The Java® Language Specification

Java SE 8 Edition

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley

ORACLE

The Java® Language Specification

Java SE 8 Edition

UnannType:
UnannPrimitiveType
UnannReferenceType

UnannPrimitiveType:
NumericType
`boolean`

UnannReferenceType:
UnannClassOrInterfaceType
UnannTypeVariable
UnannArrayType

UnannClassOrInterfaceType:
UnannClassType
UnannInterfaceType

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

UnannInterfaceType:
UnannClassType

UnannTypeVariable:
Identifier

UnannArrayType:
UnannPrimitiveType Dims
UnannClassOrInterfaceType Dims
UnannTypeVariable Dims

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

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

The *FieldModifier* clause is described in §8.3.1.

The *Identifier* in a *FieldDeclaration* may be used in a name to refer to the field.

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

The declared type of a field is denoted by the *UnannType* that appears in the field declaration, followed by any bracket pairs that follow the *Identifier* in the declarator.

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

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

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.

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

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

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

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

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

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

It is possible for a class to inherit more than one field with the same name. 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 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.

Example 8.3-1. 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 program:

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

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

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

```
interface Frob  { float v = 2.0f; }
class SuperTest { int   v = 3; }
class Test extends SuperTest implements Frob {
    public static void main(String[] args) {
        new Test().printV();
    }
    void printV() {
        System.out.println((super.v + Frob.v)/2);
    }
}
```

It compiles and prints:

```
2.5
```

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

```
interface Color        { int RED=0, GREEN=1,  BLUE=2;  }
```

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

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

Example 8.3-2. Re-inheritance of Fields

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

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

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

8.3.1 Field Modifiers

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

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

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

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 (§12.5), a new variable associated with that instance is created for every instance variable declared in that class or any of its superclasses.

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

```
class Point {
    int x, y, useCount;
    Point(int x, int y) { this.x = x; this.y = y; }
    static final Point origin = new Point(0, 0);
}
class Test {
    public static void main(String[] args) {
        Point p = new Point(1,1);
        Point q = new Point(2,2);
        p.x = 3;
        p.y = 3;
        p.useCount++;
        p.origin.useCount++;
        System.out.println("(" + q.x + "," + q.y + ")");
        System.out.println(q.useCount);
        System.out.println(q.origin == Point.origin);
        System.out.println(q.origin.useCount);
    }
}
```

This program prints:

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

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

access the same object, evidenced by the fact that the value of the reference equality expression (§15.21.3):

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

is true. Further evidence is that the incrementation:

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

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

Example 8.3.1.1-2. Hiding of Class Variables

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

This program produces the output:

```
4.7 2
```

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

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