CHAPTER 6

Names

The Tao that can be told is not the eternal Tao;
The name that can be named is not the eternal name.
The Nameless is the origin of Heaven and Earth;
The Named is the mother of all things.
—Lao-Tsu (c. 6th century BC)

NAMES are used to refer to entities declared in a program. A declared entity (§6.1) is a package, class type (normal or enum), interface type (normal or annotation type), member (class, interface, field, or method) of a reference type, type parameter (of a class, interface, method or constructor) (§4.4), parameter (to a method, constructor, or exception handler), or local variable.

Names in programs are either simple, consisting of a single identifier, or qualified, consisting of a sequence of identifiers separated by "." tokens (§6.2).

Every declaration that introduces a name has a *scope* (§6.3), which is the part of the program text within which the declared entity can be referred to by a simple name.

Packages and reference types (that is, class types, interface types, and array types) have members (§6.4). A member can be referred to using a qualified name *N*.*x*, where *N* is a simple or qualified name and *x* is an identifier. If *N* names a package, then *x* is a member of that package, which is either a class or interface type or a subpackage. If *N* names a reference type or a variable of a reference type, then *x* names a member of that type, which is either a class, an interface, a field, or a method.

In determining the meaning of a name (§6.5), the context of the occurrence is used to disambiguate among packages, types, variables, and methods with the same name.

Access control (§6.6) can be specified in a class, interface, method, or field declaration to control when *access* to a member is allowed. Access is a different concept from scope; access specifies the part of the program text within which the declared entity can be referred to by a qualified name, a field access expression

(§15.11), or a method invocation expression (§15.12) in which the method is not specified by a simple name. The default access is that a member can be accessed anywhere within the package that contains its declaration; other possibilities are `public`, `protected`, and `private`.

Fully qualified and canonical names (§6.7) and naming conventions (§6.8) are also discussed in this chapter.

The name of a field, parameter, or local variable may be used as an expression (§15.14.1). The name of a method may appear in an expression only as part of a method invocation expression (§15.12). The name of a class or interface type may appear in an expression only as part of a class literal (§15.8.2), a qualified `this` expression (§15.8.4), a class instance creation expression (§15.9), an array creation expression (§15.10), a cast expression (§15.16), or an `instanceof` expression (§15.20.2), an enum constant (§8.9), or as part of a qualified name for a field or method. The name of a package may appear in an expression only as part of a qualified name for a class or interface type.

6.1 Declarations

A *declaration* introduces an entity into a program and includes an identifier (§3.8) that can be used in a name to refer to this entity. A declared entity is one of the following:

- A package, declared in a `package` declaration (§7.4)
- An imported type, declared in a single-type-import declaration (§7.5.1) or a type-import-on-demand declaration (§7.5.2)
- A class, declared in a class type declaration (§8.1)
- An interface, declared in an interface type declaration (§9.1)
- A type variable (§4.4), declared as a formal type parameter of a generic class (§8.1.2), interface (§9.1.2) or method (§8.4.4).
- A member of a reference type (§8.2, §9.2, §10.7), one of the following:
  - A member class (§8.5, §9.5).
  - A member interface (§8.5, §9.5).
  - an enum constant (§8.9).
  - A field, one of the following:
    - A field declared in a class type (§8.3)

- A constant field declared in an interface type (§9.3)
- The field `length`, which is implicitly a member of every array type (§10.7)

- A method, one of the following:
  - A method (`abstract` or otherwise) declared in a class type (§8.4)
  - A method (always `abstract`) declared in an interface type (§9.4)

- A parameter, one of the following:
  - A parameter of a method or constructor of a class (§8.4.1, §8.8.1)
  - A parameter of an `abstract` method of an interface (§9.4)
  - A parameter of an exception handler declared in a `catch` clause of a `try` statement (§14.19)
- A local variable, one of the following:
  - A local variable declared in a block (§14.4)
  - A local variable declared in a `for` statement (§14.13)

Constructors (§8.8) are also introduced by declarations, but use the name of the class in which they are declared rather than introducing a new name.

6.2 Names and Identifiers

A *name* is used to refer to an entity declared in a program.

There are two forms of names: simple names and qualified names. A *simple name* is a single identifier. A *qualified name* consists of a name, a "`.`" token, and an identifier.

In determining the meaning of a name (§6.5), the context in which the name appears is taken into account. The rules of §6.5 distinguish among contexts where a name must denote (refer to) a package (§6.5.3), a type (§6.5.5), a variable or value in an expression (§6.5.6), or a method (§6.5.7).

Not all identifiers in programs are a part of a name. Identifiers are also used in the following situations:

- In declarations (§6.1), where an identifier may occur to specify the name by which the declared entity will be known
- In field access expressions (§15.11), where an identifier occurs after a "`.`" token to indicate a member of an object that is the value of an expression or the keyword `super` that appears before the "`.`" token
- In some method invocation expressions (§15.12), where an identifier may occur after a "`.`" token and before a "`(`" token to indicate a method to be invoked for an object that is the value of an expression or the keyword `super` that appears before the "`.`" token
- In qualified class instance creation expressions (§15.9), where an identifier occurs immediately to the right of the leftmost `new` token to indicate a type that must be a member of the compile-time type of the primary expression preceding the "`.`" preceding the leftmost `new` token.
- As labels in labeled statements (§14.7) and in `break` (§14.14) and `continue` (§14.15) statements that refer to statement labels.

In the example:

```
class Test {
   public static void main(String[] args) {
      Class c = System.out.getClass();
      System.out.println(c.toString().length() +
                         args[0].length() + args.length);
   }
}
```

the identifiers `Test`, `main`, and the first occurrences of `args` and `c` are not names; rather, they are used in declarations to specify the names of the declared entities. The names `String`, `Class`, `System.out.getClass`, `System.out.println`, `c.toString`, `args`, and `args.length` appear in the example. The first occurrence of `length` is not a name, but rather an identifier appearing in a method invocation expression (§15.12). The second occurrence of `length` is not a name, but rather an identifier appearing in a method invocation expression (§15.12).

The identifiers used in labeled statements and their associated `break` and `continue` statements are completely separate from those used in declarations. Thus, the following code is valid:

```
class TestString {
   char[] value;
   int offset, count;
   int indexOf(TestString str, int fromIndex) {
      char[] v1 = value, v2 = str.value;
      int max = offset + (count - str.count);
```

```
            int start = offset + ((fromIndex < 0) ? 0 : fromIndex);
        i:
            for (int i = start; i <= max; i++)
            {
                int n = str.count, j = i, k = str.offset;
                while (n-- != 0) {
                  if (v1[j++] != v2[k++])
                    continue i;
                }
                return i - offset;
            }
            return -1;
        }
    }
```

This code was taken from a version of the class `String` and its method `indexOf`, where the label was originally called `test`. Changing the label to have the same name as the local variable `i` does not obscure (§6.3.2) the label in the scope of the declaration of `i`. The identifier `max` could also have been used as the statement label; the label would not obscure the local variable `max` within the labeled statement.

6.3 Scope of a Declaration

The *scope* of a declaration is the region of the program within which the entity declared by the declaration can be referred to using a simple name (provided it is visible (§6.3.1)). A declaration is said to be *in scope* at a particular point in a program if and only if the declaration's scope includes that point.

The scoping rules for various constructs are given in the sections that describe those constructs. For convenience, the rules are repeated here:

The scope of the declaration of an observable (§7.4.3) top level package is all observable compilation units (§7.3). The declaration of a package that is not observable is never in scope. Subpackage declarations are never in scope.

The scope of a type imported by a single-type-import declaration (§7.5.1) or a type-import-on-demand declaration (§7.5.2) is all the class and interface type declarations (§7.6) in the compilation unit in which the import declaration appears.

The scope of a member imported by a single-static-import declaration (§7.5.3) or a static-import-on-demand declaration (§7.5.4) is all the class and interface type declarations (§7.6) in the compilation unit in which the import declaration appears.

The scope of a top level type is all type declarations in the package in which the top level type is declared.

The scope of a label declared by a labeled statement is the statement immediately enclosed by the labeled statement.

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.

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

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.

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

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.

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.

The scope of a local variable declaration in a block (§14.4.2) is the rest of the block in which the declaration appears, starting with its own initializer (§14.4) and including any further declarators to the right in the local variable declaration statement.

The scope of a local class immediately enclosed by a block (§14.2) is the rest of the immediately enclosing block, including its own class declaration. The scope of a local class immediately enclosed by in a switch block statement group (§14.11)is the rest of the immediately enclosing switch block statement group, including its own class declaration.

The scope of a local variable declared in the *ForInit* part of a basic `for` statement (§14.13) includes all of the following:

- Its own initializer
- Any further declarators to the right in the *ForInit* part of the `for` statement
- The *Expression* and *ForUpdate* parts of the `for` statement
- The contained *Statement*

The scope of a local variable declared in the *FormalParameter* part of an enhanced `for` statement (§14.13) is the contained *Statement*

The scope of a parameter of an exception handler that is declared in a `catch` clause of a `try` statement (§14.19) is the entire block associated with the `catch`.

These rules imply that declarations of class and interface types need not appear before uses of the types.

In the example:

```
package points;
class Point {
   int x, y;
   PointList list;
   Point next;
}
class PointList {
   Point first;
}
```

the use of `PointList` in class `Point` is correct, because the scope of the class declaration `PointList` includes both class `Point` and class `PointList`, as well as any other type declarations in other compilation units of package `points`.

6.3.1 Shadowing Declarations

Some declarations may be *shadowed* in part of their scope by another declaration of the same name, in which case a simple name cannot be used to refer to the declared entity.

A declaration *d* of a type named *n* shadows the declarations of any other types named *n* that are in scope at the point where *d* occurs throughout the scope of *d*.

A declaration *d* of a field, local variable, method parameter, constructor parameter or exception handler parameter named *n* shadows the declarations of any other fields, local variables, method parameters, constructor parameters or exception handler parameters named *n* that are in scope at the point where *d* occurs throughout the scope of *d*.

A declaration *d* of a label named *n* shadows the declarations of any other labels named *n* that are in scope at the point where *d* occurs throughout the scope of *d*.

A declaration *d* of a method named *n* shadows the declarations of any other methods named *n* that are in an enclosing scope at the point where *d* occurs throughout the scope of *d*.

A package declaration never shadows any other declaration.

A single-type-import declaration *d* in a compilation unit *c* of package *p* that imports a type named *n* shadows the declarations of:

- any top level type named *n* declared in another compilation unit of *p*.
- any type named *n* imported by a type-import-on-demand declaration in *c*.
- any type named *n* imported by a static-import-on-demand declaration in *c*.

throughout *c*.

A single-static-import declaration *d* in a compilation unit *c* of package *p* that imports a fi named *n* shadows the declaration of any static field named *n1d* imported by a static-import-on-demand declaration in *c*, throughout *c*.

A single-static-import declaration *d* in a compilation unit *c* of package *p* that imports a method named *n* with signature *s* shadows the declaration of any static method named *n* with signature *s* imported by a static-import-on-demand declaration in *c*, throughout *c*.

A single-static-import declaration *d* in a compilation unit *c* of package *p* that imports a type named *n* shadows the declarations of:

- any static type named *n* imported by a static-import-on-demand declaration in *c*.
- any top level type (§7.6) named *n* declared in another compilation unit (§7.3) of *p*.
- any type named *n* imported by a type-import-on-demand declaration (§7.5.2) in *c*.
- any member named *n* imported by a static-import-on-demand declaration (§7.5.2) in *c*.

throughout *c*.

A type-import-on-demand declaration never causes any other declaration to be shadowed.

A static-import-on-demand declaration never causes any other declaration to be shadowed.

A declaration *d* is said to be *visible at point p in a program* if the scope of *d* includes *p*, and *d* is not shadowed by any other declaration at *p*. When the program point we are discussing is clear from context, we will often simply say that a declaration is *visible*.

Note that shadowing is distinct from hiding (§8.3, §8.4.6.2, §8.5, §9.3, §9.5). Hiding, in the technical sense defined in this specification, applies only to members which would otherwise be inherited but are not because of a declaration in a subclass. Shadowing is also distinct from obscuring (§6.3.2).

Here is an example of shadowing of a field declaration by a local variable declaration:

```
class Test {
   static int x = 1;
   public static void main(String[] args) {
      int x = 0;
      System.out.print("x=" + x);
      System.out.println(", Test.x=" + Test.x);
   }
}
```

produces the output:

```
x=0, Test.x=1
```

This example declares:

- a class `Test`
- a class (`static`) variable `x` that is a member of the class `Test`
- a class method `main` that is a member of the class `Test`
- a parameter `args` of the `main` method.
- a local variable `x` of the `main` method

Since the scope of a class variable includes the entire body of the class (§8.2) the class variable `x` would normally be available throughout the entire body of the method `main`. In this example, however, the class variable `x` is shadowed within the body of the method `main` by the declaration of the local variable `x`.

A local variable has as its scope the rest of the block in which it is declared (§14.4.2); in this case this is the rest of the body of the `main` method, namely its initializer "`0`" and the invocations of `print` and `println`.

This means that:

- The expression "`x`" in the invocation of `print` refers to (denotes) the value of the local variable `x`.

The invocation of `println` uses a qualified name (§6.6) `Test.x`, which uses the class type name `Test` to access the class variable `x`, because the declaration of `Test.x` is shadowed at this point and cannot be referred to by its simple name.

The following example illustrates the shadowing of one type declaration by another:

```
import java.util.*;
class Vector {
   int val[] = { 1 , 2 };
}
class Test {
   public static void main(String[] args) {
      Vector v = new Vector();
      System.out.println(v.val[0]);
   }
```

```
}
```

compiles and prints:

```
1
```

using the class `Vector` declared here in preference to class `java.util.Vector` that might be imported on demand.

6.3.2 Obscured Declarations

A simple name may occur in contexts where it may potentially be interpreted as the name of a variable, a type or a package. In these situations, the rules of §6.5 specify that a variable will be chosen in preference to a type, and that a type will be chosen in preference to a package. Thus, it is may sometimes be impossible to refer to a visible type or package declaration via its simple name. We say that such a declaration is *obscured*.

Obscuring is distinct from shadowing (§6.3.1) and hiding (§8.3, §8.4.6.2, §8.5, §9.3, §9.5). The naming conventions of §6.8 help reduce obscuring.

6.4 Members and Inheritance

Packages and reference types have *members*.

This section provides an overview of the members of packages and reference types here, as background for the discussion of qualified names and the determination of the meaning of names. For a complete description of membership, see §7.1, §8.2, §9.2, and §10.7.

6.4.1 The Members of a Package

The members of a package (§7) are specified in §7.1. For convenience, we repeat that specification here:

> The members of a package are subpackages and all the top level (§7.6) class (§8) and top level interface (§9) types declared in all the compilation units (§7.3) of the package.

In general, the subpackages of a package are determined by the host system (§7.2). However, the package `java` always includes the subpackages `lang` and `io` and may include other subpackages. No two distinct members of the same package may have the same simple name (§7.1), but members of different packages may have the same simple name.

For example, it is possible to declare a package:

```
package vector;
```

```
public class Vector { Object[] vec; }
```

that has as a member a `public` class named `Vector`, even though the package `java.util` also declares a class named `Vector`. These two class types are different, reflected by the fact that they have different fully qualified names (§6.7). The fully qualified name of this example `Vector` is `vector.Vector`, whereas `java.util.Vector` is the fully qualified name of the standard `Vector` class. Because the package `vector` contains a class named `Vector`, it cannot also have a subpackage named `Vector`.

6.4.2 The Members of a Class Type

The members of a class type (§8.2) are classes (§8.5, §9.5), interfaces (§8.5, §9.5), fields (§8.3, §9.3, §10.7), and methods (§8.4, §9.4). Members are either declared in the type, or *inherited* because they are accessible members of a superclass or superinterface which are neither private nor hidden nor overridden (§8.4.6).

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

- Members inherited from its direct superclass (§8.1.3), if it has one (the class `Object` has no direct superclass)
- Members inherited from any direct superinterfaces (§8.1.4)

Members declared in the body of the class (§8.1.5)

Constructors (§8.8) are not members.

There is no restriction against a field and a method of a class type having the same simple name. Likewise, there is no restriction against a member class or member interface of a class type having the same simple name as a field or method of that class type.

A class may have two or more fields with the same simple name if they are declared in different interfaces and inherited. An attempt to refer to any of the fields by its simple name results in a compile-time error (§6.5.7.2, §8.2).

In the example:

```
interface Colors {
   int WHITE = 0, BLACK = 1;
}
interface Separates {
   int CYAN = 0, MAGENTA = 1, YELLOW = 2, BLACK = 3;
}
class Test implements Colors, Separates {
   public static void main(String[] args) {
      System.out.println(BLACK); // compile-time error: ambiguous
   }
}
```

the name BLACK in the method main is ambiguous, because class Test has two members named BLACK, one inherited from Colors and one from Separates.

A class type may have two or more methods with the same simple name if the methods have signatures that are not override-equivalent (§8.4.2). Such a method member name is said to be *overloaded*.

A class type may contain a declaration for a method with the same name and the same signature as a method that would otherwise be inherited from a superclass or superinterface. In this case, the method of the superclass or superinterface is not inherited. If the method not inherited is abstract, then the new declaration is said to *implement* it; if the method not inherited is not abstract, then the new declaration is said to *override* it.

In the example:

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

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

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

6.4.3 The Members of an Interface Type

The members of an interface type (§9.2) may be classes (§8.5, §9.5), interfaces (§8.5, §9.5), fields (§8.3, §9.3, §10.7), and methods (§8.4, §9.4).The members of an interface are:

- Those members declared in the interface.
- Those members inherited from direct superinterfaces.

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

An interface may have two or more fields with the same simple name if they are declared in different interfaces and inherited. An attempt to refer to any such field by its simple name results in a compile-time error (§6.5.6.1, §9.2).

In the example:

```
interface Colors {
   int WHITE = 0, BLACK = 1;
}
interface Separates {
   int CYAN = 0, MAGENTA = 1, YELLOW = 2, BLACK = 3;
}
interface ColorsAndSeparates extends Colors, Separates {
            int DEFAULT = BLACK; // compile-time error: ambiguous
}
```

the members of the interface `ColorsAndSeparates` include those members inherited from `Colors` and those inherited from `Separates`, namely `WHITE`, `BLACK` (first of two), `CYAN`, `MAGENTA`, `YELLOW`, and `BLACK` (second of two). The member name `BLACK` is ambiguous in the interface `ColorsAndSeparates`.

6.4.4 The Members of an Array Type

The members of an array type are specified in §10.7. For convenience, we repeat that specification here.

The members of an array type are all of the following:

- The `public final` field `length`, which contains the number of components of the array (`length` may be positive or zero)
- The `public` method `clone`, which overrides the method of the same name in class `Object` and throws no checked exceptions. The return type of the clone method of an array type *T[]* is *T[]*
- All the members inherited from class `Object`; the only method of `Object` that is not inherited is its `clone` method

The example:

```
class Test {
   public static void main(String[] args) {
      int[] ia = new int[3];
      int[] ib = new int[6];
      System.out.println(ia.getClass() == ib.getClass());
      System.out.println("ia has length=" + ia.length);
   }
}
```

produces the output:

```
true
ia has length=3
```

This example uses the method `getClass` inherited from class `Object` and the field `length`. The result of the comparison of the `Class` objects in the first `println` demonstrates that all arrays whose components are of type `int` are instances of the same array type, which is `int[]`.

6.5 Determining the Meaning of a Name

The meaning of a name depends on the context in which it is used. The determination of the meaning of a name requires three steps. First, context causes a name syntactically to fall into one of six categories: *PackageName*, *TypeName*, *ExpressionName*, *MethodName*, *PackageOrTypeName*, or *AmbiguousName*. Second, a name that is initially classified by its context as an *AmbiguousName* or as a *PackageOrTypeName* is then reclassified to be a *PackageName*, *TypeName*, or *ExpressionName*. Third, the resulting category then dictates the final determination of the meaning of the name (or a compilation error if the name has no meaning).

PackageName:
Identifier
PackageName . *Identifier*

TypeName:
Identifier
PackageOrTypeName . *Identifier*

ExpressionName:
Identifier
AmbiguousName . *Identifier*

MethodName:
Identifier
AmbiguousName . *Identifier*

PackageOrTypeName:
Identifier
PackageOrTypeName . *Identifier*

AmbiguousName:
Identifier
AmbiguousName . *Identifier*

The use of context helps to minimize name conflicts between entities of different kinds. Such conflicts will be rare if the naming conventions described in §6.8 are followed. Nevertheless, conflicts may arise unintentionally as types developed by different programmers or different organizations evolve. For example, types, methods, and fields may have the same name. It is always possible to distinguish between a method and a field with the same name, since the context of a use always tells whether a method is intended.

6.5.1 Syntactic Classification of a Name According to Context

A name is syntactically classified as a *PackageName* in these contexts:

- In a package declaration (§7.4)
- To the left of the "." in a qualified *PackageName*

A name is syntactically classified as a *TypeName* in these contexts:

- In a single-type-import declaration (§7.5.1)
- To the left of the "." in a single static import (§7.5.3) declaration
- To the left of the "." in a static import-on-demand (§7.5.4) declaration
- In an actual type argument list of a parameterized type (§4.5)
- In explicit actual type argument list in a generic method or constructor invocation
- In an `extends` clause in a type variable declaration (§8.1.2)
- In an `extends` clause in a class declaration (§8.1.3)
- In an `implements` clause in a class declaration (§8.1.4)
- In an `extends` clause in an interface declaration (§9.1.2)
- After the "@" sign in an annotation (§9.7)
- As a *Type* (or the part of a *Type* that remains after all brackets are deleted) in any of the following contexts:

- In a field declaration (§8.3, §9.3)
- As the result type of a method (§8.4, §9.4)
- As the type of a formal parameter of a method or constructor (§8.4.1, §8.8.1, §9.4)
- As the type of an exception that can be thrown by a method or constructor (§8.4.4, §8.8.4, §9.4)
- As the type of a local variable (§14.4)
- As the type of an exception parameter in a `catch` clause of a `try` statement (§14.19)
- As the type in a class literal (§15.8.2)
- As the qualifying type of a qualified `this` expression (§15.8.4).
- As the class type which is to be instantiated in an unqualified class instance creation expression (§15.9)
- As the direct superclass or direct superinterface of an anonymous class (§15.9.5) which is to be instantiated in an unqualified class instance creation expression (§15.9)
- As the element type of an array to be created in an array creation expression (§15.10)
- As the qualifying type of field access using the keyword `super` (§15.11.2)
- As the qualifying type of a method invocation using the keyword `super` (§15.12)
- As the type mentioned in the cast operator of a cast expression (§15.16)
- As the type that follows the `instanceof` relational operator (§15.20.2)

A name is syntactically classified as an *ExpressionName* in these contexts:

- As the qualifying expression in a qualified superclass constructor invocation (§8.8.5.1)
- As the qualifying expression in a qualified class instance creation expression (§15.9)
- As the array reference expression in an array access expression (§15.13)
- As a *PostfixExpression* (§15.14)
- As the left-hand operand of an assignment operator (§15.26)

A name is syntactically classified as a *MethodName* in these contexts:

- Before the "(" in a method invocation expression (§15.12)
- To the left of the "=" sign in an annotation's element value pair (§9.7)

A name is syntactically classified as a *PackageOrTypeName* in these contexts:

- To the left of the "." in a qualified *TypeName*
- In a type-import-on-demand declaration (§7.5.2)

A name is syntactically classified as an *AmbiguousName* in these contexts:

- To the left of the "." in a qualified *ExpressionName*
- To the left of the "." in a qualified *MethodName*
- To the left of the "." in a qualified *AmbiguousName*
- In the default value clause of an annotation type element declaration (§9.6)
- To the right of an "=" in an an element value pair (§9.7)

6.5.2 Reclassification of Contextually Ambiguous Names

An *AmbiguousName* is then reclassified as follows:

- If the *AmbiguousName* is a simple name, consisting of a single *Identifier*:
  - If the *Identifier* appears within the scope (§6.3) of a local variable declaration (§14.4) or parameter declaration (§8.4.1, §8.8.1, §14.19) or field declaration (§8.3) with that name, then the *AmbiguousName* is reclassified as an *ExpressionName*.
  - Otherwise, if a field of that name is declared in the compilation unit (§7.3) containing the *Identifier* by a single-static-import declaration (§7.5.3), or by a static-import-on-demand declaration (§7.5.4) then the *AmbiguousName* is reclassified as an *ExpressionName*.
  - Otherwise, if the *Identifier* appears within the scope (§6.3) of a top level class (§8) or interface type declaration (§9), a local class declaration (§14.3) or member type declaration (§8.5, §9.5) with that name, then the *AmbiguousName* is reclassified as a *TypeName*.
  - Otherwise, if a type of that name is declared in the compilation unit (§7.3) containing the *Identifier*, either by a single-type-import declaration (§7.5.1), or by a type-import-on-demand declaration (§7.5.2), or by a single-static-

import declaration (§7.5.3), or by a static-import-on-demand declaration (§7.5.4), then the *AmbiguousName* is reclassified as a *TypeName*.

- Otherwise, the *AmbiguousName* is reclassified as a *PackageName*. A later step determines whether or not a package of that name actually exists.

- If the *AmbiguousName* is a qualified name, consisting of a name, a ".", and an *Identifier*, then the name to the left of the "." is first reclassified, for it is itself an *AmbiguousName*. There is then a choice:

  - If the name to the left of the "." is reclassified as a *PackageName*, then if there is a package whose name is the name to the left of the "." and that package contains a declaration of a type whose name is the same as the *Identifier*, then this *AmbiguousName* is reclassified as a *TypeName*. Otherwise, this *AmbiguousName* is reclassified as a *PackageName*. A later step determines whether or not a package of that name actually exists.

  - If the name to the left of the "." is reclassified as a *TypeName*, then if the *Identifier* is the name of a method or field of the class or interface denoted by *TypeName*, this *AmbiguousName* is reclassified as an *ExpressionName*. Otherwise, if the *Identifier* is the name of a member type of the class or interface denoted by *TypeName*, this *AmbiguousName* is reclassified as a *TypeName*. Otherwise, a compile-time error results.

  - If the name to the left of the "." is reclassified as an *ExpressionName*, then let *T* be the type of the expression denoted by *ExpressionName*. If the *Identifier* is the name of a method or field of the class or interface denoted by *T*, this *AmbiguousName* is reclassified as an *ExpressionName*. Otherwise, if the *Identifier* is the name of a member type (§8.5, §9.5) of the class or interface denoted by *T*, then this *AmbiguousName* is reclassified as a *TypeName*. Otherwise, a compile-time error results.

As an example, consider the following contrived "library code":

```
package org.rpgpoet;
import java.util.Random;
interface Music { Random[] wizards = new Random[4]; }
```

and then consider this example code in another package:

```
package bazola;
class Gabriel {
   static int n = org.rpgpoet.Music.wizards.length;
}
```

First of all, the name `org.rpgpoet.Music.wizards.length` is classified as an *ExpressionName* because it functions as a *PostfixExpression*. Therefore, each of the names:

```
org.rpgpoet.Music.wizards
org.rpgpoet.Music
org.rpgpoet
org
```

is initially classified as an *AmbiguousName*. These are then reclassified:

- The simple name `org` is reclassified as a *PackageName* (since there is no variable or type named `org` in scope).
- Next, assuming that there is no class or interface named `rpgpoet` in any compilation unit of package `org` (and we know that there is no such class or interface because package `org` has a subpackage named `rpgpoet`), the qualified name `org.rpgpoet` is reclassified as a *PackageName*.
- Next, because package `org.rpgpoet` has an interface type named `Music`, the qualified name `org.rpgpoet.Music` is reclassified as a *TypeName*.

Finally, because the name `org.rpgpoet.Music` is a *TypeName*, the qualified name `org.rpgpoet.Music.wizards` is reclassified as an *ExpressionName*.

6.5.3 Meaning of Package Names

The meaning of a name classified as a *PackageName* is determined as follows.

6.5.3.1 *Simple Package Names*

If a package name consists of a single *Identifier*, then this identifier denotes a top level package named by that identifier. If no top level package of that name is in scope (§7.4.4), then a compile-time error occurs.

6.5.3.2 *Qualified Package Names*

If a package name is of the form *`Q.Id`*, then *`Q`* must also be a package name. The package name *`Q.Id`* names a package that is the member named *`Id`* within the package named by *`Q`*. If *`Q`* does not name an observable package (§7.4.3), or *`Id`* is not the simple name an observable subpackage of that package, then a compile-time error occurs.

6.5.4 Meaning of *PackageOrTypeNames*

6.5.4.1 *Simple PackageOrTypeNames*

If the *PackageOrTypeName*, *`Q`*, occurs in the scope of a type named *`Q`*, then the *PackageOrTypeName* is reclassified as a *TypeName*.

Otherwise, the *PackageOrTypeName* is reclassified as a *PackageName*. The meaning of the *PackageOrTypeName* is the meaning of the reclassified name.

6.5.4.2 *Qualified PackageOrTypeNames*

Given a qualified *PackageOrTypeName* of the form `Q.Id`, if the type or package denoted by `Q` has a member type named `Id`, then the qualified *PackageOrTypeName* name is reclassified as a *TypeName*.

Otherwise, it is reclassified as a *PackageName*. The meaning of the qualified *PackageOrTypeName* is the meaning of the reclassified name.

6.5.5 Meaning of Type Names

The meaning of a name classified as a *TypeName* is determined as follows.

6.5.5.1 *Simple Type Names*

If a type name consists of a single *Identifier*, then the identifier must occur in the scope of exactly one visible declaration of a type with this name, or a compile-time error occurs. The meaning of the type name is that type.

6.5.5.2 *Qualified Type Names*

If a type name is of the form `Q.Id`, then `Q` must be either a type name or a package name. If `Id` names exactly one type that is a member of the type or package denoted by `Q`, then the qualified type name denotes that type. If `Id` does not name a member type (§8.5, §9.5) within `Q`, or the member type named `Id` within `Q` is not accessible (§6.6), or `Id` names more than one member type within `Q`, then a compile-time error occurs.

The example:

```
package wnj.test;
class Test {
   public static void main(String[] args) {
      java.util.Date date =
         new java.util.Date(System.currentTimeMillis());
      System.out.println(date.toLocaleString());
   }
}
```

produced the following output the first time it was run:

```
Sun Jan 21 22:56:29 1996
```

In this example the name `java.util.Date` must denote a type, so we first use the procedure recursively to determine if `java.util` is an accessible type or a package, which it is, and then look to see if the type `Date` is accessible in this package.

6.5.6 Meaning of Expression Names

The meaning of a name classified as an *ExpressionName* is determined as follows.

6.5.6.1 *Simple Expression Names*

If an expression name consists of a single *Identifier*, then there must be exactly one visible declaration denoting either a local variable, parameter or field in scope at the point at which the the *Identifier* occurs. Otherwise, a compile-time error occurs.

If the declaration declares a final field, the meaning of the name is the value of that field. Otherwise, the meaning of the expression name is the variable declared by the declaration.

If the field is an instance variable (§8.3.1.1), the expression name must appear within the declaration of an instance method (§8.4), constructor (§8.8), or instance variable initializer (§8.3.2.2). If it appears within a `static` method (§8.4.3.2), static initializer (§8.7), or initializer for a `static` variable (§8.3.1.1, §12.4.2), then a compile-time error occurs.

The type of the expression name is the declared type of the field, local variable or parameter after capture conversion (§5.1.10).

In the example:

```
class Test {
   static int v;
   static final int f = 3;
   public static void main(String[] args) {
      int i;
      i = 1;
      v = 2;
      f = 33;                        // compile-time error
      System.out.println(i + " " + v + " " + f);
   }
}
```

the names used as the left-hand-sides in the assignments to `i`, `v`, and `f` denote the local variable `i`, the field `v`, and the value of `f` (not the variable `f`, because `f` is a `final` variable). The example therefore produces an error at compile time because the last assignment does not have a variable as its left-hand side. If the

erroneous assignment is removed, the modified code can be compiled and it will produce the output:

```
1 2 3
```

6.5.6.2 *Qualified Expression Names*

If an expression name is of the form `Q.Id`, then `Q` has already been classified as a package name, a type name, or an expression name:

- If `Q` is a package name, then a compile-time error occurs.
- If `Q` is a type name that names a class type (§8), then:
  - If there is not exactly one accessible (§6.6) member of the class type that is a field named `Id`, then a compile-time error occurs.
  - Otherwise, if the single accessible member field is not a class variable (that is, it is not declared `static`), then a compile-time error occurs.
  - Otherwise, if the class variable is declared `final`, then `Q.Id` denotes the value of the class variable. The type of the expression `Q.Id` is the declared type of the class variable after capture conversion (§5.1.10). If `Q.Id` appears in a context that requires a variable and not a value, then a compile-time error occurs.
  - Otherwise, `Q.Id` denotes the class variable. The type of the expression `Q.Id` is the declared type of the class variable after capture conversion (§5.1.10).
- If `Q` is a type name that names an interface type (§9), then:
  - If there is not exactly one accessible (§6.6) member of the interface type that is a field named `Id`, then a compile-time error occurs.
  - Otherwise, `Q.Id` denotes the value of the field. The type of the expression `Q.Id` is the declared type of the field after capture conversion (§5.1.10). If `Q.Id` appears in a context that requires a variable and not a value, then a compile-time error occurs.
- If `Q` is an expression name, let `T` be the type of the expression `Q`:
  - If `T` is not a reference type, a compile-time error occurs.
  - If there is not exactly one accessible (§6.6) member of the type `T` that is a field named `Id`, then a compile-time error occurs.
  - Otherwise, if this field is any of the following:

- A field of an interface type
- A `final` field of a class type (which may be either a class variable or an instance variable)
- The `final` field `length` of an array type

then *`Q.Id`* denotes the value of the field. The type of the expression *`Q.Id`* is the declared type of the field after capture conversion (§5.1.10). If *`Q.Id`* appears in a context that requires a variable and not a value, then a compile-time error occurs.

- Otherwise, *`Q.Id`* denotes a variable, the field *`Id`* of class *`T`*, which may be either a class variable or an instance variable. The type of the expression *`Q.Id`* is the type of the field member after capture conversion (§5.1.10).

The example:

```
class Point {
   int x, y;
   static int nPoints;
}
class Test {
   public static void main(String[] args) {
      int i = 0;
      i.x++;                  // compile-time error
      Point p = new Point();
      p.nPoints();            // compile-time error
   }
}
```

encounters two compile-time errors, because the `int` variable `i` has no members, and because `nPoints` is not a method of class `Point`.

6.5.7 Meaning of Method Names

A *MethodName* can appear only in a method invocation expression (§15.12). The meaning of a name classified as a *MethodName* is determined as follows.

6.5.7.1 *Simple Method Names*

If a method name consists of a single *Identifier*, then *Identifier* is the method name to be used for method invocation. The *Identifier* must name at least one visible (§6.3.1) method that is in scope at the point where the *Identifier* appear or a method imported by a single-static-import declaration (§7.5.3) or static-import-on-demand declaration (§7.5.4) within the compilation unit within which the *Identifier* appears.

See §15.12 for further discussion of the interpretation of simple method names in method invocation expressions.

6.5.7.2 *Qualified Method Names*

If a method name is of the form *Q*`.`*Id*, then *Q* has already been classified as a package name, a type name, or an expression name. If *Q* is a package name, then a compile-time error occurs. Otherwise, *Id* is the method name to be used for method invocation. If *Q* is a type name, then *Id* must name at least one `static` method of the type *Q*. If *Q* is an expression name, then let *T* be the type of the expression *Q*; *Id* must name at least one method of the type *T*. See §15.12 for further discussion of the interpretation of qualified method names in method invocation expressions.

6.6 Access Control

The Java programming language provides mechanisms for *access control*, to prevent the users of a package or class from depending on unnecessary details of the implementation of that package or class. If access is permitted, then the accessed entity is said to be *accessible*.

Note that accessibility is a static property that can be determined at compile time; it depends only on types and declaration modifiers. Qualified names are a means of access to members of packages and reference types; related means of access include field access expressions (§15.11) and method invocation expressions (§15.12). All three are syntactically similar in that a "`.`" token appears, preceded by some indication of a package, type, or expression having a type and followed by an *Identifier* that names a member of the package or type. These are collectively known as constructs for *qualified access*.

Access control applies to qualified access and to the invocation of constructors by class instance creation expressions (§15.9) and explicit constructor invocations (§8.8.5). Accessibility also effects inheritance of class members (§8.2), including hiding and method overriding (§8.4.6.1).

6.6.1 Determining Accessibility

- A package is always accessible.
- If a class or interface type is declared `public`, then it may be accessed by any code, provided that the compilation unit (§7.3) in which it is declared is observable. If a top level class or interface type is not declared `public`, then it may be accessed only from within the package in which it is declared.

- An array type is accessible if and only if its element type is accessible.
- A member (class, interface, field, or method) of a reference (class, interface, or array) type or a constructor of a class type is accessible only if the type is accessible and the member or constructor is declared to permit access:
  - If the member or constructor is declared `public`, then access is permitted. All members of interfaces are implicitly `public`.
  - Otherwise, if the member or constructor is declared `protected`, then access is permitted only when one of the following is true:
    - Access to the member or constructor occurs from within the package containing the class in which the `protected` member or constructor is declared.
    - Access is correct as described in §6.6.2.
  - Otherwise, if the member or constructor is declared `private`, then access is permitted if and only if it occurs within the body of the top level class (§7.6) that encloses the declaration of the member or constructor.
  - Otherwise, we say there is default access, which is permitted only when the access occurs from within the package in which the type is declared.

6.6.2 Details on `protected` Access

A `protected` member or constructor of an object may be accessed from outside the package in which it is declared only by code that is responsible for the implementation of that object.

6.6.2.1 *Access to a* `protected` *Member*

Let *C* be the class in which a `protected` member m is declared. Access is permitted only within the body of a subclass *S* of *C*. In addition, if *Id* denotes an instance field or instance method, then:

- If the access is by a qualified name *Q*`.`*Id*, where *Q* is an *ExpressionName*, then the access is permitted if and only if the type of the expression *Q* is *S* or a subclass of *S*.
- If the access is by a field access expression *E*`.`*Id*, where *E* is a *Primary* expression, or by a method invocation expression *E*`.`*Id*`(. . .)`, where *E* is a *Primary* expression, then the access is permitted if and only if the type of *E* is *S* or a subclass of *S*.

6.6.2.2 *Qualified Access to a* `protected` *Constructor*

Let *C* be the class in which a `protected` constructor is declared and let *S* be the innermost class in whose declaration the use of the `protected` constructor occurs. Then:

- If the access is by a superclass constructor invocation `super(. . .)` or by a qualified superclass constructor invocation of the form *E*`.super(. . .)`, where *E* is a *Primary* expression, then the access is permitted.
- If the access is by an anonymous class instance creation expression of the form `new` *C*`(. . .){...}` or by a qualified class instance creation expression of the form *E*`.new` *C*`(. . .){...}`, where *E* is a *Primary* expression, then the access is permitted.
- Otherwise, if the access is by a simple class instance creation expression of the form `new` *C*`(. . .)` or by a qualified class instance creation expression of the form *E*`.new` *C*`(. . .)`, where *E* is a *Primary* expression, then the access is not permitted. A `protected` constructor can be accessed by a class instance creation expression (that does not declare an anonymous class) only from within the package in which it is defined.

6.6.3 An Example of Access Control

For examples of access control, consider the two compilation units:

```
package points;
class PointVec { Point[] vec; }
```

and:

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

which declare two class types in the package `points`:

- The class type `PointVec` is not `public` and not part of the `public` interface of the package `points`, but rather can be used only by other classes in the package.
- The class type `Point` is declared `public` and is available to other packages. It is part of the `public` interface of the package `points`.

- The methods `move`, `getX`, and `getY` of the class `Point` are declared `public` and so are available to any code that uses an object of type `Point`.
- The fields `x` and `y` are declared `protected` and are accessible outside the package `points` only in subclasses of class `Point`, and only when they are fields of objects that are being implemented by the code that is accessing them.

See §6.6.7 for an example of how the `protected` access modifier limits access.

6.6.4 Example: Access to `public` and Non-`public` Classes

If a class lacks the `public` modifier, access to the class declaration is limited to the package in which it is declared (§6.6). In the example:

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

two classes are declared in the compilation unit. The class `Point` is available outside the package `points`, while the class `PointList` is available for access only within the package.

Thus a compilation unit in another package can access `points.Point`, either by using its fully qualified name:

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

or by using a single-type-import declaration (§7.5.1) that mentions the fully qualified name, so that the simple name may be used thereafter:

```
package pointsUser;
import points.Point;
class Test {
   public static void main(String[] args) {
      Point p = new Point();
      System.out.println(p.x + " " + p.y);
   }
```

```
}
```

However, this compilation unit cannot use or import `points.PointList`, which is not declared `public` and is therefore inaccessible outside package `points`.

6.6.5 Example: Default-Access Fields, Methods, and Constructors

If none of the access modifiers `public`, `protected`, or `private` are specified, a class member or constructor is accessible throughout the package that contains the declaration of the class in which the class member is declared, but the class member or constructor is not accessible in any other package.

If a `public` class has a method or constructor with default access, then this method or constructor is not accessible to or inherited by a subclass declared outside this package.

For example, if we have:

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

then a subclass in another package may declare an unrelated `move` method, with the same signature (§8.3.2) and return type. Because the original `move` method is not accessible from package `morepoints`, `super` may not be used:

```
package morepoints;
public class PlusPoint extends points.Point {
   public void move(int dx, int dy) {
      super.move(dx, dy);    // compile-time error
      moveAlso(dx, dy);
   }
}
```

Because move of `Point` is not overridden by `move` in `PlusPoint`, the method `moveAlso` in `Point` never calls the method move in `PlusPoint`.

Thus if you delete the `super.move` call from `PlusPoint` and execute the test program:

```
import points.Point;
import morepoints.PlusPoint;
class Test {
    public static void main(String[] args) {
        PlusPoint pp = new PlusPoint();
        pp.move(1, 1);
 }
}
```

it terminates normally. If move of Point were overridden by move in PlusPoint, then this program would recurse infinitely, until a StackoverflowError occurred.

6.6.6 Example: public Fields, Methods, and Constructors

A public class member or constructor is accessible throughout the package where it is declared and from any other package, provided the package in which it is declared is observable (§7.4.3). For example, in the compilation unit:

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

the public class Point has as public members the move method and the moves field. These public members are accessible to any other package that has access to package points. The fields x and y are not public and therefore are accessible only from within the package points.

6.6.7 Example: protected Fields, Methods, and Constructors

Consider this example, where the points package declares:

```
package points;
public class Point {
   protected int x, y;
   void warp(threePoint.Point3d a) {
      if (a.z > 0)    // compile-time error: cannot access a.z
         a.delta(this);
   }
}
```

and the threePoint package declares:

```
package threePoint;
import points.Point;
public class Point3d extends Point {
   protected int z;
   public void delta(Point p) {
      p.x += this.x;  // compile-time error: cannot access p.x
      p.y += this.y;  // compile-time error: cannot access p.y
```

```
        }
    public void delta3d(Point3d q) {
        q.x += this.x;
        q.y += this.y;
        q.z += this.z;
    }
}
```

which defines a class `Point3d`. A compile-time error occurs in the method `delta` here: it cannot access the protected members `x` and `y` of its parameter `p`, because while `Point3d` (the class in which the references to fields `x` and `y` occur) is a subclass of `Point` (the class in which `x` and `y` are declared), it is not involved in the implementation of a `Point` (the type of the parameter `p`). The method `delta3d` can access the protected members of its parameter `q`, because the class `Point3d` is a subclass of `Point` and is involved in the implementation of a `Point3d`.

The method `delta` could try to cast (§5.5, §15.16) its parameter to be a `Point3d`, but this cast would fail, causing an exception, if the class of `p` at run time were not `Point3d`.

A compile-time error also occurs in the method warp: it cannot access the protected member `z` of its parameter `a`, because while the class `Point` (the class in which the reference to field `z` occurs) is involved in the implementation of a `Point3d` (the type of the parameter `a`), it is not a subclass of `Point3d` (the class in which `z` is declared).

6.6.8 Example: `private` Fields, Methods, and Constructors

A `private` class member or constructor is accessible only within the body of the top level class (§7.6) that encloses the declaration of the member or constructor. .It is not inherited by subclasses. In the example:

```
class Point {
    Point() { setMasterID(); }
    int x, y;
    private int ID;
    private static int masterID = 0;
    private void setMasterID() { ID = masterID++; }
}
```

the `private` members `ID`, `masterID`, and `setMasterID` may be used only within the body of class `Point`. They may not be accessed by qualified names, field access expressions, or method invocation expressions outside the body of the declaration of `Point`.

See §8.8.8 for an example that uses a `private` constructor.

6.7 Fully Qualified Names and Canonical Names

Every package, top level class, top level interface, and primitive type has a *fully qualified name*. An array type has a fully qualified name if and only if its element type has a fully qualified name.

- The fully qualified name of a primitive type is the keyword for that primitive type, namely `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, or `double`.
- The fully qualified name of a named package that is not a subpackage of a named package is its simple name.
- The fully qualified name of a named package that is a subpackage of another named package consists of the fully qualified name of the containing package, followed by "`.`", followed by the simple (member) name of the subpackage.
- The fully qualified name of a top level class or top level interface that is declared in an unnamed package is the simple name of the class or interface.
- The fully qualified name of a top level class or top level interface that is declared in a named package consists of the fully qualified name of the package, followed by "`.`", followed by the simple name of the class or interface.
- A member class or member interface *M* of another class *C* has a fully qualified name if and only if *C* has a fully qualified name. In that case, the fully qualified name of *M* consists of the fully qualified name of *C*, followed by "`.`", followed by the simple name of *M*.
- The fully qualified name of an array type consists of the fully qualified name of the component type of the array type followed by "`[]`".

Examples:

- The fully qualified name of the type `long` is "`long`".
- The fully qualified name of the package `java.lang` is "`java.lang`" because it is subpackage `lang` of package `java`.
- The fully qualified name of the class `Object`, which is defined in the package `java.lang`, is "`java.lang.Object`".
- The fully qualified name of the interface `Enumeration`, which is defined in the package `java.util`, is "`java.util.Enumeration`".
- The fully qualified name of the type "array of `double`" is "`double[]`".
- The fully qualified name of the type "array of array of array of array of `String`" is "`java.lang.String[][][][]`".

In the example:

```
package points;
class Point { int x, y; }
class PointVec {
   Point[] vec;
}
```

the fully qualified name of the type `Point` is "`points.Point`"; the fully qualified name of the type `PointVec` is "`points.PointVec`"; and the fully qualified name of the type of the field `vec` of class `PointVec` is "`points.Point[]`".

Every package, top level class, top level interface, and primitive type has a *canonical name*. An array type has a canonical name if and only if its element type has a canonical name. A member class or member interface *M* declared in another class *C* has a canonical name if and only if *C* has a canonical name. In that case, the canonical name of *M* consists of the canonical name of *C*, followed by ".", followed by the simple name of *M*. For every package, top level class, top level interface and primitive type, the canonical name is the same as the fully qualified name. The canonical name of an array type is defined only when the component type of the array has a canonical name. In that case, the canonical name of the array type consists of the canonical name of the component type of the array type followed by "`[]`".

The difference between a fully qualified name and a canonical name can be seen in examples such as:

```
package p;
class O1 { class I{}}
class O2 extends O1{};
```

In this example both `p.O1.I` and `p.O2.I` are fully qualified names that denote the same class, but only `p.O1.I` is its canonical name.

6.8 Naming Conventions

The class libraries of the Java platform attempt to use, whenever possible, names chosen according to the conventions presented here. These conventions help to make code more readable and avoid certain kinds of name conflicts.

We recommend these conventions for use in all programs written in the Java programming language. However, these conventions should not be followed slavishly if long-held conventional usage dictates otherwise. So, for example, the `sin` and `cos` methods of the class `java.lang.Math` have mathematically conventional names, even though these method names flout the convention suggested here because they are short and are not verbs.

6.8.1 Package Names

Names of packages that are to be made widely available should be formed as described in §7.7. Such names are always qualified names whose first identifier consists of two or three lowercase letters that name an Internet domain, such as `com`, `edu`, `gov`, `mil`, `net`, `org`, or a two-letter ISO country code such as `uk` or `jp`. Here are examples of hypothetical unique names that might be formed under this convention:

```
com.JavaSoft.jag.Oak
org.npr.pledge.driver
uk.ac.city.rugby.game
```

Names of packages intended only for local use should have a first identifier that begins with a lowercase letter, but that first identifier specifically should not be the identifier `java`; package names that start with the identifier `java` are reserved by Sun for naming Java platform packages.

When package names occur in expressions:

- If a package name is obscured by a field declaration, then `import` declarations (§7.5) can usually be used to make available the type names declared in that package.
- If a package name is obscured by a declaration of a parameter or local variable, then the name of the parameter or local variable can be changed without affecting other code.

The first component of a package name is normally not easily mistaken for a type name, as a type name normally begins with a single uppercase letter. (The Java programming language does not actually rely on case distinctions to determine whether a name is a package name or a type name.)

6.8.2 Class and Interface Type Names

Names of class types should be descriptive nouns or noun phrases, not overly long, in mixed case with the first letter of each word capitalized. For example:

```
ClassLoader
SecurityManager
Thread
Dictionary
BufferedInputStream
```

Likewise, names of interface types should be short and descriptive, not overly long, in mixed case with the first letter of each word capitalized. The name may be a descriptive noun or noun phrase, which is appropriate when an interface is used as if it were an abstract superclass, such as interfaces `java.io.DataInput` and

java.io.DataOutput; or it may be an adjective describing a behavior, as for the interfaces Runnable and Cloneable.

Obscuring involving class and interface type names is rare. Names of fields, parameters, and local variables normally do not obscure type names because they conventionally begin with a lowercase letter whereas type names conventionally begin with an uppercase letter.

6.8.3 Method Names

Method names should be verbs or verb phrases, in mixed case, with the first letter lowercase and the first letter of any subsequent words capitalized. Here are some additional specific conventions for method names:

- Methods to get and set an attribute that might be thought of as a variable *V* should be named get*V* and set*V*. An example is the methods getPriority and setPriority of class Thread.
- A method that returns the length of something should be named length, as in class String.
- A method that tests a boolean condition *V* about an object should be named is*V*. An example is the method isInterrupted of class Thread.
- A method that converts its object to a particular format *F* should be named to*F*. Examples are the method toString of class Object and the methods toLocaleString and toGMTString of class java.util.Date.

Whenever possible and appropriate, basing the names of methods in a new class on names in an existing class that is similar, especially a class from the Java Application Programming Interface classes, will make it easier to use.

Method names cannot obscure or be obscured by other names (§6.5.7).

6.8.4 Field Names

Names of fields that are not final should be in mixed case with a lowercase first letter and the first letters of subsequent words capitalized. Note that well-designed classes have very few public or protected fields, except for fields that are constants (final static fields) (§6.8.5).

Fields should have names that are nouns, noun phrases, or abbreviations for nouns. Examples of this convention are the fields buf, pos, and count of the class java.io.ByteArrayInputStream and the field bytesTransferred of the class java.io.InterruptedIOException.

Obscuring involving field names is rare.

- If a field name obscures a package name, then an `import` declaration (§7.5) can usually be used to make available the type names declared in that package.
- If a field name obscures a type name, then a fully qualified name for the type can be used unless the type name denotes a local class (§14.3).
- Field names cannot obscure method names.
- If a field name is shadowed by a declaration of a parameter or local variable, then the name of the parameter or local variable can be changed without affecting other code.

6.8.5 Constant Names

The names of constants in interface types should be, and `final` variables of class types may conventionally be, a sequence of one or more words, acronyms, or abbreviations, all uppercase, with components separated by underscore "_" characters. Constant names should be descriptive and not unnecessarily abbreviated. Conventionally they may be any appropriate part of speech. Examples of names for constants include `MIN_VALUE`, `MAX_VALUE`, `MIN_RADIX`, and `MAX_RADIX` of the class `Character`.

A group of constants that represent alternative values of a set, or, less frequently, masking bits in an integer value, are sometimes usefully specified with a common acronym as a name prefix, as in:

```
interface ProcessStates {
    int PS_RUNNING = 0;
    int PS_SUSPENDED = 1;
}
```

Obscuring involving constant names is rare:

- Constant names normally have no lowercase letters, so they will not normally obscure names of packages or types, nor will they normally shadow fields, whose names typically contain at least one lowercase letter.
- Constant names cannot obscure method names, because they are distinguished syntactically.

6.8.6 Local Variable and Parameter Names

Local variable and parameter names should be short, yet meaningful. They are often short sequences of lowercase letters that are not words. For example:

- Acronyms, that is the first letter of a series of words, as in `cp` for a variable holding a reference to a `ColoredPoint`
- Abbreviations, as in `buf` holding a pointer to a `buffer` of some kind
- Mnemonic terms, organized in some way to aid memory and understanding, typically by using a set of local variables with conventional names patterned after the names of parameters to widely used classes. For example:
  - `in` and `out`, whenever some kind of input and output are involved, patterned after the fields of `System`
  - `off` and `len`, whenever an offset and length are involved, patterned after the parameters to the `read` and `write` methods of the interfaces `DataInput` and `DataOutput` of `java.io`

One-character local variable or parameter names should be avoided, except for temporary and looping variables, or where a variable holds an undistinguished value of a type. Conventional one-character names are:

- `b` for a `byte`
- `c` for a `char`
- `d` for a `double`
- `e` for an `Exception`
- `f` for a `float`
- `i`, `j`, and `k` for integers
- `l` for a `long`
- `o` for an `Object`
- `s` for a `String`
- `v` for an arbitrary value of some type

Local variable or parameter names that consist of only two or three lowercase letters should not conflict with the initial country codes and domain names that are the first component of unique package names (§7.7).

What's in a name? That which we call a rose
By any other name would smell as sweet.
—William Shakespeare, *Romeo and Juliet* (c. 1594), Act II, scene ii

Rose is a rose is a rose is a rose.
—Gertrude Stein, *Sacred Emily* (1913), in *Geographies and Plays*

. . . stat rosa pristina nomine, nomina nuda tenemus.
—Bernard of Morlay, *De contemptu mundi* (12th century),
quoted in Umberto Eco, *The Name of the Rose* (1980)

Rose, Rose, bo-Bose,
Banana-fana fo-Fose,
Fee, fie, mo-Mose—
—Rose!
—Lincoln Chase and Shirley Elliston, *The Name Game*
(#3 pop single in the U.S., January 1965),
as applied to the name "Rose"