Table of Contents

Chapter 9. Interfaces

An interface declaration introduces a new reference type whose members are classes, interfaces, constants, and methods. This type has no instance variables, and typically declares one or more `abstract` methods; otherwise unrelated classes can implement the interface by providing implementations for its `abstract` methods. Interfaces may not be directly instantiated.

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, both top level (§7.6) and nested (§8.5, §9.5), and annotation types (§9.6). 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 inherits all the member types, instance methods, and constants of the interfaces it extends, except for any members that it may override or 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 a superclass.

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* (§9.6).

```
InterfaceDeclaration:
  NormalInterfaceDeclaration
  AnnotationTypeDeclaration

NormalInterfaceDeclaration:
  {InterfaceModifier} interface Identifier [TypeParameters] [ExtendsInterfaces] InterfaceBody
```

The *Identifier* in an interface declaration specifies the name of the interface.

It is a compile-time error if an interface has the same simple name as any of its enclosing classes or interfaces.

The scope and shadowing of an interface declaration is specified in §6.3 and §6.4.

9.1.1. Interface Modifiers

An interface declaration may include *interface modifiers*.

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

The rules for annotation modifiers on an interface declaration are specified in §9.7.4 and §9.7.5.

The access modifier `public` (§6.6) pertains to every kind of interface declaration.

The access modifiers `protected` and `private` pertain only to member interfaces whose declarations are directly enclosed by a class declaration (§8.5.1).

The modifier `static` pertains only to member interfaces (§8.5.1, §9.5), not to top level interfaces (§7.6).

It is a compile-time error if the same keyword appears more than once as a modifier for an interface declaration.

If two or more (distinct) interface modifiers appear in an interface declaration, then it is customary, though not required, that they appear in the order consistent with that shown above in the production for InterfaceModifier.

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 methods declared in the interface, and 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.

The following productions from §8.1.2 and §4.4 are shown here for convenience:

```
TypeParameters:
  < TypeParameterList >

TypeParameterList:
  TypeParameter {, TypeParameter}

TypeParameter:
  {TypeParameterModifier} Identifier [TypeBound]

TypeParameterModifier:
  Annotation

TypeBound:
  extends TypeVariable
  extends ClassOrInterfaceType {AdditionalBound}

AdditionalBound:
  & InterfaceType
```

The rules for annotation modifiers on a type parameter declaration are specified in §9.7.4 and §9.7.5.

In an interface's type parameter section, a type variable T *directly depends* on a type variable S if S is the bound of T, while T *depends* on S if either T directly depends on S or T directly depends on a type variable U that depends on S (using this definition recursively). It is a compile-time error if a type variable in a interface's type parameter section depends on itself.

The scope and shadowing of an interface's type parameter is specified in §6.3.

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.

A generic interface declaration defines a set of parameterized types (§4.5), one for each possible parameterization of the type parameter section by type arguments. All of these parameterized types share the same interface at run time.

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 InterfaceTypeList
```

The following production from §8.1.5 is shown here for convenience:

```
InterfaceTypeList:
  InterfaceType {, InterfaceType}
```

Each *InterfaceType* in the `extends` clause of an interface declaration must name an accessible interface type (§6.6), or a compile-time error occurs.

If an *InterfaceType* has type arguments, it must denote a well-formed parameterized type (§4.5), and none of the type arguments may be wildcard type arguments, or a compile-time error occurs.

Given a (possibly generic) interface declaration I<F_1,...,F_n> ($n \geq 0$), the *direct superinterfaces* of the interface type I<F_1,...,F_n> are the types given in the `extends` clause of the declaration of I, if an `extends` clause is present.

Given a generic interface declaration I<F_1,...,F_n> ($n > 0$), the *direct superinterfaces* of the parameterized interface type I<T_1,...,T_n>, where T_i ($1 \leq i \leq n$) is a type, are all types J<U_1 θ,...,U_k θ>, where J<U_1,...,U_k> is a direct superinterface of I<F_1,...,F_n> and θ is the substitution $[F_1:=T_1,...,F_n:=T_n]$.

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.

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

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 in the fully qualified form of a superinterface name.

An interface I *depends* on a reference type T if any of the following is true:

- I directly depends on T.
- I directly depends on a class C that depends on T (§8.1.5).
- I directly depends on an interface J that depends on T (using this definition recursively).

It is a compile-time error if an interface depends on itself.

If circularly declared interfaces are detected at run time, as interfaces are loaded, then a `ClassCircularityError` is thrown (§12.2.1).

9.1.4. Interface Body and Member Declarations

The body of an interface may declare members of the interface, that is, fields (§9.3), methods (§9.4), classes (§9.5), and interfaces (§9.5).

```
InterfaceBody:
  { {InterfaceMemberDeclaration} }

InterfaceMemberDeclaration:
  ConstantDeclaration
  InterfaceMethodDeclaration
  ClassDeclaration
  InterfaceDeclaration
  ;
```

The scope of a declaration of a member `m` declared in or inherited by an interface type I is specified in §6.3.

9.2. Interface Members

The members of an interface type are:

- Members declared in the body of the interface (§9.1.4).
- Members inherited from any direct superinterfaces (§9.1.3).
- 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 an `abstract` 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 is a compile-time error if the interface explicitly declares a method with a signature that is override-equivalent (§8.4.2) to a `public` method of `Object`, but which has a different return type, or an incompatible `throws` clause, or is not `abstract`.

The interface inherits, from the interfaces it extends, all members of those interfaces, except for fields, classes, and interfaces that it hides; `abstract` or default methods that it overrides (§9.4.1); and `static` methods.

Fields, methods, and member types of an interface type may have the same name, since they are used in different contexts and are disambiguated by different lookup procedures (§6.5). However, this is discouraged as a matter of style.

9.3. Field (Constant) Declarations

```
ConstantDeclaration:
  {ConstantModifier} UnannType VariableDeclaratorList ;

ConstantModifier:
  (one of)
  Annotation public
  static final
```

See §8.3 for UnannType. The following productions from §4.3 and §8.3 are shown here for convenience:

```
VariableDeclaratorList:
  VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
  VariableDeclaratorId [= VariableInitializer]

VariableDeclaratorId:
  Identifier [Dims]

Dims:
  {Annotation} [ ] {{Annotation} [ ]}

VariableInitializer:
  Expression
  ArrayInitializer
```

The rules for annotation modifiers on an interface field declaration are specified in §9.7.4 and §9.7.5.

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.

It is a compile-time error if the same keyword appears more than once as a modifier for a field declaration.

If two or more (distinct) field modifiers appear in a field declaration, it is customary, though not required, that they appear in the order consistent with that shown above in the production for ConstantModifier.

The declared type of a field is denoted by *UnannType* if no bracket pairs appear in *UnannType* and *VariableDeclaratorId*, and is specified by §10.2 otherwise.

The scope and shadowing of an interface field declaration is specified in §6.3 and §6.4.

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

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 possible for an interface to inherit more than one field with the same name. Such a situation does not in itself cause a compile-time error. However, any attempt within the body of the interface to refer to any such 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.

Example 9.3-1. 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. In the program:

```
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.

Example 9.3-2. 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 previous example, 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.

9.3.1. Initialization of Fields in Interfaces

Every declarator in a field declaration of an interface must have a variable initializer, or a compile-time error occurs.

The initializer need not be a constant expression (§15.28).

It is a compile-time error if the initializer of an interface field uses the simple name of the same field or another field whose declaration occurs textually later in the same interface.

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

At run time, the initializer is evaluated and the field assignment performed exactly once, when the interface is initialized (§12.4.2).

Note that interface fields that are constant variables (§4.12.4) are initialized before other interface fields. This also applies to `static` fields that are constant variables in classes (§8.3.2). Such fields will never be observed to have their default initial values (§4.12.5), even by devious programs.

Example 9.3.1-1. Forward Reference to a Field

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

This program 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.

9.4. Method Declarations

```
InterfaceMethodDeclaration:
  {InterfaceMethodModifier} MethodHeader MethodBody

InterfaceMethodModifier:
  (one of)
  Annotation public
  abstract default static strictfp
```

The following productions from §8.4, §8.4.5, and §8.4.7 are shown here for convenience:

```
MethodHeader:
  Result MethodDeclarator [Throws]
  TypeParameters {Annotation} Result MethodDeclarator [Throws]

Result:
  UnannType
  void

MethodDeclarator:
  Identifier ( [FormalParameterList] ) [Dims]

MethodBody:
  Block
  ;
```

The rules for annotation modifiers on an interface method declaration are specified in §9.7.4 and §9.7.5.

Every method declaration in the body of an interface is implicitly `public` (§6.6). It is permitted, but discouraged as a matter of style, to redundantly specify the `public` modifier for a method declaration in an interface.

A *default method* is a method that is declared in an interface with the `default` modifier; its body is always represented by a block. It provides a default implementation for any class that implements the interface without overriding the method. Default methods are distinct from concrete methods (§8.4.3.1), which are declared in classes.

An interface can declare `static` methods, which are invoked without reference to a particular object.

It is a compile-time error to use the name of a type parameter of any surrounding declaration in the header or body of a `static` method of an interface.

The effect of the `strictfp` modifier is to make all `float` or `double` expressions within the body of a default or `static` method be explicitly FP-strict (§15.4).

An interface method lacking a `default` modifier or a `static` modifier is implicitly `abstract`, so its body is represented by a semicolon, not a block. It is permitted, but discouraged as a matter of style, to redundantly specify the `abstract` modifier for such a method declaration.

It is a compile-time error if the same keyword appears more than once as a modifier for a method declaration in an interface.

It is a compile-time error if a method is declared with more than one of the modifiers `abstract`, `default`, or `static`.

It is a compile-time error if an `abstract` method declaration contains the keyword `strictfp`.

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 `abstract` methods with such signatures (§9.4.1).

A method in an interface may be generic. The rules for 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 interface I *inherits* from its direct superinterfaces all `abstract` and default methods `m` for which all of the following are true:

- `m` is a member of a direct superinterface, J, of I.
- No method declared in I has a signature that is a subsignature (§8.4.2) of the signature of `m`.
- There exists no method `m`' that is a member of a direct superinterface, J', of I (`m` distinct from `m`', J distinct from J'), such that `m`' overrides from J' the declaration of the method `m`.

Note that methods are overridden on a signature-by-signature basis. If, for example, an interface declares two `public` methods with the same name (§9.4.2), and a subinterface overrides one of them, the subinterface still inherits the other method.

The third clause above prevents a subinterface from re-inheriting a method that has already been overridden by another of its superinterfaces. For example, in this program:

```
interface Top {
    default String name() { return "unnamed"; }
}
interface Left extends Top {
    default String name() { return getClass().getName(); }
}
interface Right extends Top {}

interface Bottom extends Left, Right {}
```

`Right` inherits `name()` from `Top`, but `Bottom` inherits `name()` from `Left`, not `Right`. This is because `name()` from `Left` overrides the declaration of `name()` in `Top`.

An interface does not inherit `static` methods from its superinterfaces.

If an interface I declares a `static` method `m`, and the signature of `m` is a subsignature of an instance method `m'` in a superinterface of I, and `m'` would otherwise be accessible to code in I, then a compile-time error occurs.

In essence, a `static` method in an interface cannot "hide" an instance method in a superinterface. This is similar to the rule in §8.4.8.2 whereby a `static` method in a class cannot hide an instance method in a superclass or superinterface. Note that the rule in §8.4.8.2 speaks of a class that "declares or inherits a `static` method", whereas the rule above speaks only of an interface that "declares a `static` method", since an interface cannot inherit a `static` method. Also note that the rule in §8.4.8.2 allows hiding of both instance and `static` methods in superclasses/superinterfaces, whereas the rule above considers only instance methods in superinterfaces.

9.4.1.1. Overriding (by Instance Methods)

An instance method m_1, declared in or inherited by an interface I, *overrides from I* another instance method, m_2, declared in interface J, iff both of the following are true:

- I is a subinterface of J.
- The signature of m_1 is a subsignature (§8.4.2) of the signature of m_2.

The presence or absence of the `strictfp` modifier has absolutely no effect on the rules for overriding methods. For example, it is permitted for a method that is not FP-strict to override an FP-strict method and it is permitted for an FP-strict method to override a method that is not FP-strict.

An overridden default method can be accessed by using a method invocation expression (§15.12) that contains the keyword `super` qualified by a superinterface name.

9.4.1.2. Requirements in Overriding

The relationship between the return type of an interface method and the return types of any overridden interface methods is specified in §8.4.8.3.

The relationship between the `throws` clause of an interface method and the `throws` clauses of any overridden interface methods are specified in §8.4.8.3.

The relationship between the signature of an interface method and the signatures of overridden interface methods are specified in §8.4.8.3.

It is a compile-time error if a default method is override-equivalent with a non-`private` method of the class `Object`, because any class implementing the interface will inherit its own implementation of the method.

The prohibition against declaring one of the `Object` methods as a default method may be surprising. There are, after all, cases like `java.util.List` in which the behavior of `toString` and `equals` are precisely defined. The motivation becomes clearer, however, when some broader design decisions are understood:

- *First, methods inherited from a superclass are allowed to override methods inherited from superinterfaces (§8.4.8.1). So, every implementing class would automatically override an interface's `toString` default. This is longstanding behavior in the Java programming language. It is not something we wish to change with the design of default methods, because that would conflict with the goal of allowing interfaces to unobtrusively evolve, only providing default behavior when a class doesn't already have it through the class hierarchy.*
- *Second, interfaces do not inherit from `Object`, but rather implicitly declare many of the same methods as `Object` (§9.2). So, there is no common ancestor for the `toString` declared in `Object` and the `toString` declared in an interface. At best, if both were candidates for inheritance by a class, they would conflict. Working around this problem would require awkward commingling of the class and interface inheritance trees.*
- *Third, use cases for declaring `Object` methods in interfaces typically assume a linear interface hierarchy; the feature does not generalize very well to multiple inheritance scenarios.*
- *Fourth, the `Object` methods are so fundamental that it seems dangerous to allow an arbitrary superinterface to silently add a default method that changes their behavior.*

An interface is free, however, to define another method that provides behavior useful for classes that override the `Object` methods. For example, the `java.util.List` interface could declare an `elementString` method that produces the string described by the contract of `toString`; implementors of `toString` in classes could then delegate to this method.

9.4.1.3. Inheriting Methods with Override-Equivalent Signatures

It is possible for an interface to inherit several methods with override-equivalent signatures (§8.4.2).

If an interface I inherits a default method whose signature is override-equivalent with another method inherited by I, then a compile-time error occurs. (This is the case whether the other method is `abstract` or `default`.)

Otherwise, all the inherited methods are `abstract`, and the interface is considered to inherit all the methods.

One of the inherited methods must be return-type-substitutable for every other inherited method, or else 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.

Naturally, when two different default methods with matching signatures are inherited by a subinterface, there is a behavioral conflict. We actively detect this conflict and notify the developer with an error, rather than waiting for the problem to arise when a concrete class is compiled. The error can be avoided by declaring a new method that overrides, and thus prevents the inheritance of, all conflicting methods.

Similarly, when an abstract and a default method with matching signatures are inherited, we produce an error. In this case, it would be possible to give priority to one or the other - perhaps we would assume that the default method provides a reasonable implementation for the abstract method, too. But this is risky, since other than the coincidental name and signature, we have no reason to believe that the default method behaves consistently with the abstract method's contract - the default method may not have even existed when the subinterface was originally developed. It is safer in this situation to ask the user to actively assert that the default implementation is appropriate (via an overriding declaration).

In contrast, the longstanding behavior for inherited concrete methods in classes is that they override abstract methods declared in interfaces (see §8.4.8). The same argument about potential contract violation applies here, but in this case there is an inherent imbalance between classes and interfaces. We prefer, in order to preserve the independent nature of class hierarchies, to minimize class-interface clashes by simply giving priority to concrete methods.

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.

Example 9.4.2-1. Overloading an `abstract` Method Declaration

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

Here, the method named `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.

9.4.3. Interface Method Body

A default method has a block body. This block of code provides an implementation of the method in the event that a class implements the interface but does not provide its own implementation of the method.

A `static` method also has a block body, which provides the implementation of the method.

It is a compile-time error if an interface method declaration is `abstract` (explicitly or implicitly) and has a block for its body.

It is a compile-time error if an interface method declaration is `default` or `static` and has a semicolon for its body.

It is a compile-time error for the body of a `static` method to attempt to reference the current object using the keyword `this` or the keyword `super`.

The rules for `return` statements in a method body are specified in §14.17.

If a method is declared to have a return type (§8.4.5), then a compile-time error occurs if the body of the method can complete normally (§14.1).

9.5. Member Type Declarations

Interfaces may contain member type declarations (§8.5).

A member type declaration in an interface is implicitly `public` and `static`. It is permitted to redundantly specify either or both of these modifiers.

It is a compile-time error if a member type declaration in an interface has the modifier `protected` or `private`.

It is a compile-time error if the same keyword appears more than once as a modifier for a member type declaration in an interface.

If an interface declares a member type with a certain name, then the declaration of that type 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. It is a compile-time error to 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* specifies a new *annotation type*, a special kind of interface type. To distinguish an annotation type declaration from a normal interface declaration, the keyword `interface` is preceded by an at-sign (`@`).

```
AnnotationTypeDeclaration:
  {InterfaceModifier} @ interface Identifier AnnotationTypeBody
```

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

The rules for annotation modifiers on an annotation type declaration are specified in §9.7.4 and §9.7.5.

The *Identifier* in an annotation type declaration specifies the name of the annotation type.

It is a compile-time error if an annotation type has the same simple name as any of its enclosing classes or interfaces.

The direct superinterface of every annotation type is `java.lang.annotation.Annotation`.

By virtue of the AnnotationTypeDeclaration syntax, an annotation type declaration cannot be generic, and no `extends` clause is permitted.

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, `java.lang.annotation.Annotation` is not itself an annotation type.

An annotation type inherits several members from `java.lang.annotation.Annotation`, including the implicitly declared methods corresponding to the instance methods of `Object`, yet these methods do not define elements of the annotation type (§9.6.1).

Because these methods do not define elements of the annotation type, it is illegal to use them in annotations of that type (§9.7). Without this rule, we could not ensure that elements were of the types representable in annotations, or that accessor methods for them would be available.

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

For example, annotation types share the same namespace as normal class and interface types; and annotation type declarations are legal wherever interface declarations are legal, and have the same scope and accessibility.

9.6.1. Annotation Type Elements

The body of an annotation type may contain method declarations, each of which defines an *element* of the annotation type. An annotation type has no elements other than those defined by the methods it explicitly declares.

```
AnnotationTypeBody:
  { {AnnotationTypeMemberDeclaration} }

AnnotationTypeMemberDeclaration:
  AnnotationTypeElementDeclaration
  ConstantDeclaration
  ClassDeclaration
  InterfaceDeclaration
  ;

AnnotationTypeElementDeclaration:
  {AnnotationTypeElementModifier} UnannType Identifier ( ) [Dims] [DefaultValue] ;

AnnotationTypeElementModifier:
  (one of)
  Annotation public
  abstract
```

By virtue of the AnnotationTypeElementDeclaration production, a method declaration in an annotation type declaration cannot have formal parameters, type parameters, or a `throws` clause. The following production from §4.3 is shown here for convenience:

```
Dims:
  {Annotation} [ ] {{Annotation} [ ]}
```

By virtue of the AnnotationTypeElementModifier production, a method declaration in an annotation type declaration cannot be `default` or `static`. Thus, an annotation type cannot declare the same variety of methods as a normal interface type. Note that it is still possible for an annotation type to inherit a default method from its implicit superinterface, `java.lang.annotation.Annotation`, though no such default method exists as of Java SE 8.

By convention, the only AnnotationTypeElementModifiers that should be present on an annotation type element are annotations.

The return type of a method declared in an annotation type must be one of the following, or a compile-time error occurs:

- A primitive type
- `String`
- `Class` or an invocation of `Class` (§4.5)
- An enum type
- An annotation type
- An array type whose component type is one of the preceding types (§10.1).

This rule precludes elements with nested array types, such as:

```
@interface Verboten {
    String[][] value();
}
```

The declaration of a method that returns an array is allowed to place the bracket pair that denotes the array type after the empty formal parameter list. This syntax is supported for compatibility with early versions of the Java programming language. It is very strongly recommended that this syntax is not used in new code.

It is a compile-time error if any method declared in an annotation type has a signature that is override-equivalent to that of any `public` or `protected` method declared in class `Object` or in the interface `java.lang.annotation.Annotation`.

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

For example, this is illegal:

```
@interface SelfRef { SelfRef value(); }
```

and so is this:

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

An annotation type with no elements is called a *marker annotation type*.

An annotation type with one element is called a *single-element annotation type*.

By convention, the name of the sole element in a single-element annotation type is `value`. Linguistic support for this convention is provided by single-element annotations (§9.7.3).

Example 9.6.1-1. Annotation Type Declaration

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

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

Example 9.6.1-2. Marker Annotation Type Declaration

The following annotation type declaration defines a marker annotation type:

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

Example 9.6.1-3. Single-Element Annotation Type Declarations

The convention that a single-element annotation type defines an element called `value` is illustrated in the following annotation type declaration:

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

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

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

The following annotation type declaration shows a `Class`-typed element whose value is constrained by a bounded wildcard:

```
interface Formatter {}

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

The following annotation type declaration contains an element whose type is also an annotation type:

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

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:

```
@interface Quality {
    enum Level { BAD, INDIFFERENT, GOOD }
    Level value();
}
```

9.6.2. Defaults for Annotation Type Elements

An annotation type element may have a *default value*, specified by following the element's (empty) parameter list with the keyword `default` and an *ElementValue* (§9.7.1).

```
DefaultValue:
  default ElementValue
```

It is a compile-time error if the type of the element is not commensurate (§9.7) with the default value specified.

Default values are not compiled into annotations, but rather applied dynamically at the time annotations are read. 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).

Example 9.6.2-1. Annotation Type Declaration With Default Values

Here is a refinement of the `RequestForEnhancement` annotation type from §9.6.1:

```
@interface RequestForEnhancementDefault {
    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]";
}
```

9.6.3. Repeatable Annotation Types

An annotation type T is *repeatable* if its declaration is (meta-)annotated with an `@Repeatable` annotation (§9.6.4.8) whose `value` element indicates a *containing annotation type of T*.

An annotation type TC is a *containing annotation type of T* if all of the following are true:

1. TC declares a `value()` method whose return type is T`[]`.
2. Any methods declared by TC other than `value()` have a default value.
3. TC is retained for at least as long as T, where retention is expressed explicitly or implicitly with the `@Retention` annotation (§9.6.4.2). Specifically:
   - If the retention of TC is `java.lang.annotation.RetentionPolicy.SOURCE`, then the retention of T is `java.lang.annotation.RetentionPolicy.SOURCE`.
   - If the retention of TC is `java.lang.annotation.RetentionPolicy.CLASS`, then the retention of T is either `java.lang.annotation.RetentionPolicy.CLASS` or `java.lang.annotation.RetentionPolicy.SOURCE`.
   - If the retention of TC is `java.lang.annotation.RetentionPolicy.RUNTIME`, then the retention of T is `java.lang.annotation.RetentionPolicy.SOURCE`, `java.lang.annotation.RetentionPolicy.CLASS`, or `java.lang.annotation.RetentionPolicy.RUNTIME`.
4. T is applicable to at least the same kinds of program element as TC (§9.6.4.1). Specifically, if the kinds of program element where T is applicable are denoted by the set m_1, and the kinds of program element

where TC is applicable are denoted by the set m_2, then each kind in m_2 must occur in m_1, except that:

- If the kind in m_2 is `java.lang.annotation.ElementType.ANNOTATION_TYPE`, then at least one of `java.lang.annotation.ElementType.ANNOTATION_TYPE` or `java.lang.annotation.ElementType.TYPE` or `java.lang.annotation.ElementType.TYPE_USE` must occur in m_1.
- If the kind in m_2 is `java.lang.annotation.ElementType.TYPE`, then at least one of `java.lang.annotation.ElementType.TYPE` or `java.lang.annotation.ElementType.TYPE_USE` must occur in m_1.
- If the kind in m_2 is `java.lang.annotation.ElementType.TYPE_PARAMETER`, then at least one of `java.lang.annotation.ElementType.TYPE_PARAMETER` or `java.lang.annotation.ElementType.TYPE_USE` must occur in m_1.

This clause implements the policy that an annotation type may be repeatable on only some of the kinds of program element where it is applicable.

5. If the declaration of T has a (meta-)annotation that corresponds to `java.lang.annotation.Documented`, then the declaration of TC must have a (meta-)annotation that corresponds to `java.lang.annotation.Documented`.

   Note that it is permissible for TC to be `@Documented` while T is not `@Documented`.

6. If the declaration of T has a (meta-)annotation that corresponds to `java.lang.annotation.Inherited`, then the declaration of TC must have a (meta)-annotation that corresponds to `java.lang.annotation.Inherited`.

   Note that it is permissible for TC to be `@Inherited` while T is not `@Inherited`.

It is a compile-time error if an annotation type T is (meta-)annotated with an `@Repeatable` annotation whose `value` element indicates a type which is not a containing annotation type of T.

Example 9.6.3-1. Ill-formed Containing Annotation Type

Consider the following declarations:

```
@Repeatable(FooContainer.class)
@interface Foo {}

@interface FooContainer { Object[] value(); }
```

Compiling the `Foo` declaration produces a compile-time error because `Foo` uses `@Repeatable` to attempt to specify `FooContainer` as its containing annotation type, but `FooContainer` is not in fact a containing annotation type of `Foo`. (The return type of `FooContainer.value()` is not `Foo[]`.)

The `@Repeatable` annotation cannot be repeated, so only one containing annotation type can be specified by a repeatable annotation type.

Allowing more than one containing annotation type to be specified would cause an undesirable choice at compile time, when multiple annotations of the repeatable annotation type are logically replaced with a container annotation (§9.7.5).

An annotation type can be the containing annotation type of at most one annotation type.

This is implied by the requirement that if the declaration of an annotation type T specifies a containing annotation type of TC, then the `value()` method of TC has a return type involving T, specifically `T[]`.

An annotation type cannot specify itself as its containing annotation type.

This is implied by the requirement on the `value()` method of the containing annotation type. Specifically, if an annotation type A specified itself (via `@Repeatable`) as its containing annotation type, then the return type of A's `value()` method would have to be `A[]`; but this would cause a compile-time error since an annotation type cannot refer to itself in its elements (§9.6.1). More generally, two annotation types cannot specify each other to be their containing annotation types, because cyclic annotation type declarations are illegal.

An annotation type TC may be the containing annotation type of some annotation type T while also having its own containing annotation type TC '. That is, a containing annotation type may itself be a repeatable annotation type.

Example 9.6.3-2. Restricting Where Annotations May Repeat

An annotation whose type declaration indicates a target of `java.lang.annotation.ElementType.TYPE` can appear in at least as many locations as an annotation whose type declaration indicates a target of `java.lang.annotation.ElementType.ANNOTATION_TYPE`. For example, given the following declarations of repeatable and containing annotation types:

```
@Target(ElementType.TYPE)
@Repeatable(FooContainer.class)
@interface Foo {}

@Target(ElementType.ANNOTATION_TYPE)
@Interface FooContainer {
    Foo[] value();
}
```

`@Foo` can appear on any type declaration while `@FooContainer` can appear on only annotation type declarations. Therefore, the following annotation type declaration is legal:

```
@Foo @Foo
@interface X {}
```

while the following interface declaration is illegal:

```
@Foo @Foo
interface X {}
```

More broadly, if `Foo` is a repeatable annotation type and `FooContainer` is its containing annotation type, then:

- *If `Foo` has no `@Target` meta-annotation and `FooContainer` has no `@Target` meta-annotation, then `@Foo` may be repeated on any program element which supports annotations.*
- *If `Foo` has no `@Target` meta-annotation but `FooContainer` has an `@Target` meta-annotation, then `@Foo` may only be repeated on program elements where `@FooContainer` may appear.*
- *If `Foo` has an `@Target` meta-annotation, then in the judgment of the designers of the Java programming language, `FooContainer` must be declared with knowledge of the `Foo`'s applicability. Specifically, the kinds of program element where `FooContainer` may appear must logically be the same as, or a subset of, `Foo`'s kinds.*

  For example, if `Foo` is applicable to field and method declarations, then `FooContainer` may legitimately serve as `Foo`'s containing annotation type if `FooContainer` is applicable to just field declarations (preventing `@Foo` from being repeated on method declarations). But if `FooContainer` is applicable only to formal parameter declarations, then `FooContainer` was a poor choice of containing annotation type by `Foo` because `@FooContainer` cannot be implicitly declared on some program elements where `@Foo` is repeated.

  Similarly, if `Foo` is applicable to field and method declarations, then `FooContainer` cannot legitimately serve as `Foo`'s containing annotation type if `FooContainer` is applicable to field and parameter declarations. While it would be possible to take the intersection of the program elements and make `Foo` repeatable on field declarations only, the presence of additional program elements for `FooContainer` indicates that `FooContainer` was not designed as a containing annotation type for `Foo`. It would therefore be dangerous for `Foo` to rely on it.

Example 9.6.3-3. A Repeatable Containing Annotation Type

The following declarations are legal:

```
// Foo: Repeatable annotation type
@Repeatable(FooContainer.class)
@interface Foo { int value(); }

// FooContainer: Containing annotation type of Foo
//               Also a repeatable annotation type itself
@Repeatable(FooContainerContainer.class)
@interface FooContainer { Foo[] value(); }

// FooContainerContainer: Containing annotation type of FooContainer
@interface FooContainerContainer { FooContainer[] value(); }
```

Thus, an annotation whose type is a containing annotation type may itself be repeated:

```
@FooContainer({@Foo(1)}) @FooContainer({@Foo(2)})
class A {}
```

An annotation type which is both repeatable and containing is subject to the rules on mixing annotations of repeatable annotation type with annotations of containing annotation type (§9.7.5). For example, it is not possible to write multiple `@Foo` annotations alongside multiple `@FooContainer` annotations, nor is it possible to write multiple `@FooContainer` annotations alongside multiple `@FooContainerContainer` annotations. However, if the `FooContainerContainer` type was itself repeatable, then it would be possible to write multiple `@Foo` annotations alongside multiple `@FooContainerContainer` annotations.

9.6.4. Predefined Annotation Types

Several annotation types are predefined in the libraries of the Java SE 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 a Java compiler or Java Virtual Machine implementation are specified here.

9.6.4.1. `@Target`

An annotation of type `java.lang.annotation.Target` is used on the declaration of an annotation type T to specify the contexts in which T is *applicable*. `java.lang.annotation.Target` has a single element, `value`, of type `java.lang.annotation.ElementType[]`, to specify contexts.

Annotation types may be applicable in *declaration contexts*, where annotations apply to declarations, or in *type contexts*, where annotations apply to types used in declarations and expressions.

There are eight declaration contexts, each corresponding to an enum constant of `java.lang.annotation.ElementType`:

1. Package declarations (§7.4.1)

   Corresponds to `java.lang.annotation.ElementType.PACKAGE`

2. Type declarations: class, interface, enum, and annotation type declarations (§8.1.1, §9.1.1, §8.5, §9.5, §8.9, §9.6)

   Corresponds to `java.lang.annotation.ElementType.TYPE`

   Additionally, annotation type declarations correspond to `java.lang.annotation.ElementType.ANNOTATION_TYPE`

3. Method declarations (including elements of annotation types) (§8.4.3, §9.4, §9.6.1)

   Corresponds to `java.lang.annotation.ElementType.METHOD`

4. Constructor declarations (§8.8.3)

   Corresponds to `java.lang.annotation.ElementType.CONSTRUCTOR`

5. Type parameter declarations of generic classes, interfaces, methods, and constructors (§8.1.2, §9.1.2, §8.4.4, §8.8.4)

   Corresponds to `java.lang.annotation.ElementType.TYPE_PARAMETER`

6. Field declarations (including enum constants) (§8.3.1, §9.3, §8.9.1)

   Corresponds to `java.lang.annotation.ElementType.FIELD`

7. Formal and exception parameter declarations (§8.4.1, §9.4, §14.20)

   Corresponds to `java.lang.annotation.ElementType.PARAMETER`

8. Local variable declarations (including loop variables of `for` statements and resource variables of `try`-with-resources statements) (§14.4, §14.14.1, §14.14.2, §14.20.3)

   Corresponds to `java.lang.annotation.ElementType.LOCAL_VARIABLE`

There are 16 type contexts (§4.11), all represented by the enum constant `TYPE_USE` of `java.lang.annotation.ElementType`.

It is a compile-time error if the same enum constant appears more than once in the `value` element of an annotation of type `java.lang.annotation.Target`.

If an annotation of type `java.lang.annotation.Target` is not present on the declaration of an annotation type T, then T is applicable in all declaration contexts except type parameter declarations, and in no type contexts.

These contexts are the syntactic locations where annotations were allowed in Java SE 7.

9.6.4.2. `@Retention`

Annotations may be present only in source code, or they may be present in the binary form of a class or interface. An annotation that is present in the binary form may or may not be available at run time via the reflection libraries of the Java SE platform. The annotation type `java.lang.annotation.Retention` is used to choose among these possibilities.

If an annotation `a` corresponds to a type T, and T has a (meta-)annotation `m` that corresponds to `java.lang.annotation.Retention`, then:

- If `m` has an element whose value is `java.lang.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 `java.lang.annotation.RetentionPolicy.CLASS` or `java.lang.annotation.RetentionPolicy.RUNTIME`, then 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.

  In addition, if `m` has an element whose value is `java.lang.annotation.RetentionPolicy.RUNTIME`, the reflection libraries of the Java SE platform must make `a` available at run time.

If T does not have a (meta-)annotation `m` that corresponds to `java.lang.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 `java.lang.annotation.RetentionPolicy.CLASS`.

9.6.4.3. `@Inherited`

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

9.6.4.4. `@Override`

Programmers occasionally overload a method declaration when they mean to override it, leading to subtle problems. The annotation type `Override` supports early detection of such problems.

> *The classic example concerns the `equals` method. Programmers write the following in class `Foo`:*
>
> ```
> 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.*

If a method declaration is annotated with the annotation `@Override`, but the method does not override or implement a method declared in a supertype, or is not override-equivalent to a `public` method of `Object`, a compile-time error occurs.

This behavior differs from Java SE 5.0, where `@Override` only caused a compile-time error if applied to a method that implemented a method from a superinterface that was not also present in a superclass.

> *The clause about overriding a `public` method is motivated by use of `@Override` in an interface. Consider the following type declarations:*
>
> ```
> class Foo     { @Override public int hashCode() {..} }
> interface Bar { @Override int hashCode(); }
> ```
>
> *The use of `@Override` in the class declaration is legal by the first clause, because `Foo.hashCode` overrides `Object.hashCode` (§8.4.8).*
>
> *For the interface declaration, consider that while an interface does not have `Object` as a supertype, an interface does have `public abstract` members that correspond to the `public` members of `Object` (§9.2). If an interface chooses to declare them explicitly (i.e. to declare members that are override-equivalent to `public` methods of `Object`), then the interface is deemed to override them (§8.4.8), and use of `@Override` is allowed.*
>
> *However, consider an interface that attempts to use `@Override` on a `clone` method: (`finalize` could also be used in this example)*
>
> ```
> interface Quux { @Override Object clone(); }
> ```
>
> *Because `Object.clone` is not `public`, there is no member called `clone` implicitly declared in `Quux`. Therefore, the explicit declaration of `clone` in `Quux` is not deemed to "implement" any other method, and it is erroneous to use `@Override`. (The fact that `Quux.clone` is `public` is not relevant.)*
>
> *In contrast, a class declaration that declares `clone` is simply overriding `Object.clone`, so is able to use `@Override`:*
>
> ```
> class Beep { @Override protected Object clone() {..} }
> ```

9.6.4.5. `@SuppressWarnings`

Java compilers are increasingly capable of issuing helpful "lint-like" warnings. To encourage the use of such warnings, there should be some way to disable a warning in a part of the program when the programmer knows that the warning is inappropriate.

The annotation type `SuppressWarnings` supports programmer control over warnings otherwise issued by a 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`".

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

9.6.4.6. `@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 deprecation warning when a type, method, field, or constructor whose declaration is annotated with `@Deprecated` is used (overridden, invoked, or referenced by name) in a construct which is explicitly or implicitly declared, unless:

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

Use of the `@Deprecated` annotation on a local variable declaration or on a parameter declaration has no effect.

The only implicitly declared construct that can cause a deprecation warning is a container annotation (§9.7.5). Namely, if T is a repeatable annotation type and TC is its containing annotation type, and TC is deprecated, then repeating the `@T` annotation will cause a deprecation warning. The warning is due to the implicit `@TC` container annotation. It is strongly discouraged to deprecate a containing annotation type without deprecating the corresponding repeatable annotation type.

9.6.4.7. `@SafeVarargs`

A variable arity parameter with a non-reifiable element type (§4.7) can cause heap pollution (§4.12.2) and give rise to compile-time unchecked warnings (§5.1.9). Such warnings are uninformative if the body of the variable arity method is well-behaved with respect to the variable arity parameter.

The annotation type `SafeVarargs`, when used to annotate a method or constructor declaration, makes a programmer assertion that prevents a Java compiler from reporting unchecked warnings for the declaration or invocation of a variable arity method or constructor where the compiler would otherwise do so due to the variable arity parameter having a non-reifiable element type.

The annotation `@SafeVarargs` has non-local effects because it suppresses unchecked warnings at method invocation expressions in addition to an unchecked warning pertaining to the declaration of the variable arity method itself (§8.4.1). In contrast, the annotation `@SuppressWarnings("unchecked")` has local effects because it only suppresses unchecked warnings pertaining to the declaration of a method.

> *The canonical target for `@SafeVarargs` is a method like `java.util.Collections.addAll`, whose declaration starts with:*
>
> ```
> public static <T> boolean
>   addAll(Collection<? super T> c, T... elements)
> ```
>
> *The variable arity parameter has declared type `T[]`, which is non-reifiable. However, the method fundamentally just reads from the input array and adds the elements to a collection, both of which are safe operations with respect to the array. Therefore, any compile-time unchecked warnings at method invocation expressions for `java.util.Collections.addAll` are arguably spurious and uninformative. Applying `@SafeVarargs` to the method declaration prevents generation of these unchecked warnings at the method invocation expressions.*

It is a compile-time error if a fixed arity method or constructor declaration is annotated with the annotation `@SafeVarargs`.

It is a compile-time error if a variable arity method declaration that is neither `static` nor `final` is annotated with the annotation `@SafeVarargs`.

Since `@SafeVarargs` is only applicable to `static` methods, `final` instance methods, and constructors, the annotation is not usable where method overriding occurs. Annotation inheritance only works on classes (not methods, interfaces, or constructors), so an `@SafeVarargs`-style annotation cannot be passed through instance methods in classes or through interfaces.

9.6.4.8. `@Repeatable`

The annotation type `java.lang.annotation.Repeatable` is used on the declaration of a *repeatable annotation type* to indicate its containing annotation type (§9.6.3).

Note that an `@Repeatable` meta-annotation on the declaration of T, indicating TC, is not sufficient to make TC the containing annotation type of T. There are numerous well-formedness rules for TC to be considered the containing annotation type of T.

9.6.4.9. `@FunctionalInterface`

The annotation type `FunctionalInterface` is used to indicate that an interface is meant to be a functional interface (§9.8). It facilitates early detection of inappropriate method declarations appearing in or inherited by an interface that is meant to be functional.

It is a compile-time error if an interface declaration is annotated with `@FunctionalInterface` but is not, in fact, a functional interface.

Because some interfaces are functional incidentally, it is not necessary or desirable that all declarations of functional interfaces be annotated with `@FunctionalInterface`.

9.7. Annotations

An *annotation* is a marker which associates information with a program construct, but has no effect at run time. An annotation denotes a specific invocation of an annotation type (§9.6) and usually provides values for the elements of that type.

There are three kinds of annotations. The first kind is the most general, while the other kinds are merely shorthands for the first kind.

```
Annotation:
  NormalAnnotation
  MarkerAnnotation
  SingleElementAnnotation
```

Normal annotations are described in §9.7.1, marker annotations in §9.7.2, and single element annotations in §9.7.3. Annotations may appear at various syntactic locations in a program, as described in §9.7.4. The number of annotations of the same type that may appear at a location is determined by their type, as described in §9.7.5.

9.7.1. Normal Annotations

A *normal annotation* specifies the name of an annotation type and optionally a list of comma-separated *element-value pairs*. Each pair contains an *element value* that is associated with an element of the annotation type (§9.6.1).

```
NormalAnnotation:
  @ TypeName ( [ElementValuePairList] )

ElementValuePairList:
  ElementValuePair {, ElementValuePair}

ElementValuePair:
  Identifier = ElementValue

ElementValue:
  ConditionalExpression
  ElementValueArrayInitializer
  Annotation

ElementValueArrayInitializer:
  { [ElementValueList] [,] }

ElementValueList:
  ElementValue {, ElementValue}
```

Note that the at-sign (`@`) is a token unto itself (§3.11). It is possible to put whitespace between it and the TypeName, but this is discouraged as a matter of style.

The *TypeName* specifies the annotation type corresponding to the annotation. The annotation is said to be "of" that type.

It is a compile-time error if *TypeName* does not specify an annotation type that is accessible (§6.6) at the point where the annotation appears.

The *Identifier* in an element-value pair must be the simple name of one of the elements (i.e. methods) of the annotation type, or a compile-time error occurs.

The return type of this method defines the *element type* of the element-value pair.

If the element type is an array type, then it is not required to use curly braces to specify the element value of the element-value pair. If the element value is not an *ElementValueArrayInitializer*, then an array value whose sole element is the element value is associated with the element. If the element value is an *ElementValueArrayInitializer*, then the array value represented by the *ElementValueArrayInitializer* is associated with the element.

It is a compile-time error if the element type is not *commensurate* with the element value. An element type T is commensurate with an element value `V` if and only if one of the following is true:

- T is an array type E`[]`, and either:
  - If `V` is a *ConditionalExpression* or an *Annotation*, then `V` is commensurate with E; or
  - If `V` is an *ElementValueArrayInitializer*, then each element value that `V` contains is commensurate with E.

    An ElementValueArrayInitializer is similar to a normal array initializer (§10.6), except that an ElementValueArrayInitializer may syntactically contain annotations as well as expressions and nested initializers. However, nested initializers are not semantically legal in an ElementValueArrayInitializer because they are never commensurate with array-typed elements in annotation type declarations (nested array types not permitted).
- T is not an array type, and the type of `V` is assignment compatible (§5.2) with T, and:
  - If T is a primitive type or `String`, then `V` is a constant expression (§15.28).
  - If T is `Class` or an invocation of `Class` (§4.5), then `V` is a class literal (§15.8.2).

- If T is an enum type (§8.9), then V is an enum constant (§8.9.1).
- V is not null.

Note that if T is not an array type or an annotation type, the element value must be a ConditionalExpression (§15.25). The use of ConditionalExpression rather than a more general production like Expression is a syntactic trick to prevent assignment expressions as element values. Since an assignment expression is not a constant expression, it cannot be a commensurate element value for a primitive or `String`-typed element.

Formally, it is invalid to speak of an ElementValue as FP-strict (§15.4) because it might be an annotation or a class literal. Still, we can speak informally of ElementValue as FP-strict when it is either a constant expression or an array of constant expressions or an annotation whose element values are (recursively) found to be constant expressions; after all, every constant expression is FP-strict.

A normal annotation 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.

A normal annotation may, but is not required to, contain element-value pairs for elements with default values.

It is customary, though not required, that element-value pairs in an annotation are presented in the same order as the corresponding elements in the annotation type declaration.

An annotation on an annotation type declaration is known as a *meta-annotation*.

An annotation of type T may appear as a meta-annotation on the declaration of type T itself. More generally, circularities in the transitive closure of the "annotates" relation are permitted.

For example, it is legal to annotate the declaration of an annotation type S with a meta-annotation of type T, and to annotate T's own declaration with a meta-annotation of type S. The pre-defined annotation types contain several such circularities.

Example 9.7.1-1. Normal Annotations

Here is an example of a normal annotation using the annotation type from §9.6.1:

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

Here is an example of a normal annotation that takes advantage of default values, using the annotation type from §9.6.2:

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

9.7.2. Marker Annotations

A *marker annotation* is a shorthand designed for use with marker annotation types (§9.6.1).

```
MarkerAnnotation:
  @ TypeName
```

It is shorthand for the normal annotation:

```
@TypeName()
```

It is legal to use marker annotations for annotation types with elements, so long as all the elements have default values (§9.6.2).

Example 9.7.2-1. Marker Annotations

Here is an example using the `Preliminary` marker annotation type from §9.6.1:

```
@Preliminary public class TimeTravel { ... }
```

9.7.3. Single-Element Annotations

A *single-element annotation*, is a shorthand designed for use with single-element annotation types (§9.6.1).

```
SingleElementAnnotation:
  @ TypeName ( ElementValue )
```

It is shorthand for the normal annotation:

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

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 (§9.6.2).

Example 9.7.3-1. Single-Element Annotations

The following annotations all use the single-element annotation types from §9.6.1.

Here is an example of a single-element annotation:

```
@Copyright("2002 Yoyodyne Propulsion Systems, Inc.")
public class OscillationOverthruster { ... }
```

Here is an example of an array-valued single-element annotation:

```
@Endorsers({"Children", "Unscrupulous dentists"})
public class Lollipop { ... }
```

Here is an example of a single-element array-valued single-element annotation: (note that the curly braces are omitted)

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

Here is an example of a single-element annotation with a `Class`-typed element whose value is constrained by a bounded wildcard.

```
class GorgeousFormatter implements Formatter { ... }

@PrettyPrinter(GorgeousFormatter.class)
public class Petunia { ... }

// Illegal; String is not a subtype of Formatter
@PrettyPrinter(String.class)
public class Begonia { ... }
```

Here is an example with of a single-element annotation that contains a normal annotation:

```
@Author(@Name(first = "Joe", last = "Hacker"))
public class BitTwiddle { ... }
```

Here is an example of a single-element annotation that uses an enum type defined inside the annotation type:

```
@Quality(Quality.Level.GOOD)
public class Karma { ... }
```

9.7.4. Where Annotations May Appear

A *declaration annotation* is an annotation that applies to a declaration, and whose own type is applicable in the declaration context (§9.6.4.1) represented by that declaration.

A *type annotation* is an annotation that applies to a type (or any part of a type), and whose own type is applicable in type contexts (§4.11).

For example, given the field declaration:

```
@Foo int f;
```

`@Foo` is a declaration annotation on `f` if `Foo` is meta-annotated by `@Target(ElementType.FIELD)`, and a type annotation on `int` if `Foo` is meta-annotated by `@Target(ElementType.TYPE_USE)`. It is possible for `@Foo` to be both a declaration annotation and a type annotation simultaneously.

Type annotations can apply to an array type or any component type thereof (§10.1). For example, assuming that `A`, `B`, and `C` are annotation types meta-annotated with `@Target(ElementType.TYPE_USE)`, then given the field declaration:

```
@C int @A [] @B [] f;
```

`@A` applies to the array type `int[][]`, `@B` applies to its component type `int[]`, and `@C` applies to the element type `int`. For more examples, see §10.2.

An important property of this syntax is that, in two declarations that differ only in the number of array levels, the annotations to the left of the type refer to the same type. For example, `@C` applies to the type `int` in all of the following declarations:

```
@C int f;
@C int[] f;
@C int[][] f;
```

It is customary, though not required, to write declaration annotations before all other modifiers, and type annotations immediately before the type to which they apply.

It is possible for an annotation to appear at a syntactic location in a program where it could plausibly apply to a declaration, or a type, or both. This can happen in any of the five declaration contexts where modifiers immediately precede the type of the declared entity:

- Method declarations (including elements of annotation types)
- Constructor declarations
- Field declarations (including enum constants)
- Formal and exception parameter declarations
- Local variable declarations (including loop variables of `for` statements and resource variables of `try`-with-resources statements)

The grammar of the Java programming language unambiguously treats annotations at these locations as modifiers for a declaration (§8.3), but that is purely a syntactic matter. Whether an annotation applies to a declaration or to the type of the declared entity - and thus, whether the annotation is a *declaration annotation* or a *type annotation* - depends on the applicability of the annotation's type:

- If the annotation's type is applicable in the declaration context corresponding to the declaration, and not in type contexts, then the annotation is deemed to apply only to the declaration.
- If the annotation's type is applicable in type contexts, and not in the declaration context corresponding to the declaration, then the annotation is deemed to apply only to the type which is closest to the annotation.
- If the annotation's type is applicable in the declaration context corresponding to the declaration *and* in type contexts, then the annotation is deemed to apply to both the declaration *and* the type which is closest to the annotation.

In the second and third cases above, the type which is *closest* to the annotation is the type written in source code for the declared entity; if that type is an array type, then the element type is deemed to be closest to the annotation.

For example, in the field declaration `@Foo public static String f;`, the type which is closest to `@Foo` is `String`. (If the type of the field declaration had been written as `java.lang.String`, then `java.lang.String` would be the type closest to `@Foo`, and later rules would prohibit a type annotation from applying to the package name `java`.) In the generic method declaration `@Foo <T> int[] m() {...}`, the type written for the declared entity is `int[]`, so `@Foo` applies to the element type `int`.

Local variable declarations are similar to formal parameter declarations of lambda expressions, in that both allow declaration annotations and type annotations in source code, but only the type annotations can be stored in the `class` file.

There are two special cases involving method/constructor declarations:

- If an annotation appears before a constructor declaration and is deemed to apply to the type which is closest to the annotation, that type is the type of the newly constructed object. The type of the newly constructed object is the fully qualified name of the type immediately enclosing the constructor declaration. Within that fully qualified name, the annotation applies to the simple type name indicated by the constructor declaration.
- **If an annotation appears before a `void` method declaration and is deemed to apply only to the type which is closest to the annotation, a compile-time error occurs.**

It is a compile-time error if an annotation of type T is syntactically a modifier for:

- a package declaration, but T is not applicable to package declarations.
- a class, interface, or enum declaration, but T is not applicable to type declarations or type contexts; or an annotation type declaration, but T is not applicable to annotation type declarations or type declarations or type contexts.
- a method declaration (including an element of an annotation type), but T is not applicable to method declarations or type contexts.
- a constructor declaration, but T is not applicable to constructor declarations or type contexts.
- a type parameter declaration of a generic class, interface, method, or constructor, but T is not applicable to type parameter declarations or type contexts.
- a field declaration (including an enum constant), but T is not applicable to field declarations or type contexts.
- a formal or exception parameter declaration, but T is not applicable to either formal and exception parameter declarations or type contexts.
- a receiver parameter, but T is not applicable to type contexts.
- a local variable declaration (including a loop variable of a `for` statement or a resource variable of a `try`-with-resources statement), but T is not applicable to local variable declarations or type contexts.

Note that most of the clauses above mention "... or type contexts", because even if an annotation does not apply to the declaration, it may still apply to the type of the declared entity.

A type annotation is *admissible* if both of the following are true:

- The simple name to which the annotation is closest is classified as a *TypeName*, not a *PackageName*.
- If the simple name to which the annotation is closest is followed by "`.`" and another *TypeName* - that is, the annotation appears as `@Foo T.U` - then `U` denotes an inner class of `T`.

The intuition behind the second clause is that if `Outer.this` is legal in a nested class enclosed by `Outer`, then `Outer` may be annotated because it represents the type of some object at run time. On the other hand, if `Outer.this` is not legal - because the class where it appears has no enclosing instance of `Outer` at run time - then `Outer` may not be annotated because it is logically just a name, akin to components of a package name in a fully qualified type name.

For example, in the following program, it is not possible to write `A.this` in the body of `B`, as `B` has no lexically enclosing instances (8.5.1). Therefore, it is not possible to apply `@Foo` to `A` in the type `A.B`, because `A` is logically just a name, not a type.

```
@Target(ElementType.TYPE_USE)
@interface Foo {}

class Test {
  class A {
    static class B {}
  }

  @Foo A.B x;  // Illegal
}
```

On the other hand, in the following program, it is possible to write `C.this` in the body of `D`. Therefore, it is possible to apply `@Foo` to `C` in the type `C.D`, because `C` represents the type of some object at run time.

```
@Target(ElementType.TYPE_USE)
@interface Foo {}

class Test {
  static class C {
    class D {}
  }

  @Foo C.D x;  // Legal
}
```

Finally, note that the second clause looks only one level deeper in a qualified type. This is because a `static` class may only be nested in a top level class or another `static` nested class. It is not possible to write a nest like:

```
@Target(ElementType.TYPE_USE)
@interface Foo {}

class Test {
  class E {
    class F {
      static class G {}
    }
  }

  @Foo E.F.G x;
}
```

Assume for a moment that the nest was legal. In the type of field `x`, `E` and `F` would logically be names qualifying `G`, as `E.F.this` would be illegal in the body of `G`. Then, `@Foo` should not be legal next to `E`. Technically, however, `@Foo` would be admissible next to `E` because the next deepest term `F` denotes an inner class; but this is moot as the class nest is illegal in the first place.

It is a compile-time error if an annotation of type T applies to the outermost level of a type in a type context, and T is not applicable in type contexts or the declaration context (if any) which occupies the same syntactic location.

It is a compile-time error if an annotation of type T applies to a part of a type (that is, not the outermost level) in a type context, and T is not applicable in type contexts.

It is a compile-time error if an annotation of type T applies to a type (or any part of a type) in a type context, and T is applicable in type contexts, and the annotation is not admissible.

For example, assume an annotation type `TA` which is meta-annotated with just `@Target(ElementType.TYPE_USE)`. The terms `@TA java.lang.Object` and `java.@TA lang.Object` are illegal because the simple name to which `@TA` is closest is classified as a package name. On the other hand, `java.lang.@TA Object` is legal.

Note that the illegal terms are illegal "everywhere". The ban on annotating package names applies broadly: to locations which are solely type contexts, such as `class ... extends @TA java.lang.Object {...}`, and to locations which are both declaration and type contexts, such as `@TA java.lang.Object f;`. (There are no locations which are solely declaration contexts where a package name could be annotated, as class, package, and type parameter declarations use only simple names.)

If `TA` is additionally meta-annotated with `@Target(ElementType.FIELD)`, then the term `@TA java.lang.Object` is legal in locations which are both declaration and type contexts, such as a field declaration `@TA java.lang.Object f;`. Here, `@TA` is deemed to apply to the declaration of `f` (and not to the type `java.lang.Object`) because `TA` is applicable in the field declaration context.

9.7.5. Multiple Annotations of the Same Type

It is a compile-time error if multiple annotations of the same type T appear in a declaration context or type context, unless T is repeatable (§9.6.3) and both T and the containing annotation type of T are applicable in the declaration context or type context (§9.6.4.1).

It is customary, though not required, for multiple annotations of the same type to appear contiguously.

If a declaration context or type context has multiple annotations of a repeatable annotation type T, then it is as if the context has no explicitly declared annotations of type T and one implicitly declared annotation of the containing annotation type of T.

The implicitly declared annotation is called the *container annotation*, and the multiple annotations of type T which appeared in the context are called the *base annotations*. The elements of the (array-typed) `value` element of the container annotation are all the base annotations in the left-to-right order in which they appeared in the context.

It is a compile-time error if, in a declaration context or type context, there are multiple annotations of a repeatable annotation type T and any annotations of the containing annotation type of T.

In other words, it is not possible to repeat annotations where an annotation of the same type as their container also appears. This prohibits obtuse code like:

```
@Foo(0) @Foo(1) @FooContainer({@Foo(2)})
class A {}
```

If this code was legal, then multiple levels of containment would be needed: first the annotations of type `Foo` would be contained by an implicitly declared container annotation of type `FooContainer`, then that annotation and the explicitly declared annotation of type `FooContainer` would be contained in yet another implicitly declared annotation. This complexity is undesirable in the judgment of the designers of the Java programming language. Another approach, treating the annotations of type `Foo` as if they had occurred alongside `@Foo(2)` in the explicit `@FooContainer` annotation, is undesirable because it could change how reflective programs interpret the `@FooContainer` annotation.

It is a compile-time error if, in a declaration context or type context, there is one annotation of a repeatable annotation type T and multiple annotations of the containing annotation type of T.

This rule is designed to allow the following code:

```
@Foo(1) @FooContainer({@Foo(2)})
class A {}
```

With only one annotation of the repeatable annotation type `Foo`, no container annotation is implicitly declared, even if `FooContainer` is the containing annotation type of `Foo`. However, repeating the annotation of type `FooContainer`, as in:

```
@Foo(1) @FooContainer({@Foo(2)}) @FooContainer({@Foo(3)})
class A {}
```

is prohibited, even if `FooContainer` is repeatable with a containing annotation type of its own. It is obtuse to repeat annotations which are themselves containers when an annotation of the underlying repeatable type is present.

9.8. Functional Interfaces

A *functional interface* is an interface that has just one abstract method (aside from the methods of `Object`), and thus represents a single function contract. This "single" method may take the form of multiple abstract methods with override-equivalent signatures inherited from superinterfaces; in this case, the inherited methods logically represent a single method.

For an interface I, let `M` be the set of `abstract` methods that are members of I that do not have the same signature as any `public` instance method of the class `Object`. Then, I is a *functional interface* if there exists a method `m` in `M` for which both of the following are true:

- The signature of `m` is a subsignature (§8.4.2) of every method's signature in `M`.
- `m` is return-type-substitutable (§8.4.5) for every method in `M`.

In addition to the usual process of creating an interface instance by declaring and instantiating a class (§15.9), instances of functional interfaces can be created with method reference expressions and lambda expressions (§15.13, §15.27).

The definition of functional interface excludes methods in an interface that are also `public` methods in `Object`. This is to allow functional treatment of an interface like `java.util.Comparator<T>` that declares multiple `abstract` methods of which only one is really "new" - `int compare(T,T)`. The other method - `boolean equals(Object)` - is an explicit declaration of an `abstract` method that would otherwise be implicitly declared, and will be automatically implemented by every class that `implements` the interface.

Note that if non-`public` methods of `Object`, such as `clone()`, are declared in an interface, they are not automatically implemented by every class that `implements` the interface. The implementation inherited from `Object` is `protected` while the interface method is necessarily `public`. The only way to implement such an interface would be for a class to override the non-`public` `Object` method with a `public` method.

Example 9.8-1. Functional Interfaces

A simple example of a functional interface is:

```
interface Runnable {
    void run();
}
```

The following interface is not functional because it declares nothing which is not already a member of `Object`:

```
interface NonFunc {
    boolean equals(Object obj);
}
```

However, its subinterface can be functional by declaring an `abstract` method which is not a member of `Object`:

```
interface Func extends NonFunc {
    int compare(String o1, String o2);
}
```

Similarly, the well known interface `java.util.Comparator<T>` is functional because it has one `abstract` non-`Object` method:

```
interface Comparator<T> {
    boolean equals(Object obj);
    int compare(T o1, T o2);
}
```

The following interface is not functional because while it only declares one `abstract` method which is not a member of `Object`, it declares two `abstract` methods which are not `public` members of `Object`:

```
interface Foo {
    int m();
    Object clone();
}
```

Example 9.8-2. Functional Interfaces and Erasure

In the following interface hierarchy, Z *is a functional interface because while it inherits two* abstract *methods which are not members of* Object*, they have the same signature, so the inherited methods logically represent a single method:*

```
interface X { int m(Iterable<String> arg); }
interface Y { int m(Iterable<String> arg); }
interface Z extends X, Y {}
```

Similarly, Z *is a functional interface in the following interface hierarchy because* Y.m *is a subsignature of* X.m *and is return-type-substitutable for* X.m*:*

```
interface X { Iterable m(Iterable<String> arg); }
interface Y { Iterable<String> m(Iterable arg); }
interface Z extends X, Y {}
```

The definition of functional interface respects the fact that an interface cannot have two members which are not subsignatures of each other, yet have the same erasure ([§9.4.1.2](#)). Thus, in the following three interface hierarchies where Z *causes a compile-time error,* Z *is not a functional interface: (because none of its* abstract *members are subsignatures of all other* abstract *members)*

```
interface X { int m(Iterable<String> arg); }
interface Y { int m(Iterable<Integer> arg); }
interface Z extends X, Y {}

interface X { int m(Iterable<String> arg, Class c); }
interface Y { int m(Iterable arg, Class<?> c); }
interface Z extends X, Y {}

interface X<T> { void m(T arg); }
interface Y<T> { void m(T arg); }
interface Z<A, B> extends X<A>, Y<B> {}
```

Similarly, the definition of "functional interface" respects the fact that an interface may only have methods with override-equivalent signatures if one is return-type-substitutable for all the others. Thus, in the following interface hierarchy where Z *causes a compile-time error,* Z *is not a functional interface: (because none of its* abstract *members are return-type-substitutable for all other* abstract *members)*

```
interface X { long m(); }
interface Y { int  m(); }
interface Z extends X, Y {}
```

In the following example, the declarations of Foo<T,N> *and* Bar *are legal: in each, the methods called* m *are not subsignatures of each other, but do have different erasures. Still, the fact that the methods in each are not subsignatures means* Foo<T,N> *and* Bar *are not functional interfaces. However,* Baz *is a functional interface because the methods it inherits from* Foo<Integer,Integer> *have the same signature and so logically represent a single method.*

```
interface Foo<T, N extends Number> {
    void m(T arg);
    void m(N arg);
}
interface Bar extends Foo<String, Integer> {}
interface Baz extends Foo<Integer, Integer> {}
```

Finally, the following examples demonstrate the same rules as above, but with generic methods:

```
interface Exec { <T> T execute(Action<T> a); }
  // Functional

interface X { <T> T execute(Action<T> a); }
interface Y { <S> S execute(Action<S> a); }
interface Exec extends X, Y {}
  // Functional: signatures are logically "the same"

interface X { <T>   T execute(Action<T> a); }
interface Y { <S,T> S execute(Action<S> a); }
interface Exec extends X, Y {}
  // Error: different signatures, same erasure
```

Example 9.8-3. Generic Functional Interfaces

Functional interfaces can be generic, such as java.util.function.Predicate<T>*. Such a functional interface may be parameterized in a way that produces distinct* abstract *methods - that is, multiple methods that cannot be legally overridden with a single declaration. For example:*

```
interface I    { Object m(Class c); }
interface J<S> { S m(Class<?> c); }
interface K<T> { T m(Class<?> c); }
interface Functional<S,T> extends I, J<S>, K<T> {}
```

Functional<S,T> *is a functional interface -* I.m *is return-type-substitutable for* J.m *and* K.m *- but the functional interface type* Functional<String,Integer> *clearly cannot be implemented with a single method. However, other parameterizations of* Functional<S,T> *which are functional interface types are possible.*

The declaration of a functional interface allows a *functional interface type* to be used in a program. There are four kinds of functional interface type:

- The type of a non-generic ([§6.1](#)) functional interface
- A parameterized type that is a parameterization ([§4.5](#)) of a generic functional interface
- The raw type ([§4.8](#)) of a generic functional interface
- An intersection type ([§4.9](#)) that induces a notional functional interface

In special circumstances, it is useful to treat an intersection type as a functional interface type. Typically, this will look like an intersection of a functional interface type with one or more marker interface types, such as Runnable & java.io.Serializable*. Such an intersection can be used in casts ([§15.16](#)) that force a lambda expression to conform to a certain type. If one of the interface types in the intersection is* java.io.Serializable*, special run-time support for serialization is triggered ([§15.27.4](#)).*

9.9. Function Types

The *function type* of a functional interface I is a method type ([§8.2](#)) that can be used to override ([§8.4.8](#)) the abstract method(s) of I.

Let M be the set of abstract methods defined for I. The function type of I consists of the following:

- Type parameters, formal parameters, and return type:

  Let m be a method in M with:

  1. a signature that is a subsignature of every method's signature in M; and
  2. a return type that is a subtype of every method's return type in M (after adapting for any type parameters ([§8.4.4](#))).

  If no such method exists, then let m be a method in M that:

  1. has a signature that is a subsignature of every method's signature in M; and
  2. is return-type-substitutable ([§8.4.5](#)) for every method in M.

  The function type's type parameters, formal parameter types, and return type are as given by m.

- throws clause:

  The function type's throws clause is derived from the throws clauses of the methods in M. If the function type is generic, these clauses are first adapted to the type parameters of the function type ([§8.4.4](#)). If the function type is not generic but at least one method in M is generic, these clauses are first erased. Then, the function type's throws clause includes every type, E, which satisfies the following constraints:

  - E is mentioned in one of the throws clauses.
  - For each throws clause, E is a subtype of some type named in that clause.

When some return types in M *are raw and others are not, the definition of a function type tries to choose the most specific type, if possible. For example, if the return types are* LinkedList *and* LinkedList<String>*, then the latter is immediately* *chosen as the function type's return type. When there is no most specific type, the definition compensates by finding the most substitutable return type. For example, if there is a third return type, `List<?>`, then it is not the case that one of the return types is a subtype of every other (as raw `LinkedList` is not a subtype of `List<?>`); instead, `LinkedList<String>` is chosen as the function type's return type because it is return-type-substitutable for both `LinkedList` and `List<?>`.*

The goal driving the definition of a function type's thrown exception types is to support the invariant that a method with the resulting `throws` clause could override each abstract method of the functional interface. Per §8.4.6, this means the function type cannot throw "more" exceptions than any single method in the set `M`, so we look for as many exception types as possible that are "covered" by every method's `throws` clause.

The function type of a functional interface type is specified as follows:

- The function type of the type of a non-generic functional interface I is simply the function type of the functional interface I, as defined above.
- The function type of a parameterized functional interface type I<A_1...A_n>, where A_1...A_n are types and the corresponding type parameters of I are P_1...P_n, is derived by applying the substitution $[P_1:=A_1, ..., P_n:=A_n]$ to the function type of the generic functional interface I<P_1...P_n>.
- The function type of a parameterized functional interface type I<A_1...A_n>, where one or more of A_1...A_n is a wildcard, is the function type of the *non-wildcard parameterization* of I, I<T_1...T_n>. The non-wildcard parameterization is determined as follows.

  Let P_1...P_n be the type parameters of I with corresponding bounds B_1...B_n. For all *i* ($1 \le i \le n$), T_i is derived according to the form of A_i:

  - If A_i is a type, then $T_i = A_i$.
  - If A_i is a wildcard, and the corresponding type parameter's bound, B_i, mentions one of P_1...P_n, then T_i is undefined and there is no function type.
  - Otherwise:
    - If A_i is an unbound wildcard `?`, then $T_i = B_i$.
    - If A_i is a upper-bounded wildcard `? extends` U_i, then T_i = glb(U_i, B_i) (§5.1.10).
    - If A_i is a lower-bounded wildcard `? super` L_i, then $T_i = L_i$.
- The function type of the raw type of a generic functional interface I<...> is the erasure of the function type of the generic functional interface I<...>.
- The function type of an intersection type that induces a notional functional interface is the function type of the notional functional interface.

Example 9.9-1. Function Types

Given the following interfaces:

```
interface X { void m() throws IOException; }
interface Y { void m() throws EOFException; }
interface Z { void m() throws ClassNotFoundException; }
```

the function type of:

```
interface XY extends X, Y {}
```

is:

```
()->void throws EOFException
```

while the function type of:

```
interface XYZ extends X, Y, Z {}
```

is:

```
()->void (throws nothing)
```

Given the following interfaces:

```
interface A {
    List<String> foo(List<String> arg)
      throws IOException, SQLTransientException;
}
interface B {
    List foo(List<String> arg)
      throws EOFException, SQLException, TimeoutException;
}
interface C {
    List foo(List arg) throws Exception;
}
```

the function type of:

```
interface D extends A, B {}
```

is:

```
(List<String>)->List<String>
  throws EOFException, SQLTransientException
```

while the function type of:

```
interface E extends A, B, C {}
```

is:

```
(List)->List throws EOFException, SQLTransientException
```

The function type of a functional interface is defined nondeterministically: while the signatures in `M` are "the same", they may be syntactically different (`HashMap.Entry` and `Map.Entry`, for example); the return type may be a subtype of every other return type, but there may be other return types that are also subtypes (`List<?>` and `List<? extends Object>`, for example); and the order of thrown types is unspecified. These distinctions are subtle, but they can sometimes be important. However, function types are not used in the Java programming language in such a way that the nondeterminism matters. Note that the return type and `throws` clause of a "most specific method" are also defined nondeterministically when there are multiple abstract methods (§15.12.2.5).

When a generic functional interface is parameterized by wildcards, there are many different instantiations that could satisfy the wildcard and produce different function types. For example, each of `Predicate<Integer>` (function type `Integer -> boolean`), `Predicate<Number>` (function type `Number -> boolean`), and `Predicate<Object>` (function type `Object -> boolean`) is a `Predicate<? super Integer>`. Sometimes, it is possible to known from the context, such as the parameter types of a lambda expression, which function type is intended (§15.27.3). Other times, it is necessary to pick one; in these circumstances, the bounds are used. (This simple strategy cannot guarantee that the resulting type will satisfy certain complex bounds, so not all complex cases are supported.)

Example 9.9-2. Generic Function Types

A function type may be generic, as a functional interface's abstract method may be generic. For example, in the following interface hierarchy:

```
interface G1 {
    <E extends Exception> Object m() throws E;
}
interface G2 {
    <F extends Exception> String m() throws Exception;
}
interface G extends G1, G2 {}
```

the function type of `G` is:

```
<F extends Exception> ()->String throws F
```

A generic function type for a functional interface may be implemented by a method reference expression (§15.13), but not by a lambda expression (§15.27) as there is no syntax for generic lambda expressions.