations*$_{opt}$ `}`

InterfaceMemberDeclarations:
InterfaceMemberDeclaration
InterfaceMemberDeclarations InterfaceMemberDeclaration

InterfaceMemberDeclaration:
ConstantDeclaration
AbstractMethodDeclaration
ClassDeclaration
InterfaceDeclaration
`;`

The scope of the declaration of a member *m* declared in or inherited by an interface type *I* is the entire body of *I*, including any nested type declarations.

9.1.5 Access to Interface Member Names

All interface members are implicitly `public`. They are accessible outside the package where the interface is declared if the interface is also declared `public` or `protected`, in accordance with the rules of §6.6.

9.2 Interface Members

The members of an interface are:

- Those members declared in the interface.
- Those members inherited from direct superinterfaces.
- If an interface has no direct superinterfaces, then the interface implicitly declares a public abstract member method *m* with signature *s*, return type *r*, and `throws` clause *t* corresponding to each public instance method *m* with signature *s*, return type *r*, and `throws` clause *t* declared in `Object`, unless a method with the same signature, same return type, and a compatible `throws` clause is explicitly declared by the interface. It is a compile-time error if the interface explicitly declares such a method *m* in the case where *m* is declared to be `final` in `Object`.

It follows that is a compile-time error if the interface declares a method with a signature that is override-equivalent (§8.4.2) to a public method of `Object`, but has a different return type or incompatible `throws` clause.

The interface inherits, from the interfaces it extends, all members of those interfaces, except for fields, classes, and interfaces that it hides and methods that it overrides.

9.3 Field (Constant) Declarations

The materials of action are variable,
but the use we make of them should be constant.
—Epictetus (circa 60 A.D.),
translated by Thomas Wentworth Higginson

ConstantDeclaration:
ConstantModifiers$_{opt}$ *Type VariableDeclarators* ;

ConstantModifiers:
ConstantModifier
ConstantModifier ConstantModifers

ConstantModifier: one of
Annotation `public static final`

Every field declaration in the body of an interface is implicitly `public`, `static`, and `final`. It is permitted to redundantly specify any or all of these modifiers for such fields.

If an annotation *a* on a field declaration corresponds to an annotation type *T*, and *T* has a (meta-)annotation *m* that corresponds to `annotation.Target`, then *m* must have an element whose value is `annotation.ElementType.FIELD`, or a compile-time error occurs. Annotation modifiers are described further in (§9.7).

If the interface declares a field with a certain name, then the declaration of that field is said to *hide* any and all accessible declarations of fields with the same name in superinterfaces of the interface.

It is a compile-time error for the body of an interface declaration to declare two fields with the same name.

It is possible for an interface to inherit more than one field with the same name (§8.3.3.3). Such a situation does not in itself cause a compile-time error. However, any attempt within the body of the interface to refer to either field by its simple name will result in a compile-time error, because such a reference is ambiguous.

There might be several paths by which the same field declaration might be inherited from an interface. In such a situation, the field is considered to be inherited only once, and it may be referred to by its simple name without ambiguity.

9.3.1 Initialization of Fields in Interfaces

Every field in the body of an interface must have an initialization expression, which need not be a constant expression. The variable initializer is evaluated and the assignment performed exactly once, when the interface is initialized (§12.4).

A compile-time error occurs if an initialization expression for an interface field contains a reference by simple name to the same field or to another field whose declaration occurs textually later in the same interface.

Thus:

```
interface Test {
    float f = j;
    int j = 1;
    int k = k+1;
}
```

causes two compile-time errors, because `j` is referred to in the initialization of `f` before `j` is declared and because the initialization of `k` refers to `k` itself.

One subtlety here is that, at run time, fields that are initialized with compile-time constant values are initialized first. This applies also to `static final` fields in classes (§8.3.2.1). This means, in particular, that these fields will never be observed to have their default initial values (§4.5.5), even by devious programs. See §12.4.2 and §13.4.8 for more discussion.

If the keyword `this` (§15.8.3) or the keyword `super` (15.11.2, 15.12) occurs in an initialization expression for a field of an interface, then unless the occurrence is within the body of an anonymous class (§15.9.5), a compile-time error occurs.

9.3.2 Examples of Field Declarations

The following example illustrates some (possibly subtle) points about field declarations.

9.3.2.1 *Ambiguous Inherited Fields*

If two fields with the same name are inherited by an interface because, for example, two of its direct superinterfaces declare fields with that name, then a single *ambiguous member* results. Any use of this ambiguous member will result in a compile-time error. Thus in the example:

```
interface BaseColors {
    int RED = 1, GREEN = 2, BLUE = 4;
}
interface RainbowColors extends BaseColors {
    int YELLOW = 3, ORANGE = 5, INDIGO = 6, VIOLET = 7;
}
```

```
interface PrintColors extends BaseColors {
   int YELLOW = 8, CYAN = 16, MAGENTA = 32;
}
interface LotsOfColors extends RainbowColors, PrintColors {
   int FUCHSIA = 17, VERMILION = 43, CHARTREUSE = RED+90;
}
```

the interface `LotsOfColors` inherits two fields named `YELLOW`. This is all right as long as the interface does not contain any reference by simple name to the field `YELLOW`. (Such a reference could occur within a variable initializer for a field.)

Even if interface `PrintColors` were to give the value `3` to `YELLOW` rather than the value `8`, a reference to field `YELLOW` within interface `LotsOfColors` would still be considered ambiguous.

9.3.2.2 *Multiply Inherited Fields*

If a single field is inherited multiple times from the same interface because, for example, both this interface and one of this interface's direct superinterfaces extend the interface that declares the field, then only a single member results. This situation does not in itself cause a compile-time error.

In the example in the previous section, the fields `RED`, `GREEN`, and `BLUE` are inherited by interface `LotsOfColors` in more than one way, through interface `RainbowColors` and also through interface `PrintColors`, but the reference to field `RED` in interface `LotsOfColors` is not considered ambiguous because only one actual declaration of the field `RED` is involved.

9.4 Abstract Method Declarations

AbstractMethodDeclaration:
    AbstractMethodModifiers$_{opt}$ *TypeParameters*$_{opt}$ *ResultType* *MethodDeclarator* *Throws*$_{opt}$ `;`

AbstractMethodModifiers:
    AbstractMethodModifier
    AbstractMethodModifiers *AbstractMethodModifier*

AbstractMethodModifier: one of
    Annotation `public` `abstract`

The access modifier `public` is discussed in §6.6. A compile-time error occurs if the same modifier appears more than once in an abstract method declaration.

Every method declaration in the body of an interface is implicitly `abstract`, so its body is always represented by a semicolon, not a block.

Every method declaration in the body of an interface is implicitly `public`.

For compatibility with older versions of the Java platform, it is permitted but discouraged, as a matter of style, to redundantly specify the `abstract` modifier for methods declared in interfaces.

It is permitted, but strongly discouraged as a matter of style, to redundantly specify the `public` modifier for interface methods.

Note that a method declared in an interface must not be declared `static`, or a compile-time error occurs, because `static` methods cannot be `abstract`.

Note that a method declared in an interface must not be declared `strictfp` or `native` or `synchronized`, or a compile-time error occurs, because those keywords describe implementation properties rather than interface properties. However, a method declared in an interface may be implemented by a method that is declared `strictfp` or `native` or `synchronized` in a class that implements the interface.

If an annotation *a* on a method declaration corresponds to an annotation type *T*, and *T* has a (meta-)annotation *m* that corresponds to `annotation.Target`, then *m* must have an element whose value is `annotation.ElementType.METHOD`, or a compile-time error occurs. Annotation modifiers are described further in (§9.7).

It is a compile-time error for the body of an interface to declare, explicitly or implicitly, two methods with override-equivalent signatures (§8.4.2). However, an interface may inherit several methods with such signatures (§9.4.1).

Note that a method declared in an interface must not be declared `final` or a compile-time error occurs. However, a method declared in an interface may be implemented by a method that is declared `final` in a class that implements the interface.

A method in an interface may be generic. The rules for formal type parameters of a generic method in an interface are the same as for a generic method in a class (§8.4.4).

9.4.1 Inheritance and Overriding

An instance method *m1* declared in an interface *I* *overrides* another instance method, *m2*, declared in interface *J* iff both of the following are true:

1. *I* is a subinterface of *J*.
2. The signature of *m1* is a subsignature (§8.4.2) of the signature of *m*2.

If a method declaration d_1 with return type R_1 overrides or hides the declaration of another method d_2 with return type R_2, then d_1 must be return-type substitutable

(§8.4.5) for d_2, or a compile-time error occurs. Furthermore, if R_1 is not a subtype of R_2, an unchecked warning must be issued.

Moreover, a method declaration must not have a `throws` clause that conflicts (§8.4.4) with that of any method that it overrides; otherwise, a compile-time error occurs.

It is a compile time error if a type declaration T has a member method m_1 and there exists a method m_2 declared in T or a supertype of T such that all of the following conditions hold:

- m_1 and m_2 have the same name.
- m_2 is accessible from T.
- The signature of m_1 is not a subsignature (§8.4.2) of the signature of m_2.
- m_1 or some method m_1 overrides (directly or indirectly) has the same erasure as m_2 or some method m_2 overrides (directly or indirectly).

Methods are overridden on a signature-by-signature basis. If, for example, an interface declares two `public` methods with the same name, and a subinterface overrides one of them, the subinterface still inherits the other method.

An interface inherits from its direct superinterfaces all methods of the superinterfaces that are not overridden by a declaration in the interface.

It is possible for an interface to inherit several methods with override-equivalent signatures (§8.4.2). Such a situation does not in itself cause a compile-time error. The interface is considered to inherit all the methods. However, one of the inherited methods must must be return type substitutable for any other inherited method; otherwise, a compile-time error occurs (The `throws` clauses do not cause errors in this case.)

There might be several paths by which the same method declaration is inherited from an interface. This fact causes no difficulty and never of itself results in a compile-time error.

9.4.2 Overloading

If two methods of an interface (whether both declared in the same interface, or both inherited by an interface, or one declared and one inherited) have the same name but different signatures that are not override-equivalent (§8.4.2), then the method name is said to be *overloaded*. This fact causes no difficulty and never of itself results in a compile-time error. There is no required relationship between the return types or between the `throws` clauses of two methods with the same name but different signatures that are not override-equivalent.

9.4.3 Examples of Abstract Method Declarations

The following examples illustrate some (possibly subtle) points about abstract method declarations.

9.4.3.1 *Example: Overriding*

Methods declared in interfaces are `abstract` and thus contain no implementation. About all that can be accomplished by an overriding method declaration, other than to affirm a method signature, is to refine the return type or to restrict the exceptions that might be thrown by an implementation of the method. Here is a variation of the example shown in §8.4.3.1:

```
class BufferEmpty extends Exception {
   BufferEmpty() { super(); }
   BufferEmpty(String s) { super(s); }
}

class BufferException extends Exception {
   BufferException() { super(); }
   BufferException(String s) { super(s); }
}

public interface Buffer {
   char get() throws BufferEmpty, BufferException;
}

public interface InfiniteBuffer extends Buffer {
    char get() throws BufferException;// override
}
```

9.4.3.2 *Example: Overloading*

In the example code:

```
interface PointInterface {
   void move(int dx, int dy);
}

interface RealPointInterface extends PointInterface {
   void move(float dx, float dy);
   void move(double dx, double dy);
}
```

the method name `move` is overloaded in interface `RealPointInterface` with three different signatures, two of them declared and one inherited. Any non-

`abstract` class that implements interface `RealPointInterface` must provide implementations of all three method signatures.

9.5 Member Type Declarations

Interfaces may contain member type declarations (§8.5). A member type declaration in an interface is implicitly `static` and `public`.

If a member type declared with simple name *C* is directly enclosed within the declaration of an interface with fully qualified name *N*, then the member type has the fully qualified name *N*`.`*C*.

If the interface declares a member type with a certain name, then the declaration of that field is said to *hide* any and all accessible declarations of member types with the same name in superinterfaces of the interface.

An interface inherits from its direct superinterfaces all the non-private member types of the superinterfaces that are both accessible to code in the interface and not hidden by a declaration in the interface.

An interface may inherit two or more type declarations with the same name. A compile-time error occurs on any attempt to refer to any ambiguously inherited class or interface by its simple name. If the same type declaration is inherited from an interface by multiple paths, the class or interface is considered to be inherited only once; it may be referred to by its simple name without ambiguity.

9.6 Annotation Types

An *annotation type* declaration is a special kind of interface declaration. To distinguish an annotation type declaration from an ordinary interface declaration, the keyword `interface` is preceded by an at sign (`@`).

DISCUSSION

Note that the at sign (`@`) and the keyword `interface` are two distinct tokens; technically it is possible to separate them with whitespace, but this is strongly discouraged as a matter of style.

AnnotationTypeDeclaration:
InterfaceModifiers$_{opt}$ *@ interface Identifier AnnotationTypeBody*

AnnotationTypeBody:
{ AnnotationTypeElementDeclarations$_{opt}$ *}*

AnnotationTypeElementDeclarations:
AnnotationTypeElementDeclaration
AnnotationTypeElementDeclarations AnnotationTypeElementDeclaration

AnnotationTypeElementDeclaration:
AbstractMethodModifiers$_{opt}$ *Type Identifier ( ) DefaultValue*$_{opt}$ *;*
ConstantDeclaration
ClassDeclaration
InterfaceDeclaration
EnumDeclaration
AnnotationTypeDeclaration
;

DefaultValue:
default ElementValue

DISCUSSION

The following restrictions are imposed on annotation type declarations by virtue of their context free syntax:

- Annotation type declarations cannot be generic.
- No extends clause is permitted. (Annotation types implicitly extend `annotation.Annotation`.)
- Methods cannot have any parameters.
- Methods cannot have any type parameters.
- Method declarations cannot have a throws clause.

Unless explicitly modified herein, all of the rules that apply to ordinary interface declarations apply to annotation type declarations.

The *Identifier* in an annotation type declaration specifies the name of the annotation type. A compile-time error occurs if an annotation type has the same simple name as any of its enclosing classes or interfaces.

DISCUSSION

For example, annotation types share the same namespace as ordinary class and interface types.

Annotation type declarations are legal wherever interface declarations are legal, and have the same scope and accessibility.

If an annotation *a* on an annotation type declaration corresponds to an annotation type *T*, and *T* has a (meta-)annotation *m* that corresponds to `annotation.Target`, then *m* must have either an element whose value is `annotation.ElementType.ANNOTATION_TYPE`, or an element whose value is `annotation.ElementType.TYPE`, or a compile-time error occurs.

DISCUSSION

By convention, no *AbstractMethodModifiers* should be present except for annotations.

The direct superinterface of an annotation type is always `annotation.Annotation`.

DISCUSSION

A consequence of the fact that an annotation type cannot explicitly declare a superclass or superinterface is that a subclass or subinterface of an annotation type is never itself an annotation type. Similarly, `annotation.Annotation` is not itself an annotation type.

It is a compile-time error if the return type of a method declared in an annotation type is any type other than one of the following: one of the primitive types, `String`, `Class` and any invocation of `Class`, an enum type (§8.9), an annotation type, or an array (§10) of one of the preceding types. It is also a compile-time error if any method declared in an annotation type has the a signature that is override-equivalent to that of any `public` or `protected` method declared in class `Object` or in the interface `annotation.Annotation`.

DISCUSSION

Note that this does not conflict with the prohibition on generic methods, as wildcards eliminate the need for an explicit type parameter.

Each method declaration in an annotation type declaration defines an *element* of the annotation type. Annotation types can have zero or more elements. An annotation type has no elements other than those defined by the method it explicitly declares.

DISCUSSION

Thus, an annotation type declaration inherits several members from `annotation.Annotation`, including the implicitly declared methods corresponding to the instance methods in `Object`, yet these methods do not define elements of the annotation type and it is illegal to use them in annotations.

Without this rule, we could not ensure that the elements were of the types representable in annotations, or that access methods for them would be available.

It is a compile-time error if an annotation type *T* contains an element of type *T*, either directly or indirectly.

DISCUSSION

For example, this is illegal:

```
// Illegal self-reference!!
@interface SelfRef {
    SelfRef value();
}
```

and so is this:

```
// Illegal circularity!!
@interface Ping {
    Pong value();
}

@interface Pong {
    Ping value();
}
```

Note also that this specification precludes elements whose types are nested arrays. For example, this annotation type declaration is illegal:

```
// Illegal nested array!!
@interface Verboten {
    String[][] value();
}
```

An annotation type element may have a default value specified for it. This is done by following its (empty) parameter list with the keyword `default` and the default value of the element.

Defaults are applied dynamically at the time annotations are read; default values are not compiled into annotations. Thus, changing a default value affects annotations even in classes that were compiled before the change was made (presuming these annotations lack an explicit value for the defaulted element).

An *ElementValue* is used to specify a default value. It is a compile-time error if the type of the element is not commensurate (§9.7) with the default value specified.

DISCUSSION

The following annotation type declaration defines an annotation type with several elements:

```
// Normal annotation type declaration with several elements

/**
    * Describes the "request-for-enhancement" (RFE)
    * that led to the presence of
    * the annotated API element.
```

```
 */
public @interface RequestForEnhancement {
    int    id();        // Unique ID number associated with RFE
    String synopsis();  // Synopsis of RFE
    String engineer();  // Name of engineer who implemented RFE
    String date();      // Date RFE was implemented
}
```

The following annotation type declaration defines an annotation type with no elements, termed a marker annotation type:

```
// Marker annotation type declaration

/**
 * Annotation with this type indicates that the specification of
the
 * annotated API element is preliminary and subject to change.
 */
public @interface Preliminary { }
```

By convention, the name of the sole element in a single-element annotation type is `value`.

DISCUSSION

Linguistic support for this convention is provided by the single element annotation construct; one must obey the convention in order to take advantage of the construct.

DISCUSSION

The convention is illustrated in the following annotation type declaration:

```
// Single-element annotation type declaration

/**
 * Associates a copyright notice with the annotated API element.
 */
public @interface Copyright {
    String value();
}
```

The following annotation type declaration defines a single-element annotation type whose sole element has an array type:

```
    // Single-element annotation type declaration with array-typed ele-
ment

    /**
     * Associates a list of endorsers with the annotated class.
     */
    public @interface Endorsers {
        String[] value();
    }
```

Here is an example of complex annotation types, annotation types that contain one or more elements whose types are also annotation types.

```
    // Complex Annotation Type

    /**
     * A person's name.  This annotation type is not designed to be used
     * directly to annotate program elements, but to define elements
     * of other annotation types.
     */
    public @interface Name {
        String first();
        String last();
    }

    /**
     * Indicates the author of the annotated program element.
     */
    public @interface Author {
        Name value();
    }

    /**
     * Indicates the reviewer of the annotated program element.
     */
    public @interface Reviewer {
        Name value();
    }
```

The following annotation type declaration provides default values for two of its four elements:

```
    // Annotation type declaration with defaults on some elements
    public @interface RequestForEnhancement {
         int    id();        // No default - must be specified in each
    annotation
         String synopsis(); // No default - must be specified in each
    annotation
        String engineer()  default "[unassigned]";
        String date()      default "[unimplemented]";
    }
```

The following annotation type declaration shows a `Class` annotation whose value is restricted by a bounded wildcard.

```
    // Annotation type declaration with bounded wildcard to restrict
Class annotation
    /**
     * The annotation type declaration below presumes the existence of
    this interface,
     * which describes a formatter for Java programming language source
    code.
     */
    public interface Formatter { ... }

    /**
     * Designates a formatter to pretty-print the annotated class.
     */
    public @interface PrettyPrinter {
        Class<? extends Formatter> value();
    }
```

Note that the grammar for annotation type declarations permits other element declarations besides method declarations. For example, one might choose to declare a nested enum for use in conjunction with an annotation type:

```
    // Annotation type declaration with nested enum type declaration
    public @interface Quality {
        enum Level { BAD, INDIFFERENT, GOOD }

        Level value();
    }
```

9.6.1 Predefined Annotation Types

Several annotation types are predefined in the libraries of the Java platform. Some of these predefined annotation types have special semantics. These semantics are specified in this section. This section does not provide a complete specification for the predefined annotations contained here in; that is the role of the appropriate API specifications. Only those semantics that require special behavior on the part of the Java compiler or virtual machine are specified here.

9.6.1.1 *Target*

The annotation type `annotation.Target` is intended to be used in meta-annotations that indicate the kind of program element that an annotation type is applicable to. `Target` has one element, of type `annotation.ElementType[]`. It is a compile-time error if a given enum constant appears more than once in an annotation whose corresponding type is `annotation.Target`. See sections §7.4.1,

§8.1.1, §8.3.1, §8.4.1, §8.4.3, §8.8.3, §8.9, §9.1.1, §9.3, §9.4, §9.6 and §14.4 for the other effects of `@annotation.Target` annotations.

9.6.1.2 *Retention*

Annotations may be present only in the source code, or they may be present in the binary form of a class or interface. An annotation that is present in the binary may or may not be available at run-time via the reflective libraries of the Java platform.

The annotation type `annotation.Retention` is used to choose among the above possibilities. If an annotation *a* corresponds to a type *T*, and *T* has a (meta-)annotation *m* that corresponds to `annotation.Retention`, then:

- If *m* has an element whose value is `annotation.RetentionPolicy.SOURCE`, then a Java compiler must ensure that *a* is not present in the binary representation of the class or interface in which *a* appears.
- If *m* has an element whose value is `annotation.RetentionPolicy.CLASS`, or `annotation.RetentionPolicy.RUNTIME` a Java compiler must ensure that *a* is represented in the binary representation of the class or interface in which *a* appears, unless *m* annotates a local variable declaration. An annotation on a local variable declaration is never retained in the binary representation.

If *T* does not have a (meta-)annotation *m* that corresponds to `annotation.Retention`, then a Java compiler must treat *T* as if it does have such a meta-annotation *m* with an element whose value is `annotation.RetentionPolicy.CLASS`.

DISCUSSION

If *m* has an element whose value is `annotation.RetentionPolicy.RUNTIME`, the reflective libraries of the Java platform will make *a* available at run-time as well.

9.6.1.3 *Inherited*

The annotation type `annotation.Inherited` is used to indicate that annotations on a class *C* corresponding to a given annotation type are inherited by subclasses of *C*.

9.6.1.4 *Override*

Programmers occasionally overload a method declaration when they mean to override it.

DISCUSSION

The classic example concerns the equals method. Programmers write the following:

```
public boolean equals(Foo that) { ... }
```

when they mean to write:

```
public boolean equals(Object that) { ... }
```

This is perfectly legal, but class Foo inherits the equals implementation from Object, which can cause some very subtle bugs.

The annotation type `Override` supports early detection of such problems. If a method declaration is annotated with the annotation `@Override`, but the method does not in fact override any method declared in a superclass, a compile-time error will occur.

DISCUSSION

Note that if a method overrides a method from a superinterface but not from a superclass, using @Override will cause a compile-time error.

The rationale for this is that a concrete class that implements an interface will necessarily override all the interface's methods irrespective of the @Override annotation, and so it would be confusing to have the semantics of this annotation interact with the rules for implementing interfaces.

A by product of this rule is that it is never possible to use the @Override annotation in an interface declaration.

9.6.1.5 *SuppressWarnings*

The annotation type `SuppressWarnings` supports programmer control over warnings otherwise issued by the Java compiler. It contains a single element that is an array of `String`. If a program declaration is annotated with the annotation

`@SuppressWarnings(value = {`S_1, ... , S_k`})`, then a Java compiler must not report any warning identified by one of S_1, ... , S_k if that warning would have been generated as a result of the annotated declaration or any of its parts.

Unchecked warnings are identified by the string `"unchecked"`.

DISCUSSION

Recent Java compilers issue more warnings than previous ones did, and these "lint-like" warnings are very useful. It is likely that more such warnings will be added over time. To encourage their use, there should be some way to disable a warning in a particular part of the program when the programmer knows that the warning is inappropriate.

DISCUSSION

Compiler vendors should document the warning names they support in conjunction with this annotation type. They are encouraged to cooperate to ensure that the same names work across multiple compilers.

9.6.1.6 *Deprecated*

A program element annotated `@Deprecated` is one that programmers are discouraged from using, typically because it is dangerous, or because a better alternative exists. A Java compiler must produce a warning when a deprecated entity (type, method, field, or constructor) is used (overridden, invoked, or referenced by name) unless:

- The use is within an entity that itself is is annotated with the annotation `@Deprecated`; or
- The declaration and use are both within the same outermost class; or
- The use site is within an entity that is annotated to suppress the warning with the annotation `@SuppressWarnings("deprecation")`

9.7 Annotations

An *annotation* is a modifier consisting of an annotation type (§9.6) and zero or more element-value pairs, each of which associates a value with a different element of the annotation type. The purpose of an annotation is simply to associate information with the annotated program element.

Annotations must contain an element-value pair for every element of the corresponding annotation type, except for those elements with default values, or a compile-time error occurs. Annotations may, but are not required to, contain element-value pairs for elements with default values.

Annotations may be used as modifiers in any declaration, whether package (§7.4), class (§8), interface, field (§8.3, §9.3), method (§8.4, §9.4), parameter, constructor (§8.8), or local variable (§14.4).

DISCUSSION

Note that classes include enums (§8.9), and interfaces include annotation types (§9.6)

Annotations may also be used on enum constants. Such annotations are placed immediately before the enum constant they annotate.

It is a compile-time error if a declaration is annotated with more than one annotation for a given annotation type.

DISCUSSION

Annotations are conventionally placed before all other modifiers, but this is not a requirement; they may be freely intermixed with other modifiers.

There are three kinds of annotations. The first (normal annotation) is fully general. The others (marker annotation and single-element annotation) are merely shorthands.

Annotations:
Annotation
Annotations Annotation

Annotation:
NormalAnnotation
MarkerAnnotation
SingleElementAnnotation

A normal annotation is used to annotate a program element:

NormalAnnotation:
@ TypeName (ElementValuePairs$_{opt}$)

ElementValuePairs:
ElementValuePair
ElementValuePairs , ElementValuePair

ElementValuePair:
Identifier = ElementValue

ElementValue:
ConditionalExpression
Annotation
ElementValueArrayInitializer

ElementValueArrayInitializer:
{ ElementValues$_{opt}$,$_{opt}$ }

ElementValues:
ElementValue
ElementValues , ElementValue

DISCUSSION

Note that the at-sign (@) is a token unto itself. Technically it is possible to put whitespace in between the at-sign and the *TypeName*, but this is discouraged.

TypeName names the annotation type corresponding to the annotation. It is a compile-time error if *TypeName* does not name an annotation type. The annotation type named by an annotation must be accessible (§6.6) at the point where the annotation is used, or a compile-time error occurs.

The *Identifier* in an *ElementValuePair* must be the simple name of one of the elements of the annotation type identified by *TypeName* in the containing annotation. Otherwise, a compile-time error occurs. (In other words, the identifier in an element-value pair must also be a method name in the interface identified by *TypeName*.) The return type of this method defines the element type of the element-value pair. An *ElementValueArrayInitializer* is similar to a normal array initializer (§10.6), except that annotations are permitted in place of expressions.

An element type *T* is *commensurate* with an element value *V* if and only if one of the following conditions is true:

- *T* is an array type *E[]* and either:
  - *V* is an *ElementValueArrayInitializer* and each *ElementValueInitializer* (analogous to a variable initializer in an array initializer) in *V* is commensurate with *E*. Or
  - *V* is an *ElementValue* that is commensurate with *T*.
- The type of *V* is assignment compatible (§5.2) with *T* and, furthermore:
  - If *T* is a primitive type or `String`, *V* is a constant expression (§15.28).
  - *V* is not null.
  - if *T* is `Class`, or an invocation of `Class`, and *V* is a class literal (§15.8.2).
  - If *T* is an enum type, and *V* is an enum constant.

It is a compile-time error if the element type is not commensurate with the *ElementValue*.

If the element type is not an annotation type or an array type, *ElementValue* must be a *ConditionalExpression* (§15.25).

DISCUSSION

Note that `null` is not a legal element value for any element type.

If the element type is an array type and the corresponding *ElementValue* is not an *ElementValueArrayInitializer*, an array value whose sole element is the value represented by the *ElementValue* is associated with the element. Otherwise, the value represented by *ElementValue* is associated with the element.

DISCUSSION

In other words, it is permissible to omit the curly braces when a single-element array is to be associated with an array-valued annotation type element.

Note that the array's element type cannot be an array type, that is, nested array types are not permitted as element types. (While the annotation syntax would permit this, the annotation type declaration syntax would not.)

An annotation on an annotation type declaration is known as a *meta-annotation*. An annotation type may be used to annotate its own declaration. More generally, circularities in the transitive closure of the "annotates" relation are permitted. For example, it is legal to annotate an annotation type declaration with another annotation type, and to annotate the latter type's declaration with the former type. (The pre-defined meta-annotation types contain several such circularities.)

DISCUSSION

Here is an example of a normal annotation:

```
// Normal annotation
@RequestForEnhancement(
    id       = 2868724,
    synopsis = "Provide time-travel functionality",
    engineer = "Mr. Peabody",
    date     = "4/1/2004"
)
public static void travelThroughTime(Date destination) { ... }
```

Note that the types of the annotations in the examples in this section are the annotation types defined in the examples in §9.6. Note also that the elements are in the above annotation are in the same order as in the corresponding annotation type declaration. This is not required, but unless specific circumstances dictate otherwise, it is a reasonable convention fo follow.

The second form of annotation, marker annotation, is a shorthand designed for use with marker annotation types:

MarkerAnnotation:
@ TypeName

It is simply a shorthand for the normal annotation:

```
@TypeName()
```

DISCUSSION

Example:

```
// Marker annotation
@Preliminary public class TimeTravel { ... }
```

Note that it is legal to use marker annotations for annotation types with elements, so long as all the elements have default values.

The third form of annotation, single-element annotation, is a shorthand designed for use with single-element annotation types:

SingleElementAnnotation:
@ TypeName (ElementValue)

It is shorthand for the normal annotation:

```
@TypeName ( value = ElementValue )
```

DISCUSSION

Example:

```
// Single-element annotation
@Copyright("2002 Yoyodyne Propulsion Systems, Inc., All rights
reserved.")
public class OscillationOverthruster { ... }
```

Example with array-valued single-element annotation:

```
// Array-valued single-element annotation
@Endorsers({"Children", "Unscrupulous dentists"})
public class Lollipop { ... }
```

Example with single-element array-valued single-element annotation (note that the curly braces are omitted):

```
// Single-element array-valued single-element annotation
```

```
@Endorsers("Epicurus")
public class Pleasure { ... }
```

Example with complex annotation:

```
// Single-element complex annotation
@Author(@Name(first = "Joe", last = "Hacker"))
public class BitTwiddle { ... }
```

Note that it is legal to use single-element annotations for annotation types with multiple elements, so long as one element is named value, and all other elements have default values.

Here is an example of an annotation that takes advantage of default values:

```
// Normal annotation with default values
@RequestForEnhancement(
    id       = 4561414,
    synopsis = "Balance the federal budget"
)
public static void balanceFederalBudget() {
    throw new UnsupportedOperationException("Not implemented");
}
```

Here is an example of an annotation with a Class element whose value is restricted by the use of a bounded wildcard.

```
// Single-element annotation with Class element restricted by
bounded wildcard
// The annotation presumes the existence of this class.
class GorgeousFormatter implements Formatter { ... }
@PrettyPrinter(GorgeousFormatter.class) public class Petunia {...}
// This annotation is illegal, as String is not a subtype of Format-
ter!!
@PrettyPrinter(String.class) public class Begonia { ... }
```

Here is an example of an annotation using an enum type defined inside the annotation type:

```
// Annotation using enum type declared inside the annotation type
@Quality(Quality.Level.GOOD)
public class Karma {
   ...
}
```

Death, life, and sleep, reality and thought,
Assist me, God, their boundaries to know . . .
—William Wordsworth, *Maternal Grief*