CHAPTER 8

Classes

A class declaration defines a new class and describes how it is implemented (§8.1).

A *top level class* (§7.6) is a class declared directly in a compilation unit.

A *nested class* is any class whose declaration occurs within the body of another class or interface declaration. A nested class may be a member class (§8.5, §9.5), a local class (§14.3), or an anonymous class (§15.9.5).

Some kinds of nested class are an *inner class* (§8.1.3), which is a class that can refer to enclosing class instances, local variables, and type variables.

An *enum class* (§8.9) is a class declared with abbreviated syntax that defines a small set of named class instances.

A *record class* (§8.10) is a class declared with abbreviated syntax that defines a simple aggregate of values.

This chapter discusses the common semantics of all classes. Details that are specific to particular kinds of classes are discussed in the sections dedicated to these constructs.

A class may be declared `public` (§8.1.1) so it can be referred to from code in any package of its module and potentially from code in other modules.

A class may be declared `abstract` (§8.1.1.1), and must be declared `abstract` if it is incompletely implemented; such a class cannot be instantiated, but can be extended by subclasses. The degree to which a class can be extended can be controlled explicitly (§8.1.1.2): it may be declared `sealed` to limit its subclasses, or it may be declared `final` to ensure no subclasses. Each class except `Object` is an extension of (that is, a subclass of) a single existing class (§8.1.4) and may implement interfaces (§8.1.5).

A class may be *generic* (§8.1.2), that is, its declaration may introduce type variables whose bindings differ among different instances of the class.

Class declarations may be decorated with annotations (§9.7) just like any other kind of declaration.

The body of a class declares members (fields, methods, classes, and interfaces), instance and static initializers, and constructors (§8.1.7). The scope (§6.3) of a member (§8.2) is the entire body of the declaration of the class to which the member belongs. Field, method, member class, member interface, and constructor declarations may include the access modifiers `public`, `protected`, or `private` (§6.6). The members of a class include both declared and inherited members (§8.2). Newly declared fields can hide fields declared in a superclass or superinterface. Newly declared member classes and member interfaces can hide member classes and member interfaces declared in a superclass or superinterface. Newly declared methods can hide, implement, or override methods declared in a superclass or superinterface.

Field declarations (§8.3) describe class variables, which are incarnated once, and instance variables, which are freshly incarnated for each instance of the class. A field may be declared `final` (§8.3.1.2), in which case it can be assigned to only once. Any field declaration may include an initializer.

Member class declarations (§8.5) describe nested classes that are members of the surrounding class. Member classes may be `static`, in which case they have no access to the instance variables of the surrounding class; or they may be inner classes.

Member interface declarations (§8.5) describe nested interfaces that are members of the surrounding class.

Method declarations (§8.4) describe code that may be invoked by method invocation expressions (§15.12). A class method is invoked relative to the class; an instance method is invoked with respect to some particular object that is an instance of a class. A method whose declaration does not indicate how it is implemented must be declared `abstract`. A method may be declared `final` (§8.4.3.3), in which case it cannot be hidden or overridden. A method may be implemented by platform-dependent `native` code (§8.4.3.4). A `synchronized` method (§8.4.3.6) automatically locks an object before executing its body and automatically unlocks the object on return, as if by use of a `synchronized` statement (§14.19), thus allowing its activities to be synchronized with those of other threads (§17 (*Threads and Locks*)).

Method names may be overloaded (§8.4.9).

Instance initializers (§8.6) are blocks of executable code that may be used to help initialize an instance when it is created (§15.9).

Static initializers (§8.7) are blocks of executable code that may be used to help initialize a class.

Constructors (§8.8) are similar to methods, but cannot be invoked directly by a method call; they are used to initialize new class instances. Like methods, they may be overloaded (§8.8.8).

8.1 Class Declarations

A *class declaration* specifies a class.

There are three kinds of class declarations: *normal class declarations*, *enum declarations* (§8.9), and *record declarations* (§8.10).

ClassDeclaration:
  NormalClassDeclaration
  EnumDeclaration
  RecordDeclaration

NormalClassDeclaration:
  {ClassModifier} `class` *TypeIdentifier [TypeParameters]*
    [ClassExtends] [ClassImplements] [ClassPermits] ClassBody

A class is also implicitly declared by a class instance creation expression (§15.9.5) and an enum constant that ends with a class body (§8.9.1).

The *TypeIdentifier* in a class declaration specifies the name of the class.

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

The scope and shadowing of a class declaration is specified in §6.3 and §6.4.1.

8.1.1 Class Modifiers

A class declaration may include *class modifiers*.

ClassModifier:
  (one of)
  Annotation `public` `protected` `private`
  `abstract` `static` `final` `sealed` `non-sealed` `strictfp`

The rules concerning annotation modifiers for a class declaration are specified in §9.7.4 and §9.7.5.

The access modifier `public` (§6.6) pertains only to top level classes (§7.6) and member classes (§8.5, §9.5), not to local classes (§14.3) or anonymous classes (§15.9.5).

The access modifiers `protected` and `private` pertain only to member classes.

The modifier `static` pertains only to member classes and local classes.

It is a compile-time error if the same keyword appears more than once as a modifier for a class declaration, or if a class declaration has more than one of the access modifiers `public`, `protected`, and `private`.

It is a compile-time error if a class declaration has more than one of the modifiers `sealed`, `non-sealed`, and `final`.

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

8.1.1.1 `abstract` *Classes*

An `abstract` class is a class that is incomplete, or to be considered incomplete.

It is a compile-time error if an attempt is made to create an instance of an `abstract` class using a class instance creation expression (§15.9.1).

A subclass of an `abstract` class that is not itself `abstract` may be instantiated, resulting in the execution of a constructor for the `abstract` class and, therefore, the execution of the field initializers for instance variables of that class.

A normal class may have `abstract` methods, that is, methods that are declared but not yet implemented (§8.4.3.1), only if it is an `abstract` class. It is a compile-time error if a normal class that is not `abstract` has an `abstract` method.

A class *`C`* has `abstract` methods if either of the following is true:

- Any of the member methods (§8.2) of *`C`* - either declared or inherited - is `abstract`.
- Any of *`C`*'s superclasses has an `abstract` method declared with package access, and there exists no method that overrides the `abstract` method from *`C`* or from a superclass of *`C`*.

It is a compile-time error to declare an `abstract` class type such that it is not possible to create a subclass that implements all of its `abstract` methods. This situation can occur if the class would have as members two `abstract` methods that have the same method signature (§8.4.2) but return types for which no type is return-type-substitutable with both (§8.4.5).

Example 8.1.1.1-1. Abstract Class Declaration

```
abstract class Point {
    int x = 1, y = 1;
    void move(int dx, int dy) {
        x += dx;
        y += dy;
        alert();
    }
    abstract void alert();
}
abstract class ColoredPoint extends Point {
    int color;
}
class SimplePoint extends Point {
    void alert() { }
}
```

Here, a class `Point` is declared that must be declared `abstract`, because it contains a declaration of an `abstract` method named `alert`. The subclass of `Point` named `ColoredPoint` inherits the `abstract` method `alert`, so it must also be declared `abstract`. On the other hand, the subclass of `Point` named `SimplePoint` provides an implementation of `alert`, so it need not be `abstract`.

The statement:

```
Point p = new Point();
```

would result in a compile-time error; the class `Point` cannot be instantiated because it is `abstract`. However, a `Point` variable could correctly be initialized with a reference to any subclass of `Point`, and the class `SimplePoint` is not `abstract`, so the statement:

```
Point p = new SimplePoint();
```

would be correct. Instantiation of a `SimplePoint` causes the default constructor and field initializers for `x` and `y` of `Point` to be executed.

Example 8.1.1.1-2. Abstract Class Declaration that Prohibits Subclasses

```
interface Colorable {
    void setColor(int color);
}
abstract class Colored implements Colorable {
    public abstract int setColor(int color);
}
```

These declarations result in a compile-time error: it would be impossible for any subclass of class `Colored` to provide an implementation of a method named `setColor`, taking one argument of type `int`, that can satisfy both `abstract` method specifications, because the one in interface `Colorable` requires the same method to return no value, while the one in class `Colored` requires the same method to return a value of type `int` (§8.4).

A class type should be declared `abstract` only if the intent is that subclasses can be created to complete the implementation. If the intent is simply to prevent instantiation of a class, the proper way to express this is to declare a constructor (§8.8.10) of no arguments, make it `private`, never invoke it, and declare no other constructors. A class of this form usually contains class methods and variables.

The class `Math` is an example of a class that cannot be instantiated; its declaration looks like this:

```
public final class Math {
    private Math() { }  // never instantiate this class
    . . . declarations of class variables and methods . . .
}
```

8.1.1.2 `sealed`, `non-sealed`, *and* `final` *Classes*

A class can be declared `sealed` if all its direct subclasses are known when the class is declared (§8.1.6), and no other direct subclasses are desired or required .

Explicit and exhaustive control over a class's direct subclasses is useful when the class hierarchy is used to model the kinds of values in a domain, rather than as a mechanism for code inheritance and reuse. The direct subclasses may themselves be declared `sealed` in order to further control the class hierarchy.

A class can be declared `final` if its definition is complete and no subclasses are desired or required.

It is a compile-time error if a class is declared both `final` and `abstract`, because the implementation of such a class could never be completed (§8.1.1.1).

Because a `final` class never has any subclasses, the methods of a `final` class are never overridden (§8.4.8.1).

A class is *freely extensible* if its direct superclass is not `sealed` (§8.1.4), and none of its direct superinterfaces are `sealed` (§8.1.5), and it is neither `sealed` nor `final` itself.

A class that has a `sealed` direct superclass or a `sealed` direct superinterface is freely extensible if and only if it is declared `non-sealed`.

It is a compile-time error if a class has a `sealed` direct superclass or a `sealed` direct superinterface, and is not declared `final`, `sealed`, or `non-sealed` either explicitly or implicitly.

Thus, an effect of the `sealed` keyword is to force all direct subclasses to explicitly declare whether they are `final`, `sealed`, or `non-sealed`. This avoids accidentally exposing a sealed class hierarchy to unwanted subclassing.

An enum class is either implicitly `final` or implicitly `sealed`, so it can implement a `sealed` interface. Similarly, a record class is implicitly `final`, so it can also implement a sealed interface.

It is a compile-time error if a class is declared `non-sealed` but has neither a `sealed` direct superclass nor a `sealed` direct superinterface.

Thus, a subclass of a `non-sealed` class cannot itself be declared `non-sealed`.

8.1.1.3 `strictfp` *Classes*

The `strictfp` modifier on a class declaration is obsolete and should not be used in new code. Its presence or absence has no effect at compile time or run time.

8.1.1.4 `static` *Classes*

The `static` modifier specifies that a nested class is not an inner class (§8.1.3). Just as a `static` method of a class has no current instance of the class in its body, a `static` nested class has no immediately enclosing instance in its body.

References from a `static` nested class to type parameters, instance variables, local variables, formal parameters, exception parameters, or instance methods of lexically enclosing class, interface, or method declarations are disallowed (§6.5.5.1, §6.5.6.1, and §15.12.3).

The `static` modifier does not pertain to all nested classes. It pertains only to member classes, whose declarations may use the `static` modifier, and not to local classes or anonymous classes, whose declarations may not use the `static` modifier (§14.3, §15.9.5). However, some local classes are implicitly `static`, namely local enum classes and local record classes, because all nested enum classes and nested record classes are implicitly `static` (§8.9, §8.10).

8.1.2 Generic Classes and Type Parameters

A class is *generic* if the class declaration declares one or more type variables (§4.4).

These type variables are known as the *type parameters* of the class. The type parameter section follows the class name and is delimited by angle brackets.

TypeParameters:
  < *TypeParameterList* >

TypeParameterList:
  TypeParameter {`,` *TypeParameter*}

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

TypeParameter:
  {*TypeParameterModifier*} *TypeIdentifier* [*TypeBound*]

TypeParameterModifier:
  Annotation

TypeBound:
  `extends` *TypeVariable*
  `extends` *ClassOrInterfaceType* {*AdditionalBound*}

AdditionalBound:
  `&` *InterfaceType*

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

In a class'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 class's type parameter section depends on itself.

The scope and shadowing of a class's type parameter is specified in §6.3 and §6.4.1.

References to a class's type parameter from a static context or a nested class or interface are restricted, as specified in §6.5.5.1.

A generic class 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 class at run time.

For instance, executing the code:

```
Vector<String>  x = new Vector<String>();
Vector<Integer> y = new Vector<Integer>();
boolean b = x.getClass() == y.getClass();
```

will result in the variable `b` holding the value `true`.

It is a compile-time error if a generic class is a direct or indirect subclass of `Throwable` (§11.1.1).

This restriction is needed since the catch mechanism of the Java Virtual Machine works only with non-generic classes.

Example 8.1.2-1. Mutually Recursive Type Variable Bounds

```
interface ConvertibleTo<T> {
    T convert();
}
class ReprChange<T extends ConvertibleTo<S>,
                 S extends ConvertibleTo<T>> {
    T t;
    void set(S s) { t = s.convert();    }
    S get()       { return t.convert(); }
}
```

Example 8.1.2-2. Nested Generic Classes

```
class Seq<T> {
    T      head;
    Seq<T> tail;

    Seq() { this(null, null); }
    Seq(T head, Seq<T> tail) {
        this.head = head;
        this.tail = tail;
    }
    boolean isEmpty() { return tail == null; }

    class Zipper<S> {
        Seq<Pair<T,S>> zip(Seq<S> that) {
            if (isEmpty() || that.isEmpty()) {
                return new Seq<Pair<T,S>>();
            } else {
                Seq<T>.Zipper<S> tailZipper =
                    tail.new Zipper<S>();
                return new Seq<Pair<T,S>>(
                    new Pair<T,S>(head, that.head),
                    tailZipper.zip(that.tail));
            }
        }
    }
}
class Pair<T, S> {
    T fst; S snd;
    Pair(T f, S s) { fst = f; snd = s; }
}
class Test {
    public static void main(String[] args) {
        Seq<String> strs =
            new Seq<String>(
                "a",
                new Seq<String>("b",
                                new Seq<String>()));
```

```
            Seq<Number> nums =
                new Seq<Number>(
                    new Integer(1),
                    new Seq<Number>(new Double(1.5),
                                    new Seq<Number>()));

            Seq<String>.Zipper<Number> zipper =
                strs.new Zipper<Number>();

            Seq<Pair<String,Number>> combined =
                zipper.zip(nums);
        }
    }
```

8.1.3 Inner Classes and Enclosing Instances

An *inner class* is a nested class that is not explicitly or implicitly `static`.

An inner class is one of the following:

- a member class that is not explicitly or implicitly `static` (§8.5)
- a local class that is not implicitly `static` (§14.3)
- an anonymous class (§15.9.5)

The following nested classes are implicitly `static`, so are not inner classes:

- a member enum class (§8.9)
- a local enum class (§14.3)
- a member record class (§8.10)
- a local record class (§14.3)
- a member class of an interface (§9.5)

All of the rules that apply to nested classes apply to inner classes. In particular, an inner class may declare and inherit `static` members (§8.2), and declare static initializers (§8.7), even though the inner class itself is not `static`.

There are no "inner interfaces" because every nested interface is implicitly `static` (§9.1.1.3).

Example 8.1.3-1. Inner Class Declarations and Static Members

```
class HasStatic {
    static int j = 100;
}

class Outer {
```

```
        class Inner extends HasStatic {
            static {
                System.out.println("Hello from Outer.Inner");
            }

            static       int x = 3;
            static final int y = 4;

            static void hello() {
                System.out.println("Hello from Outer.Inner.hello");
            }

            static class VeryNestedButNotInner
                extends NestedButNotInner {}
        }

        static class NestedButNotInner {
            int z = Inner.x;
        }

        interface NeverInner {}  // Implicitly static, so never inner
    }
```

Prior to Java SE 16, an inner class could not declare static initializers, and could only declare `static` members that were constant variables (§4.12.4).

A construct (statement, local variable declaration statement, local class declaration, local interface declaration, or expression) *occurs in a static context* if the innermost:

- method declaration,
- field declaration,
- constructor declaration,
- instance initializer,
- static initializer, or
- explicit constructor invocation statement

which encloses the construct is one of the following:

- a `static` method declaration (§8.4.3.2, §9.4)
- a `static` field declaration (§8.3.1.1, §9.3)
- a static initializer (§8.7)
- an explicit constructor invocation statement (§8.8.7.1)

Note that a construct which appears in a constructor declaration or an instance initializer does not occur in a static context.

The purpose of a static context is to demarcate code that must not refer explicitly or implicitly to the current instance of the class whose declaration lexically encloses the static context. Consequently, code that occurs in a static context is restricted in the following ways:

- `this` expressions (both unqualified and qualified) are disallowed (§15.8.3, §15.8.4).
- Field accesses, method invocations, and method references may not be qualified by `super` (§15.11.2, §15.12.3, §15.13.1).
- Unqualified references to instance variables of any lexically enclosing class or interface declaration are disallowed (§6.5.6.1).
- Unqualified invocations of instance methods of any lexically enclosing class or interface declaration are disallowed (§15.12.3).
- References to type parameters of any lexically enclosing class or interface declarations are disallowed (§6.5.5.1).
- References to type parameters, local variables, formal parameters, and exception parameters declared by methods or constructors of any lexically enclosing class or interface declaration *that is outside the immediately enclosing class or interface declaration* are disallowed (§6.5.5.1, §6.5.6.1).
- Declarations of local normal classes (as opposed to local enum classes) and declarations of anonymous classes both specify classes that are inner, yet when instantiated have no immediately enclosing instances (§15.9.2).
- Class instance creation expressions that instantiate inner member classes must be qualified (§15.9).

An inner class `C` is a *direct inner class of a class or interface* `O` if `O` is the immediately enclosing class or interface declaration of `C` and the declaration of `C` does not occur in a static context.

If an inner class is a local class or an anonymous class, it may be declared in a static context, and in that case is not considered an inner class of any enclosing class or interface.

A class `C` is an *inner class of class or interface* `O` if it is either a direct inner class of `O` or an inner class of an inner class of `O`.

It is unusual, but possible, for the immediately enclosing class or interface declaration of an inner class to be an interface. This only occurs if the class is a local or anonymous class declared in a `default` or `static` method body (§9.4).

A class or interface `O` is the *zeroth lexically enclosing class or interface declaration of itself*.

A class `O` is the *n'th lexically enclosing class declaration of a class* `C` if it is the immediately enclosing class declaration of the *n-1*'th lexically enclosing class declaration of `C`.

An instance `i` of a direct inner class `C` of a class or interface `O` is associated with an instance of `O`, known as the *immediately enclosing instance of* `i`. The immediately enclosing instance of an object, if any, is determined when the object is created (§15.9.2).

An object `o` is the *zeroth lexically enclosing instance of itself*.

An object `o` is the *n'th lexically enclosing instance of an instance* `i` if it is the immediately enclosing instance of the *n-1*'th lexically enclosing instance of `i`.

An instance of an inner local class or an anonymous class whose declaration occurs in a static context has no immediately enclosing instance. Also, an instance of a `static` nested class (§8.1.1.4) has no immediately enclosing instance.

For every superclass `S` of `C` which is itself a direct inner class of a class or interface `SO`, there is an instance of `SO` associated with `i`, known as the *immediately enclosing instance of* `i` *with respect to* `S`. The immediately enclosing instance of an object with respect to its class' direct superclass, if any, is determined when the superclass constructor is invoked via an explicit constructor invocation statement (§8.8.7.1).

When an inner class (whose declaration does not occur in a static context) refers to an instance variable that is a member of a lexically enclosing class or interface declaration, the variable of the corresponding lexically enclosing instance is used.

Any local variable, formal parameter, or exception parameter used but not declared in an inner class must either be `final` or effectively final (§4.12.4), as specified in §6.5.6.1.

Any local variable used but not declared in an inner class must be definitely assigned (§16 (*Definite Assignment*)) before the body of the inner class, or a compile-time error occurs.

Similar rules on variable use apply in the body of a lambda expression (§15.27.2).

A blank `final` field (§4.12.4) of a lexically enclosing class or interface declaration may not be assigned within an inner class, or a compile-time error occurs.

Example 8.1.3-2. Inner Class Declarations

```
class Outer {
    int i = 100;
    static void classMethod() {
        final int l = 200;
        class LocalInStaticContext {
            int k = i;  // Compile-time error
            int m = l;  // OK
        }
    }
```

```
        void foo() {
            class Local {  // A local class
                int j = i;
            }
        }
    }
```

The declaration of class `LocalInStaticContext` occurs in a static context due to being within the static method `classMethod`. Instance variables of class `Outer` are not available within the body of a static method. In particular, instance variables of `Outer` are not available inside the body of `LocalInStaticContext`. However, local variables from the surrounding method may be referred to without error (provided they are declared `final` or are effectively final).

Inner classes whose declarations do not occur in a static context may freely refer to the instance variables of their enclosing class declaration. An instance variable is always defined with respect to an instance. In the case of instance variables of an enclosing class declaration, the instance variable must be defined with respect to an enclosing instance of the inner class. For example, the class `Local` above has an enclosing instance of class `Outer`. As a further example:

```
class WithDeepNesting {
    boolean toBe;
    WithDeepNesting(boolean b) { toBe = b; }

    class Nested {
        boolean theQuestion;
        class DeeplyNested {
            DeeplyNested(){
                theQuestion = toBe || !toBe;
            }
        }
    }
}
```

Here, every instance of `WithDeepNesting.Nested.DeeplyNested` has an enclosing instance of class `WithDeepNesting.Nested` (its immediately enclosing instance) and an enclosing instance of class `WithDeepNesting` (its 2nd lexically enclosing instance).

8.1.4 Superclasses and Subclasses

The optional `extends` clause in a normal class declaration specifies the *direct superclass type* of the class being declared.

ClassExtends:
  `extends` *ClassType*

The `extends` clause must not appear in the definition of the class `Object`, or a compile-time error occurs, because it is the primordial class and has no direct superclass type.

The *ClassType* must name an accessible class (§6.6), or a compile-time error occurs.

It is a compile-time error if the *ClassType* names a class that is `sealed` (§8.1.1.2) and the class being declared is not a permitted direct subclass of the named class (§8.1.6).

It is a compile-time error if the *ClassType* names a class that is `final`, because `final` classes are not allowed to have subclasses (§8.1.1.2).

It is a compile-time error if the *ClassType* names the class `Enum`, which can only be extended by an enum class (§8.9), or names the class `Record`, which can only be extended by a record class (§8.10).

If the *ClassType* 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.

The direct superclass type of a class whose declaration lacks an `extends` clause is as follows:

- The class `Object` has no direct superclass type.
- For a class other than `Object` with a normal class declaration, the direct superclass type is `Object`.
- For an enum class *`E`*, the direct superclass type is `Enum<`*`E`*`>`.
- For an anonymous class, the direct superclass type is defined in §15.9.5.

The *direct superclass* of a class is the class named by its direct superclass type. The direct superclass is important because its implementation is used to derive the implementation of the class being declared.

The *superclass* relationship is the transitive closure of the direct superclass relationship. A class *`A`* is a superclass of class *`C`* if either of the following is true:

- *`A`* is the direct superclass of *`C`*.
- Where a class *`B`* is the direct superclass of *`C`*, *`A`* is a superclass of *`B`*, applying this definition recursively.

A class is said to be a *direct subclass* of its direct superclass, and a *subclass* of each of its superclasses.

Example 8.1.4-1. Direct Superclasses and Subclasses

```
class Point { int x, y; }
final class ColoredPoint extends Point { int color; }
class Colored3DPoint extends ColoredPoint { int z; }  // error
```

Here, the relationships are as follows:

- The class `Point` is a direct subclass of `Object`.
- The class `Object` is the direct superclass of the class `Point`.
- The class `ColoredPoint` is a direct subclass of class `Point`.
- The class `Point` is the direct superclass of class `ColoredPoint`.

The declaration of class `Colored3dPoint` causes a compile-time error because it attempts to extend the final class `ColoredPoint`.

Example 8.1.4-2. Superclasses and Subclasses

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
final class Colored3dPoint extends ColoredPoint { int z; }
```

Here, the relationships are as follows:

- The class `Point` is a superclass of class `ColoredPoint`.
- The class `Point` is a superclass of class `Colored3dPoint`.
- The class `ColoredPoint` is a subclass of class `Point`.
- The class `ColoredPoint` is a superclass of class `Colored3dPoint`.
- The class `Colored3dPoint` is a subclass of class `ColoredPoint`.
- The class `Colored3dPoint` is a subclass of class `Point`.

A class *`C`* *directly depends* on a class or interface *`A`* if *`A`* is mentioned in the `extends` or `implements` clause of *`C`* either as a superclass or superinterface, or as a qualifier in the fully qualified form of a superclass or superinterface name.

A class *`C`* *depends* on a class or interface *`A`* if any of the following is true:

- *`C`* directly depends on *`A`*.
- *`C`* directly depends on an interface *`I`* that depends (§9.1.3) on *`A`*.
- *`C`* directly depends on a class *`B`* that depends on *`A`*, applying this definition recursively.

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

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

Example 8.1.4-3. Class Depends on Itself

```
class Point extends ColoredPoint { int x, y; }
class ColoredPoint extends Point { int color; }
```

This program causes a compile-time error because class `Point` depends on itself.

8.1.5 Superinterfaces

The optional `implements` clause in a class declaration specifies the *direct superinterface types* of the class being declared.

ClassImplements:
  `implements` *InterfaceTypeList*

InterfaceTypeList:
  InterfaceType {`,` *InterfaceType*}

Each *InterfaceType* must name an accessible interface (§6.6), or a compile-time error occurs.

It is a compile-time error if any *InterfaceType* names a interface that is `sealed` (§9.1.1.4) and the class being declared is not a permitted direct subclass of the named interface (§9.1.4).

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.

It is a compile-time error if the same interface is named by a direct superinterface type more than once in a single `implements` clause. This is true even if the interface is named in different ways.

Example 8.1.5-1. Illegal Superinterfaces

```
class Redundant implements java.lang.Cloneable, Cloneable {
    int x;
}
```

This program results in a compile-time error because the names `java.lang.Cloneable` and `Cloneable` refer to the same interface.

A class whose declaration lacks an `implements` clause has no direct superinterface types, with one exception: an anonymous class may have a superinterface type (§15.9.5).

An interface is a *direct superinterface* of a class if the interface is named by one of the direct superinterface types of the class.

An interface `I` is a *superinterface* of class `C` if any of the following is true:

- `I` is a direct superinterface of `C`.
- `C` has some direct superinterface `J` for which `I` is a superinterface, using the definition of "superinterface of an interface" given in §9.1.3.
- `I` is a superinterface of the direct superclass of `C`.

A class can have a superinterface in more than one way.

A class is said to *directly implement* its direct superinterfaces, and to *implement* all of its superinterfaces.

A class is said to be a *direct subclass* of its direct superinterfaces, and a *subclass* of all of its superinterfaces.

A class may not declare a direct superclass type and a direct superinterface type, or two direct superinterface types, which are, or which have supertypes (§4.10.2) which are, different parameterizations of the same generic interface (§9.1.2), or a parameterization of a generic interface and a raw type naming that same generic interface. In the case of such a conflict, a compile-time error occurs.

This requirement was introduced in order to support translation by type erasure (§4.6).

Example 8.1.5-2. Superinterfaces

```
interface Colorable {
    void setColor(int color);
    int getColor();
}
enum Finish { MATTE, GLOSSY }
interface Paintable extends Colorable {
    void setFinish(Finish finish);
    Finish getFinish();
}

class Point { int x, y; }
class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
    public int getColor() { return color; }
}
```

```
class PaintedPoint extends ColoredPoint implements Paintable {
    Finish finish;
    public void setFinish(Finish finish) {
        this.finish = finish;
    }
    public Finish getFinish() { return finish; }
}
```

Here, the relationships are as follows:

- The interface `Paintable` is a superinterface of class `PaintedPoint`.
- The interface `Colorable` is a superinterface of class `ColoredPoint` and of class `PaintedPoint`.
- The interface `Paintable` is a subinterface of the interface `Colorable`, and `Colorable` is a superinterface of `Paintable`, as defined in §9.1.3.

The class `PaintedPoint` has `Colorable` as a superinterface both because it is a superinterface of `ColoredPoint` and because it is a superinterface of `Paintable`.

Example 8.1.5-3. Illegal Multiple Inheritance of an Interface

```
interface I<T> {}
class B implements I<Integer> {}
class C extends B implements I<String> {}
```

Class `C` causes a compile-time error because it attempts to be a subtype of both `I<Integer>` and `I<String>`.

Unless the class being declared is `abstract`, all the `abstract` member methods of each direct superinterface must be implemented (§8.4.8.1) either by a declaration in this class or by an existing method declaration inherited from the direct superclass or a direct superinterface, because a class that is not `abstract` is not permitted to have `abstract` methods (§8.1.1.1).

Each default method (§9.4.3) of a superinterface of the class may optionally be overridden by a method in the class; if not, the default method is typically inherited and its behavior is as specified by its default body.

It is permitted for a single method declaration in a class to implement methods of more than one superinterface.

Example 8.1.5-4. Implementing Methods of a Superinterface

```
interface Colorable {
    void setColor(int color);
    int getColor();
}
class Point { int x, y; };
class ColoredPoint extends Point implements Colorable {
```

```
        int color;
    }
```

This program causes a compile-time error, because `ColoredPoint` is not an `abstract` class but fails to provide an implementation of methods `setColor` and `getColor` of the interface `Colorable`.

In the following program:

```
    interface Fish  { int getNumberOfScales(); }
    interface Piano { int getNumberOfScales(); }
    class Tuna implements Fish, Piano {
        // You can tune a piano, but can you tuna fish?
        public int getNumberOfScales() { return 91; }
    }
```

the method `getNumberOfScales` in class `Tuna` has a name, signature, and return type that matches the method declared in interface `Fish` and also matches the method declared in interface `Piano`; it is considered to implement both.

On the other hand, in a situation such as this:

```
    interface Fish       { int    getNumberOfScales(); }
    interface StringBass { double getNumberOfScales(); }
    class Bass implements Fish, StringBass {
        // This declaration cannot be correct,
        // no matter what type is used.
        public ?? getNumberOfScales() { return 91; }
    }
```

it is impossible to declare a method named `getNumberOfScales` whose signature and return type are compatible with those of both the methods declared in interface `Fish` and in interface `StringBass`, because a class cannot have multiple methods with the same signature and different primitive return types (§8.4). Therefore, it is impossible for a single class to implement both interface `Fish` and interface `StringBass` (§8.4.8).

8.1.6 Permitted Direct Subclasses

The optional `permits` clause in a normal class declaration specifies all the classes intended as direct subclasses of the class being declared (§8.1.1.2).

ClassPermits:
  `permits` *TypeName* {`,` *TypeName*}

It is a compile-time error if a class declaration has a `permits` clause but no `sealed` modifier.

Every *TypeName* must name an accessible class (§6.6), or a compile-time error occurs.

It is a compile-time error if the same class is specified more than once in a `permits` clause. This is true even if the class is named in different ways.

> The canonical name of a class does not need to be used in a `permits` clause, but a `permits` clause can only specify a class once. For example, the following program fails to compile:

```
package p;

sealed class A     permits B, C, p.B {}  // error

non-sealed class B extends A {}
non-sealed class C extends A {}
```

If a `sealed` class *C* is associated with a named module (§7.3), then every class specified in the `permits` clause of *C*'s declaration must be associated with the same module as *C*, or a compile-time error occurs.

If a `sealed` class *C* is associated with an unnamed module (§7.7.5), then every class specified in the `permits` clause of *C*'s declaration must belong to the same package as *C*, or a compile-time error occurs.

> A `sealed` class and its direct subclasses need to refer to each other in a circular fashion, in `permits` and `extends` clauses, respectively. Therefore, in a modular codebase, they must be co-located in the same module, as classes in different modules cannot refer to each other in a circular fashion. Co-location is desirable in any case because a sealed class hierarchy should always be declared within a single maintenance domain, where the same developer or group of developers is responsible for maintaining the hierarchy. A named module typically represents a maintenance domain in a modular codebase.

If the declaration of a `sealed` class *C* has a `permits` clause, then the *permitted direct subclasses* of *C* are the classes specified by the `permits` clause.

Every permitted direct subclass specified by the `permits` clause must be a direct subclass of *C* (§8.1.4), or a compile-time error occurs.

If the declaration of a `sealed` class *C* lacks a `permits` clause, then the permitted direct subclasses of *C* are as follows:

- If `C` is not an enum class, then its permitted direct subclasses are those classes declared in the same compilation unit as `C` (§7.3) which have a canonical name (§6.7) and whose direct superclass is `C`.

  That is, the permitted direct subclasses are inferred as the classes in the same compilation unit that specify `C` as their direct superclass. The requirement for a canonical name means that no local classes or anonymous classes will be considered.

  It is a compile-time error if the declaration of a `sealed` class `C` lacks a `permits` clause and `C` has no permitted direct subclasses.

- If `C` is an enum class, then its permitted direct subclasses, if any, are specified in §8.9.

8.1.7 Class Body and Member Declarations

A *class body* may contain declarations of members of the class, that is, fields (§8.3), methods (§8.4), classes, and interfaces (§8.5).

A class body may also contain instance initializers (§8.6), static initializers (§8.7), and declarations of constructors (§8.8) for the class.

ClassBody:
  `{` *{ClassBodyDeclaration}* `}`

ClassBodyDeclaration:
  ClassMemberDeclaration
  InstanceInitializer
  StaticInitializer
  ConstructorDeclaration

ClassMemberDeclaration:
  FieldDeclaration
  MethodDeclaration
  ClassDeclaration
  InterfaceDeclaration
  `;`

The scope and shadowing of a declaration of a member `m` declared in or inherited by a class `C` is specified in §6.3 and §6.4.1.

If `C` is a nested class, there may be definitions of the same kind (variable, method, or type) and name as `m` in enclosing scopes. (The scopes may be blocks, classes, or packages.) In all such cases, the member `m` declared in or inherited by `C` shadows the other definitions of the same kind and name.

8.2 Class Members

The members of a class are all of the following:

- Members inherited from its direct superclass type (§8.1.4), except in the class `Object`, which has no direct superclass type
- Members inherited from any direct superinterface types (§8.1.5)
- Members declared in the body of the class (§8.1.7)

Members of a class that are declared `private` are not inherited by subclasses of that class.

Only members of a class that are declared `protected` or `public` are inherited by subclasses declared in a package other than the one in which the class is declared.

Constructors, static initializers, and instance initializers are not members and therefore are not inherited.

We use the phrase *the type of a member* to denote:

- For a field, its type.
- For a method, an ordered 4-tuple consisting of:
  - type parameters: the declarations of any type parameters of the method member.
  - argument types: a list of the types of the arguments to the method member.
  - return type: the return type of the method member.
  - `throws` clause: exception types declared in the `throws` clause of the method member.

Fields, methods, member classes, and member interfaces of a class 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.

Example 8.2-1. Use of Class Members

```
class Point {
    int x, y;
    private Point() { reset(); }
```

```
    Point(int x, int y) { this.x = x; this.y = y; }
    private void reset() { this.x = 0; this.y = 0; }
}
class ColoredPoint extends Point {
    int color;
    void clear() { reset(); }  // error
}
class Test {
    public static void main(String[] args) {
        ColoredPoint c = new ColoredPoint(0, 0);  // error
        c.reset();  // error
    }
}
```

This program causes four compile-time errors.

One error occurs because `ColoredPoint` has no constructor declared with two `int` parameters, as requested by the use in `main`. This illustrates the fact that `ColoredPoint` does not inherit the constructors of its superclass `Point`.

Another error occurs because `ColoredPoint` declares no constructors, and therefore a default constructor for it is implicitly declared (§8.8.9), and this default constructor is equivalent to:

```
ColoredPoint() { super(); }
```

which invokes the constructor, with no arguments, for the direct superclass of the class `ColoredPoint`. The error is that the constructor for `Point` that takes no arguments is `private`, and therefore is not accessible outside the class `Point`, even through a superclass constructor invocation (§8.8.7).

Two more errors occur because the method `reset` of class `Point` is `private`, and therefore is not inherited by class `ColoredPoint`. The method invocations in method `clear` of class `ColoredPoint` and in method `main` of class `Test` are therefore not correct.

Example 8.2-2. Inheritance of Class Members with Package Access

Consider the example where the `points` package declares two compilation units:

```
package points;
public class Point {
    int x, y;
    public void move(int dx, int dy) { x += dx; y += dy; }
}
```

and:

```
package points;
public class Point3d extends Point {
    int z;
```

```
        public void move(int dx, int dy, int dz) {
            x += dx; y += dy; z += dz;
        }
    }
```

and a third compilation unit, in another package, is:

```
    import points.Point3d;
    class Point4d extends Point3d {
        int w;
        public void move(int dx, int dy, int dz, int dw) {
            x += dx; y += dy; z += dz; w += dw; // compile-time errors
        }
    }
```

Here both classes in the `points` package compile. The class `Point3d` inherits the fields `x` and `y` of class `Point`, because it is in the same package as `Point`. The class `Point4d`, which is in a different package, does not inherit the fields `x` and `y` of class `Point` or the field `z` of class `Point3d`, and so fails to compile.

A better way to write the third compilation unit would be:

```
    import points.Point3d;
    class Point4d extends Point3d {
        int w;
        public void move(int dx, int dy, int dz, int dw) {
            super.move(dx, dy, dz); w += dw;
        }
    }
```

using the `move` method of the superclass `Point3d` to process `dx`, `dy`, and `dz`. If `Point4d` is written in this way, it will compile without errors.

Example 8.2-3. Inheritance of `public` and `protected` Class Members

Given the class `Point`:

```
    package points;
    public class Point {
        public int x, y;
        protected int useCount = 0;
        static protected int totalUseCount = 0;
        public void move(int dx, int dy) {
            x += dx; y += dy; useCount++; totalUseCount++;
        }
    }
```

the `public` and `protected` fields `x`, `y`, `useCount`, and `totalUseCount` are inherited in all subclasses of `Point`.

Therefore, this test program, in another package, can be compiled successfully:

```
class Test extends points.Point {
    public void moveBack(int dx, int dy) {
        x -= dx; y -= dy; useCount++; totalUseCount++;
    }
}
```

Example 8.2-4. Inheritance of `private` Class Members

```
class Point {
    int x, y;
    void move(int dx, int dy) {
        x += dx; y += dy; totalMoves++;
    }
    private static int totalMoves;
    void printMoves() { System.out.println(totalMoves); }
}
class Point3d extends Point {
    int z;
    void move(int dx, int dy, int dz) {
        super.move(dx, dy); z += dz; totalMoves++; // error
    }
}
```

Here, the class variable `totalMoves` can be used only within the class `Point`; it is not inherited by the subclass `Point3d`. A compile-time error occurs because method move of class `Point3d` tries to increment `totalMoves`.

Example 8.2-5. Accessing Members of Inaccessible Classes

Even though a class might not be declared `public`, instances of the class might be available at run time to code outside the package in which it is declared by means of a `public` superclass or superinterface. An instance of the class can be assigned to a variable of such a `public` type. An invocation of a `public` method of the object referred to by such a variable may invoke a method of the class if it implements or overrides a method of the `public` superclass or superinterface. (In this situation, the method is necessarily declared `public`, even though it is declared in a class that is not `public`.)

Consider the compilation unit:

```
package points;
public class Point {
    public int x, y;
    public void move(int dx, int dy) {
        x += dx; y += dy;
    }
}
```

and another compilation unit of another package:

```
package morePoints;
class Point3d extends points.Point {
    public int z;
    public void move(int dx, int dy, int dz) {
        super.move(dx, dy); z += dz;
    }
    public void move(int dx, int dy) {
        move(dx, dy, 0);
    }
}
public class OnePoint {
    public static points.Point getOne() {
        return new Point3d();
    }
}
```

An invocation `morePoints.OnePoint.getOne()` in yet a third package would return a `Point3d` that can be used as a `Point`, even though the type `Point3d` is not available outside the package `morePoints`. The two-argument version of method `move` could then be invoked for that object, which is permissible because method `move` of `Point3d` is `public` (as it must be, for any method that overrides a `public` method must itself be `public`, precisely so that situations such as this will work out correctly). The fields `x` and `y` of that object could also be accessed from such a third package.

While the field `z` of class `Point3d` is `public`, it is not possible to access this field from code outside the package `morePoints`, given only a reference to an instance of class `Point3d` in a variable `p` of type `Point`. This is because the expression `p.z` is not correct, as `p` has type `Point` and class `Point` has no field named `z`; also, the expression `((Point3d)p).z` is not correct, because the class type `Point3d` cannot be referred to outside package `morePoints`.

The declaration of the field `z` as `public` is not useless, however. If there were to be, in package `morePoints`, a `public` subclass `Point4d` of the class `Point3d`:

```
package morePoints;
public class Point4d extends Point3d {
    public int w;
    public void move(int dx, int dy, int dz, int dw) {
        super.move(dx, dy, dz); w += dw;
    }
}
```

then class `Point4d` would inherit the field `z`, which, being `public`, could then be accessed by code in packages other than `morePoints`, through variables and expressions of the `public` type `Point4d`.

8.3 Field Declarations

The variables of a class are introduced by *field declarations*.

FieldDeclaration:
{FieldModifier} UnannType VariableDeclaratorList `;`

VariableDeclaratorList:
VariableDeclarator {`,` *VariableDeclarator}*

VariableDeclarator:
VariableDeclaratorId [= VariableInitializer]

VariableDeclaratorId:
Identifier [Dims]

VariableInitializer:
Expression
ArrayInitializer

UnannType:
UnannPrimitiveType
UnannReferenceType

UnannPrimitiveType:
NumericType
`boolean`

UnannReferenceType:
UnannClassOrInterfaceType
UnannTypeVariable
UnannArrayType

UnannClassOrInterfaceType:
UnannClassType
UnannInterfaceType

UnannClassType:
TypeIdentifier [TypeArguments]
PackageName `.` *{Annotation} TypeIdentifier [TypeArguments]*
UnannClassOrInterfaceType `.` *{Annotation} TypeIdentifier [TypeArguments]*

UnannInterfaceType:
UnannClassType

UnannTypeVariable:
  TypeIdentifier

UnannArrayType:
  UnannPrimitiveType Dims
  UnannClassOrInterfaceType Dims
  UnannTypeVariable Dims

The following production from §4.3 is shown here for convenience:

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

Each declarator in a *FieldDeclaration* declares one field. The *Identifier* in a declarator may be used in a name to refer to the field.

More than one field may be declared in a single *FieldDeclaration* by using more than one declarator; the *FieldModifier*s and *UnannType* apply to all the declarators in the declaration.

The *FieldModifier* clause is described in §8.3.1.

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 a field declaration is specified in §6.3 and §6.4.1.

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

If a class 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 superclasses, and superinterfaces of the class.

In this respect, hiding of fields differs from hiding of methods (§8.4.8.3), for there is no distinction drawn between `static` and non-`static` fields in field hiding whereas a distinction is drawn between `static` and non-`static` methods in method hiding.

A hidden field can be accessed by using a qualified name (§6.5.6.2) if it is `static`, or by using a field access expression that contains the keyword `super` (§15.11.2) or a cast to a superclass type.

In this respect, hiding of fields is similar to hiding of methods.

If a field declaration hides the declaration of another field, the two fields need not have the same type.

A class inherits from its direct superclass and direct superinterfaces all the non-`private` fields of the superclass and superinterfaces that are both accessible (§6.6) to code in the class and not hidden by a declaration in the class.

A `private` field of a superclass might be accessible to a subclass - for example, if both classes are members of the same class. Nevertheless, a `private` field is never inherited by a subclass.

It is possible for a class to inherit more than one field with the same name, either from its superclass and superinterfaces or from its superinterfaces alone. Such a situation does not in itself cause a compile-time error. However, any attempt within the body of the class to refer to any such field by its simple name will result in a compile-time error, because the reference is ambiguous.

There might be several paths by which the same field declaration is 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 8.3-1. Multiply Inherited Fields

A class may inherit two or more fields with the same name, either from its superclass and a superinterface or from two superinterfaces. A compile-time error occurs on any attempt to refer to any ambiguously inherited field by its simple name. A qualified name or a field access expression that contains the keyword `super` (§15.11.2) may be used to access such fields unambiguously. In the program:

```
interface Frob  { float v = 2.0f; }
class SuperTest { int   v = 3; }
class Test extends SuperTest implements Frob {
    public static void main(String[] args) {
        new Test().printV();
    }
    void printV() { System.out.println(v); }
}
```

the class `Test` inherits two fields named `v`, one from its superclass `SuperTest` and one from its superinterface `Frob`. This in itself is permitted, but a compile-time error occurs because of the use of the simple name `v` in method `printV`: it cannot be determined which `v` is intended.

The following variation uses the field access expression `super.v` to refer to the field named `v` declared in class `SuperTest` and uses the qualified name `Frob.v` to refer to the field named `v` declared in interface `Frob`:

```
interface Frob  { float v = 2.0f; }
class SuperTest { int   v = 3; }
class Test extends SuperTest implements Frob {
    public static void main(String[] args) {
        new Test().printV();
```

```
        }
        void printV() {
            System.out.println((super.v + Frob.v)/2);
        }
    }
```

It compiles and prints:

```
2.5
```

Even if two distinct inherited fields have the same type, the same value, and are both `final`, any reference to either field by simple name is considered ambiguous and results in a compile-time error. In the program:

```
interface Color        { int RED=0, GREEN=1,  BLUE=2;  }
interface TrafficLight { int RED=0, YELLOW=1, GREEN=2; }
class Test implements Color, TrafficLight {
    public static void main(String[] args) {
        System.out.println(GREEN);  // compile-time error
        System.out.println(RED);    // compile-time error
    }
}
```

it is not astonishing that the reference to `GREEN` should be considered ambiguous, because class `Test` inherits two different declarations for `GREEN` with different values. The point of this example is that the reference to `RED` is also considered ambiguous, because two distinct declarations are inherited. The fact that the two fields named `RED` happen to have the same type and the same unchanging value does not affect this judgment.

Example 8.3-2. Re-inheritance of Fields

If the same field declaration is inherited from an interface by multiple paths, the field is considered to be inherited only once. It may be referred to by its simple name without ambiguity. For example, in the code:

```
interface Colorable {
    int RED = 0xff0000, GREEN = 0x00ff00, BLUE = 0x0000ff;
}
interface Paintable extends Colorable {
    int MATTE = 0, GLOSSY = 1;
}
class Point { int x, y; }
class ColoredPoint extends Point implements Colorable {}
class PaintedPoint extends ColoredPoint implements Paintable {
    int p = RED;
}
```

the fields `RED`, `GREEN`, and `BLUE` are inherited by the class `PaintedPoint` both through its direct superclass `ColoredPoint` and through its direct superinterface `Paintable`. The simple names `RED`, `GREEN`, and `BLUE` may nevertheless be used without ambiguity within the class `PaintedPoint` to refer to the fields declared in interface `Colorable`.

8.3.1 Field Modifiers

FieldModifier:
 (one of)
 Annotation `public` `protected` `private`
 `static` `final` `transient` `volatile`

The rules concerning annotation modifiers for a field declaration are specified in §9.7.4 and §9.7.5.

It is a compile-time error if the same keyword appears more than once as a modifier for a field declaration, or if a field declaration has more than one of the access modifiers `public`, `protected`, and `private` (§6.6).

> 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 *FieldModifier*.

8.3.1.1 `static` *Fields*

If a field is declared `static`, there exists exactly one incarnation of the field, no matter how many instances (possibly zero) of the class may eventually be created. A `static` field, sometimes called a *class variable*, is incarnated when the class is initialized (§12.4).

A field that is not declared `static` is called an *instance variable*, and sometimes called a non-`static` field. Whenever a new instance of a class is created (§12.5), a new variable associated with that instance is created for every instance variable declared in that class or any of its superclasses.

The declaration of a class variable introduces a static context (§8.1.3), which limits the use of constructs that refer to the current object. Notably, the keywords `this` and `super` are prohibited in a static context (§15.8.3, §15.11.2), as are unqualified references to instance variables, instance methods, and type parameters of lexically enclosing declarations (§6.5.5.1, §6.5.6.1, §15.12.3).

References to an instance variable from a static context or a nested class or interface are restricted, as specified in §6.5.6.1.

Example 8.3.1.1-1. `static` **Fields**

```
class Point {
    int x, y, useCount;
    Point(int x, int y) { this.x = x; this.y = y; }
    static final Point origin = new Point(0, 0);
}
class Test {
```

```
        public static void main(String[] args) {
            Point p = new Point(1,1);
            Point q = new Point(2,2);
            p.x = 3;
            p.y = 3;
            p.useCount++;
            p.origin.useCount++;
            System.out.println("(" + q.x + "," + q.y + ")");
            System.out.println(q.useCount);
            System.out.println(q.origin == Point.origin);
            System.out.println(q.origin.useCount);
        }
    }
```

This program prints:

```
(2,2)
0
true
1
```

showing that changing the fields `x`, `y`, and `useCount` of `p` does not affect the fields of `q`, because these fields are instance variables in distinct objects. In this example, the class variable `origin` of the class `Point` is referenced both using the class name as a qualifier, in `Point.origin`, and using variables of the class type in field access expressions (§15.11), as in `p.origin` and `q.origin`. These two ways of accessing the `origin` class variable access the same object, evidenced by the fact that the value of the reference equality expression (§15.21.3):

```
q.origin==Point.origin
```

is true. Further evidence is that the incrementation:

```
p.origin.useCount++;
```

causes the value of `q.origin.useCount` to be `1`; this is so because `p.origin` and `q.origin` refer to the same variable.

Example 8.3.1.1-2. Hiding of Class Variables

```
class Point {
    static int x = 2;
}
class Test extends Point {
    static double x = 4.7;
    public static void main(String[] args) {
        new Test().printX();
    }
    void printX() {
        System.out.println(x + " " + super.x);
    }
```

```
}
```

This program produces the output:

```
4.7 2
```

because the declaration of `x` in class `Test` hides the definition of `x` in class `Point`, so class `Test` does not inherit the field `x` from its superclass `Point`. Within the declaration of class `Test`, the simple name `x` refers to the field declared within class `Test`. Code in class `Test` may refer to the field `x` of class `Point` as `super.x` (or, because `x` is `static`, as `Point.x`). If the declaration of `Test.x` is deleted:

```
class Point {
    static int x = 2;
}
class Test extends Point {
    public static void main(String[] args) {
        new Test().printX();
    }
    void printX() {
        System.out.println(x + " " + super.x);
    }
}
```

then the field `x` of class `Point` is no longer hidden within class `Test`; instead, the simple name `x` now refers to the field `Point.x`. Code in class `Test` may still refer to that same field as `super.x`. Therefore, the output from this variant program is:

```
2 2
```

Example 8.3.1.1-3. Hiding of Instance Variables

```
class Point {
    int x = 2;
}
class Test extends Point {
    double x = 4.7;
    void printBoth() {
        System.out.println(x + " " + super.x);
    }
    public static void main(String[] args) {
        Test sample = new Test();
        sample.printBoth();
        System.out.println(sample.x + " " + ((Point)sample).x);
    }
}
```

This program produces the output:

```
4.7 2
4.7 2
```

because the declaration of `x` in class `Test` hides the definition of `x` in class `Point`, so class `Test` does not inherit the field `x` from its superclass `Point`. It must be noted, however, that while the field `x` of class `Point` is not inherited by class `Test`, it is nevertheless *implemented* by instances of class `Test`. In other words, every instance of class `Test` contains two fields, one of type `int` and one of type `double`. Both fields bear the name `x`, but within the declaration of class `Test`, the simple name `x` always refers to the field declared within class `Test`. Code in instance methods of class `Test` may refer to the instance variable `x` of class `Point` as `super.x`.

Code that uses a field access expression to access field `x` will access the field named `x` in the class indicated by the type of reference expression. Thus, the expression `sample.x` accesses a `double` value, the instance variable declared in class `Test`, because the type of the variable `sample` is `Test`, but the expression `((Point)sample).x` accesses an `int` value, the instance variable declared in class `Point`, because of the cast to type `Point`.

If the declaration of `x` is deleted from class `Test`, as in the program:

```
class Point {
    static int x = 2;
}
class Test extends Point {
    void printBoth() {
        System.out.println(x + " " + super.x);
    }
    public static void main(String[] args) {
        Test sample = new Test();
        sample.printBoth();
        System.out.println(sample.x + " " + ((Point)sample).x);
    }
}
```

then the field `x` of class `Point` is no longer hidden within class `Test`. Within instance methods in the declaration of class `Test`, the simple name `x` now refers to the field declared within class `Point`. Code in class `Test` may still refer to that same field as `super.x`. The expression `sample.x` still refers to the field `x` within type `Test`, but that field is now an inherited field, and so refers to the field `x` declared in class `Point`. The output from this variant program is:

```
2 2
2 2
```

8.3.1.2 `final` *Fields*

A field can be declared `final` (§4.12.4). Both class and instance variables (`static` and non-`static` fields) may be declared `final`.

A blank `final` class variable must be definitely assigned by a static initializer of the class in which it is declared, or a compile-time error occurs (§8.7, §16.8).

A blank `final` instance variable must be definitely assigned and moreover not definitely unassigned at the end of every constructor of the class in which it is declared, or a compile-time error occurs (§8.8, §16.9).

8.3.1.3 `transient` *Fields*

Variables may be marked `transient` to indicate that they are not part of the persistent state of an object.

Example 8.3.1.3-1. Persistence of `transient` Fields

If an instance of the class `Point`:

```
class Point {
    int x, y;
    transient float rho, theta;
}
```

were saved to persistent storage by a system service, then only the fields `x` and `y` would be saved. This specification does not specify details of such services; see the specification of `java.io.Serializable` for an example of such a service.

8.3.1.4 `volatile` *Fields*

The Java programming language allows threads to access shared variables (§17.1). As a rule, to ensure that shared variables are consistently and reliably updated, a thread should ensure that it has exclusive use of such variables by obtaining a lock that, conventionally, enforces mutual exclusion for those shared variables.

The Java programming language provides a second mechanism, `volatile` fields, that is more convenient than locking for some purposes.

A field may be declared `volatile`, in which case the Java Memory Model ensures that all threads see a consistent value for the variable (§17.4).

It is a compile-time error if a `final` variable is also declared `volatile`.

Example 8.3.1.4-1. `volatile` Fields

If, in the following example, one thread repeatedly calls the method `one` (but no more than `Integer.MAX_VALUE` times in all), and another thread repeatedly calls the method `two`:

```
class Test {
    static int i = 0, j = 0;
    static void one() { i++; j++; }
    static void two() {
        System.out.println("i=" + i + " j=" + j);
    }
```

```
}
```

then method `two` could occasionally print a value for `j` that is greater than the value of `i`, because the example includes no synchronization and, under the rules explained in §17.4, the shared values of `i` and `j` might be updated out of order.

One way to prevent this out-or-order behavior would be to declare methods `one` and `two` to be `synchronized` (§8.4.3.6):

```
class Test {
    static int i = 0, j = 0;
    static synchronized void one() { i++; j++; }
    static synchronized void two() {
        System.out.println("i=" + i + " j=" + j);
    }
}
```

This prevents method `one` and method `two` from being executed concurrently, and furthermore guarantees that the shared values of `i` and `j` are both updated before method `one` returns. Therefore method `two` never observes a value for `j` greater than that for `i`; indeed, it always observes the same value for `i` and `j`.

Another approach would be to declare `i` and `j` to be `volatile`:

```
class Test {
    static volatile int i = 0, j = 0;
    static void one() { i++; j++; }
    static void two() {
        System.out.println("i=" + i + " j=" + j);
    }
}
```

This allows method `one` and method `two` to be executed concurrently, but guarantees that accesses to the shared values for `i` and `j` occur exactly as many times, and in exactly the same order, as they appear to occur during execution of the program text by each thread. Therefore, the shared value for `j` is never greater than that for `i`, because each update to `i` must be reflected in the shared value for `i` before the update to `j` occurs. It is possible, however, that any given invocation of method `two` might observe a value for `j` that is much greater than the value observed for `i`, because method `one` might be executed many times between the moment when method `two` fetches the value of `i` and the moment when method two fetches the value of `j`.

See §17.4 for more discussion and examples.

8.3.2 Field Initialization

If a declarator in a field declaration has a *variable initializer*, then the declarator has the semantics of an assignment (§15.26) to the declared variable.

If the declarator is for a class variable (that is, a `static` field) (§8.3.1.1), then the following rules apply to its initializer:

- The initializer may not refer to the current object using the keyword `this` or the keyword `super`, as specified in §15.8.3 and §15.11.2, nor refer by simple name to any instance variable or instance method, as specified in §6.5.6.1 and §15.12.3.
- At run time, the initializer is evaluated and the assignment performed exactly once, when the class is initialized (§12.4.2).

Note that `static` fields that are constant variables (§4.12.4) are initialized before other `static` fields (§12.4.2, step 6). This also applies in interfaces (§9.3.1). When such fields are referenced by simple name, they will never be observed to have their default initial values (§4.12.5).

If the declarator is for an instance variable (that is, a field that is not `static`), then the following rules apply to its initializer:

- The initializer may refer to the current object using the keyword `this` or the keyword `super`, and may refer by simple name to any class variable declared in or inherited by the class, even one whose declaration occurs to the right of the initializer (§3.5).
- At run time, the initializer is evaluated and the assignment performed each time an instance of the class is created (§12.5).

References from variable initializers to fields that may not yet be initialized are restricted, as specified in §8.3.3 and §16 (*Definite Assignment*).

Exception checking for a variable initializer in a field declaration is specified in §11.2.3.

Variable initializers are also used in local variable declaration statements (§14.4), where the initializer is evaluated and the assignment performed each time the local variable declaration statement is executed.

Example 8.3.2-1. Field Initialization

```
class Point {
    int x = 1, y = 5;
}
class Test {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println(p.x + ", " + p.y);
    }
}
```

This program produces the output:

```
1, 5
```

because the assignments to `x` and `y` occur whenever a new `Point` is created.

Example 8.3.2-2. Forward Reference to a Class Variable

```
class Test {
    float f = j;
    static int j = 1;
}
```

This program compiles without error; it initializes `j` to `1` when class `Test` is initialized, and initializes `f` to the current value of `j` every time an instance of class `Test` is created.

8.3.3 Restrictions on Field References in Initializers

References to a field are sometimes restricted, even through the field is in scope. The following rules constrain forward references to a field (where the use textually precedes the field declaration) as well as self-reference (where the field is used in its own initializer).

For a reference by simple name to a class variable *`f`* declared in class or interface *`C`*, it is a compile-time error if:

- The reference appears either in a class variable initializer of *`C`* or in a static initializer of *`C`* (§8.7); and
- The reference appears either in the initializer of *`f`*'s own declarator or at a point to the left of *`f`*'s declarator; and
- The reference is *not* on the left hand side of an assignment expression (§15.26); and
- The innermost class or interface enclosing the reference is *`C`*.

For a reference by simple name to an instance variable *`f`* declared in class *`C`*, it is a compile-time error if:

- The reference appears either in an instance variable initializer of *`C`* or in an instance initializer of *`C`* (§8.6); and
- The reference appears in the initializer of *`f`*'s own declarator or at a point to the left of *`f`*'s declarator; and
- The reference is *not* on the left hand side of an assignment expression (§15.26); and
- The innermost class enclosing the reference is *`C`*.

Example 8.3.3-1. Restrictions on Field References

A compile-time error occurs for this program:

```
class Test1 {
    int i = j;  // compile-time error:
                // incorrect forward reference
    int j = 1;
}
```

whereas the following program compiles without error:

```
class Test2 {
    Test2() { k = 2; }
    int j = 1;
    int i = j;
    int k;
}
```

even though the constructor for `Test2` (§8.8) refers to the field `k` that is declared three lines later.

The restrictions above are designed to catch, at compile time, circular or otherwise malformed initializations. Thus, both:

```
class Z {
    static int i = j + 2;
    static int j = 4;
}
```

and:

```
class Z {
    static { i = j + 2; }
    static int i, j;
    static { j = 4; }
}
```

result in compile-time errors. Accesses by methods are not checked in this way, so:

```
class Z {
    static int peek() { return j; }
    static int i = peek();
    static int j = 1;
}
class Test {
    public static void main(String[] args) {
        System.out.println(Z.i);
    }
}
```

produces the output:

```
0
```

because the variable initializer for `i` uses the class method `peek` to access the value of the variable `j` before `j` has been initialized by its variable initializer, at which point it still has its default value (§4.12.5).

A more elaborate example is:

```
class UseBeforeDeclaration {
    static {
        x = 100;
          // ok - assignment
        int y = x + 1;
          // error - read before declaration
        int v = x = 3;
          // ok - x at left hand side of assignment
        int z = UseBeforeDeclaration.x * 2;
          // ok - not accessed via simple name

        Object o = new Object() {
            void foo() { x++; }
              // ok - occurs in a different class
            { x++; }
              // ok - occurs in a different class
        };
    }

    {
        j = 200;
          // ok - assignment
        j = j + 1;
          // error - right hand side reads before declaration
        int k = j = j + 1;
          // error - illegal forward reference to j
        int n = j = 300;
          // ok - j at left hand side of assignment
        int h = j++;
          // error - read before declaration
        int l = this.j * 3;
          // ok - not accessed via simple name

        Object o = new Object() {
            void foo(){ j++; }
              // ok - occurs in a different class
            { j = j + 1; }
              // ok - occurs in a different class
        };
    }

    int w = x = 3;
```

```
          // ok - x at left hand side of assignment
        int p = x;
          // ok - instance initializers may access static fields

        static int u =
            (new Object() { int bar() { return x; } }).bar();
         // ok - occurs in a different class

        static int x;

        int m = j = 4;
          // ok - j at left hand side of assignment
        int o =
            (new Object() { int bar() { return j; } }).bar();
            // ok - occurs in a different class
        int j;
    }
```

8.4 Method Declarations

A *method* declares executable code that can be invoked, passing a fixed number of values as arguments.

MethodDeclaration:
  {MethodModifier} MethodHeader MethodBody

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

MethodDeclarator:
  Identifier `(` *[ReceiverParameter* `,`*] [FormalParameterList]* `)` *[Dims]*

ReceiverParameter:
  {Annotation} UnannType [Identifier `.`*]* `this`

The following production from §4.3 is shown here for convenience:

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

The *FormalParameterList* clause is described in §8.4.1, the *MethodModifier* clause in §8.4.3, the *TypeParameters* clause in §8.4.4, the *Result* clause in §8.4.5, the *Throws* clause in §8.4.6, and the *MethodBody* in §8.4.7.

The *Identifier* in a *MethodDeclarator* may be used in a name to refer to the method (§6.5.7.1, §15.12).

The scope and shadowing of a method declaration is specified in §6.3 and §6.4.1.

The *receiver parameter* is an optional syntactic device for an instance method or an inner class's constructor. For an instance method, the receiver parameter represents the object for which the method is invoked. For an inner class's constructor, the receiver parameter represents the immediately enclosing instance of the newly constructed object. In both cases, the receiver parameter exists solely to allow the type of the represented object to be denoted in source code, so that the type may be annotated (§9.7.4). The receiver parameter is not a formal parameter; more precisely, it is not a declaration of any kind of variable (§4.12.3), it is never bound to any value passed as an argument in a method invocation expression or class instance creation expression, and it has no effect whatsoever at run time.

A receiver parameter may appear either in the *MethodDeclarator* of an instance method or in the *ConstructorDeclarator* of a constructor of an inner class where the inner class is not declared in a static context (§8.1.3). If a receiver parameter appears in any other kind of method or constructor, then a compile-time error occurs.

The type and name of a receiver parameter are constrained as follows:

- In an instance method, the type of the receiver parameter must be the class or interface in which the method is declared, and the name of the receiver parameter must be `this`; otherwise, a compile-time error occurs.
- In an inner class's constructor, the type of the receiver parameter must be the class or interface which is the immediately enclosing type declaration of the inner class, and the name of the receiver parameter must be *Identifier* `. this` where *Identifier* is the simple name of the class or interface which is the immediately enclosing type declaration of the inner class; otherwise, a compile-time error occurs.

It is a compile-time error for the body of a class declaration to declare as members two methods with override-equivalent signatures (§8.4.2).

The declaration of a method that returns an array is allowed to place some or all of the bracket pairs that denote the array type after the 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.

8.4.1 Formal Parameters

The *formal parameters* of a method or constructor, if any, are specified by a list of comma-separated parameter specifiers. Each parameter specifier consists of a type (optionally preceded by the `final` modifier and/or one or more annotations) and an identifier (optionally followed by brackets) that specifies the name of the parameter.

If a method or constructor has no formal parameters, and no receiver parameter, then an empty pair of parentheses appears in the declaration of the method or constructor.

FormalParameterList:
  FormalParameter {`,` *FormalParameter*}

FormalParameter:
  {*VariableModifier*} *UnannType VariableDeclaratorId*
  VariableArityParameter

VariableArityParameter:
  {*VariableModifier*} *UnannType* {*Annotation*} `...` *Identifier*

VariableModifier:
  Annotation
  `final`

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

VariableDeclaratorId:
  Identifier [*Dims*]

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

A formal parameter of a method or constructor may be a *variable arity parameter*, indicated by an ellipsis following the type. At most one variable arity parameter is permitted for a method or constructor. It is a compile-time error if a variable arity parameter appears anywhere in the list of parameter specifiers except the last position.

In the grammar for *VariableArityParameter*, note that the ellipsis (`...`) is a token unto itself (§3.11). It is possible to put whitespace between it and the type, but this is discouraged as a matter of style.

If the last formal parameter of a method is a variable arity parameter, the method is a *variable arity method*. Otherwise, it is a *fixed arity method*.

The rules concerning annotation modifiers for a formal parameter declaration and for a receiver parameter are specified in §9.7.4 and §9.7.5.

It is a compile-time error if `final` appears more than once as a modifier for a formal parameter declaration.

The scope and shadowing of a formal parameter is specified in §6.3 and §6.4.

References to a formal parameter from a nested class or interface, or a lambda expression, are restricted, as specified in §6.5.6.1.

It is a compile-time error for a method or constructor to declare two formal parameters with the same name. (That is, their declarations mention the same *Identifier*.)

It is a compile-time error if a formal parameter that is declared `final` is assigned to within the body of the method or constructor.

The declared type of a formal parameter depends on whether it is a variable arity parameter:

- If the formal parameter is not a variable arity parameter, then the declared type is denoted by *UnannType* if no bracket pairs appear in *UnannType* and *VariableDeclaratorId*, and specified by §10.2 otherwise.
- If the formal parameter is a variable arity parameter, then the declared type is an array type specified by §10.2.

If the declared type of a variable arity parameter has a non-reifiable element type (§4.7), then a compile-time unchecked warning occurs for the declaration of the variable arity method, unless the method is annotated with `@SafeVarargs` (§9.6.4.7) or the warning is suppressed by `@SuppressWarnings` (§9.6.4.5).

When the method or constructor is invoked (§15.12), the values of the actual argument expressions initialize newly created parameter variables, each of the declared type, before execution of the body of the method or constructor. The *Identifier* that appears in the *FormalParameter* may be used as a simple name in the body of the method or constructor to refer to the formal parameter.

Invocations of a variable arity method may contain more actual argument expressions than formal parameters. All the actual argument expressions that do not correspond to the formal parameters preceding the variable arity parameter will be evaluated and the results stored into an array that will be passed to the method invocation (§15.12.4.2).

Here are some examples of receiver parameters in instance methods and inner classes' constructors:

```
class Test {
    Test(/* ?? ?? */) {}
      // No receiver parameter is permitted in the constructor of
      // a top level class, as there is no conceivable type or name.

    void m(Test this) {}
      // OK: receiver parameter in an instance method

    static void n(Test this) {}
      // Illegal: receiver parameter in a static method

    class A {
        A(Test Test.this) {}
          // OK: the receiver parameter represents the instance
          // of Test which immediately encloses the instance
          // of A being constructed.

        void m(A this) {}
          // OK: the receiver parameter represents the instance
          // of A for which A.m() is invoked.

        class B {
            B(Test.A A.this) {}
              // OK: the receiver parameter represents the instance
              // of A which immediately encloses the instance of B
              // being constructed.

            void m(Test.A.B this) {}
              // OK: the receiver parameter represents the instance
              // of B for which B.m() is invoked.
        }
    }
}
```

`B`'s constructor and instance method show that the type of the receiver parameter may be denoted with a qualified *TypeName* like any other type; but that the name of the receiver parameter in an inner class's constructor must use the simple name of the enclosing class.

8.4.2 Method Signature

Two methods or constructors, *`M`* and *`N`*, have the *same signature* if they have the same name, the same type parameters (if any) (§8.4.4), and, after adapting the formal parameter types of *`N`* to the type parameters of *`M`*, the same formal parameter types.

The signature of a method m_1 is a *subsignature* of the signature of a method m_2 if either:

- m_2 has the same signature as m_1, or
- the signature of m_1 is the same as the erasure (§4.6) of the signature of m_2.

Two method signatures m_1 and m_2 are *override-equivalent* iff either m_1 is a subsignature of m_2 or m_2 is a subsignature of m_1.

It is a compile-time error to declare two methods with override-equivalent signatures in a class.

Example 8.4.2-1. Override-Equivalent Signatures

```
class Point {
    int x, y;
    abstract void move(int dx, int dy);
    void move(int dx, int dy) { x += dx; y += dy; }
}
```

This program causes a compile-time error because it declares two `move` methods with the same (and hence, override-equivalent) signature. This is an error even though one of the declarations is `abstract`.

The notion of subsignature is designed to express a relationship between two methods whose signatures are not identical, but in which one may override the other. Specifically, it allows a method whose signature does not use generic types to override any generified version of that method. This is important so that library designers may freely generify methods independently of clients that define subclasses or subinterfaces of the library.

Consider the example:

```
class CollectionConverter {
    List toList(Collection c) {...}
}
class Overrider extends CollectionConverter {
    List toList(Collection c) {...}
}
```

Now, assume this code was written before the introduction of generics, and now the author of class `CollectionConverter` decides to generify the code, thus:

```
class CollectionConverter {
    <T> List<T> toList(Collection<T> c) {...}
}
```

Without special dispensation, `Overrider.toList` would no longer override `CollectionConverter.toList`. Instead, the code would be illegal. This would significantly inhibit the use of generics, since library writers would hesitate to migrate existing code.

8.4.3 Method Modifiers

MethodModifier:
(one of)
Annotation `public` `protected` `private`
`abstract` `static` `final` `synchronized` `native` `strictfp`

The rules concerning annotation modifiers for a method declaration are specified in §9.7.4 and §9.7.5.

It is a compile-time error if the same keyword appears more than once as a modifier for a method declaration, or if a method declaration has more than one of the access modifiers `public`, `protected`, and `private` (§6.6).

It is a compile-time error if a method declaration that contains the keyword `abstract` also contains any one of the keywords `private`, `static`, `final`, `native`, `strictfp`, or `synchronized`.

It is a compile-time error if a method declaration that contains the keyword `native` also contains `strictfp`.

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

8.4.3.1 `abstract` *Methods*

An `abstract` method declaration introduces the method as a member, providing its signature (§8.4.2), result (§8.4.5), and `throws` clause if any (§8.4.6), but does not provide an implementation (§8.4.7). A method that is not `abstract` may be referred to as a *concrete* method.

The declaration of an `abstract` method *`m`* must appear directly within an `abstract` class (call it *`A`*) unless it occurs within an enum declaration (§8.9); otherwise, a compile-time error occurs.

Every subclass of *`A`* that is not `abstract` (§8.1.1.1) must provide an implementation for *`m`*, or a compile-time error occurs.

An `abstract` class can override an `abstract` method by providing another `abstract` method declaration.

> This can provide a place to put a documentation comment, to refine the return type, or to declare that the set of checked exceptions that can be thrown by that method, when it is implemented by its subclasses, is to be more limited.

An instance method that is not `abstract` can be overridden by an `abstract` method.

Example 8.4.3.1-1. Abstract/Abstract Method Overriding

```
class BufferEmpty extends Exception {
    BufferEmpty() { super(); }
    BufferEmpty(String s) { super(s); }
}
class BufferError extends Exception {
    BufferError() { super(); }
    BufferError(String s) { super(s); }
}
interface Buffer {
    char get() throws BufferEmpty, BufferError;
}
abstract class InfiniteBuffer implements Buffer {
    public abstract char get() throws BufferError;
}
```

The overriding declaration of method `get` in class `InfiniteBuffer` states that method `get` in any subclass of `InfiniteBuffer` never throws a `BufferEmpty` exception, putatively because it generates the data in the buffer, and thus can never run out of data.

Example 8.4.3.1-2. Abstract/Non-Abstract Overriding

We can declare an `abstract` class `Point` that requires its subclasses to implement `toString` if they are to be complete, instantiable classes:

```
abstract class Point {
    int x, y;
    public abstract String toString();
}
```

This `abstract` declaration of `toString` overrides the non-`abstract` `toString` method of the class `Object`. (`Object` is the implicit direct superclass of class `Point`.) Adding the code:

```
class ColoredPoint extends Point {
    int color;
    public String toString() {
        return super.toString() + ": color " + color;  // error
    }
}
```

results in a compile-time error because the invocation `super.toString()` refers to method `toString` in class `Point`, which is `abstract` and therefore cannot be invoked. Method `toString` of class `Object` can be made available to class `ColoredPoint` only if class `Point` explicitly makes it available through some other method, as in:

```
abstract class Point {
```

```
    int x, y;
    public abstract String toString();
    protected String objString() { return super.toString(); }
}
class ColoredPoint extends Point {
    int color;
    public String toString() {
        return objString() + ": color " + color;  // correct
    }
}
```

8.4.3.2 `static` *Methods*

A method that is declared `static` is called a *class method*.

A class method is always invoked without reference to a particular object. The declaration of a class method introduces a static context (§8.1.3), which limits the use of constructs that refer to the current object. Notably, the keywords `this` and `super` are prohibited in a static context (§15.8.3, §15.11.2), as are unqualified references to instance variables, instance methods, and type parameters of lexically enclosing declarations (§6.5.5.1, §6.5.6.1, §15.12.3).

A method that is not declared `static` is called an *instance method*, and sometimes called a non-`static` method.

An instance method is always invoked with respect to an object, which becomes the current object to which the keywords `this` and `super` refer during execution of the method body.

References to an instance method from a static context or a nested class or interface are restricted, as specified in §15.12.3.

8.4.3.3 `final` *Methods*

A method can be declared `final` to prevent subclasses from overriding or hiding it.

It is a compile-time error to attempt to override or hide a `final` method.

A `private` method and all methods declared immediately within a `final` class (§8.1.1.2) behave as if they are `final`, since it is impossible to override them.

> At run time, a machine-code generator or optimizer can "inline" the body of a `final` method, replacing an invocation of the method with the code in its body. The inlining process must preserve the semantics of the method invocation. In particular, if the target of an instance method invocation is `null`, then a `NullPointerException` must be thrown even if the method is inlined. A Java compiler must ensure that the exception will be thrown at the correct point, so that the actual arguments to the method will be seen to have been evaluated in the correct order prior to the method invocation.

Consider the example:

```
final class Point {
    int x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
}
class Test {
    public static void main(String[] args) {
        Point[] p = new Point[100];
        for (int i = 0; i < p.length; i++) {
            p[i] = new Point();
            p[i].move(i, p.length-1-i);
        }
    }
}
```

Inlining the method `move` of class `Point` in method `main` would transform the `for` loop to the form:

```
for (int i = 0; i < p.length; i++) {
        p[i] = new Point();
        Point pi = p[i];
        int j = p.length-1-i;
        pi.x += i;
        pi.y += j;
    }
```

The loop might then be subject to further optimizations.

Such inlining cannot be done at compile time unless it can be guaranteed that `Test` and `Point` will always be recompiled together, so that whenever `Point` - and specifically its `move` method - changes, the code for `Test.main` will also be updated.

8.4.3.4 `native` *Methods*

A method that is `native` is implemented in platform-dependent code, typically written in another programming language such as C. The body of a `native` method is given as a semicolon only, indicating that the implementation is omitted, instead of a block (§8.4.7).

For example, the class `RandomAccessFile` of the package `java.io` might declare the following `native` methods:

```
package java.io;
public class RandomAccessFile
    implements DataOutput, DataInput {
    . . .
    public native void open(String name, boolean writeable)
        throws IOException;
    public native int readBytes(byte[] b, int off, int len)
```

```
            throws IOException;
        public native void writeBytes(byte[] b, int off, int len)
            throws IOException;
        public native long getFilePointer() throws IOException;
        public native void seek(long pos) throws IOException;
        public native long length() throws IOException;
        public native void close() throws IOException;
    }
```

8.4.3.5 `strictfp` *Methods*

The `strictfp` modifier on a method declaration is obsolete and should not be used in new code. Its presence or absence has has no effect at run time.

8.4.3.6 `synchronized` *Methods*

A `synchronized` method acquires a monitor (§17.1) before it executes.

For a class (`static`) method, the monitor associated with the `Class` object for the method's class is used.

For an instance method, the monitor associated with `this` (the object for which the method was invoked) is used.

Example 8.4.3.6-1. `synchronized` **Monitors**

These are the same monitors that can be used by the `synchronized` statement (§14.19).

Thus, the code:

```
class Test {
    int count;
    synchronized void bump() {
        count++;
    }
    static int classCount;
    static synchronized void classBump() {
        classCount++;
    }
}
```

has exactly the same effect as:

```
class BumpTest {
    int count;
    void bump() {
        synchronized (this) { count++; }
    }
    static int classCount;
```

```
        static void classBump() {
            try {
                synchronized (Class.forName("BumpTest")) {
                    classCount++;
                }
            } catch (ClassNotFoundException e) {}
        }
    }
```

Example 8.4.3.6-2. `synchronized` **Methods**

```
    public class Box {
        private Object boxContents;
        public synchronized Object get() {
            Object contents = boxContents;
            boxContents = null;
            return contents;
        }
        public synchronized boolean put(Object contents) {
            if (boxContents != null) return false;
            boxContents = contents;
            return true;
        }
    }
```

This program defines a class which is designed for concurrent use. Each instance of the class `Box` has an instance variable `boxContents` that can hold a reference to any object. You can put an object in a `Box` by invoking `put`, which returns `false` if the box is already full. You can get something out of a `Box` by invoking `get`, which returns a null reference if the box is empty.

If `put` and `get` were not `synchronized`, and two threads were executing methods for the same instance of `Box` at the same time, then the code could misbehave. It might, for example, lose track of an object because two invocations to `put` occurred at the same time.

8.4.4 Generic Methods

A method is *generic* if it declares one or more type variables (§4.4).

These type variables are known as the *type parameters* of the method. The form of the type parameter section of a generic method is identical to the type parameter section of a generic class (§8.1.2).

A generic method declaration defines a set of methods, one for each possible invocation of the type parameter section by type arguments. Type arguments may not need to be provided explicitly when a generic method is invoked, as they can often be inferred (§18 (*Type Inference*)).

The scope and shadowing of a method's type parameter is specified in §6.3 and §6.4.1.

References to a method's type parameter from a nested class or interface are restricted, as specified in §6.5.5.1.

Two methods or constructors `M` and `N` have the *same type parameters* if both of the following are true:

- `M` and `N` have same number of type parameters (possibly zero).
- Where $A_1, ..., A_n$ are the type parameters of `M` and $B_1, ..., B_n$ are the type parameters of `N`, let $\theta=[B_1:=A_1, ..., B_n:=A_n]$. Then, for all i ($1 \le i \le n$), the bound of A_i is the same type as θ applied to the bound of B_i.

Where two methods or constructors `M` and `N` have the same type parameters, a type mentioned in `N` can be *adapted to the type parameters* of `M` by applying θ, as defined above, to the type.

8.4.5 Method Result

The *result* of a method declaration either declares the type of value that the method returns (the *return type*), or uses the keyword `void` to indicate that the method does not return a value.

Result:
UnannType
`void`

If the result is not `void`, then the return type of a method is denoted by *UnannType* if no bracket pairs appear after the formal parameter list, and is specified by §10.2 otherwise.

Return types may vary among methods that override each other if the return types are reference types. The notion of return-type-substitutability supports *covariant returns*, that is, the specialization of the return type to a subtype.

A method declaration d_1 with return type R_1 is *return-type-substitutable* for another method d_2 with return type R_2 iff any of the following is true:

- If R_1 is `void` then R_2 is `void`.
- If R_1 is a primitive type then R_2 is identical to R_1.

- If R_1 is a reference type then one of the following is true:
  – R_1, adapted to the type parameters of d_2 (§8.4.4), is a subtype of R_2.
  – R_1 can be converted to a subtype of R_2 by unchecked conversion (§5.1.9).
  – d_1 does not have the same signature as d_2 (§8.4.2), and $R_1 = |R_2|$.

An unchecked conversion is allowed in the definition, despite being unsound, as a special allowance to allow smooth migration from non-generic to generic code. If an unchecked conversion is used to determine that R_1 is return-type-substitutable for R_2, then R_1 is necessarily not a subtype of R_2 and the rules for overriding (§8.4.8.3, §9.4.1) will require a compile-time unchecked warning.

8.4.6 Method Throws

A `throws` clause is used to denote any checked exception classes (§11.1.1) that the statements in a method or constructor body can throw (§11.2.2).

Throws:
  `throws` *ExceptionTypeList*

ExceptionTypeList:
  ExceptionType {`,` *ExceptionType*}

ExceptionType:
  ClassType
  TypeVariable

It is a compile-time error if an *ExceptionType* mentioned in a `throws` clause is not a subtype (§4.10) of `Throwable`.

Type variables are allowed in a `throws` clause even though they are not allowed in a `catch` clause (§14.20).

It is permitted but not required to mention unchecked exception classes (§11.1.1) in a `throws` clause.

The relationship between a `throws` clause and the exception checking for a method or constructor body is specified in §11.2.3.

Essentially, for each checked exception that can result from execution of the body of a method or constructor, a compile-time error occurs unless its exception type or a supertype of its exception type is mentioned in a `throws` clause in the declaration of the method or constructor.

The requirement to declare checked exceptions allows a Java compiler to ensure that code for handling such error conditions has been included. Methods or constructors that fail to handle exceptional conditions thrown as checked exceptions in their bodies will normally cause compile-time errors if they lack proper exception types in their `throws` clauses. The Java programming language thus encourages a programming style where rare and otherwise truly exceptional conditions are documented in this way.

The relationship between the `throws` clause of a method and the `throws` clauses of overridden or hidden methods is specified in §8.4.8.3.

Example 8.4.6-1. Type Variables as Thrown Exception Types

```
import java.io.FileNotFoundException;
interface PrivilegedExceptionAction<E extends Exception> {
    void run() throws E;
}
class AccessController {
    public static <E extends Exception>
    Object doPrivileged(PrivilegedExceptionAction<E> action) throws E {
        action.run();
        return "success";
    }
}
class Test {
    public static void main(String[] args) {
        try {
            AccessController.doPrivileged(
              new PrivilegedExceptionAction<FileNotFoundException>() {
                  public void run() throws FileNotFoundException {
                      // ... delete a file ...
                  }
              });
        } catch (FileNotFoundException f) { /* Do something */ }
    }
}
```

8.4.7 Method Body

A *method body* is either a block of code that implements the method or simply a semicolon, indicating the lack of an implementation.

MethodBody:
  Block
  `;`

The body of a method must be a semicolon if the method is `abstract` or `native` (§8.4.3.1, §8.4.3.4). More precisely:

- It is a compile-time error if a method declaration is either `abstract` or `native` and has a block for its body.
- It is a compile-time error if a method declaration is neither `abstract` nor `native` and has a semicolon for its body.

If an implementation is to be provided for a method declared `void`, but the implementation requires no executable code, the method body should be written as a block that contains no statements: "`{ }`".

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).

In other words, a method with a return type must return only by using a `return` statement that provides a value return; the method is not allowed to "drop off the end of its body". See §14.17 for the precise rules about `return` statements in a method body.

It is possible for a method to have a return type and yet contain no `return` statements. Here is one example:

```
class DizzyDean {
    int pitch() { throw new RuntimeException("90 mph?!"); }
}
```

8.4.8 Inheritance, Overriding, and Hiding

A class *C* *inherits* from its direct superclass type *D* all concrete methods *m* (both `static` and instance) for which all of the following are true:

- *m* is a member of *D*.
- *m* is `public`, `protected`, or declared with package access in the same package as *C*.
- No method declared in *C* has a signature that is a subsignature (§8.4.2) of the signature of *m* as a member of *D*.

A class *C* *inherits* from its direct superclass type and direct superinterface types all `abstract` and default (§9.4) methods *m* for which all of the following are true:

- *m* is a member of the direct superclass type or a direct superinterface type of *C*, known in either case as *D*.
- *m* is `public`, `protected`, or declared with package access in the same package as *C*.

- No method declared in *C* has a signature that is a subsignature (§8.4.2) of the signature of *m* as a member of *D*.
- No concrete method inherited by *C* from its direct superclass type has a signature that is a subsignature of the signature of *m* as a member of *D*.
- There exists no method *m'* that is a member of the direct superclass type or a direct superinterface type of *C*, *D'* (*m* distinct from *m'*, *D* distinct from *D'*), such that *m'* overrides from the class or interface of *D'* the declaration of the method *m* (§8.4.8.1, §9.4.1.1).

A class does not inherit `private` or `static` methods from its superinterface types.

Note that methods are overridden or hidden on a signature-by-signature basis. If, for example, a class declares two `public` methods with the same name (§8.4.9), and a subclass overrides one of them, the subclass still inherits the other method.

Example 8.4.8-1. Inheritance

```
interface I1 {
    int foo();
}

interface I2 {
    int foo();
}

abstract class Test implements I1, I2 {}
```

Here, the `abstract` class `Test` inherits the `abstract` method `foo` from interface `I1` and also the `abstract` method `foo` from interface `I2`. The key question in determining the inheritance of `foo` from `I1` is: does the method `foo` in `I2` override "from `I2`" (§9.4.1.1) the method `foo` in `I1`? No, because `I1` and `I2` are not subinterfaces of each other. Thus, from the viewpoint of class `Test`, the inheritance of `foo` from `I1` is unfettered; similarly for the inheritance of `foo` from `I2`. Per §8.4.8.4, class `Test` can inherit both `foo` methods; obviously it must be declared `abstract`, or else override both `abstract foo` methods with a concrete method.

Note that it is possible for an inherited concrete method to prevent the inheritance of an `abstract` or default method. (The concrete method will override the `abstract` or default method "from *C*", per §8.4.8.1 and §9.4.1.1.) Also, it is possible for one supertype method to prevent the inheritance of another supertype method if the former "already" overrides the latter - this is the same as the rule for interfaces (§9.4.1), and prevents conflicts in which multiple default methods are inherited and one implementation is clearly meant to supersede the other.

8.4.8.1 *Overriding (by Instance Methods)*

An instance method m_C declared in or inherited by class `C`, *overrides from* `C` another method m_A declared in class `A`, iff all of the following are true:

- `C` is a subclass of `A`.
- `C` does not inherit m_A.
- The signature of m_C is a subsignature (§8.4.2) of the signature of m_A as a member of the supertype of `C` that names `A`.
- One of the following is true:
  - m_A is `public`.
  - m_A is `protected`.
  - m_A is declared with package access in the same package as `C`, and either `C` declares m_C or m_A is a member of the direct superclass type of `C`.
  - m_A is declared with package access and m_C overrides m_A from some superclass of `C`.
  - m_A is declared with package access and m_C overrides a method `m'` from `C` (`m'` distinct from m_C and m_A), such that `m'` overrides m_A from some superclass of `C`.

If m_C is non-`abstract` and overrides from `C` an `abstract` method m_A, then m_C is said to *implement* m_A *from* `C`.

It is a compile-time error if the overridden method, m_A, is a `static` method.

> In this respect, overriding of methods differs from hiding of fields (§8.3), for it is permissible for an instance variable to hide a `static` variable.

An instance method m_C declared in or inherited by class `C`, *overrides from* `C` another method m_I declared in interface `I`, iff all of the following are true:

- `I` is a superinterface of `C`.
- m_I is not `static`.
- `C` does not inherit m_I.
- The signature of m_C is a subsignature (§8.4.2) of the signature of m_I as a member of the supertype of `C` that names `I`.
- m_I is `public`.

> The signature of an overriding method may differ from the overridden one if a formal parameter in one of the methods has a raw type, while the corresponding parameter in the

other has a parameterized type. This accommodates migration of pre-existing code to take advantage of generics.

The notion of overriding includes methods that override another from some subclass of their declaring class. This can happen in two ways:

- A concrete method in a generic superclass can, under certain parameterizations, have the same signature as an `abstract` method in that class. In this case, the concrete method is inherited and the `abstract` method is not (as described above). The inherited method should then be considered to override its abstract peer *from* `C`. (This scenario is complicated by package access: if `C` is in a different package, then m_A would not have been inherited anyway, and should not be considered overridden.)
- A method inherited from a class can override a superinterface method. (Happily, package access is not a concern here.)

An overridden method can be accessed by using a method invocation expression (§15.12) that contains the keyword `super`. A qualified name or a cast to a superclass type is not effective in attempting to access an overridden method.

In this respect, overriding of methods differs from hiding of fields.

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

Example 8.4.8.1-1. Overriding

```
class Point {
    int x = 0, y = 0;
    void move(int dx, int dy) { x += dx; y += dy; }
}
class SlowPoint extends Point {
    int xLimit, yLimit;
    void move(int dx, int dy) {
        super.move(limit(dx, xLimit), limit(dy, yLimit));
    }
    static int limit(int d, int limit) {
        return d > limit ? limit : d < -limit ? -limit : d;
    }
}
```

Here, the class `SlowPoint` overrides the declarations of method `move` of class `Point` with its own `move` method, which limits the distance that the point can move on each invocation of the method. When the `move` method is invoked for an instance of class `SlowPoint`, the overriding definition in class `SlowPoint` will always be called, even if the reference to the `SlowPoint` object is taken from a variable whose type is `Point`.

Example 8.4.8.1-2. Overriding

Overriding makes it easy for subclasses to extend the behavior of an existing class, as shown in this example:

```
import java.io.OutputStream;
import java.io.IOException;

class BufferOutput {
    private OutputStream o;
    BufferOutput(OutputStream o) { this.o = o; }
    protected byte[] buf = new byte[512];
    protected int pos = 0;
    public void putchar(char c) throws IOException {
        if (pos == buf.length) flush();
        buf[pos++] = (byte)c;
    }
    public void putstr(String s) throws IOException {
        for (int i = 0; i < s.length(); i++)
            putchar(s.charAt(i));
    }
    public void flush() throws IOException {
        o.write(buf, 0, pos);
        pos = 0;
    }
}
class LineBufferOutput extends BufferOutput {
    LineBufferOutput(OutputStream o) { super(o); }
    public void putchar(char c) throws IOException {
        super.putchar(c);
        if (c == '\n') flush();
    }
}
class Test {
    public static void main(String[] args) throws IOException {
        LineBufferOutput lbo = new LineBufferOutput(System.out);
        lbo.putstr("lbo\nlbo");
        System.out.print("print\n");
        lbo.putstr("\n");
    }
}
```

This program produces the output:

```
lbo
print
lbo
```

The class `BufferOutput` implements a very simple buffered version of an `OutputStream`, flushing the output when the buffer is full or `flush` is invoked. The subclass `LineBufferOutput` declares only a constructor and a single method `putchar`, which overrides the method `putchar` of `BufferOutput`. It inherits the methods `putstr` and `flush` from class `BufferOutput`.

In the `putchar` method of a `LineBufferOutput` object, if the character argument is a newline, then it invokes the `flush` method. The critical point about overriding in this example is that the method `putstr`, which is declared in class `BufferOutput`, invokes the `putchar` method defined by the current object `this`, which is not necessarily the `putchar` method declared in class `BufferOutput`.

Thus, when `putstr` is invoked in `main` using the `LineBufferOutput` object `lbo`, the invocation of `putchar` in the body of the `putstr` method is an invocation of the `putchar` of the object `lbo`, the overriding declaration of `putchar` that checks for a newline. This allows a subclass of `BufferOutput` to change the behavior of the `putstr` method without redefining it.

Documentation for a class such as `BufferOutput`, which is designed to be extended, should clearly indicate what is the contract between the class and its subclasses, and should clearly indicate that subclasses may override the `putchar` method in this way. The implementor of the `BufferOutput` class would not, therefore, want to change the implementation of `putstr` in a future implementation of `BufferOutput` not to use the method `putchar`, because this would break the pre-existing contract with subclasses. See the discussion of binary compatibility in §13 (*Binary Compatibility*), especially §13.2.

8.4.8.2 *Hiding (by Class Methods)*

If a class *C* declares or inherits a `static` method *m*, then *m* is said to *hide* any method *m'* declared in a class or interface *A* for which all of the following are true:

- *A* is a superclass or superinterface of *C*.
- If *A* is an interface, *m'* is an instance method.
- *m'* is accessible to *C* (§6.6).
- The signature of *m* is a subsignature (§8.4.2) of the signature of *m'* as a member of the supertype of *C* that names *A*.

It is a compile-time error if a `static` method hides an instance method.

In this respect, hiding of methods differs from hiding of fields (§8.3), for it is permissible for a `static` variable to hide an instance variable. Hiding is also distinct from shadowing (§6.4.1) and obscuring (§6.4.2).

A hidden method can be accessed by using a qualified name or by using a method invocation expression (§15.12) that contains the keyword `super` or a cast to a superclass type.

In this respect, hiding of methods is similar to hiding of fields.

Example 8.4.8.2-1. Invocation of Hidden Class Methods

A class (`static`) method that is hidden can be invoked by using a reference whose type is the type of the class that actually contains the declaration of the method. In this respect, hiding of `static` methods is different from overriding of instance methods. The example:

```
class Super {
    static String greeting() { return "Goodnight"; }
    String name() { return "Richard"; }
}
class Sub extends Super {
    static String greeting() { return "Hello"; }
    String name() { return "Dick"; }
}
class Test {
    public static void main(String[] args) {
        Super s = new Sub();
        System.out.println(s.greeting() + ", " + s.name());
    }
}
```

produces the output:

```
Goodnight, Dick
```

because the invocation of `greeting` uses the type of `s`, namely `Super`, to figure out, at compile time, which class method to invoke, whereas the invocation of `name` uses the class of `s`, namely `Sub`, to figure out, at run time, which instance method to invoke.

8.4.8.3 *Requirements in Overriding and Hiding*

If a method declaration d_1 with return type R_1 overrides or hides the declaration of another method d_2 with return type R_2, then d_1 must be return-type-substitutable (§8.4.5) for d_2, or a compile-time error occurs.

This rule allows for covariant return types - refining the return type of a method when overriding it.

If R_1 is not a subtype of R_2, then a compile-time unchecked warning occurs, unless suppressed by `@SuppressWarnings` (§9.6.4.5).

A method that overrides or hides another method, including methods that implement `abstract` methods defined in interfaces, may not be declared to throw more checked exceptions than the overridden or hidden method.

In this respect, overriding of methods differs from hiding of fields (§8.3), for it is permissible for a field to hide a field of another type.

More precisely, suppose that `B` is a class or interface, and `A` is a superclass or superinterface of `B`, and a method declaration m_2 in `B` overrides or hides a method declaration m_1 in `A`. Then:

- If m_2 has a `throws` clause that mentions any checked exception types, then m_1 must have a `throws` clause, or a compile-time error occurs.
- For every checked exception type listed in the `throws` clause of m_2, that same exception class or one of its supertypes must occur in the erasure (§4.6) of the `throws` clause of m_1; otherwise, a compile-time error occurs.
- If the unerased `throws` clause of m_1 does not contain a supertype of each exception type in the `throws` clause of m_2 (adapted, if necessary, to the type parameters of m_1), then a compile-time unchecked warning occurs, unless suppressed by `@SuppressWarnings` (§9.6.4.5).

It is a compile-time error if a class or interface `C` has a member method m_1 and there exists a method m_2 declared in `C` or a superclass or superinterface of `C`, `A`, such that all of the following are true:

- m_1 and m_2 have the same name.
- m_2 is accessible (§6.6) from `C`.
- The signature of m_1 is not a subsignature (§8.4.2) of the signature of m_2 as a member of the supertype of `C` that names `A`.
- The declared signature of m_1 or some method m_1 overrides (directly or indirectly) has the same erasure as the declared signature of m_2 or some method m_2 overrides (directly or indirectly).

These restrictions are necessary because generics are implemented via erasure. The rule above implies that methods declared in the same class with the same name must have different erasures. It also implies that a class or interface cannot implement or extend two distinct parameterizations of the same generic interface.

The access modifier of an overriding or hiding method must provide at least as much access as the overridden or hidden method, as follows:

- If the overridden or hidden method is `public`, then the overriding or hiding method must be `public`; otherwise, a compile-time error occurs.
- If the overridden or hidden method is `protected`, then the overriding or hiding method must be `protected` or `public`; otherwise, a compile-time error occurs.
- If the overridden or hidden method has package access, then the overriding or hiding method must *not* be `private`; otherwise, a compile-time error occurs.

Note that a `private` method cannot be overridden or hidden in the technical sense of those terms. This means that a subclass can declare a method with the same signature as a `private` method in one of its superclasses, and there is no requirement that the return type or `throws` clause of such a method bear any relationship to those of the `private` method in the superclass.

Example 8.4.8.3-1. Covariant Return Types

The following declarations are legal in the Java programming language from Java SE 5.0 onwards:

```
class C implements Cloneable {
    C copy() throws CloneNotSupportedException {
        return (C)clone();
    }
}
class D extends C implements Cloneable {
    D copy() throws CloneNotSupportedException {
        return (D)clone();
    }
}
```

The relaxed rule for overriding also allows one to relax the conditions on abstract classes implementing interfaces.

Example 8.4.8.3-2. Unchecked Warning from Return Type

Consider:

```
class StringSorter {
    // turns a collection of strings into a sorted list
    List toList(Collection c) {...}
}
```

and assume that someone subclasses `StringSorter`:

```
class Overrider extends StringSorter {
    List toList(Collection c) {...}
}
```

Now, at some point the author of `StringSorter` decides to generify the code:

```
class StringSorter {
    // turns a collection of strings into a sorted list
    List<String> toList(Collection<String> c) {...}
}
```

An unchecked warning would be given when compiling `Overrider` against the new definition of `StringSorter` because the return type of `Overrider.toList` is `List`, which is not a subtype of the return type of the overridden method, `List<String>`.

Example 8.4.8.3-3. Incorrect Overriding because of `throws`

This program uses the usual and conventional form for declaring a new exception type, in its declaration of the class `BadPointException`:

```
class BadPointException extends Exception {
    BadPointException() { super(); }
    BadPointException(String s) { super(s); }
}
class Point {
    int x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
}
class CheckedPoint extends Point {
    void move(int dx, int dy) throws BadPointException {
        if ((x + dx) < 0 || (y + dy) < 0)
            throw new BadPointException();
        x += dx; y += dy;
    }
}
```

The program results in a compile-time error, because the override of method `move` in class `CheckedPoint` declares that it will throw a checked exception that the `move` in class `Point` has not declared. If this were not considered an error, an invoker of the method `move` on a reference of type `Point` could find the contract between it and `Point` broken if this exception were thrown.

Removing the `throws` clause does not help:

```
class CheckedPoint extends Point {
    void move(int dx, int dy) {
        if ((x + dx) < 0 || (y + dy) < 0)
            throw new BadPointException();
        x += dx; y += dy;
    }
}
```

A different compile-time error now occurs, because the body of the method `move` cannot throw a checked exception, namely `BadPointException`, that does not appear in the `throws` clause for `move`.

Example 8.4.8.3-4. Erasure Affects Overriding

A class cannot have two member methods with the same name and type erasure:

```
class C<T> {
    T id (T x) {...}
}
class D extends C<String> {
    Object id(Object x) {...}
}
```

This is illegal since `D.id(Object)` is a member of `D`, `C<String>.id(String)` is declared in a supertype of `D`, and:

- The two methods have the same name, `id`
- `C<String>.id(String)` is accessible to `D`
- The signature of `D.id(Object)` is not a subsignature of that of `C<String>.id(String)`
- The two methods have the same erasure

Two different methods of a class may not override methods with the same erasure:

```
class C<T> {
    T id(T x) {...}
}
interface I<T> {
    T id(T x);
}
class D extends C<String> implements I<Integer> {
   public String  id(String x)  {...}
   public Integer id(Integer x) {...}
}
```

This is also illegal, since `D.id(String)` is a member of `D`, `D.id(Integer)` is declared in `D`, and:

- The two methods have the same name, `id`
- `D.id(Integer)` is accessible to `D`
- The two methods have different signatures (and neither is a subsignature of the other)
- `D.id(String)` overrides `C<String>.id(String)` and `D.id(Integer)` overrides `I.id(Integer)` yet the two overridden methods have the same erasure

8.4.8.4 *Inheriting Methods with Override-Equivalent Signatures*

It is possible for a class to inherit multiple methods with override-equivalent signatures (§8.4.2).

It is a compile-time error if a class *C* inherits a concrete method whose signature is override-equivalent with another method inherited by *C*.

It is a compile-time error if a class *C* inherits a default method whose signature is override-equivalent with another method inherited by *C*, unless there exists an `abstract` method declared in a superclass of *C* and inherited by *C* that is override-equivalent with the two methods.

This exception to the strict default-`abstract` and default-default conflict rules is made when an `abstract` method is declared in a superclass: the assertion of abstract-ness coming

> from the superclass hierarchy essentially trumps the default method, making the default method act as if it were `abstract`. However, the `abstract` method from a class does not override the default method(s), because interfaces are still allowed to refine the *signature* of the `abstract` method coming from the class hierarchy.
>
> Note that the exception does not apply if all override-equivalent `abstract` methods inherited by `C` were declared in interfaces.

Otherwise, the set of override-equivalent methods consists of at least one `abstract` method and zero or more default methods; then the class is necessarily an `abstract` class and is considered to inherit all the methods.

One of the inherited methods must be return-type-substitutable for every other inherited method; otherwise, a compile-time error occurs. (The `throws` clauses do not cause errors in this case.)

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

8.4.9 Overloading

If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but signatures that are not override-equivalent, 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, unless their signatures are override-equivalent.

When a method is invoked (§15.12), the number of actual arguments (and any explicit type arguments) and the compile-time types of the arguments are used, at compile time, to determine the signature of the method that will be invoked (§15.12.2). If the method that is to be invoked is an instance method, the actual method to be invoked will be determined at run time, using dynamic method lookup (§15.12.4).

Example 8.4.9-1. Overloading

```
class Point {
    float x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
    void move(float dx, float dy) { x += dx; y += dy; }
    public String toString() { return "("+x+","+y+")"; }
}
```

Here, the class `Point` has two members that are methods with the same name, `move`. The overloaded `move` method of class `Point` chosen for any particular method invocation is determined at compile time by the overloading resolution procedure given in §15.12.

In total, the members of the class `Point` are the `float` instance variables `x` and `y` declared in `Point`, the two declared `move` methods, the declared `toString` method, and the members that `Point` inherits from its implicit direct superclass `Object` (§4.3.2), such as the method `hashCode`. Note that `Point` does not inherit the `toString` method of class `Object` because that method is overridden by the declaration of the `toString` method in class `Point`.

Example 8.4.9-2. Overloading, Overriding, and Hiding

```
class Point {
    int x = 0, y = 0;
    void move(int dx, int dy) { x += dx; y += dy; }
    int color;
}
class RealPoint extends Point {
    float x = 0.0f, y = 0.0f;
    void move(int dx, int dy) { move((float)dx, (float)dy); }
    void move(float dx, float dy) { x += dx; y += dy; }
}
```

Here, the class `RealPoint` hides the declarations of the `int` instance variables `x` and `y` of class `Point` with its own `float` instance variables `x` and `y`, and overrides the method `move` of class `Point` with its own `move` method. It also overloads the name `move` with another method with a different signature (§8.4.2).

In this example, the members of the class `RealPoint` include the instance variable `color` inherited from the class `Point`, the `float` instance variables `x` and `y` declared in `RealPoint`, and the two `move` methods declared in `RealPoint`.

Which of these overloaded `move` methods of class `RealPoint` will be chosen for any particular method invocation will be determined at compile time by the overloading resolution procedure described in §15.12.

This following program is an extended variation of the preceding program:

```
class Point {
    int x = 0, y = 0, color;
    void move(int dx, int dy) { x += dx; y += dy; }
    int getX() { return x; }
    int getY() { return y; }
}
class RealPoint extends Point {
    float x = 0.0f, y = 0.0f;
    void move(int dx, int dy) { move((float)dx, (float)dy); }
    void move(float dx, float dy) { x += dx; y += dy; }
    float getX() { return x; }
    float getY() { return y; }
```

```
}
```

Here, the class `Point` provides methods `getX` and `getY` that return the values of its fields `x` and `y`; the class `RealPoint` then overrides these methods by declaring methods with the same signature. The result is two errors at compile time, one for each method, because the return types do not match; the methods in class `Point` return values of type `int`, but the wanna-be overriding methods in class `RealPoint` return values of type `float`.

This program corrects the errors of the preceding program:

```
class Point {
    int x = 0, y = 0;
    void move(int dx, int dy) { x += dx; y += dy; }
    int getX() { return x; }
    int getY() { return y; }
    int color;
}
class RealPoint extends Point {
    float x = 0.0f, y = 0.0f;
    void move(int dx, int dy) { move((float)dx, (float)dy); }
    void move(float dx, float dy) { x += dx; y += dy; }
    int getX() { return (int)Math.floor(x); }
    int getY() { return (int)Math.floor(y); }
}
```

Here, the overriding methods `getX` and `getY` in class `RealPoint` have the same return types as the methods of class `Point` that they override, so this code can be successfully compiled.

Consider, then, this test program:

```
class Test {
    public static void main(String[] args) {
        RealPoint rp = new RealPoint();
        Point p = rp;
        rp.move(1.71828f, 4.14159f);
        p.move(1, -1);
        show(p.x, p.y);
        show(rp.x, rp.y);
        show(p.getX(), p.getY());
        show(rp.getX(), rp.getY());
    }
    static void show(int x, int y) {
        System.out.println("(" + x + ", " + y + ")");
    }
    static void show(float x, float y) {
        System.out.println("(" + x + ", " + y + ")");
    }
}
```

The output from this program is:

```
(0, 0)
(2.7182798, 3.14159)
(2, 3)
(2, 3)
```

The first line of output illustrates the fact that an instance of `RealPoint` actually contains the two integer fields declared in class `Point`; it is just that their names are hidden from code that occurs within the declaration of class `RealPoint` (and those of any subclasses it might have). When a reference to an instance of class `RealPoint` in a variable of type `Point` is used to access the field `x`, the integer field `x` declared in class `Point` is accessed. The fact that its value is zero indicates that the method invocation `p.move(1, -1)` did not invoke the method `move` of class `Point`; instead, it invoked the overriding method `move` of class `RealPoint`.

The second line of output shows that the field access `rp.x` refers to the field `x` declared in class `RealPoint`. This field is of type `float`, and this second line of output accordingly displays floating-point values. Incidentally, this also illustrates the fact that the method name `show` is overloaded; the types of the arguments in the method invocation dictate which of the two definitions will be invoked.

The last two lines of output show that the method invocations `p.getX()` and `rp.getX()` each invoke the `getX` method declared in class `RealPoint`. Indeed, there is no way to invoke the `getX` method of class `Point` for an instance of class `RealPoint` from outside the body of `RealPoint`, no matter what the type of the variable we may use to hold the reference to the object. Thus, we see that fields and methods behave differently: hiding is different from overriding.

8.5 Member Class and Interface Declarations

A *member class* is a class whose declaration is directly enclosed in the body of another class or interface declaration (§8.1.7, §9.1.5).

A *member interface* is an interface whose declaration is directly enclosed in the body of another class or interface declaration.

A member class may be a normal class (§8.1), an enum class (§8.9), or a record class (§8.10).

A member interface may be a normal interface (§9.1) or an annotation interface (§9.6).

The accessibility of a member class or interface declaration in the body of a class declaration is specified by its access modifier, or by §6.6 if lacking an access modifier.

The rules for modifiers of a member class declaration in the body of a class declaration are specified in §8.1.1.

The rules for modifiers of a member interface declaration in the body of a class declaration are specified in §9.1.1.

The scope and shadowing of a member class or interface is specified in §6.3 and §6.4.1.

If a class declares a member class or interface with a certain name, then the declaration of the member class or interface is said to *hide* any and all accessible declarations of member classes and interfaces with the same name in superclasses and superinterfaces of the class.

In this respect, hiding of member class and interfaces is similar to hiding of fields (§8.3).

A class inherits from its direct superclass and direct superinterfaces all the non-`private` member classes and interfaces of the superclass and superinterfaces that are both accessible to code in the class and not hidden by a declaration in the class.

It is possible for a class to inherit more than one member class or interface with the same name, either from its superclass and superinterfaces or from its superinterfaces alone. Such a situation does not in itself cause a compile-time error. However, any attempt within the body of the class to refer to any such member class or interface by its simple name will result in a compile-time error, because the reference is ambiguous.

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

8.6 Instance Initializers

An *instance initializer* declared in a class is executed when an instance of the class is created (§12.5, §15.9, §8.8.7.1).

InstanceInitializer:
  Block

It is a compile-time error if an instance initializer cannot complete normally (§14.22).

It is a compile-time error if a `return` statement (§14.17) appears anywhere within an instance initializer.

An instance initializer is permitted to refer to the current object using the keyword `this` (§15.8.3) or the keyword `super` (§15.11.2, §15.12), and to use any type variables in scope.

Restrictions on how an instance initializer may refer to instance variables, even when the instance variables are in scope, are specified in §8.3.3.

Exception checking for an instance initializer is specified in §11.2.3.

8.7 Static Initializers

A *static initializer* declared in a class is executed when the class is initialized (§12.4.2). Together with any field initializers for class variables (§8.3.2), static initializers may be used to initialize the class variables of the class.

StaticInitializer:
  `static` *Block*

It is a compile-time error if a static initializer cannot complete normally (§14.22).

It is a compile-time error if a `return` statement (§14.17) appears anywhere within a static initializer.

A static initializer introduces a static context (§8.1.3, which limits the use of constructs that refer to the current object. Notably, the keywords `this` and `super` are prohibited in a static context (§15.8.3, §15.11.2), as are unqualified references to instance variables, instance methods, and type parameters of lexically enclosing declarations (§6.5.5.1, §6.5.6.1, §15.12.3).

Restrictions on how a static initializer may refer to class variables, even when the class variables are in scope, are specified in §8.3.3.

Exception checking for a static initializer is specified in §11.2.3.

8.8 Constructor Declarations

A *constructor* is used in the creation of an object that is an instance of a class (§12.5, §15.9).

ConstructorDeclaration:
  {ConstructorModifier} ConstructorDeclarator [Throws] ConstructorBody

ConstructorDeclarator:
  [TypeParameters] SimpleTypeName
    `(` *[ReceiverParameter* `,` *] [FormalParameterList]* `)`

SimpleTypeName:
  TypeIdentifier

The rules in this section apply to constructors in all class declarations, including enum declarations and record declarations. However, special rules apply to enum declarations with regard to constructor modifiers, constructor bodies, and default constructors; these rules are stated in §8.9.2. Special rules also apply to record declarations with regard to constructors, as stated in §8.10.4.

The *SimpleTypeName* in the *ConstructorDeclarator* must be the simple name of the class that contains the constructor declaration, or a compile-time error occurs.

In all other respects, a constructor declaration looks just like a method declaration that has no result (§8.4.5).

Constructor declarations are not members. They are never inherited and therefore are not subject to hiding or overriding.

Constructors are invoked by class instance creation expressions (§15.9), by the conversions and concatenations caused by the string concatenation operator `+` (§15.18.1), and by explicit constructor invocations from other constructors (§8.8.7). Access to constructors is governed by access modifiers (§6.6), so it is possible to prevent class instantiation by declaring an inaccessible constructor (§8.8.10).

Constructors are never invoked by method invocation expressions (§15.12).

Example 8.8-1. Constructor Declarations

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
}
```

8.8.1 Formal Parameters

The formal parameters of a constructor are identical in syntax and semantics to those of a method (§8.4.1).

If the last formal parameter of a constructor is a variable arity parameter, the constructor is a *variable arity constructor*. Otherwise, it is a *fixed arity constructor*.

The constructor of a non-`private` inner member class implicitly declares, as the first formal parameter, a variable representing the immediately enclosing instance of the class (§15.9.2, §15.9.3).

The rationale for why only this kind of class has an implicitly declared constructor parameter is subtle. The following explanation may be helpful:

1. In a class instance creation expression for a non-`private` inner member class, §15.9.2 specifies the immediately enclosing instance of the member class. The member class may have been emitted by a compiler which is different than the compiler of the class instance creation expression. Therefore, there must be a standard way for the compiler of the creation expression to pass a reference (representing the immediately enclosing instance) to the member class's constructor. Consequently, the Java programming language deems in this section that a non-`private` inner member class's constructor implicitly declares an initial parameter for the immediately enclosing instance. §15.9.3 specifies that the instance is passed to the constructor.
2. In a class instance creation expression for an inner local class or an anonymous class (not in a static context), §15.9.2 specifies the immediately enclosing instance of the local/anonymous class. The local/anonymous class is necessarily emitted by the same compiler as the class instance creation expression. That compiler can represent the immediately enclosing instance how ever it wishes. There is no need for the Java programming language to implicitly declare a parameter in the local/anonymous class's constructor.
3. In a class instance creation expression for an anonymous class, and where the anonymous class's superclass is an inner class (not in a static context), §15.9.2 specifies the anonymous class's immediately enclosing instance with respect to the superclass. This instance must be transmitted from the anonymous class to its superclass, where it will serve as the immediately enclosing instance. Since the superclass may have been emitted by a compiler which is different than the compiler of the class instance creation expression, it is necessary to transmit the instance in a standard way, by passing it as the first argument to the superclass's constructor. Note that the anonymous class itself is necessarily emitted by the same compiler as the class instance creation expression, so it would be possible for the compiler to transmit the immediately enclosing instance with respect to the superclass to the anonymous class how ever it wishes, before the anonymous class passes the instance to the superclass's constructor. However, for consistency, the Java programming language deems in §15.9.5.1 that, in some circumstances, an anonymous class's constructor implicitly declares an initial parameter for the immediately enclosing instance with respect to the superclass.

The fact that a non-`private` inner member class may be accessed by a different compiler than compiled it, whereas an inner local class or an anonymous class is always accessed by the same compiler that compiled it, explains why the binary name of a non-`private` inner member class is defined to be predictable but the binary name of an inner local class or an anonymous class is not (§13.1).

8.8.2 Constructor Signature

It is a compile-time error to declare two constructors with override-equivalent signatures (§8.4.2) in a class.

It is a compile-time error to declare two constructors whose signatures have the same erasure (§4.6) in a class.

8.8.3 Constructor Modifiers

ConstructorModifier:
(one of)
Annotation `public` `protected` `private`

The rules concerning annotation modifiers for a constructor declaration are specified in §9.7.4 and §9.7.5.

It is a compile-time error if the same keyword appears more than once as a modifier in a constructor declaration, or if a constructor declaration has more than one of the access modifiers `public`, `protected`, and `private` (§6.6).

In a normal class declaration, a constructor declaration with no access modifiers has package access.

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

Unlike methods, a constructor cannot be `abstract`, `static`, `final`, `native`, `strictfp`, or `synchronized`:

- A constructor is not inherited, so there is no need to declare it `final`.
- An `abstract` constructor could never be implemented.
- A constructor is always invoked with respect to an object, so it makes no sense for a constructor to be `static`.
- There is no practical need for a constructor to be `synchronized`, because it would lock the object under construction, which is normally not made available to other threads until all constructors for the object have completed their work.
- The lack of `native` constructors is an arbitrary language design choice that makes it easy for an implementation of the Java Virtual Machine to verify that superclass constructors are always properly invoked during object creation.
- The inability to declare a constructor as `strictfp` (in contrast to a method (§8.4.3)) is an intentional language design choice that stemmed from the (now obsolete) ability to declare a class as `strictfp`.

8.8.4 Generic Constructors

A constructor is *generic* if it declares one or more type variables (§4.4).

These type variables are known as the *type parameters* of the constructor. The form of the type parameter section of a generic constructor is identical to the type parameter section of a generic class (§8.1.2).

It is possible for a constructor to be generic independently of whether the class the constructor is declared in is itself generic.

A generic constructor declaration defines a set of constructors, one for each possible invocation of the type parameter section by type arguments. Type arguments may not need to be provided explicitly when a generic constructor is invoked, as they can often by inferred (§18 (*Type Inference*)).

The scope and shadowing of a constructor's type parameter is specified in §6.3 and §6.4.1.

References to a constructor's type parameter from an explicit constructor invocation statement or a nested class or interface are restricted, as specified in §6.5.5.1.

8.8.5 Constructor Throws

The `throws` clause for a constructor is identical in structure and behavior to the `throws` clause for a method (§8.4.6).

8.8.6 The Type of a Constructor

The type of a constructor consists of its signature and the exception types given by its `throws` clause.

8.8.7 Constructor Body

The first statement of a constructor body may be an explicit invocation of another constructor of the same class or of the direct superclass (§8.8.7.1).

ConstructorBody:
`{` *[ExplicitConstructorInvocation] [BlockStatements]* `}`

It is a compile-time error for a constructor to directly or indirectly invoke itself through a series of one or more explicit constructor invocations involving `this`.

If a constructor body does not begin with an explicit constructor invocation and the constructor being declared is not part of the primordial class `Object`, then the constructor body implicitly begins with a superclass constructor invocation "`super();`", an invocation of the constructor of its direct superclass that takes no arguments.

Except for the possibility of explicit constructor invocations, and the prohibition on explicitly returning a value (§14.17), the body of a constructor is like the body of a method (§8.4.7).

A `return` statement (§14.17) may be used in the body of a constructor if it does not include an expression.

Example 8.8.7-1. Constructor Bodies

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
}
class ColoredPoint extends Point {
    static final int WHITE = 0, BLACK = 1;
    int color;
    ColoredPoint(int x, int y) {
        this(x, y, WHITE);
    }
    ColoredPoint(int x, int y, int color) {
        super(x, y);
        this.color = color;
    }
}
```

Here, the first constructor of `ColoredPoint` invokes the second, providing an additional argument; the second constructor of `ColoredPoint` invokes the constructor of its superclass `Point`, passing along the coordinates.

8.8.7.1 *Explicit Constructor Invocations*

ExplicitConstructorInvocation:
  [TypeArguments] `this` `(` *[ArgumentList]* `)` `;`
  [TypeArguments] `super` `(` *[ArgumentList]* `)` `;`
  ExpressionName `.` *[TypeArguments]* `super` `(` *[ArgumentList]* `)` `;`
  Primary `.` *[TypeArguments]* `super` `(` *[ArgumentList]* `)` `;`

The following productions from §4.5.1 and §15.12 are shown here for convenience:

TypeArguments:
  < *TypeArgumentList* >

ArgumentList:
  Expression {`,` *Expression*}

Explicit constructor invocation statements are divided into two kinds:

- *Alternate constructor invocations* begin with the keyword `this` (possibly prefaced with explicit type arguments). They are used to invoke an alternate constructor of the same class.
- *Superclass constructor invocations* begin with either the keyword `super` (possibly prefaced with explicit type arguments) or a *Primary* expression or an *ExpressionName*. They are used to invoke a constructor of the direct superclass. They are further divided:
  - *Unqualified superclass constructor invocations* begin with the keyword `super` (possibly prefaced with explicit type arguments).
  - *Qualified superclass constructor invocations* begin with a *Primary* expression or an *ExpressionName*. They allow a subclass constructor to explicitly specify the newly created object's immediately enclosing instance with respect to the direct superclass (§8.1.3). This may be necessary when the superclass is an inner class.

An explicit constructor invocation statement introduces a static context (§8.1.3), which limits the use of constructs that refer to the current object. Notably, the keywords `this` and `super` are prohibited in a static context (§15.8.3, §15.11.2), as are unqualified references to instance variables, instance methods, and type parameters of lexically enclosing declarations (§6.5.5.1, §6.5.6.1, §15.12.3).

If *TypeArguments* is present to the left of `this` or `super`, then it is a compile-time error if any of the type arguments are wildcards (§4.5.1).

Let *`C`* be the class being instantiated, and let *`S`* be the direct superclass of *`C`*.

If a superclass constructor invocation statement is unqualified, then:

- If *`S`* is an inner member class, but *`S`* is not a member of a class enclosing *`C`*, then a compile-time error occurs.

  Otherwise, let *`O`* be the innermost enclosing class of *`C`* of which *`S`* is a member. *`C`* must be an inner class of *`O`* (§8.1.3), or a compile-time error occurs.
- If *`S`* is an inner local class, and *`S`* does not occur in a static context, let *`O`* be the immediately enclosing class or interface declaration of *`S`*. *`C`* must be an inner class of *`O`*, or a compile-time error occurs.

If a superclass constructor invocation statement is qualified, then:

- If *`S`* is not an inner class, or if the declaration of *`S`* occurs in a static context, then a compile-time error occurs.
- Otherwise, let *`p`* be the *Primary* expression or the *ExpressionName* immediately preceding "`.super`", and let *`O`* be the immediately enclosing class of *`S`*. It is a

compile-time error if the type of `p` is not `O` or a subclass of `O`, or if the type of `p` is not accessible (§6.6).

The exception types that an explicit constructor invocation statement can throw are specified in §11.2.2.

Evaluation of an alternate constructor invocation statement proceeds by first evaluating the arguments to the constructor, left-to-right, as in an ordinary method invocation; and then invoking the constructor.

Evaluation of a superclass constructor invocation statement proceeds as follows:

1. Let `i` be the instance being created. The immediately enclosing instance of `i` with respect to `S` (if any) must be determined:

   - If `S` is not an inner class, or if the declaration of `S` occurs in a static context, then no immediately enclosing instance of `i` with respect to `S` exists.

   - Otherwise, if the superclass constructor invocation is unqualified, then `S` is necessarily an inner local class or an inner member class.

     If `S` is an inner local class, let `O` be the immediately enclosing class or interface declaration of `S`.

     If `S` is an inner member class, let `O` be the innermost enclosing class of `C` of which `S` is a member.

     Let *n* be an integer ($n \geq 1$) such that `O` is the *n*'th lexically enclosing class or interface declaration of `C`.

     The immediately enclosing instance of `i` with respect to `S` is the *n*'th lexically enclosing instance of `this`.

       While it may be the case that `S` is a member of `C` due to inheritance, the zeroth lexically enclosing instance of `this` (that is, `this` itself) is never used as the immediately enclosing instance of *i* with respect to `S`.

   - Otherwise, if the superclass constructor invocation is qualified, then the *Primary* expression or the *ExpressionName* immediately preceding "`.super`", *p*, is evaluated.

     If `p` evaluates to `null`, a `NullPointerException` is raised, and the superclass constructor invocation completes abruptly.

     Otherwise, the result of this evaluation is the immediately enclosing instance of `i` with respect to `S`.

2. After determining the immediately enclosing instance of `i` with respect to `S` (if any), evaluation of the superclass constructor invocation statement proceeds

by evaluating the arguments to the constructor, left-to-right, as in an ordinary method invocation; and then invoking the constructor.

3. Finally, if the superclass constructor invocation statement completes normally, then all instance variable initializers of `C` and all instance initializers of `C` are executed. If an instance initializer or instance variable initializer `I` textually precedes another instance initializer or instance variable initializer `J`, then `I` is executed before `J`.

   Execution of instance variable initializers and instance initializers is performed regardless of whether the superclass constructor invocation actually appears as an explicit constructor invocation statement or is provided implicitly. (An alternate constructor invocation does not perform this additional implicit execution.)

Example 8.8.7.1-1. Restrictions on Explicit Constructor Invocation Statements

If the first constructor of `ColoredPoint` in the example from §8.8.7 were changed as follows:

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
}
class ColoredPoint extends Point {
    static final int WHITE = 0, BLACK = 1;
    int color;
    ColoredPoint(int x, int y) {
        this(x, y, color);  // Changed to color from WHITE
    }
    ColoredPoint(int x, int y, int color) {
        super(x, y);
        this.color = color;
    }
}
```

then a compile-time error would occur, because the instance variable `color` cannot be used by a explicit constructor invocation statement.

Example 8.8.7.1-2. Qualified Superclass Constructor Invocation

In the code below, `ChildOfInner` has no lexically enclosing class or interface declaration, so an instance of `ChildOfInner` has no enclosing instance. However, the superclass of `ChildOfInner` (`Inner`) has a lexically enclosing class declaration (`Outer`), and an instance of `Inner` must have an enclosing instance of `Outer`. The enclosing instance of `Outer` is set when an instance of `Inner` is created. Therefore, when we create an instance of `ChildOfInner`, which is implicitly an instance of `Inner`, we must provide the enclosing instance of `Outer` via a qualified superclass invocation statement in `ChildOfInner`'s constructor. The instance of `Outer` is called the immediately enclosing instance of `ChildOfInner` with respect to `Inner`.

```
class Outer {
    class Inner {}
}
class ChildOfInner extends Outer.Inner {
    ChildOfInner() { (new Outer()).super(); }
}
```

Perhaps surprisingly, the same instance of `Outer` may serve as the immediately enclosing instance of `ChildOfInner` with respect to `Inner` *for multiple instances of* `ChildOfInner`. These instances of `ChildOfInner` are implicitly linked to the same instance of `Outer`. The program below achieves this by passing an instance of `Outer` to the constructor of `ChildOfInner`, which uses the instance in a qualified superclass constructor invocation statement. The rules for an explicit constructor invocation statement do not prohibit using formal parameters of the constructor that contains the statement.

```
class Outer {
    int secret = 5;
    class Inner {
        int  getSecret()      { return secret; }
        void setSecret(int s) { secret = s; }
    }
}
class ChildOfInner extends Outer.Inner {
    ChildOfInner(Outer x) { x.super(); }
}

public class Test {
    public static void main(String[] args) {
        Outer x = new Outer();
        ChildOfInner a = new ChildOfInner(x);
        ChildOfInner b = new ChildOfInner(x);
        System.out.println(b.getSecret());
        a.setSecret(6);
        System.out.println(b.getSecret());
    }
}
```

This program produces the output:

```
5
6
```

The effect is that manipulation of instance variables in the common instance of `Outer` is visible through references to different instances of `ChildOfInner`, even though such references are not aliases in the conventional sense.

8.8.8 Constructor Overloading

Overloading of constructors is identical in behavior to overloading of methods (§8.4.9). The overloading is resolved at compile time by each class instance creation expression (§15.9).

8.8.9 Default Constructor

If a class contains no constructor declarations, then a default constructor is implicitly declared. The form of the default constructor for a top level class, member class, or local class is as follows:

- The default constructor has the same access modifier as the class, unless the class lacks an access modifier, in which case the default constructor has package access (§6.6).
- The default constructor has no formal parameters, except in a non-`private` inner member class, where the default constructor implicitly declares one formal parameter representing the immediately enclosing instance of the class (§8.8.1, §15.9.2, §15.9.3).
- The default constructor has no `throws` clause.
- If the class being declared is the primordial class `Object`, then the default constructor has an empty body. Otherwise, the default constructor simply invokes the superclass constructor with no arguments.

The form of the default constructor for an anonymous class is specified in §15.9.5.1.

It is a compile-time error if a default constructor is implicitly declared but the superclass does not have an accessible constructor that takes no arguments and has no `throws` clause.

Example 8.8.9-1. Default Constructors

The declaration:

```
public class Point {
    int x, y;
}
```

is equivalent to the declaration:

```
public class Point {
    int x, y;
    public Point() { super(); }
}
```

where the default constructor is `public` because the class `Point` is `public`.

Example 8.8.9-2. Accessibility of Constructors v. Classes

The rule that the default constructor of a class has the same accessibility as the class itself is simple and intuitive. Note, however, that this does not imply that the constructor is accessible whenever the class is accessible. Consider:

```
package p1;
public class Outer {
    protected class Inner {}
}
package p2;
class SonOfOuter extends p1.Outer {
    void foo() {
        new Inner();  // compile-time access error
    }
}
```

The default constructor for `Inner` is `protected`. However, the constructor is `protected` relative to `Inner`, while `Inner` is `protected` relative to `Outer`. So, `Inner` is accessible in `SonOfOuter`, since it is a subclass of `Outer`. `Inner`'s constructor is not accessible in `SonOfOuter`, because the class `SonOfOuter` is not a subclass of `Inner`! Hence, even though `Inner` is accessible, its default constructor is not.

8.8.10 Preventing Instantiation of a Class

A class can be designed to prevent code outside the class declaration from creating instances of the class by declaring at least one constructor, to prevent the creation of a default constructor, and by declaring all constructors to be `private` (§6.6.1).

A `public` class can likewise prevent the creation of instances outside its package by declaring at least one constructor, to prevent creation of a default constructor with `public` access, and by declaring no constructor that is `public` or `protected` (§6.6.2).

Example 8.8.10-1. Preventing Instantiation via Constructor Accessibility

```
class ClassOnly {
    private ClassOnly() { }
    static String just = "only the lonely";
}
```

Here, the class `ClassOnly` cannot be instantiated, while in the following code:

```
package just;
public class PackageOnly {
    PackageOnly() { }
    String[] justDesserts = { "cheesecake", "ice cream" };
```

```
}
```

the `public` class `PackageOnly` can be instantiated only within the package `just`, in which it is declared. This restriction would also apply if the constructor of `PackageOnly` was `protected`, although in that case, it would be possible for code in other packages to instantiate subclasses of `PackageOnly`.

8.9 Enum Classes

An *enum declaration* specifies a new *enum class*, a restricted kind of class that defines a small set of named class instances.

EnumDeclaration:
  {ClassModifier} `enum` *TypeIdentifier [ClassImplements] EnumBody*

An enum declaration may specify a top level enum class (§7.6), a member enum class (§8.5, §9.5), or a local enum class (§14.3).

The *TypeIdentifier* in an enum declaration specifies the name of the enum class.

It is a compile-time error if an enum declaration has the modifier `abstract`, `final`, `sealed`, or `non-sealed`.

An enum class is either implicitly `final` or implicitly `sealed`, as follows:

- An enum class is implicitly `final` if its declaration contains no enum constants that have a class body (§8.9.1).
- An enum class *`E`* is implicitly `sealed` if its declaration contains at least one enum constant that has a class body. The permitted direct subclasses (§8.1.6) of *`E`* are the anonymous classes implicitly declared by the enum constants that have a class body.

A nested enum class is implicitly `static`. That is, every member enum class and local enum class is `static`. It is permitted for the declaration of a member enum class to redundantly specify the `static` modifier, but it is not permitted for the declaration of a local enum class (§14.3).

It is a compile-time error if the same keyword appears more than once as a modifier for an enum declaration, or if an enum declaration has more than one of the access modifiers `public`, `protected`, and `private` (§6.6).

The direct superclass type of an enum class *`E`* is `Enum<`*`E`*`>` (§8.1.4).

An enum declaration does not have an `extends` clause, so it is not possible to explicitly declare a direct superclass type, even `Enum<E>`.

An enum class has no instances other than those defined by its enum constants. It is a compile-time error to attempt to explicitly instantiate an enum class (§15.9.1).

In addition to the compile-time error, three further mechanisms ensure that no instances of an enum class exist beyond those defined by its enum constants:

- The `final clone` method in `Enum` ensures that enum constants can never be cloned.
- Reflective instantiation of enum classes is prohibited.
- Special treatment by the serialization mechanism ensures that duplicate instances are never created as a result of deserialization.

8.9.1 Enum Constants

The body of an enum declaration may contain *enum constants*. An enum constant defines an instance of the enum class.

EnumBody:
  `{` *[EnumConstantList]* *[*`,`*]* *[EnumBodyDeclarations]* `}`

EnumConstantList:
  EnumConstant {`,` *EnumConstant}*

EnumConstant:
  {EnumConstantModifier} Identifier [`(` *[ArgumentList]* `)`*] [ClassBody]*

EnumConstantModifier:
  Annotation

The following production from §15.12 is shown here for convenience:

ArgumentList:
  Expression {`,` *Expression}*

The rules concerning annotation modifiers for an enum constant declaration are specified in §9.7.4 and §9.7.5.

The *Identifier* in a *EnumConstant* may be used in a name to refer to the enum constant.

The scope and shadowing of an enum constant is specified in §6.3 and §6.4.1.

An enum constant may be followed by arguments, which are passed to the constructor of the enum when the constant is created during class initialization as described later in this section. The constructor to be invoked is chosen using the normal rules of overload resolution (§15.12.2). If the arguments are omitted, an empty argument list is assumed.

The optional class body of an enum constant implicitly declares an anonymous class (§15.9.5) that (i) is a direct subclass of the immediately enclosing enum class (§8.1.4), and (ii) is `final` (§8.1.1.2). The class body is governed by the usual rules of anonymous classes; in particular it cannot contain any constructors. Instance methods declared in these class bodies may be invoked outside the enclosing enum class only if they override accessible methods in the enclosing enum class (§8.4.8).

It is a compile-time error for the class body of an enum constant to declare an `abstract` method.

Because there is only one instance of each enum constant, it is permitted to use the == operator in place of the `equals` method when comparing two object references if it is known that at least one of them refers to an enum constant.

> The `equals` method in `Enum` is a `final` method that merely invokes `super.equals` on its argument and returns the result, thus performing an identity comparison.

8.9.2 Enum Body Declarations

In addition to enum constants, the body of an enum declaration may contain constructor and member declarations as well as instance and static initializers.

EnumBodyDeclarations:
  `;` *{ClassBodyDeclaration}*

The following productions from §8.1.7 are shown here for convenience:

ClassBodyDeclaration:
  ClassMemberDeclaration
  InstanceInitializer
  StaticInitializer
  ConstructorDeclaration

ClassMemberDeclaration:
  FieldDeclaration
  MethodDeclaration
  ClassDeclaration
  InterfaceDeclaration
  `;`

Any constructor or member declarations in the body of an enum declaration apply to the enum class exactly as if they had been present in the body of a normal class declaration, unless explicitly stated otherwise.

It is a compile-time error if a constructor declaration in an enum declaration is `public` or `protected` (§6.6).

It is a compile-time error if a constructor declaration in an enum declaration contains a superclass constructor invocation statement (§8.8.7.1).

It is a compile-time error to refer to a `static` field of an enum class from a constructor, instance initializer, or instance variable initializer in the enum declaration of the class, unless the field is a constant variable (§4.12.4).

In an enum declaration, a constructor declaration with no access modifiers is `private`.

In an enum declaration with no constructor declarations, a default constructor is implicitly declared. The default constructor is `private`, has no formal parameters, and has no `throws` clause.

In practice, a compiler is likely to mirror the `Enum` class by declaring `String` and `int` parameters in the default constructor of an enum class. However, these parameters are not specified as "implicitly declared" because different compilers do not need to agree on the form of the default constructor. Only the compiler of an enum declaration knows how to instantiate the enum constants; other compilers can simply rely on the implicitly declared `public static` fields of the enum class (§8.9.3) without regard for how those fields were initialized.

It is a compile-time error if an enum declaration *E* has an `abstract` method *m* as a member, unless *E* has at least one enum constant and all of *E*'s enum constants have class bodies that provide concrete implementations of *m*.

It is a compile-time error for an enum declaration to declare a finalizer (§12.6). An instance of an enum class may never be finalized.

Example 8.9.2-1. Enum Body Declarations

```
enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
    Coin(int value) { this.value = value; }

    private final int value;
    public int value() { return value; }
}
```

Each enum constant arranges for a different value in the field `value`, passed in via a constructor. The field represents the value, in cents, of an American coin. Note that there are no restrictions on the parameters that may be declared by an enum class's constructor.

Example 8.9.2-2. Restriction On Enum Constant Self-Reference

Without the rule on `static` field access, apparently reasonable code would fail at run time due to the initialization circularity inherent in enum classes. (A circularity exists in any class with a "self-typed" `static` field.) Here is an example of the sort of code that would fail:

```
import java.util.Map;
import java.util.HashMap;

enum Color {
    RED, GREEN, BLUE;
    Color() { colorMap.put(toString(), this); }

    static final Map<String,Color> colorMap =
        new HashMap<String,Color>();
}
```

Static initialization of this enum would throw a `NullPointerException` because the `static` variable `colorMap` is uninitialized when the constructors for the enum constants run. The restriction above ensures that such code cannot be compiled. However, the code can easily be refactored to work properly:

```
import java.util.Map;
import java.util.HashMap;

enum Color {
    RED, GREEN, BLUE;

    static final Map<String,Color> colorMap =
        new HashMap<String,Color>();
    static {
        for (Color c : Color.values())
            colorMap.put(c.toString(), c);
    }
}
```

The refactored version is clearly correct, as static initialization occurs top to bottom.

8.9.3 Enum Members

The members of an enum class *`E`* are all of the following:

- Members declared in the body of the declaration of *`E`*.
- Members inherited from `Enum<`*`E`*`>`.
- For each enum constant *`c`* declared in the body of the declaration of *`E`*, *`E`* has an implicitly declared `public static final` field of type *`E`* that has the same name as *`c`*. The field has a variable initializer which instantiates *`E`* and passes any

arguments of *c* to the constructor chosen for *E*. The field has the same annotations as *c* (if any).

These fields are implicitly declared in the same order as the corresponding enum constants, before any `static` fields explicitly declared in the body of the declaration of *E*.

An enum constant is said to be *created* when the corresponding implicitly declared field is initialized.

- An implicitly declared method `public static E[] values()`, which returns an array containing the enum constants of *E*, in the same order as they appear in the body of the declaration of *E*.
- An implicitly declared method `public static E valueOf(String name)`, which returns the enum constant of *E* with the specified name.

It follows that the declaration of enum class *E* cannot contain fields that conflict with the implicitly declared fields corresponding to *E*'s enum constants, nor contain methods that conflict with implicitly declared methods or override `final` methods of class `Enum<E>`.

Example 8.9.3-1. Iterating Over Enum Constants With An Enhanced `for` Loop

```
public class Test {
    enum Season { WINTER, SPRING, SUMMER, FALL }

    public static void main(String[] args) {
        for (Season s : Season.values())
            System.out.println(s);
    }
}
```

This program produces the output:

```
WINTER
SPRING
SUMMER
FALL
```

Example 8.9.3-2. Switching Over Enum Constants

A `switch` statement (§14.11) is useful for simulating the addition of a method to an enum class from outside the class. This example "adds" a `color` method to the `Coin` class from §8.9.2, and prints a table of coins, their values, and their colors.

```
class Test {
    enum CoinColor { COPPER, NICKEL, SILVER }

    static CoinColor color(Coin c) {
        switch (c) {
```

```
                case PENNY:
                    return CoinColor.COPPER;
                case NICKEL:
                    return CoinColor.NICKEL;
                case DIME: case QUARTER:
                    return CoinColor.SILVER;
                default:
                    throw new AssertionError("Unknown coin: " + c);
            }
        }

        public static void main(String[] args) {
            for (Coin c : Coin.values())
                System.out.println(c + "\t\t" +
                                   c.value() + "\t" + color(c));
        }
    }
```

This program produces the output:

```
PENNY           1       COPPER
NICKEL          5       NICKEL
DIME            10      SILVER
QUARTER         25      SILVER
```

Example 8.9.3-3. Enum Constants with Class Bodies

Rather than using a `switch` statement to "add" behavior to an enum class from the outside, it is possible to use class bodies to attach behaviors to enum constants directly.

```
enum Operation {
    PLUS {
        double eval(double x, double y) { return x + y; }
    },
    MINUS {
        double eval(double x, double y) { return x - y; }
    },
    TIMES {
        double eval(double x, double y) { return x * y; }
    },
    DIVIDED_BY {
        double eval(double x, double y) { return x / y; }
    };

    // Each constant supports an arithmetic operation
    abstract double eval(double x, double y);

    public static void main(String args[]) {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
        for (Operation op : Operation.values())
            System.out.println(x + " " + op + " " + y +
```

```
                                        " = " + op.eval(x, y));
        }
    }
```

The program produces the output:

```
java Operation 2.0 4.0
2.0 PLUS 4.0 = 6.0
2.0 MINUS 4.0 = -2.0
2.0 TIMES 4.0 = 8.0
2.0 DIVIDED_BY 4.0 = 0.5
```

This pattern is much safer than using a `switch` statement because the pattern precludes the possibility of forgetting to add a behavior for a new constant (since the enum declaration would cause a compile-time error).

Example 8.9.3-4. Multiple Enum Classes

In the following program, a playing card class is built atop two simple enums.

```
import java.util.List;
import java.util.ArrayList;
class Card implements Comparable<Card>,
                      java.io.Serializable {
    public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX, SEVEN,
                       EIGHT, NINE, TEN,JACK, QUEEN, KING, ACE }

    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

    private final Rank rank;
    private final Suit suit;
    public Rank rank() { return rank; }
    public Suit suit() { return suit; }

    private Card(Rank rank, Suit suit) {
        if (rank == null || suit == null)
            throw new NullPointerException(rank + ", " + suit);
        this.rank = rank;
        this.suit = suit;
    }

    public String toString() { return rank + " of " + suit; }

    // Primary sort on suit, secondary sort on rank
    public int compareTo(Card c) {
        int suitCompare = suit.compareTo(c.suit);
        return (suitCompare != 0 ?
                    suitCompare :
                    rank.compareTo(c.rank));
    }
```

```
    private static final List<Card> prototypeDeck =
        new ArrayList<Card>(52);

    static {
        for (Suit suit : Suit.values())
            for (Rank rank : Rank.values())
                prototypeDeck.add(new Card(rank, suit));
    }

    // Returns a new deck
    public static List<Card> newDeck() {
        return new ArrayList<Card>(prototypeDeck);
    }
}
```

The following program exercises the `Card` class. It takes two integer parameters on the command line, representing the number of hands to deal and the number of cards in each hand:

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
class Deal {
    public static void main(String args[]) {
        int numHands     = Integer.parseInt(args[0]);
        int cardsPerHand = Integer.parseInt(args[1]);
        List<Card> deck  = Card.newDeck();
        Collections.shuffle(deck);
        for (int i=0; i < numHands; i++)
            System.out.println(dealHand(deck, cardsPerHand));
    }

    /**
     * Returns a new ArrayList consisting of the last n
     * elements of deck, which are removed from deck.
     * The returned list is sorted using the elements'
     * natural ordering.
     */
    public static <E extends Comparable<E>>
    ArrayList<E> dealHand(List<E> deck, int n) {
        int deckSize = deck.size();
        List<E> handView = deck.subList(deckSize - n, deckSize);
        ArrayList<E> hand = new ArrayList<E>(handView);
        handView.clear();
        Collections.sort(hand);
        return hand;
    }
}
```

The program produces the output:

```
java Deal 4 3
[DEUCE of CLUBS, SEVEN of CLUBS, QUEEN of DIAMONDS]
[NINE of HEARTS, FIVE of SPADES, ACE of SPADES]
[THREE of HEARTS, SIX of HEARTS, TEN of SPADES]
[TEN of CLUBS, NINE of DIAMONDS, THREE of SPADES]
```

8.10 Record Classes

A *record declaration* specifies a new record class, a restricted kind of class that defines a simple aggregate of values.

RecordDeclaration:
{ClassModifier} `record` *TypeIdentifier [TypeParameters] RecordHeader*
[ClassImplements] RecordBody

A record declaration may specify a top level record class (§7.6), a member record class (§8.5, §9.5), or a local record class (§14.3).

The *TypeIdentifier* in a record declaration specifies the name of the record class.

It is a compile-time error if a record declaration has the modifier `abstract`, `sealed`, or `non-sealed`.

A record class is implicitly `final`. It is permitted for the declaration of a record class to redundantly specify the `final` modifier.

A nested record class is implicitly `static`. That is, every member record class and local record class is `static`. It is permitted for the declaration of a member record class to redundantly specify the `static` modifier, but it is not permitted for the declaration of a local record class (§14.3).

It is a compile-time error if the same keyword appears more than once as a modifier for a record declaration, or if a record declaration has more than one of the access modifiers `public`, `protected`, and `private` (§6.6).

The direct superclass type of a record class is `Record` (§8.1.4).

A record declaration does not have an `extends` clause, so it is not possible to explicitly declare a direct superclass type, even `Record`.

The serialization mechanism treats instances of a record class differently than ordinary serializable or externalizable objects. In particular, a record object is deserialized using the canonical constructor (§8.10.4).

8.10.1 Record Components

The *record components* of a record class, if any, are specified in the header of a record declaration. Each record component consists of a type (optionally preceded by one or more annotations) and an identifier that specifies the name of the record component. A record component corresponds to two members of the record class: a `private` field declared implicitly, and a `public` accessor method declared explicitly or implicitly (§8.10.3).

If a record class has no record components, then an empty pair of parentheses appears in the header of the record declaration.

RecordHeader:
  `(` *[RecordComponentList]* `)`

RecordComponentList:
  RecordComponent {`,` *RecordComponent*}

RecordComponent:
  {RecordComponentModifier} UnannType Identifier
  VariableArityRecordComponent

VariableArityRecordComponent:
  {RecordComponentModifier} UnannType {Annotation} `...` *Identifier*

RecordComponentModifier:
  Annotation

A record component may be a *variable arity record component*, indicated by an ellipsis following the type. At most one variable arity record component is permitted for a record class. It is a compile-time error if a variable arity record component appears anywhere in the list of record components except the last position.

The rules concerning annotation modifiers for a record component declaration are specified in §9.7.4 and §9.7.5.

Annotations on a record component declaration are available via reflection if their annotation interfaces are applicable in the record component context (§9.6.4.1). Independently, annotations on a record component declaration are propagated to the declarations of members and constructors of the record class if their annotation interfaces are applicable in other contexts (§8.10.3, §8.10.4).

It is a compile-time error for a record declaration to declare a record component with the name `clone`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, or `wait`.

> These are the names of the no-args `public` and `protected` methods in `Object`. Disallowing them as the names of record components avoids confusion in a number of ways. First, every record class provides implementations of `hashCode` and `toString` that return representations of a record object as a whole; they cannot serve as accessor methods (§8.10.3) for record components called `hashCode` or `toString`, and there would be no way to access such record components from outside the record class. Similarly, some record classes may provide implementations of `clone` and (regrettably) `finalize`, so a record component called `clone` or `finalize` could not be accessed via an accessor method. Finally, the `getClass`, `notify`, `notifyAll`, and `wait` methods in `Object` are `final`, so record components with the same names could not have accessor methods. (The accessor methods would have the same signatures as the `final` methods, and would thus attempt, unsuccessfully, to override them.)

It is a compile-time error for a record declaration to declare two record components with the same name.

The declared type of a record component depends on whether it is a variable arity record component:

- If the record component is not a variable arity record component, then the declared type is denoted by *UnannType*.
- If the record component is a variable arity record component, then the declared type is an array type specified by §10.2.

If the declared type of a variable arity record component has a non-reifiable element type (§4.7), then a compile-time unchecked warning occurs for the declaration of the variable arity record component, unless the canonical constructor (§8.10.4) is annotated with `@SafeVarargs` (§9.6.4.7) or the warning is suppressed by `@SuppressWarnings` (§9.6.4.5).

8.10.2 Record Body Declarations

The body of a record declaration may contain constructor and member declarations as well as static initializers.

RecordBody:
  `{` *{RecordBodyDeclaration}* `}`

RecordBodyDeclaration:
  ClassBodyDeclaration
  CompactConstructorDeclaration

The following productions from §8.1.7 are shown here for convenience:

ClassBodyDeclaration:
ClassMemberDeclaration
InstanceInitializer
StaticInitializer
ConstructorDeclaration

ClassMemberDeclaration:
FieldDeclaration
MethodDeclaration
ClassDeclaration
InterfaceDeclaration
`;`

The *CompactConstructorDeclaration* clause is described in §8.10.4.2.

It is a compile-time error for the body of a record declaration to contain a non-`static` field declaration (§8.3.1.1).

It is a compile-time error for the body of a record declaration to contain a method declaration that is `abstract` or `native` (§8.4.3.1, §8.4.3.4).

It is a compile-time error for the body of a record declaration to contain an instance initializer (§8.6).

8.10.3 Record Members

For each record component, a record class has a field with the same name as the record component and the same type as the declared type of the record component. This field, which is declared implicitly, is known as a *component field*.

A component field is `private`, `final`, and non-`static`.

A component field is annotated with the annotations, if any, that appear on the corresponding record component and whose annotation interfaces are applicable in the field declaration context, or in type contexts, or both (§9.7.4).

Furthermore, for each record component, a record class has a method with the same name as the record component and an empty formal parameter list. This method, which is declared explicitly or implicitly, is known as an *accessor method*.

If an accessor method for a record component is declared explicitly, then all of the following must be true, or a compile-time error occurs:

- The return type of the accessor method (§8.4.5) must be the same as the declared type of the record component.
- The accessor method must not be generic (§8.4.4).

- The accessor method must be a `public` instance method with no formal parameters and no `throws` clause.

If a record class has a record component for which an accessor method is not declared explicitly, then an accessor method for that record component is declared implicitly, with the following properties:

- Its name is the same as the name of the record component.
- Its return type is the same as the declared type of the record component.
- It is not generic.
- It is a `public` instance method with no formal parameters and no `throws` clause.
- It is annotated with the annotations, if any, that appear on the corresponding record component and whose annotation interfaces are applicable in the method declaration context, or in type contexts, or both (§9.7.4).
- Its body returns the value of the corresponding component field.

> The restrictions on record component names (§8.10.1) mean that no implicitly declared accessor method has a signature that is override-equivalent with a non-`private` method of class `Object`. An explicit method declaration that takes one of the restricted names, such as `public void wait() {...}`, is not an accessor method, since `wait` is never a record component name.
>
> Annotations that appear on a record component are not propagated to an explicitly declared accessor method for that record component. In some situations, the programmer may need to duplicate a record component's annotations on an explicitly declared accessor method, but this is not generally necessary.
>
> Annotations that are propagated to an implicitly declared accessor method must result in a legally annotated method. For example, in the following record declaration, the implicitly declared accessor method `x()` would be annotated with `@SafeVarargs`, but such an annotation is illegal on a fixed arity method (§9.6.4.7):

```
record BadRecord(@SafeVarargs int x) {}  // Error
```

Record classes may explicitly declare instance methods other than accessor methods, but may not explicitly declare instance variables (§8.10.2). Explicit declarations of class methods and class variables are permitted.

All members of record classes, including implicitly declared members, are subject to the usual rules for member declarations in a class (§8.3, §8.4, §8.5).

All of the rules concerning inheritance that apply to normal classes apply to record classes. In particular, record classes may inherit members from superinterfaces, although a superinterface method will never be inherited as an accessor method because the record class will always declare, explicitly or implicitly, an accessor method that overrides the superinterface method.

For example, a record class can inherit default methods from its direct superinterfaces, although the default method bodies have no knowledge of the component fields of the record class. The following program prints `Logged`:

```
public class Test {
    interface Logging {
        default void logAction() {
            System.out.println("Logged");
        }
    }

    record Point(int i, int j) implements Logging {}

    public static void main(String[] args) {
        Point p = new Point(10, 20);
        p.logAction();
    }
}
```

A record class provides implementations of all the `abstract` methods declared in class `Record`. For each of the following methods, if a record class *R* does not explicitly declare a method with the same modifiers, name, and signature (§8.4.2), then the method is implicitly declared as follows:

- A method `public final boolean equals(Object)` that returns `true` if and only if the argument is an instance of *R*, and the current instance is equal to the argument instance at every record component of *R*; otherwise `false` is returned.

  Equality of an instance *a* of a record class *R* with another instance *b* of the same record class at a record component *c* is determined as follows:

  – If the type of the record component *c* is a reference type, equality is determined as follows: if the value of the component field *c* of both *a* and *b* is the null reference then `true` is returned; if the value of the component field *c* of either *a* or *b*, but not both, is the null reference then `false` is returned; otherwise equality is determined by invoking the `equals` method on the value of the component field *c* of *a*, with an argument that is the value of the component field *c* of *b*.

  – If the type of the record component *c* is a primitive type *T*, equality is determined as if by invoking the `static` method `compare` of the wrapper class corresponding to *T* (§5.1.7), with the first argument given by the value of the component field *c* of *a*, and the second argument given by the value of the component field `c` of `b`; if the method would return `0` then `true` is returned, otherwise `false` is returned.

The use of `compare` in wrapper classes ensures that the implicitly declared `equals` method is reflexive and behaves consistently with the implicitly declared `hashCode` method for record classes that have floating-point components.

- A method `public final int hashCode()` that returns a hash code value derived from the hash code values at every record component of `R`.

  The hash code value of an instance `a` of a record class at a record component `c` is as follows:

  – If the type of the record component `c` is a reference type, then the hash code value is determined as if by invoking the `hashCode` method on the value of the component field `c` of `a`.

  – If the type of the record component `c` is a primitive type `T`, then the hash code value is determined as if by subjecting the value of the component field `c` of `a` to boxing conversion (§5.1.7) and then invoking the method `hashCode` of the wrapper class corresponding to `T` on the resulting object.

- A method `public final String toString()` that returns a string derived from the name of the record class and the names and string representations of every record component of `R`.

  The string representation of a record component `c` of an instance `a` of a record class is as follows:

  – If the type of the record component `c` is a reference type, then the string representation is determined as if by invoking the `toString` method on the value of the component field `c` of `a`.

  – If the type of the record component `c` is a primitive type `T`, then the string representation is determined as if by subjecting the value of the component field `c` of `a` to boxing conversion (§5.1.7) and then invoking the method `toString` method of the wrapper class corresponding to `T` on the resulting object.

Note that equality, hash code values, and string representations are determined by looking at the values of component fields directly, rather than by invoking accessor methods.

Consider a record class `R` that has components c_1, ..., c_n, and an implicitly declared accessor method for every component, and an implicitly declared `equals` method. If an instance `r1` of `R` is copied in the following way:

```
R r2 = new R(r1.c1(), r1.c2(), ..., r1.cn());
```

then, assuming `r1` is not the null reference, it is always the case that the expression `r1.equals(r2)` will evaluate to `true`. Explicitly declared accessor methods and `equals` methods should respect this invariant. It is not generally possible for a compiler to check whether explicitly declared methods respect the invariant. The following record declaration is bad style because its accessor methods clip the `x` and `y` components and therefore prevent `p3` from being `equals` to `p1`:

```
record SmallPoint(int x, int y) {
    public int x() { return this.x < 100 ? this.x : 100; }
    public int y() { return this.y < 100 ? this.y : 100; }

    public static void main(String[] args) {
        SmallPoint p1 = new SmallPoint(200,300);
        SmallPoint p2 = new SmallPoint(200,300);
        System.out.println(p1.equals(p2));  // prints true

        SmallPoint p3 = new SmallPoint(p1.x(), p1.y());
        System.out.println(p1.equals(p3));  // prints false
    }
}
```

8.10.4 Record Constructor Declarations

To ensure proper initialization of its record components, a record class does not implicitly declare a default constructor (§8.8.9). Instead, a record class has a *canonical constructor*, declared explicitly or implicitly, that initializes all the component fields of the record class.

There are two ways to explicitly declare a canonical constructor in a record declaration: by declaring a normal constructor with a suitable signature (§8.10.4.1) or by declaring a compact constructor (§8.10.4.2).

Given the signature of a normal constructor that qualifies as canonical, and the signature derived for a compact constructor, the rules of constructor signatures (§8.8.2) mean it is a compile-time error if a record declaration has both a normal constructor that qualifies as canonical *and* a compact constructor.

Either way, an explicitly declared canonical constructor must provide at least as much access as the record class, as follows:

- If the record class is `public`, then the canonical constructor must be `public`; otherwise, a compile-time error occurs.
- If the record class is `protected`, then the canonical constructor must be `protected` or `public`; otherwise, a compile-time error occurs.
- If the record class has package access, then the canonical constructor must *not* be `private`; otherwise, a compile-time error occurs.

- If the record class is `private`, then the canonical constructor may be declared with any accessibility.

An explicitly declared canonical constructor may be a fixed arity constructor or a variable arity constructor (§8.8.1).

If a canonical constructor is not explicitly declared in the declaration of a record class *R*, then a canonical constructor *r* is implicitly declared in *R* with the following properties:

- The signature of *r* has no type parameters, and has formal parameters given by the derived formal parameter list of *R*, defined below.
- *r* has the same access modifier as *R*, unless *R* lacks an access modifier, in which case *r* has package access.
- *r* has no `throws` clause.
- The body of *r* initializes each component field of the record class with the corresponding formal parameter of *r*, in the order that record components (corresponding to the component fields) appear in the record header.

The *derived formal parameter list* of a record class is formed by deriving a formal parameter from each record component in the record header, in order, as follows:

- If the record component is not a variable arity record component, then the derived formal parameter has the same name and declared type as the record component.

  If the record component is a variable arity record component, then the derived formal parameter is a variable arity parameter (§8.4.1) with the same name and declared type as the record component.
- The derived formal parameter is annotated with the annotations, if any, that appear on the record component and whose annotation interfaces are applicable in the formal parameter context, or in type contexts, or both (§9.7.4).

A record declaration may contain declarations of constructors that are not canonical constructors. The body of every non-canonical constructor in a record declaration must start with an alternate constructor invocation (§8.8.7.1), or a compile-time error occurs.

8.10.4.1 *Normal Canonical Constructors*

A (non-compact) constructor in the declaration of record class *R* is the *canonical constructor* of *R* if its signature is override-equivalent (§8.4.2) to the derived constructor signature of *R*.

The *derived constructor signature* of a record class `R` is a signature that consists of the name `R`, no type parameters, and the formal parameter types derived from the record header of `R` by taking the declared type of each record component in order.

As a canonical constructor has a signature that is override-equivalent to the derived constructor signature of the record class, there can be only one canonical constructor declared explicitly in the record class.

The declaration of a (non-compact) canonical constructor must satisfy all of the following conditions, or a compile-time error occurs:

- Each formal parameter in the formal parameter list must have the same name and declared type as the corresponding record component.

  A formal parameter must be a variable arity parameter if and only if the corresponding record component is a variable arity record component.

- The constructor must not be generic (§8.8.4).
- The constructor must not have a `throws` clause.
- The constructor body must not contain an explicit constructor invocation statement (§8.8.7.1).
- All the other rules for constructor declarations in a normal class declaration must be satisfied (§8.8).

A consequence of these rules is that the annotations on a record component can differ from the annotations on the corresponding formal parameter of an explicitly declared canonical constructor. For example, the following record declaration is valid:

```
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@interface Foo {}
@interface Bar {}

record Person(@Foo String name) {
    Person(@Bar String name) {
        this.name = name;
    }
}
```

8.10.4.2 *Compact Canonical Constructors*

A *compact constructor declaration* is a succinct form of constructor declaration, only available in a record declaration. It declares the canonical constructor of a record class without requiring the record components of the class to be manually repeated as formal parameters of the constructor.

CompactConstructorDeclaration:
{ConstructorModifier} SimpleTypeName ConstructorBody

The following productions from §8.8, §8.8.3, and §8.8.7 are shown here for convenience:

ConstructorModifier:
(one of)
Annotation `public` `protected` `private`

SimpleTypeName:
TypeIdentifier

ConstructorBody:
`{` *[ExplicitConstructorInvocation] [BlockStatements]* `}`

It is a compile-time error for a record declaration to have more than one compact constructor declaration.

The formal parameters of a compact constructor of a record class are implicitly declared. They are given by the derived formal parameter list of the record class (§8.10.4).

The compact constructor of a record class is a variable arity constructor (§8.8.1) if the record class has a variable arity record component.

The signature of a compact constructor declaration is equal to the derived constructor signature of the record class (§8.10.4.1).

The body of a compact constructor declaration must satisfy all of the following conditions, or a compile-time error occurs:

- The body must not contain a `return` statement (§14.17).
- The body must not contain an explicit constructor invocation statement (§8.8.7.1).
- The body must not contain an assignment to a component field of the record class.
- All the other rules for a constructor in a normal class declaration must be satisfied (§8.8), *except* for the requirement that the component fields of the record class must be definitely assigned and moreover not definitely unassigned at the end of the compact constructor (§8.3.1.2).

If a record declaration has a record component named `c`, then the simple name `c` in the body of a compact constructor denotes the implicit formal parameter named `c`, and not the component field named `c`.

After the last statement, if any, in the body of the compact constructor has completed normally (§14.1), all component fields of the record class are implicitly initialized to the values of the corresponding formal parameters. The component fields are initialized in the order that the corresponding record components are declared in the record header.

The intent of a compact constructor declaration is that only code to validate or normalize parameters needs to be given in the constructor body; the remaining initialization code is supplied by the compiler. For example, the following record class has a compact constructor that simplifies a rational number:

```
record Rational(int num, int denom) {
    private static int gcd(int a, int b) {
        if (b == 0) return Math.abs(a);
        else return gcd(b, a % b);
    }

    Rational {
        int gcd = gcd(num, denom);
        num    /= gcd;
        denom  /= gcd;
    }
}
```

The compact constructor `Rational {...}` behaves the same as this normal constructor:

```
Rational(int num, int demon) {
    int gcd = gcd(num, denom);
    num    /= gcd;
    denom  /= gcd;
    this.num   = num;
    this.denom = denom;
}
```