CHAPTER 8

Classes

class 1. The noun *class* derives from
Medieval French and French *classe* from Latin *classis*,
probably originally a summons,
hence a summoned collection of persons,
a group liable to be summoned:
perhaps for *callassis* from *calare*,
to call, hence to summon.
—Eric Partridge
Origins: A Short Etymological Dictionary of Modern English

CLASS declarations define new reference types and describe how they are implemented (§8.1).

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

This chapter discusses the common semantics of all classes—top level (§7.6) and nested (including member classes (§8.5, §9.5), local classes (§14.3) and anonymous classes (§15.9.5)). Details that are specific to particular kinds of classes are discussed in the sections dedicated to these constructs.

A named 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. A class may be declared `final` (§8.1.1.2), in which case it cannot have subclasses. If a class is declared `public`, then it can be referred to from other packages. 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). Classes may be *generic*, that is, they may declare type variables (§4.4) whose bindings may differ among different instances of the class.

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

The body of a class declares members (fields and methods and nested classes and interfaces), instance and static initializers, and constructors (§8.1.6). The

scope (§6.3) of a member (§8.2) is the entire declaration of the class to which the member belongs. Field, method, member class, member interface, and constructor declarations may include the access modifiers (§6.6) `public`, `protected`, or `private`. 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 class members and interface members can hide class or interface members 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 (§8.1.3).

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 type; an instance method is invoked with respect to some particular object that is an instance of a class type. 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 (§14.18), thus allowing its activities to be synchronized with those of other threads (§17).

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 Declaration

A *class declaration* specifies a new named reference type. There are two kinds of class declarations - *normal class declarations* and *enum declarations*:

ClassDeclaration:
NormalClassDeclaration
EnumDeclaration

NormalClassDeclaration:
ClassModifiers$_{opt}$ `class` *Identifier* *TypeParameters*$_{opt}$ *Super*$_{opt}$ *Interfaces*$_{opt}$ *ClassBody*

The rules in this section apply to all class declarations unless this specification explicitly states otherwise. In many cases, special restrictions apply to enum declarations. Enum declarations are described in detail in §8.9.

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

8.1.1 Class Modifiers

A class declaration may include *class modifiers*.

ClassModifiers:
ClassModifier
ClassModifiers *ClassModifier*

ClassModifier: one of
Annotation `public` `protected` `private`
`abstract` `static` `final` `strictfp`

Not all modifiers are applicable to all kinds of class declarations. The access modifier `public` pertains only to top level classes (§7.6) and to member classes (§8.5, §9.5), and is discussed in §6.6, §8.5 and §9.5. The access modifiers `protected` and `private` pertain only to member classes within a directly enclosing class declaration (§8.5) and are discussed in §8.5.1. The access modifier `static` pertains only to member classes (§8.5, §9.5). A compile-time error occurs if the same modifier appears more than once in a class declaration.

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

If two or more 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. Normal classes may have `abstract` methods (§8.4.3.1, §9.4), that is methods that are declared but not yet implemented, only if they are `abstract` classes. If a normal class that is not `abstract` contains an `abstract` method, then a compile-time error occurs.

Enum types (§8.9) must not be declared `abstract`; doing so will result in a compile-time error. It is a compile-time error for an enum type *E* to have an abstract method *m* as a member unless *E* has one or more enum constants, and all of *E*'s enum constants have class bodies that provide concrete implementations of *m*. It is a compile-time error for the class body of an enum constant to declare an abstract method.

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

- *C* explicitly contains a declaration of an `abstract` method (§8.4.3).
- Any of *C*'s superclasses has an `abstract` method that has not been implemented (§8.4.8.1) in *C* or any of its superclasses.
- A direct superinterface (§8.1.5) of *C* declares or inherits a method (which is therefore necessarily `abstract`) and *C* neither declares nor inherits a method that implements it.

In the example:

```
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() { }
}
```

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.

A compile-time error occurs if an attempt is made to create an instance of an abstract class using a class instance creation expression (§15.9).

Thus, continuing the example just shown, 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.

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. Thus, in the example just given, instantiation of a SimplePoint causes the default constructor and field initializers for x and y of Point to be executed.

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 incompatible return types.

As an example, the declarations:

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

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 `final` *Classes*

A class can be declared `final` if its definition is complete and no subclasses are desired or required. A compile-time error occurs if the name of a `final` class appears in the `extends` clause (§8.1.4) of another `class` declaration; this implies that a `final` class cannot have any subclasses. A compile-time error occurs 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).

8.1.1.3 `strictfp` *Classes*

The effect of the `strictfp` modifier is to make all `float` or `double` expressions within the class declaration be explicitly FP-strict (§15.4). This implies that all methods declared in the class, and all nested types declared in the class, are implicitly `strictfp`.

Note also that all `float` or `double` expressions within all variable initializers, instance initializers, static initializers and constructors of the class will also be explicitly FP-strict.

8.1.2 Generic Classes and Type Parameters

A class is *generic* if it 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. It defines one or more type variables that act as parameters. A generic class declaration defines a set of parameterized types, one for each possible invocation of the type parameter section. All of these parameterized types share the same class at runtime.

DISCUSSION

For instance, the code

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

will yield true.

TypeParameters ::= < TypeParameterList >

TypeParameterList ::= TypeParameterList , TypeParameter

| *TypeParameter*

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

DISCUSSION

This restriction is needed since the throw and catch mechanism of the Java virtual machine works only with non-generic classes.

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

It is a compile-time error to refer to a type parameter of a class *C* anywhere in the declaration of a static member of *C* or the declaration of a static member of any type declaration nested within *C*. It is a compile-time error to refer to a type parameter of a class C within a static initializer of *C* or any class nested within *C*.

DISCUSSION

Example: Mutually recursive type variable bounds.

```
interface ConvertibleTo<A> {
   A convert();
}

class ReprChange<A implements ConvertibleTo<B>,
                 B implements ConvertibleTo<A>> {
   A a;
   void set(B x) { a = x.convert(); }
   B get() { return a.convert(); }
}
```

Parameterized class declarations can be nested inside other declarations.

DISCUSSION

This is illustrated in the following example:

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

   class Zipper<B> {
      Seq<Pair<A,B>> zip(Seq<B> that) {
     if (this.isEmpty() || that.isEmpty())
             return new Seq<Pair<A,B>>();
     else
             return new Seq<Pair<A,B>>(
                 new Pair<A,B>(this.head, that.head),
             this.tail.zip(that.tail));
      }
   }
}

class Pair<T, S> {
    T fst; S Snd;
    Pair(T f, S s) {fst = f; snd = s;}
}

class Client {{
   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 declared `static`. Inner classes may not declare static initializers (§8.7) or member interfaces. Inner classes may not declare static members, unless they are compile-time constant fields (§15.28).

To illustrate these rules, consider the example below:

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

class Outer{
   class Inner extends HasStatic{
      static final int x = 3;// ok - compile-time constant
      static int y = 4; // compile-time error, an inner class
   }

   static class NestedButNotInner{
      static int z = 5; // ok, not an inner class
   }

   interface NeverInner{}// interfaces are never inner
}
```

Inner classes may inherit static members that are not compile-time constants even though they may not declare them. Nested classes that are not inner classes may declare static members freely, in accordance with the usual rules of the Java programming language. Member interfaces (§8.5) are always implicitly static so they are never considered to be inner classes.

A statement or expression *occurs in a static context* if and only if the innermost method, constructor, instance initializer, static initializer, field initializer, or explicit constructor invocation statement enclosing the statement or expression is a static method, a static initializer, the variable initializer of a static variable, or an explicit constructor invocation statement (§8.8.7).

An inner class C is a *direct inner class of a class* O if O is the immediately lexically enclosing class of C and the declaration of C does not occur in a static context. A class C is an *inner class of class* O if it is either a direct inner class of O or an inner class of an inner class of O.

A class O is the *zeroth lexically enclosing class of itself*. A class O is the *nth lexically enclosing class of a class* C if it is the immediately enclosing class of the $n-1$ st lexically enclosing class of C.

An instance i of a direct inner class C of a class 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 *nth lexically enclosing instance of an instance i* if it is the immediately enclosing instance of the $n-1$ st lexically enclosing instance of *i*.

When an inner class refers to an instance variable that is a member of a lexically enclosing class, the variable of the corresponding lexically enclosing instance is used. A blank final (§4.5.4) field of a lexically enclosing class may not be assigned within an inner class.

An instance of an inner class *I* whose declaration occurs in a static context has no lexically enclosing instances. However, if *I* is immediately declared within a static method or static initializer then *I* does have an *enclosing block*, which is the innermost block statement lexically enclosing the declaration of *I*.

Furthermore, for every superclass *S* of *C* which is itself a direct inner class of a class *S0*, there is an instance of *S0* 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.

Any local variable, formal method parameter or exception handler parameter used but not declared in an inner class must be declared `final`. Any local variable, used but not declared in an inner class must be definitely assigned (§16) before the body of the inner class.

Inner classes include local (§14.3), anonymous (§15.9.5) and non-static member classes (§8.5). Here are some examples:

```
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—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 marked `final`).

Inner classes whose declarations do not occur in a static context may freely refer to the instance variables of their enclosing class. An instance variable is always defined with respect to an instance. In the case of instance variables of an enclosing class, the instance variable must be defined with respect to an enclosing instance of that class. So, 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* of the current class.

Super:
`extends` *ClassType*

The following is repeated from §4.3 to make the presentation here clearer:

ClassType:
TypeName TypeArguments$_{opt}$

A class is said to be a *direct subclass* of its direct superclass. The direct superclass is the class from whose implementation the implementation of the current class is derived. The direct superclass of an enum type `E` is `Enum<E>`. The `extends` clause must not appear in the definition of the class `Object`, because it is the primordial class and has no direct superclass.

Given a (possibly generic) class declaration for `C<A1,...,An>`, $n \geq 0$, $C \neq Object$, the *direct superclass* of the class type (§4.5) $C<A_1,...,A_n>$ is the type given in the extends clause of the declaration of `C` if an extends clause is present, or `Object` otherwise.

Let `C<A1,...,An>`, $n > 0$, be a generic class declaration. The direct superclass of the parameterized class type $C<T_1,...,T_n>$, where T_i, `1 <= i <= n`, is a type, is $D<U_1$ `theta , ...,` U_k `theta>`, where $D<U_1,...,U_k>$ is the direct superclass of $C<A_1,...,A_n>$, and *theta* is the substitution $[A_1 := T_1, ..., A_n := T_n]$.

The *ClassType* must name an accessible (§6.6) class type, or a compile-time error occurs. If the specified *ClassType* names a class that is `final` (§8.1.1.2), then a compile-time error occurs; `final` classes are not allowed to have subclasses. It is a compile-time error if the *ClassType* names the class `Enum` or any invocation of it. If the *TypeName* is followed by any type arguments, it must be correct invocation of the type declaration denoted by *TypeName*, and none of the type arguments may be wildcard type arguments, or a compile-time error occurs.

In the example:

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

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`.

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

- `A` is the direct subclass of `C`.
- There exists a class `B` such that `A` is a subclass of `B`, and `B` is a subclass of `C`, applying this definition recursively.

Class `C` is said to be a *superclass* of class `A` whenever `A` is a subclass of `C`.

In the example:

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

```
final class Colored3dPoint extends ColoredPoint { int z; }
```

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 type *T* if *T* is mentioned in the `extends` or `implements` clause of *C* either as a superclass or superinterface, or as a qualifier of a superclass or superinterface name. A class *C depends* on a reference type *T* if any of the following conditions hold:

- *C* directly depends on *T*.
- *C* directly depends on an interface *I* that depends (§9.1.2) on *T*.
- *C* directly depends on a class *D* that depends on *T* (using this definition recursively).

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

For example:

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

causes a compile-time error.

If circularly declared classes are detected at run time, as classes are loaded (§12.2), then a `ClassCircularityError` is thrown.

8.1.5 Superinterfaces

The optional `implements` clause in a class declaration lists the names of interfaces that are *direct superinterfaces* of the class being declared:

Interfaces:
`implements` *InterfaceTypeList*

InterfaceTypeList:
InterfaceType
InterfaceTypeList `,` *InterfaceType*

The following is repeated from §4.3 to make the presentation here clearer:

InterfaceType:
TypeName TypeArguments$_{opt}$

Given a (possibly generic) class declaration for `C<A1,...,An>`, $n \geq 0$, $C \neq Object$, the *direct superinterfaces* of the class type (§4.5) `C<A1,...,An>` are the types given in the implements clause of the declaration of `C` if an implements clause is present.

Let `C<A1,...,An>`, $n > 0$, be a generic class declaration. The direct superinterfaces of the parameterized class type `C<T1,...,Tn>`, where `Ti`, `1 <= i <= n`, is a type, are all types `I<U1 theta , ..., Uk theta>`, where `I<U1,...,Uk>` is a direct superinterface of `C<A1,...,An>`, and theta is the substitution [`A1` := `T1`, ..., `An` := `Tn`].

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

A compile-time error occurs if the same interface is mentioned as a direct superinterface two or more times in a single `implements` clause names.

This is true even if the interface is named in different ways; for example, the code:

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

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

An interface type `I` is a *superinterface* of class type `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 §9.1.2.
- `I` is a superinterface of the direct superclass of `C`.

A class is said to *implement* all its superinterfaces.

In the example:

```
public interface Colorable {
   void setColor(int color);
   int getColor();
}

public interface Paintable extends Colorable {
   int MATTE = 0, GLOSSY = 1;
   void setFinish(int finish);
```

```
    int 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
{
    int finish;
    public void setFinish(int finish) {
        this.finish = finish;
    }
    public int getFinish() { return finish; }
}
```

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 §9.1.2.

A class can have a superinterface in more than one way. In this example, the class `PaintedPoint` has `Colorable` as a superinterface both because it is a superinterface of `ColoredPoint` and because it is a superinterface of `Paintable`. Unless the class being declared is `abstract`, the declarations of all the method members of each direct superinterface must be implemented either by a declaration in this class or by an existing method declaration inherited from the direct superclass, because a class that is not `abstract` is not permitted to have `abstract` methods (§8.1.1.1).

Thus, the example:

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

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

It is permitted for a single method declaration in a class to implement methods of more than one superinterface. For example, in the code:

```
interface Fish { int getNumberOfScales(); }
interface Piano { int getNumberOfScales(); }
class Tuna implements Fish, Piano {
    // You can tune a piano, but can you tuna fish?
    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` with the same signature and return type as 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).

A class may not at the same time be a subtype of two interface types which are different parameterizations of the same interface.

DISCUSSION

Hence, every superclass and implemented interface of a parameterized type or type variable (§4.4) can be augmented by parameterization to exactly one supertype. Here is an example of an illegal multiple inheritance of an interface:

class B implements I<Integer>
class C extends B implements I<String>
This requirement was introduced in order to support translation by type erasure (§4.6).

8.1.6 Class Body and Member Declarations

A *class body* may contain declarations of members of the class, that is, fields (§8.3), classes (§8.5), interfaces (§8.5) and methods (§8.4). A class body may also contain instance initializers (§8.6), static initializers (§8.7), and declarations of constructors (§8.8) for the class.

ClassBody:
{ *ClassBodyDeclarations*$_{opt}$ }

ClassBodyDeclarations:
ClassBodyDeclaration
ClassBodyDeclarations ClassBodyDeclaration

ClassBodyDeclaration:
ClassMemberDeclaration
InstanceInitializer
StaticInitializer
ConstructorDeclaration

ClassMemberDeclaration:
FieldDeclaration
MethodDeclaration
ClassDeclaration
InterfaceDeclaration
;

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

If *C* itself 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 or inherited in C shadows (§6.3.1) the other definitions of the same kind and name.

8.2 Class Members

I wouldn't want to belong to any club that would accept me as a member.
—Groucho Marx

The members of a class type are all of the following:

- Members inherited from its direct superclass (§8.1.4), except in class `Object`, which has no direct superclass
- Members inherited from any direct superinterfaces (§8.1.5)
- Members declared in the body of the class (§8.1.6)

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.

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

- The type of a field member.
- An ordered 3-tuple consisting of:
  - **argument types:** a list of the types of the arguments to the method member.
  - **return type:** the return type of the method member and the
  - **throws clause:** exception types declared in the throws clause of the method member.

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

The example:

```
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
   }
}
```

causes four compile-time errors:

- An error occurs because `ColoredPoint` has no constructor declared with two integer 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 automatically created (§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.

8.2.1 Examples of Inheritance

This section illustrates inheritance of class members through several examples.

8.2.1.1 *Example: Inheritance with Default 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.

8.2.1.2 *Inheritance with* `public` *and* `protected`

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++;
```

```
    }
}
```

8.2.1.3 *Inheritance with* `private`

In the example:

```
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++;
    }
}
```

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`.

8.2.1.4 *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 if it has 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

Poetic fields encompass me around,
And still I seem to tread on classic ground.
—Joseph Addison (1672–1719), *A Letter from Italy*

The variables of a class type are introduced by *field declarations*:

FieldDeclaration:
FieldModifiers$_{opt}$ *Type VariableDeclarators* `;`

VariableDeclarators:
VariableDeclarator
VariableDeclarators `,` *VariableDeclarator*

VariableDeclarator:
VariableDeclaratorId
VariableDeclaratorId `=` *VariableInitializer*

VariableDeclaratorId:
Identifier
VariableDeclaratorId `[ ]`

VariableInitializer:
Expression
ArrayInitializer

The *FieldModifiers* are described in §8.3.1. The *Identifier* in a *FieldDeclarator* may be used in a name to refer to the field. Fields are members; the scope (§6.3) of a field declaration is specified in §8.1.6. More than one field may be declared in a single field declaration by using more than one declarator; the *FieldModifiers* and *Type* apply to all the declarators in the declaration. Variable declarations involving array types are discussed in §10.2.

It is a compile-time error for the body of a class declaration to declare two fields with the same name. Methods, types, and fields may have the same name, since they are used in different contexts and are disambiguated by different lookup procedures (§6.5).

If the 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. The field declaration also shadows (§6.3.1) declarations of any accessible fields in enclosing classes or interfaces, and any local variables, formal method parameters, and exception handler parameters with the same name in any enclosing blocks.

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 to code in the class and not hidden by a declaration in the class.

Note that 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 (§8.3.3.3). 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 such a reference is ambiguous.

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

A hidden field can be accessed by using a qualified name (if it is `static`) or by using a field access expression (§15.11) that contains the keyword `super` or a cast to a superclass type. See §15.11.2 for discussion and an example.

A value stored in a field of type `float` is always an element of the float value set (§4.2.3); similarly, a value stored in a field of type `double` is always an element of the double value set. It is not permitted for a field of type `float` to contain an element of the float-extended-exponent value set that is not also an element of the float value set, nor for a field of type `double` to contain an element of the double-extended-exponent value set that is not also an element of the double value set.

8.3.1 Field Modifiers

FieldModifiers:
FieldModifier
FieldModifiers FieldModifier

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

The access modifiers `public`, `protected`, and `private` are discussed in §6.6. A compile-time error occurs if the same modifier appears more than once in a field declaration, or if a field declaration has more than one of the access modifiers `public`, `protected`, and `private`.

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

If 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` (sometimes called a non-`static` field) is called an *instance variable*. Whenever a new instance of a class is created, a new variable associated with that instance is created for every instance variable declared in that class or any of its superclasses. The example program:

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

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.

8.3.1.2 final *Fields*

A field can be declared final (§4.5.4). Both class and instance variables (static and non-static fields) may be declared final.

It is a compile-time error if a blank final (§4.5.4) class variable is not definitely assigned (§16.7) by a static initializer (§8.7) of the class in which it is declared.

A blank final instance variable must be definitely assigned (§16.8) at the end of every constructor (§8.8) of the class in which it is declared; otherwise a compile-time error occurs.

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.

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*

As described in §17, the Java programming language allows threads that access shared variables to keep private working copies of the variables; this allows a more efficient implementation of multiple threads. These working copies need be reconciled with the master copies in the shared main memory only at prescribed

synchronization points, namely when objects are locked or unlocked. 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 for some purposes.

A field may be declared `volatile`, in which case a thread must reconcile its working copy of the field with the master copy every time it accesses the variable. Moreover, operations on the master copies of one or more volatile variables on behalf of a thread are performed by the main memory in exactly the order that the thread requested.

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, 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 for more discussion and examples.

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

8.3.2 Initialization of Fields

If a field declarator contains a *variable initializer*, then it has the semantics of an assignment (§15.26) to the declared variable, and:

- If the declarator is for a class variable (that is, a `static` field), then the variable initializer is evaluated and the assignment performed exactly once, when the class is initialized (§12.4).
- If the declarator is for an instance variable (that is, a field that is not `static`), then the variable initializer is evaluated and the assignment performed each time an instance of the class is created (§12.5).

The example:

```
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);
    }
}
```

produces the output:

```
1, 5
```

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

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.

It is a compile-time error if the evaluation of a variable initializer for a `static` field of a named class (or of an interface) can complete abruptly with a checked exception (§11.2).

It is compile-time error if an instance variable initializer of a named class can throw a checked exception unless that exception or one of its supertypes is explicitly declared in the `throws` clause of each constructor of its class and the class has at least one explicitly declared constructor. An instance variable initializer in an anonymous class (§15.9.5) can throw any exceptions.

8.3.2.1 *Initializers for Class Variables*

If a reference by simple name to any instance variable occurs in an initialization expression for a class variable, then a compile-time error occurs.

If the keyword `this` (§15.8.3) or the keyword `super` (§15.11.2, §15.12) occurs in an initialization expression for a class variable, then a compile-time error occurs.

One subtlety here is that, at run time, `static` variables that are `final` and that are initialized with compile-time constant values are initialized first. This also applies to such fields in interfaces (§9.3.1). These variables are "constants" that will never be observed to have their default initial values (§4.5.5), even by devious programs. See §12.4.2 and §13.4.8 for more discussion.

Use of class variables whose declarations appear textually after the use is sometimes restricted, even though these class variables are in scope. See §8.3.2.3 for the precise rules governing forward reference to class variables.

8.3.2.2 *Initializers for Instance Variables*

Initialization expressions for instance variables may use the simple name of any `static` variable declared in or inherited by the class, even one whose declaration occurs textually later.

Thus the example:

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

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.

Initialization expressions for instance variables are permitted to refer to the current object `this` (§15.8.3) and to use the keyword `super` (§15.11.2, §15.12).

Use of instance variables whose declarations appear textually after the use is sometimes restricted, even though these instance variables are in scope. See §8.3.2.3 for the precise rules governing forward reference to instance variables.

8.3.2.3 *Restrictions on the use of Fields during Initialization*

The declaration of a member needs to appear textually before it is used only if the member is an instance (respectively `static`) field of a class or interface *C* and all of the following conditions hold:

- The usage occurs in an instance (respectively `static`) variable initializer of *C* or in an instance (respectively `static`) initializer of *C*.
- The usage is not on the left hand side of an assignment.The usage is via a simple name.
- *C* is the innermost class or interface enclosing the usage.

A compile-time error occurs if any of the three requirements above are not met.

This means that a compile-time error results from the test program:

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

whereas the following example compiles without error:

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

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

These restrictions 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 (§4.5.5).

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;
      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;
}
```

8.3.3 Examples of Field Declarations

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

8.3.3.1 *Example: Hiding of Class Variables*

The example:

```
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);
    }
}
```

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
```

8.3.3.2 *Example: Hiding of Instance Variables*

This example is similar to that in the previous section, but uses instance variables rather than static variables. The code:

```
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);
   }
}
```

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
```

8.3.3.3 *Example: Multiply Inherited Fields*

A class may inherit two or more fields with the same name, either from two interfaces or from its superclass and an interface. 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 example:

```
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 example:

```
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.

8.3.3.4 *Example: 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:

```
public interface Colorable {
   int RED = 0xff0000, GREEN = 0x00ff00, BLUE = 0x0000ff;
}
public 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
{
   ...  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.

8.4 Method Declarations

The diversity of physical arguments and opinions embraces all sorts of methods.
—Michael de Montaigne (1533–1592), *Of Experience*

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

MethodDeclaration:
MethodHeader MethodBody

MethodHeader:
MethodModifiers$_{opt}$ *TypeParameters*$_{opt}$ *ResultType MethodDeclarator Throws*$_{opt}$

ResultType:
Type
`void`

MethodDeclarator:
Identifier `(` *FormalParameterList*$_{opt}$ `)`

The *MethodModifiers* are described in §8.4.3, the *TypeParameters* clause of a method in §8.4.4, the *Throws* clause in §8.4.6, and the *MethodBody* in §8.4.7. A method declaration either specifies the type of value that the method returns or uses the keyword `void` to indicate that the method does not return a value.

The *Identifier* in a *MethodDeclarator* may be used in a name to refer to the method. A class can declare a method with the same name as the class or a field, member class or member interface of the class, but this is discouraged as a matter of syle.

For compatibility with older versions of the Java platform, a declaration form for a method that returns an array is allowed to place (some or all of) the empty bracket pairs that form the declaration of the array type after the parameter list. This is supported by the obsolescent production:

MethodDeclarator:
MethodDeclarator `[ ]`

but should not be used in new code.

It is a compile-time error for the body of a class to declare as members two methods with override-equivalent signatures (§8.4.2) (name, number of parameters, and types of any parameters). Methods and fields may have the same name, since they are used in different contexts and are disambiguated by different lookup procedures (§6.5).

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 or one or more annotations (§9.7)) and an identifier (optionally followed by brackets) that specifies the name of the

parameter. The last formal parameter in a list is special; it may be a *variable arity parameter*, indicated by an elipsis following the type:

FormalParameterList:
LastFormalParameter
FormalParameters `,` *LastFormalParameter*

FormalParameters:
FormalParameter
FormalParameters `,` *FormalParameter*

FormalParameter:
VariableModifiers Type VariableDeclaratorId

VariableModifiers:
VariableModifier
VariableModifiers VariableModifier

VariableModifier: one of
`final` *Annotation*

LastFormalParameter:
VariableModifiers Type `...`$_{opt}$ *VariableDeclaratorId*

The following is repeated from §8.3 to make the presentation here clearer:

VariableDeclaratorId:
Identifier
VariableDeclaratorId `[ ]`

If a method or constructor has no parameters, only an empty pair of parentheses appears in the declaration of the method or constructor.

If two formal parameters of the same method or constructor are declared to have the same name (that is, their declarations mention the same *Identifier*), then a compile-time error occurs.

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

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

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 *DeclaratorId* may be used as a simple name in the body of the method or constructor to refer to the formal parameter.

If the last formal parameter is a variable arity parameter of type *T*, it is considered to define a formal parameter of type *T[]*. The method is then a *variable arity method*. Otherwise, it is a *fixed arity method*. 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).

The scope of a parameter of a method (§8.4.1) or constructor (§8.8.1) is the entire body of the method or constructor.

These parameter names may not be redeclared as local variables of the method, or as exception parameters of catch clauses in a try statement of the method or constructor. However, a parameter of a method or constructor may be shadowed anywhere inside a class declaration nested within that method or constructor. Such a nested class declaration could declare either a local class (§14.3) or an anonymous class (§15.9).

Formal parameters are referred to only using simple names, never by using qualified names (§6.6).

A method or constructor parameter of type `float` always contains an element of the float value set (§4.2.3); similarly, a method or constructor parameter of type `double` always contains an element of the double value set. It is not permitted for a method or constructor parameter of type `float` to contain an element of the float-extended-exponent value set that is not also an element of the float value set, nor for a method parameter of type `double` to contain an element of the double-extended-exponent value set that is not also an element of the double value set.

Where an actual argument expression corresponding to a parameter variable is not FP-strict (§15.4), evaluation of that actual argument expression is permitted to use intermediate values drawn from the appropriate extended-exponent value sets. Prior to being stored in the parameter variable the result of such an expression is mapped to the nearest value in the corresponding standard value set by method invocation conversion (§5.3).

8.4.2 Method Signature

It is a compile-time error to declare two methods with the override-equivalent signatures (defined below) in a class.

Two methods have the same signature if they have the same name and argument types. Two method or constructor declarations `M` and `N` have *the same argument types* if all of the following conditions hold:

- They have the same number of formal parameters (possibly zero)
- They have the same number of type parameters (possibly zero)

- Let $<A_1,\ldots,A_n>$ be the formal type parameters of M and let $<B_1,\ldots,B_n>$ be the formal type parameters of N. After renaming each occurrence of a B_i in N's type to A_i the bounds of corresponding type variables and the argument types of M and N are the same.

The example:

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

causes a compile-time error because it declares two `move` methods with the same signature. This is an error even though one of the declarations is `abstract`.

The signature of a method *m1* is a *subsignature* for the signature of a method *m2* if either

- *m2* has the same signature as *m1*, or
- the signature of *m1* is the same as the erasure of the signature of *m2*.

DISCUSSION

The notion of subsignature defined here 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 that 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 genericity, and now the author of class Overridden 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 genericity, since library writers would hesitate to migrate existing code.

Two method signatures *m1* and *m2* are *override-equivalent* iff either *m1* is a subsignature of *m2* or *m2* is a subsignature of *m1*.

8.4.3 Method Modifiers

MethodModifiers:
MethodModifier
MethodModifiers MethodModifier

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

The access modifiers `public`, `protected`, and `private` are discussed in §6.6. A compile-time error occurs if the same modifier appears more than once in a method declaration, or if a method declaration has more than one of the access modifiers `public`, `protected`, and `private`. A compile-time error occurs if a method declaration that contains the keyword `abstract` also contains any one of the keywords `private`, `static`, `final`, `native`, `strictfp`, or `synchronized`. A compile-time error occurs if a method declaration that contains the keyword `native` also contains `strictfp`.If an annotation *a* on a method declaration corresponds to an annotation type *T*, and *T* has a (meta-)annotation *m* that corresponds to `annotation.Target`, then *m* must have an element whose value is `annotation.ElementType.METHOD`, or a compile-time error occurs. Annotations are discussed further in (§9.7).

If two or more 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), return type, and `throws` clause (if any), but does not provide an implementation. The declaration of an `abstract` method *m* must appear directly within an `abstract` class (call it *A*) unless it occurs within an enum (§8.9); otherwise a compile-time error results. Every subclass of *A* that is not

`abstract` must provide an implementation for *m*, or a compile-time error occurs as specified in §8.1.1.1.

It is a compile-time error for a `private` method to be declared `abstract`.

It would be impossible for a subclass to implement a `private abstract` method, because `private` methods are not inherited by subclasses; therefore such a method could never be used.

It is a compile-time error for a `static` method to be declared `abstract`.

It is a compile-time error for a `final` method to be declared `abstract`.

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 (§11.2) that can be thrown by that method, when it is implemented by its subclasses, is to be more limited. For example, consider this code:

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

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

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

public 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.

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

For example, 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 class Object. (Class 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. An attempt to reference the current object using the keyword this or the keyword super or the type parameters of any surrounding declaration in the body of a class method results in a compile-time error. It is a compile-time error for a static method to be declared abstract.

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.

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.

It is a compile-time error for a `final` method to be declared `abstract`.

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. The 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);
      }
   }
}
```

Here, 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, C++, FORTRAN,or assembly language. The body of a `native` method is given as a semicolon only, indicating that the implementation is omitted, instead of a block.

A compile-time error occurs if a `native` method is declared `abstract`.

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

8.4.3.6 `synchronized` *Methods*

A `synchronized` method acquires a lock (§17.1) before it executes. For a class (`static`) method, the lock associated with the `Class` object for the method's class is used. For an instance method, the lock associated with `this` (the object for which the method was invoked) is used.

These are the same locks that can be used by the `synchronized` statement (§14.18); 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) {
              ...
        }
    }
}
```

The more elaborate example:

```
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;
    }

}
```

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.

See §17 for more discussion of threads and locks.

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 *formal type parameters* of the method. The form of the formal type parameter list is identical to a type parameter list of a class or interface, as described in §8.1.2.

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

Type parameters of generic methods need not be provided explicitly when a generic method is invoked. Instead, they are almost always inferred as specified in §15.12.2.11

8.4.5 Method Return Type

The return type of a method declares the type of value a method returns, if it returns a value, or states that the method is `void`.

A method declaration d_1 with return type R_1 is *return-type-substitutable* for another method d_2 with return type R_2, if and only if the following conditions hold:

- If R_1 is a primitive type, then R_2 is identical to R_1.
- If R_1 is a reference type then:
  - If the signature of d_1 is a the same as that of d_2, then R_1 is either a subtype of R_2 or R_1 can be converted to a subtype of R_2 by unchecked conversion (§5.1.9).
  - Otherwise, R_1 is a subtype of $|R_2|$.
- If d_1 is `void` then d_2 is `void`.

DISCUSSION

The notion of return-type substitutability summarizes the ways in which return types may vary among methods that override each other.

Note that this definition supports *covariant returns* - that is, the specialization of the return type to a subtype (but only for reference types).

Also note that unchecked conversions are allowed as well. This is unsound, and requires an unchecked warning whenever it is used; it is a special allowance is made to allow smooth migration from non-generic to generic code.

8.4.6 Method Throws

A *throws clause* is used to declare any checked exceptions (§11.2) that can result from the execution of a method or constructor:

Throws:
`throws` *ExceptionTypeList*

ExceptionTypeList:
ExceptionType
ExceptionTypeList **,** *ExceptionType*

ExceptionType:
ClassType
TypeVariable

A compile-time error occurs if any *ExceptionType* mentioned in a `throws` clause is not a subtype (§4.10) of `Throwable`. It is permitted but not required to mention other (unchecked) exceptions in a `throws` clause.

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

The requirement to declare checked exceptions allows the 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 will normally result in a compile-time error because of the lack of a proper exception type in a `throws` clause. The Java programming language thus encourages a programming style where rare and otherwise truly exceptional conditions are documented in this way.

The predefined exceptions that are not checked in this way are those for which declaring every possible occurrence would be unimaginably inconvenient:

- Exceptions that are represented by the subclasses of class `Error`, for example `OutOfMemoryError`, are thrown due to a failure in or of the virtual machine. Many of these are the result of linkage failures and can occur at unpredictable points in the execution of a program. Sophisticated programs may yet wish to catch and attempt to recover from some of these conditions.
- The exceptions that are represented by the subclasses of the class `RuntimeException`, for example `NullPointerException`, result from run-time integrity checks and are thrown either directly from the program or in library routines. It is beyond the scope of the Java programming language, and perhaps beyond the state of the art, to include sufficient information in the program to reduce to a manageable number the places where these can be proven not to occur.

A method that overrides or hides another method (§8.4.8), including methods that implement `abstract` methods defined in interfaces, may not be declared to throw more checked exceptions than the overridden or hidden method.

More precisely, suppose that *B* is a class or interface, and *A* is a superclass or superinterface of *B*, and a method declaration *n* in *B* overrides or hides a method declaration *m* in *A*. If *n* has a `throws` clause that mentions any checked exception types, then *m* must have a `throws` clause, and for every checked exception type listed in the `throws` clause of *n*, that same exception class or one of its supertypes must occur in the erasure of the `throws` clause of *m*; otherwise, a compile-time error occurs.

If any exception type in the `throws` clause of *n* does not appear in the (unerased) `throws` clause of *m* an unchecked warning must be issued.

DISCUSSION

See §11 for more information about exceptions and a large example.

Type variables are allowed in throws lists even though they are not allowed in catch clauses.

```
interface PrivilegedExceptionAction<E extends Exception> {
  void run() throws E;
}

class AccessController {
  public static <E extends Exception>
  Object doPrivileged(PrivilegedExceptionAction<E> action) throws E
```

```
  { ... }
}

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. The body of a method must be a semicolon if and only if the method is either `abstract` (§8.4.3.1) or `native` (§8.4.3.4).

MethodBody:
Block
;

A compile-time error occurs if a method declaration is either `abstract` or `native` and has a block for its body. A compile-time error occurs 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: "`{ }`".

If a method is declared `void`, then its body must not contain any `return` statement (§14.16) that has an *Expression*.

If a method is declared to have a return type, then every `return` statement (§14.16) in its body must have an *Expression*. 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; it is not allowed to "drop off the end of its body."

Note that it is possible for a method to have a declared 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 and direct superinterfaces all non-private methods (whether `abstract` or not) of the superclass and superinterfaces that are public, protected or declared with default access in the same package as *C* and are neither overridden (§8.4.8.1) nor hidden (§8.4.8.2) by a declaration in the class.

8.4.8.1 *Overriding (by Instance Methods)*

An instance method *m1* declared in a class *C overrides* another instance method, *m2*, declared in class *A* iff all of the following are true:

1. C is a subclass of *A*.
2. The signature of *m1* is a subsignature (§8.4.2) of the signature of *m2*.
3. Either
   - *m2* is public, protected or declared with default access in the same package as *C*, or
   - *m1* overrides a method *m3*, *m3* distinct from *m1*, *m3* distinct from *m2*, such that *m3* overrides *m2*.

Moreover, if *m1* is not `abstract`, then *m1* is said to *implement* any and all declarations of `abstract` methods that it overrides.

DISCUSSION

The rules allow the signature of the overriding method to differ from the overridden one, to accommodate migration of pre-existing code to take advantage of genericity.

A compile-time error occurs if an instance method overrides 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 overridden method can be accessed by using a method invocation expression (§15.12) that contains the keyword `super`. Note that 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. See §15.12.4.9 for discussion and examples of this point.

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 FP-strict to override an FP-strict method and it is permitted for an FP-strict method to override a method that is not FP-strict.

8.4.8.2 *Hiding (by Class Methods)*

If a class declares a `static` method *m*, then the declaration *m* is said to *hide* any and method *m'*, where the signature of *m* is a subsignature (§8.4.2) of the signature of *m'*, in the superclasses and superinterfaces of the class that would otherwise be accessible to code in the class. A compile-time error occurs 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 (§6.3.1) and obscuring (§6.3.2).

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.

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 for d_2, or a compile-time error occurs. Furthermore, if R_1 is not a subtype of R_2, an unchecked warning must be issued (unless suppressed (§9.6.1.5)).

A method declaration must not have a `throws` clause that conflicts (§8.4.6) with that of any method that it overrides or hides; otherwise, a compile-time error occurs.

DISCUSSION

The rules above allow for *covariant return types* - refining the return type of a method when overriding it..

For example, the following declarations are legal although they were illegal in prior versions of the Java programming language:

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

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

DISCUSSION

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

In the example of Overrider given above, an unchecked warning would be given because the return type of Overrider.toList() is List, which is not a subtype of the return type of the overridden method, List<String>.

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

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

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

DISCUSSION

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 type declaration cannot implement or extend two distinct invocations of the same generic interface. Here are some further examples.

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

```
class C<A> { A id (A 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.

DISCUSSION

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

```
class C<A> { A id (A x) {...} }
interface I<A> { A id(A x); }
class D extends C<String> implements I<Integer> {
   String id(String x) {...}
   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.
- the two methods have different signatures.
- D.id(Integer) is accessible to D.

• 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.

The access modifier (§6.6) of an overriding or hiding method must provide at least as much access as the overridden or hidden method, or a compile-time error occurs. In more detail:

- 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 default (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 hidden or overridden 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.

8.4.8.4 *Inheriting Methods with Override-Equivalent Signatures*

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

It is a compile time error if a class *C* inherits a concrete method whose signatures is a subsignature of another concrete method inherited by *C*.

DISCUSSION

This can happen, if a superclass is parametric, and it has two methods that were distinct in the generic declaration, but have the same signature in the particular invocation used.

Otherwise, there are two possible cases:

- If one of the inherited methods is not `abstract`, then there are two subcases:
  - If the method that is not `abstract` is `static`, a compile-time error occurs.
  - Otherwise, the method that is not `abstract` is considered to override, and therefore to implement, all the other methods on behalf of the class that inherits it. If the signature of the non-abstract method is not a subsignature of each of the other inherited methods an unchecked warning must be issued (unless suppressed (§9.6.1.5)). A compile-time error also occurs if the return type of the non-abstract method is not return type substitutable (§8.4.5) for each of the other inherited methods. If the return type of the non-abstract method is not a subtype of the return type of any of the other inherited methods, an unchecked warning must be issued. Moreover, a compile-time error occurs if the inherited method that is not `abstract` has a `throws` clause that conflicts (§8.4.6) with that of any other of the inherited methods.
- If all the inherited methods are `abstract`, then the class is necessarily an `abstract` class and is considered to inherit all the `abstract` methods. A compile-time error occurs if, for any two such inherited methods, one of the methods is not return type substitutable for the other (The `throws` clauses do not cause errors in this case.)

There might be several paths by which the same method declaration might be 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.

Methods are overridden on a signature-by-signature basis.

If, for example, a class declares two `public` methods with the same name, and a subclass overrides one of them, the subclass still inherits the other method.

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).

8.4.10 Examples of Method Declarations

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

8.4.10.1 *Example: Overriding*

In the example:

```
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;
    }
}
```

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`.

8.4.10.2 *Example: Overloading, Overriding, and Hiding*

In the example:

```
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; }
}
```

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.

8.4.10.3 *Example: Incorrect Overriding*

This example is an extended variation of that in the preceding section:

```
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`.

8.4.10.4 *Example: Overriding versus Hiding*

This example corrects the errors of the example in the preceding section:

```
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.

8.4.10.5 *Example: Invocation of Hidden Class Methods*

A hidden class (`static`) method can be invoked by using a reference whose type is 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.

8.4.10.6 *Large Example of 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 example 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 preexisting contract with subclasses. See the further discussion of binary compatibility in §13, especially §13.2.

8.4.10.7 *Example: Incorrect Overriding because of Throws*

This example 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;
   }
}
```

This example 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`.

8.5 Member Type Declarations

A *member class* is a class whose declaration is directly enclosed in another class or interface declaration. Similarly, a *member interface* is an interface whose declaration is directly enclosed in another class or interface declaration. The scope (§6.3) of a member class or interface is specified in §8.1.6.

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

Within a class *C*, a declaration *d* of a member type named *n* shadows the declarations of any other types named *n* that are in scope at the point where *d* occurs.

If a member class or interface declared with simple name *C* is directly enclosed within the declaration of a class with fully qualified name *N*, then the member class or interface has the fully qualified name *N*`.`*C*. A class inherits from its direct superclass and direct superinterfaces all the non-private member types of the superclass and superinterfaces that are both accessible to code in the class and not hidden by a declaration in the class.

A class may inherit two or more type declarations with the same name, either from two interfaces or from its superclass and an interface. A compile-time error occurs on any attempt to refer to any ambiguously inherited class or interface by its simple name

If the same type declaration is inherited from an interface by multiple paths, the class or interface is considered to be inherited only once. It may be referred to by its simple name without ambiguity.

8.5.1 Modifiers

The access modifiers `public`, `protected`, and `private` are discussed in §6.6. A compile-time error occurs if a member type declaration has more than one of the access modifiers `public`, `protected`, and `private`.

Member type declarations may have annotation modifers just like any type or member declaration.

8.5.2 Static Member Type Declarations

The `static` keyword may modify the declaration of a member type *C* within the body of a non-inner class *T*. Its effect is to declare that *C* is not an inner class. Just as a static method of *T* has no current instance of *T* in its body, *C* also has no current instance of *T*, nor does it have any lexically enclosing instances.

It is a compile-time error if a `static` class contains a usage of a non-`static` member of an enclosing class.

Member interfaces are always implicitly `static`. It is permitted but not required for the declaration of a member interface to explicitly list the `static` modifier.

8.6 Instance Initializers

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

InstanceInitializer:
Block

It is compile-time error if an instance initializer of a named class can throw a checked exception unless that exception or one of its supertypes is explicitly declared in the `throws` clause of each constructor of its class and the class has at least one explicitly declared constructor. An instance initializer in an anonymous class (§15.9.5) can throw any exceptions.

The rules above distinguish between instance initializers in named and anonymous classes. This distinction is deliberate. A given anonymous class is only instantiated at a single point in a program. It is therefore possible to directly propagate information about what exceptions might be raised by an anonymous class' instance initializer to the surrounding expression. Named classes, on the other hand, can be instantiated in many places. Therefore the only way to propagate information about what exceptions might be raised by an instance initializer of a named class is through the `throws` clauses of its constructors. It follows that a more liberal rule can be used in the case of anonymous classes. Similar comments apply to instance variable initializers.

It is a compile-time error if an instance initializer cannot complete normally (§14.20). If a `return` statement (§14.16) appears anywhere within an instance initializer, then a compile-time error occurs.

Use of instance variables whose declarations appear textually after the use is sometimes restricted, even though these instance variables are in scope. See §8.3.2.3 for the precise rules governing forward reference to instance variables.

Instance initializers are permitted to refer to the current object `this` (§15.8.3), to any type variables (§4.4) in scope and to use the keyword `super` (§15.11.2, §15.12).

8.7 Static Initializers

Any *static initializers* declared in a class are executed when the class is initialized and, together with any field initializers (§8.3.2) for class variables, may be used to initialize the class variables of the class (§12.4).

StaticInitializer:
`static` *Block*

It is a compile-time error for a static initializer to be able to complete abruptly (§14.1, §15.6) with a checked exception (§11.2). It is a compile-time error if a static initializer cannot complete normally (§14.20).

The static initializers and class variable initializers are executed in textual order.

Use of class variables whose declarations appear textually after the use is sometimes restricted, even though these class variables are in scope. See §8.3.2.3 for the precise rules governing forward reference to class variables.

If a `return` statement (§14.16) appears anywhere within a static initializer, then a compile-time error occurs.

If the keyword `this` (§15.8.3) or any type variable (§4.4) defined outside the initializer or the keyword `super` (§15.11, §15.12) appears anywhere within a static initializer, then a compile-time error occurs.

8.8 Constructor Declarations

The constructor of wharves, bridges, piers, bulk-heads,
floats, stays against the sea . . .
—Walt Whitman, *Song of the Broad-Axe* (1856)

A *constructor* is used in the creation of an object that is an instance of a class:

ConstructorDeclaration:
ConstructorModifiers$_{opt}$ *ConstructorDeclarator*
Throws$_{opt}$ *ConstructorBody*

ConstructorDeclarator:
TypeParameters$_{opt}$ *SimpleTypeName* (*FormalParameterList*$_{opt}$)

The *SimpleTypeName* in the *ConstructorDeclarator* must be the simple name of the class that contains the constructor declaration; otherwise a compile-time error occurs. In all other respects, the constructor declaration looks just like a method declaration that has no result type.

Here is a simple example:

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

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

Access to constructors is governed by access modifiers (§6.6).

This is useful, for example, in preventing instantiation by declaring an inaccessible constructor (§8.8.10).

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

8.8.1 Formal Parameters and Formal Type Parameter

The formal parameters and formal type parameters of a constructor are identical in structure and behavior to the formal parameters of a method (§8.4.1).

8.8.2 Constructor Signature

It is a compile-time error to declare two constructors with the override-equivalent (§8.4.2) *signatures* in a class. It is a compile-time error to declare two constructors whose signature has the same erasure (§4.6) in a class.

8.8.3 Constructor Modifiers

ConstructorModifiers:
ConstructorModifier
ConstructorModifiers *ConstructorModifier*

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

The access modifiers `public`, `protected`, and `private` are discussed in §6.6. A compile-time error occurs if the same modifier appears more than once in a constructor declaration, or if a constructor declaration has more than one of the access modifiers `public`, `protected`, and `private`.

If no access modifier is specified for the constructor of a normal class, the constructor has default access. If no access modifier is specified for the constructor of an enum type, the constructor is `private`. It is a compile-time error if the constructor of an enum type (§8.9) is declared `public` or `protected`.

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

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` and 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.

Note that a *ConstructorModifier* cannot be declared `strictfp`. This difference in the definitions for *ConstructorModifier* and *MethodModifier* (§8.4.3) is an intentional language design choice; it effectively ensures that a constructor is FP-strict (§15.4) if and only if its class is FP-strict.

8.8.4 Generic Constructors

It is possible for a constructor to be declared generic, independently of whether the class the constructor is declared in is itself generic. A constructor is *generic* if it declares one or more type variables (§4.4). These type variables are known as

the *formal type parameters* of the constructor. The form of the formal type parameter list is identical to a type parameter list of a generic class or interface, as described in §8.1.2.

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

Type parameters of generic constructor need not be provided explicitly when a generic constructor is invoked. When they are not provided, they are inferred as specified in §15.12.2.11.

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 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*$_{opt}$ *BlockStatements*$_{opt}$ `}`

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 the constructor is a constructor for an enum type (§8.9), it is a compile-time error for it to invoke the superclass constructor explicitly.

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 is implicitly assumed by the compiler to begin 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, the body of a constructor is like the body of a method (§8.4.7). A `return` statement (§14.16) may be used in the body of a constructor if it does not include an expression.

In the example:

```
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;
   }
}
```

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.

§12.5 and §15.9 describe the creation and initialization of new class instances.

8.8.7.1 *Explicit Constructor Invocations*

ExplicitConstructorInvocation:
NonWildTypeArguments$_{opt}$ `this` `(` *ArgumentList*$_{opt}$ `)` `;`
NonWildTypeArguments$_{opt}$ `super` `(` *ArgumentList*$_{opt}$ `)` `;`
Primary. NonWildTypeArguments$_{opt}$ `super` `(` *ArgumentList*$_{opt}$ `)` `;`

NonWildTypeArguments:
`<` *ReferenceTypeList* `>`

ReferenceTypeList:
ReferenceType
ReferenceTypeList `,` *ReferenceType*

Explicit constructor invocation statements can be 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 invocation*s begin with either the keyword `super` (possibly prefaced with explicit type arguments) or a *Primary* expression. They are used to invoke a constructor of the direct superclass. Superclass constructor invocations may be further subdivided:
  - *Unqualified superclass constructor invocations* begin with the keyword `super` (possibly prefaced with explicit type arguments).
  - *Qualified superclass constructor invocations* begin with a *Primary* expression . 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.

Here is an example of a qualified superclass constructor invocation:

```
class Outer {
   class Inner{}
}

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

An explicit constructor invocation statement in a constructor body may not refer to any instance variables or instance methods declared in this class or any superclass, or use `this` or `super` in any expression; otherwise, a compile-time error occurs.

For example, if the first constructor of `ColoredPoint` in the example above were changed to:

```
ColoredPoint(int x, int y) {
   this(x, y, color);
}
```

then a compile-time error would occur, because an instance variable cannot be used within a superclass constructor invocation.

A explicit constructor invocation statement can throw an exception type *E* iff either:

- Some subexpression of the constructor invocation list can throw *E*; or
- *E* is declared in the throws clause of the constructor that is invoked.

If an anonymous class instance creation expression appears within an explicit constructor invocation statement, then the anonymous class may not refer to any of the enclosing instances of the class whose constructor is being invoked.

For example:

```
class Top {
   int x;

   class Dummy {
      Dummy(Object o) {}
   }

   class Inside extends Dummy {
      Inside() {
         super(new Object() { int r = x; }); // error
      }
      Inside(final int y) {
         super(new Object() { int r = y; }); // correct
      }
   }
}
```

Let *C* be the class being instantiated, let *S* be the direct superclass of *C*, and let *i* be the instance being created. The evaluation of an explicit constructor invocation proceeds as follows:

- First, if the constructor invocation statement is a superclass constructor invocation, then the immediately enclosing instance of *i* with respect to *S* (if any) must be determined. Whether or not *i* has an immediately enclosing instance with respect to *S* is determined by the superclass constructor invocation as follows:
  - If *S* is not an inner class, or if the declaration of *S* occurs in a static context, no immediately enclosing instance of *i* with respect to *S* exists. A compile-time error occurs if the superclass constructor invocation is a qualified superclass constructor invocation.
  - Otherwise:
    - If the superclass constructor invocation is qualified, then the *Primary* expression *p* immediately preceding "`.super`" is evaluated. If the primary expression 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*. Let *O* be the immediately lexically enclosing class of *S*; it is a compile-time error if the type of *p* is not *O* or a subclass of *O*.
    - Otherwise:

- If *S* is a local class (§14.3), then let *O* be the innermost lexically enclosing class of *S*. Let *n* be an integer such that *O* is the *n*th lexically enclosing class of *C*. The immediately enclosing instance of *i* with respect to *S* is the *n*th lexically enclosing instance of `this`.
- Otherwise, *S* is an inner member class (§8.5). It is a compile-time error if *S* is not a member of a lexically enclosing class, or of a superclass or superinterface thereof. Let *O* be the innermost lexically enclosing class of which *S* is a member, and let *n* be an integer such that *O* is the *n*th lexically enclosing class of *C*. The immediately enclosing instance of *i* with respect to *S* is the *n*th lexically enclosing instance of `this`.

- Second, the arguments to the constructor are evaluated, left-to-right, as in an ordinary method invocation.
- Next, the constructor is invoked.
- Finally, if the constructor invocation statement is a superclass constructor invocation and the 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*. This action is performed regardless of whether the superclass constructor invocation actually appears as an explicit constructor invocation statement or is provided automatically. An alternate constructor invocation does not perform this additional implicit action.

8.8.8 Constructor Overloading

Overloading of constructors is identical in behavior to overloading of methods. 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* that takes no parameters is automatically provided:

- If the class being declared is the primordial class `Object`, then the default constructor has an empty body.
- Otherwise, the default constructor takes no parameters and simply invokes the superclass constructor with no arguments.

A compile-time error occurs if a default constructor is provided by the compiler but the superclass does not have an accessible constructor that takes no arguments.

A default constructor has no `throws` clause.

It follows that if the nullary constructor of the superclass has a `throws` clause, then a compile-time error will occur.

In an enum type (§8.9), the default constructor is implicitly `private`. Otherwise, if the class is declared `public`, then the default constructor is implicitly given the access modifier `public` (§6.6); if the class is declared `protected`, then the default constructor is implicitly given the access modifier `protected` (§6.6); if the class is declared `private`, then the default constructor is implicitly given the access modifier `private` (§6.6); otherwise, the default constructor has the default access implied by no access modifier.

Thus, the example:

```
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`.

The rule that the default constructor of a class has the same access modifier 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 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 accessi-

ble 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 an implicit constructor, and declaring all constructors to be private. 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 declaring no constructor that is public.

Thus, in the example:

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

the class ClassOnly cannot be instantiated, while in the example:

```
package just;

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

the class PackageOnly can be instantiated only within the package just, in which it is declared.

8.9 Enums

An enum declaration has the form:

EnumDeclaration:
ClassModifiers$_{opt}$ `enum` *Identifier Interfaces*$_{opt}$ *EnumBody*

EnumBody:
{ EnumConstants$_{opt}$ *,*$_{opt}$ *EnumBodyDeclarations*$_{opt}$ *}*

Bow, bow, ye lower middle classes!
Bow, bow, ye tradesmen, bow, ye masses!
Blow the trumpets, bang the brasses!
Tantantara! Tzing! Boom!

—W. S. Gilbert, *Iolanthe*

The body of an enum type may contain *enum constants*. An enum constant defines an instance of the enum type. An enum type has no instances other than those defined by its enum constants.

DISCUSSION

It is a compile-time error to attempt to explicitly instantiate an enum type (§15.9.1). The final clone method in Enum ensures that enum constants can never be cloned, and the special treatment by the serialization mechanism ensures that duplicate instances are never created as a result of deserialization. Reflective instantiation of enum types is prohibited. Together, these four things ensure that no instances of an enum type exist beyond those defined by the enum constants.

Because there is only one instance of each enum constant, it is permissible 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.)

EnumConstants:
EnumConstant
EnumConstants , EnumConstant

EnumConstant:
Annotations Identifier $Arguments_{opt}$ $ClassBody_{opt}$

Arguments:
($ArgumentList_{opt}$ *)*

EnumBodyDeclarations:
; $ClassBodyDeclarations_{opt}$

An enum constant may be preceded by annotation (§9.7) modifiers. If an annotation *a* on an enum constant corresponds to an annotation type *T*, and *T* has a (meta-)annotation *m* that corresponds to `annotation.Target`, then *m* must have an element whose value is `annotation.ElementType.FIELD`, or a compile-time error occurs.

An enum constant may be followed by arguments, which are passed to the constructor of the enum type when the constant is created during class initialization as described later in this section. The constructor to be invoked is chosen using the normal overloading rules (§15.12.2). If the arguments are omitted, an empty argument list is assumed. If the enum type has no constructor declarations,

a parameterless default constructor is provided (which matches the implicit empty argument list). This default constructor is `private`.

The optional class body of an enum constant implicitly defines an anonymous class declaration (§15.9.5) that extends the immediately enclosing enum type. The class body is governed by the usual rules of anonymous classes; in particular it cannot contain any constructors.

DISCUSSION

Instance methods declared in these class bodies are accessible outside the enclosing enum type only if they override accessible methods in the enclosing enum type.

Enum types (§8.9) must not be declared `abstract`; doing so will result in a compile-time error. It is a compile-time error for an enum type *E* to have an abstract method *m* as a member unless *E* has one or more enum constants, and all of *E*'s enum constants have class bodies that provide concrete implementations of *m*. It is a compile-time error for the class body of an enum constant to declare an abstract method.

An enum type is implicitly `final` unless it contains at least one enum constant that has a class body. In any case, it is a compile-time error to explicitly declare an enum type to be `final`.

Nested enum types are implicitly `static`. It is permissable to explicitly declare a nested enum type to be `static`.

DISCUSSION

This implies that it is impossible to define a local enum, or to define an enum in an inner class.

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

The direct superclass of an enum type named *E* is `Enum<E>`. In addition to the members it inherits from `Enum<E>`, for each declared enum constant with the name *n* the enum type has an implicitly declared public `static final` field named *n* of type `E`. These fields are considered to be declared in the same order as the corresponding enum constants, before any static fields explicitly declared in the enum type. Each such field is initialized to the enum constant that corresponds to it. Each such field is also considered to be annotated by the same annotations as the corresponding enum constant. The enum constant is said to be *created* when the corresponding field is initialized.

It is a compile-time error for an enum to declare a finalizer. An instance of an enum may never be finalized.

In addition, if *E* is the name of an enum type, then that type has the following implicitly declared static methods:

```
/**
* Returns an array containing the constants of this enum
* type, in the order they're declared.  This method may be
* used to iterate over the constants as follows:
*
*    for(E c : E.values())
*        System.out.println(c);
*
* @return an array containing the constants of this enum
* type, in the order they're declared
*/
public static E[] values();

/**
* Returns the enum constant of this type with the specified
* name.
* The string must match exactly an identifier used to declare
* an enum constant in this type.  (Extraneous whitespace
* characters are not permitted.)
*
* @return the enum constant with the specified name
* @throws IllegalArgumentException if this enum type has no
* constant with the specified name
*/
public static E valueOf(String name);
```

DISCUSSION

It follows that enum type declarations cannot contain fields that conflict with the enum constants, and cannot contain methods that conflict with the automatically generated methods (values() andvalueOf(String) or methods that override the final methods in Enum:

equals(Object), hashCode(), clone(), compareTo(Object), name(), ordinal(), and getDeclaringClass().

It is a compile-time error to reference a non-constant (§15.28) static field of an enum type from its constructors, instance initializer blocks, or instance variable initializer expressions.

DISCUSSION

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

```
enum Color {
        RED, GREEN, BLUE;
        static final Map<String,Color> colorMap =
                                        new HashMap<String,Color>();
        Color() {
            colorMap.put(toString(), this);
        }
    }
```

Static initialization of this enum type 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 won't compile. Note that the example can easily be refactored to work properly:

```
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.

DISCUSSION

Here is program with a nested enum declaration that uses an enhanced for loop to iterate over the constants in the enum:

```
public class Example1 {
    public enum Season { WINTER, SPRING, SUMMER, FALL }

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

Running this program produces the following output:

```
WINTER
SPRING
SUMMER
FALL
```

Here is a program illustrating the use of EnumSet to work with subranges:

```
import java.util.*;

public class Example2 {
    enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATUR-
DAY, SUNDAY }

    public static void main(String[] args) {
        System.out.print("Weekdays: ");
        for (Day d : EnumSet.range(Day.MONDAY, Day.FRIDAY))
            System.out.print(d + " ");
        System.out.println();
    }
}
```

Running this program produces the following output:

```
Weekdays: MONDAY TUESDAY WEDNESDAY THURSDAY FRIDAY
```

EnumSet contains a rich family of static factories, so this technique can be generalized to work non-contiguous subsets as well as subranges. At first glance, it might appear wasteful to generate an EnumSet for a single iteration, but they are so cheap that this is the recommended idiom for iteration over a subrange. Internally, an EnumSet is represented with a single long assuming the enum type has 64 or fewer elements.

Here is a slightly more complex enum declaration for an enum type with an explicit instance field and an accessor for this field. Each member has a different value in the field, and the values are passed in via a constructor. In this example, the field represents the value, in cents, of an American coin. Note, however, that their are no restrictions on the type or number of parameters that may be passed to an enum constructor.

```
public 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; }
}
```

Switch statements are useful for simulating the addition of a method to an enum type from outside the type. This example "adds" a color method to the Coin type, and prints a table of coins, their values, and their colors.

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

    private enum CoinColor { COPPER, NICKEL, SILVER }

    private 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);
        }
    }
}
```

Running the program prints:

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

In the following example, a playing card class is built atop two simple enum types. Note that each enum type would be as long as the entire example in the absence of the enum facility:

```
import java.util.*;

public 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;

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

        public Rank rank() { return rank; }
        public Suit suit() { return 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.comp-
    areTo(c.rank));
        }

         private static final List<Card> prototypeDeck = new ArrayL-
    ist<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);
        }
    }
```

Here's a little program that 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.*;

    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;
    }
}
```

Running the program produces results like this:

```
java Deal 4 5
[FOUR of SPADES, NINE of CLUBS, NINE of SPADES, QUEEN of SPADES,
KING of SPADES]
[THREE of DIAMONDS, FIVE of HEARTS, SIX of SPADES, SEVEN of DIA-
MONDS, KING of DIAMONDS]
[FOUR of DIAMONDS, FIVE of SPADES, JACK of CLUBS, ACE of DIAMONDS,
ACE of HEARTS]
[THREE of HEARTS, FIVE of DIAMONDS, TEN of HEARTS, JACK of HEARTS,
QUEEN of HEARTS]
```

The next example demonstrates the use of constant-specific class bodies to attach behaviors to the constants. (It is anticipated that the need for this will be rare.):

```
import java.util.*;

public 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; }
    };

    // Perform the arithmetic operation represented by this constant
    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));
    }
}
```

Running this program produces the following 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
```

The above pattern is suitable for moderately sophisticated programmers. It is admittedly a bit tricky, but it is much safer than using a case statement in the base type (Operation), as the pattern precludes the possibility of forgetting to add a behavior for a new constant (you'd get a compile-time error).