CHAPTER 7

Packages

Good things come in small packages.
—Traditional proverb

PROGRAMS are organized as sets of packages. Each package has its own set of names for types, which helps to prevent name conflicts. A **top level** type is accessible (§6.6) outside the package that declares it only if the type is declared `public`.

The naming structure for packages is hierarchical (§7.1). The members of a package are class and interface types (§7.6), which are declared in compilation units of the package, and subpackages, which may contain compilation units and subpackages of their own.

A package can be stored in a file system (§7.2.1) or in a database (§7.2.2). Packages that are stored in a file system may have certain constraints on the organization of their compilation units to allow a simple implementation to find classes easily.

A package consists of a number of compilation units (§7.3). A compilation unit automatically has access to all types declared in its package and also automatically imports all of the public types declared in the predefined package `java.lang`.

For small programs and casual development, a package can be unnamed (§7.4.2) or have a simple name, but if code is to be widely distributed, unique package names should be chosen (§7.7). This can prevent the conflicts that would otherwise occur if two development groups happened to pick the same package name and these packages were later to be used in a single program.

7.1 Package Members

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.

For example, in the Java Application Programming Interface:

- The package `java` has subpackages `awt`, `applet`, `io`, `lang`, `net`, and `util`, but no compilation units.
- The package `java.awt` has a subpackage named `image`, as well as a number of compilation units containing declarations of class and interface types.

If the fully qualified name (§6.7) of a package is *P*, and *Q* is a subpackage of *P*, then *P*`.`*Q* is the fully qualified name of the subpackage.

A package may not contain two members of the same name, or a compile-time error results.

Here are some examples:

- Because the package `java.awt` has a subpackage `image`, it cannot (and does not) contain a declaration of a class or interface type named `image`.
- If there is a package named `mouse` and a member type `Button` in that package (which then might be referred to as `mouse.Button`), then there cannot be any package with the fully qualified name `mouse.Button` or `mouse.Button.Click`.

If `com.sun.java.jag` is the fully qualified name of a type, then there cannot be any package whose fully qualified name is either `com.sun.java.jag` or `com.sun.java.jag.scrabble`.

The hierarchical naming structure for packages is intended to be convenient for organizing related packages in a conventional manner, but has no significance in itself other than the prohibition against a package having a subpackage with the same simple name as a **top level** type (§7.6) **declared in** that package. There is no special access relationship between a package named `oliver` and another package named `oliver.twist`, or between packages named `evelyn.wood` and `evelyn.waugh`. For example, the code in a package named `oliver.twist` has no better access to the types declared within package `oliver` than code in any other package.

7.2 Host Support for Packages

Each host determines how packages, compilation units, and subpackages are created and stored, and which compilation units are observable (§7.3) in a particular compilation.

The observability of compilation units in turn determines which packages are observable, and which packages are in scope.

The packages may be stored in a local file system in simple implementations of the Java platform. Other implementations may use a distributed file system or some form of database to store source and/or binary code.

7.2.1 Storing Packages in a File System

As an extremely simple example, all the packages and source and binary code on a system might be stored in a single directory and its subdirectories. Each immediate subdirectory of this directory would represent a top level package, that is, one whose fully qualified name consists of a single simple name. The directory might contain the following immediate subdirectories:

```
com
gls
jag
java
wnj
```

where directory `java` would contain the Java Application Programming Interface packages; the directories `jag`, `gls`, and `wnj` might contain packages that three of the authors of this specification created for their personal use and to share with each other within this small group; and the directory `com` would contain packages procured from companies that used the conventions described in §7.7 to generate unique names for their packages.

Continuing the example, the directory `java` would contain, among others, the following subdirectories:

```
applet
awt
io
lang
net
util
```

corresponding to the packages `java.applet`, `java.awt`, `java.io`, `java.lang`, `java.net`, and `java.util` that are defined as part of the Java Application Programming Interface.

Still continuing the example, if we were to look inside the directory `util`, we might see the following files:

```
BitSet.java                 Observable.java
BitSet.class                Observable.class
Date.java                   Observer.java
Date.class                  Observer.class
...
```

where each of the `.java` files contains the source for a compilation unit (§7.3) that contains the definition of a class or interface whose binary compiled form is contained in the corresponding `.class` file.

Under this simple organization of packages, an implementation of the Java platform would transform a package name into a pathname by concatenating the components of the package name, placing a file name separator (directory indicator) between adjacent components.

For example, if this simple organization were used on a UNIX system, where the file name separator is `/`, the package name:

```
jag.scrabble.board
```

would be transformed into the directory name:

```
jag/scrabble/board
```

and:

```
com.sun.sunsoft.DOE
```

would be transformed to the directory name:

```
com/sun/sunsoft/DOE
```

A package name component or class name might contain a character that cannot correctly appear in a host file system's ordinary directory name, such as a Unicode character on a system that allows only ASCII characters in file names. As a convention, the character can be escaped by using, say, the `@` character followed by four hexadecimal digits giving the numeric value of the character, as in the `\u`*xxxx* escape (§3.3), so that the package name:

```
children.activities.crafts.papierM\u00e2ch\u00e9
```

which can also be written using full Unicode as:

```
children.activities.crafts.papierMâché
```

might be mapped to the directory name:

```
children/activities/crafts/papierM@00e2ch@00e9
```

If the `@` character is not a valid character in a file name for some given host file system, then some other character that is not valid in a identifier could be used instead.

7.2.2 Storing Packages in a Database

A host system may store packages and their compilation units and subpackages in a database.

Such a database must not impose the optional restrictions (§7.6) on compilation units in file-based implementations. For example, a system that uses a database to store packages may not enforce a maximum of one `public` class or interface per compilation unit.

Systems that use a database must, however, provide an option to convert a program to a form that obeys the restrictions, for purposes of export to file-based implementations.

7.3 Compilation Units

CompilationUnit is the goal symbol (§2.1) for the syntactic grammar (§2.3) of Java programs. It is defined by the following productions:

CompilationUnit:
PackageDeclaration$_{opt}$ *ImportDeclarations*$_{opt}$ *TypeDeclarations*$_{opt}$

ImportDeclarations:
ImportDeclaration
ImportDeclarations ImportDeclaration

TypeDeclarations:
TypeDeclaration
TypeDeclarations TypeDeclaration

Types declared in different compilation units can depend on each other, circularly. A Java compiler must arrange to compile all such types at the same time.

A *compilation unit* consists of three parts, each of which is optional:

- A `package` declaration (§7.4), giving the fully qualified name (§6.7) of the package to which the compilation unit belongs. A compilation unit that has no package declaration is part of an unnamed package (§7.4.2).
- `import` declarations (§7.5) that allow types from other packages and static members of types to be referred to using their simple names
- Top level type declarations (§7.6) of class and interface types

Which compilation units are *observable* is determined by the host system. However, all the compilation units of the package `java` and its subpackages `lang` and `io` must always be observable. The observability of a compilation unit influences the observability of its package (§7.4.3).

Every compilation unit automatically and implicitly imports every `public` type name declared by the predefined package `java.lang`, so that the names of all those types are available as simple names, as described in §7.5.5.

7.4 Package Declarations

A package declaration appears within a compilation unit to indicate the package to which the compilation unit belongs.

7.4.1 Named Packages

A *package declaration* in a compilation unit specifies the name (§6.2) of the package to which the compilation unit belongs.

PackageDeclaration:
Annotations$_{opt}$ `package` *PackageName* `;`

The keyword `package` may optionally be preceded by annotation modifiers (§9.7). If an annotation *a* on a package 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.PACKAGE`, or a compile-time error occurs.

The package name mentioned in a package declaration must be the fully qualified name (§6.7) of the package.

7.4.1.1 *Package Annotations*

Annotations may be used on package declarations, with the restriction that at most one annotated package declaration is permitted for a given package.

DISCUSSION

The manner in which this restriction is enforced must, of necessity, vary from implementation to implementation. The following scheme is strongly recommended for file-system-based implementations: The sole annotated package declaration, if it exists, is placed in a source file called `package-info.java` in the directory containing the source files for the package. This file does not contain the source for a class called package-info.java; indeed it would be illegal for it to do so, as package-info is not a legal identifier. Typically `package-info.java` contains only a package declaration, preceded immediately by the annotations on the package. While the file could technically contain the source code for one or more package-private classes, it would be very bad form.

It is recommended that `package-info.java`, if it is present, take the place of `package.html` for javadoc and other similar documentation generation systems. If this file is present, the documentation generation tool should ignore package.html and look instead for the package documentation comment immediately preceding the (possibly annotated) package declaration in `package-info.java`. In this way, package-info.java becomes the sole repository for package level annotations and documentation. If, in future, it becomes

desirable to add any other package-level information, this file should prove a convenient home for this information.

7.4.2 Unnamed Packages

A compilation unit that has no package declaration is part of an unnamed package.

Note that an unnamed package cannot have subpackages, since the syntax of a package declaration always includes a reference to a named top level package. As an example, the compilation unit:

```
class FirstCall {
    public static void main(String[] args) {
        System.out.println("Mr. Watson, come here. "
                           + "I want you.");
    }
}
```

defines a very simple compilation unit as part of an unnamed package.

An implementation of the Java platform must support at least one unnamed package; it may support more than one unnamed package but is not required to do so. Which compilation units are in each unnamed package is determined by the host system.

In implementations of the Java platform that use a hierarchical file system for storing packages, one typical strategy is to associate an unnamed package with each directory; only one unnamed package is observable at a time, namely the one that is associated with the "current working directory." The precise meaning of "current working directory" depends on the host system.

Unnamed packages are provided by the Java platform principally for convenience when developing small or temporary applications or when just beginning development.

7.4.3 Observability of a Package

A package is *observable* if and only if either:

- A compilation unit containing a declaration of the package is observable.
- A subpackage of the package is observable.

One can conclude from the rule above and from the requirements on observable compilation units, that the packages `java`, `java.lang`, and `java.io` are always observable.

7.4.4 Scope of a Package Declaration

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.

It follows that the package `java` is always in scope (§6.3).

Package declarations never shadow other declarations.

7.5 Import Declarations

An *import declaration* allows a static member or a **named** type to be referred to by a simple name (§6.2) that consists of a single identifier. Without the use of an appropriate `import` declaration, the only way to refer to a type declared in another package, or a static member of another type, is to use a fully qualified name (§6.7).

ImportDeclaration:
SingleTypeImportDeclaration
TypeImportOnDemandDeclaration
SingleStaticImportDeclaration
StaticImportOnDemandDeclaration

A single-type-import declaration (§7.5.1) imports a single **named** type, by mentioning its **canonical** name.

A type-import-on-demand declaration (§7.5.2) imports all the accessible types of a named **type** or package as needed. It is a compile time error to import a type from the unnamed package.

A single static import declaration (§7.5.3) imports all single static members with a given name from a type, by giving its canonical name.

A static-import-on-demand declaration (§7.5.4) imports all accessible static members of a named type as needed.

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.

An `import` declaration makes types available by their simple names only within the compilation unit that actually contains the `import` declaration. The scope of the entities(s) it introduces specifically does not include the `package`

statement, other `import` declarations in the current compilation unit, or other compilation units in the same package. See §7.5.6 for an illustrative example.

7.5.1 Single-Type-Import Declaration

A *single-type-import declaration* imports a single type by giving its canonical name, making it available under a simple name in the class and interface declarations of the compilation unit in which the single-type import declaration appears.

SingleTypeImportDeclaration:
`import` *TypeName* `;`

The *TypeName* must be the canonical name of a class or interface type; a compile-time error occurs if the named type does not exist. The named type must be accessible (§6.6) or a compile-time error occurs.

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*.

The example:

```
import java.util.Vector;
```

causes the simple name `Vector` to be available within the class and interface declarations in a compilation unit. Thus, the simple name `Vector` refers to the type declaration `Vector` in the package `java.util` in all places where it is not shadowed (§6.3.1) or obscured (§6.3.2) by a declaration of a field, parameter, local variable, or nested type declaration with the same name.

DISCUSSION

Note that Vector is declared as a generic type. Once imported, the name Vector can be used without qualification in a parameterized type such as Vector<String>, or as the raw type Vector

This highlights a limitation of the import declaration. A type nested inside a generic type declaration can be imported, but its outer type is always erased.

If two single-type-import declarations in the same compilation unit attempt to import types with the same simple name, then a compile-time error occurs, unless the two types are the same type, in which case the duplicate declaration is ignored. If the type imported by the the single-type-import declaration is declared in the compilation unit that contains the import declaration, the import declaration is ignored. If a compilation unit contains both a single-static-import (§7.5.3) declaration that imports a type whose simple name is *n*, and a single-type-import declaration (§7.5.1) that imports a type whose simple name is *n*, a compile-time error occurs.

If another top level type with the same simple name is otherwise declared in the current compilation unit except by a type-import-on-demand declaration (§7.5.2) or a static-import-on-demand declaration (§7.5.4), then a compile-time error occurs.

So the sample program:

```
import java.util.Vector;
class Vector { Object[] vec; }
```

causes a compile-time error because of the duplicate declaration of `Vector`, as does:

```
import java.util.Vector;
import myVector.Vector;
```

where `myVector` is a package containing the compilation unit:

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

The compiler keeps track of types by their binary names (§13.1).

Note that an import statement cannot import a subpackage, only a type. For example, it does not work to try to import `java.util` and then use the name `util.Random` to refer to the type `java.util.Random`:

```
import java.util;                // incorrect: compile-time error
class Test { util.Random generator; }
```

7.5.2 Type-Import-on-Demand Declaration

A *type-import-on-demand declaration* allows all accessible (§6.6) types declared in the type or package named by a canonical name to be imported as needed.

TypeImportOnDemandDeclaration:
`import` *PackageOrTypeName* `. * ;`

It is a compile-time error for a type-import-on-demand declaration to name a type or package that is not accessible. Two or more type-import-on-demand declarations in the same compilation unit may name the same type or package. All but one of these declarations are considered *redundant*; the effect is as if that type was imported only once.

If a compilation unit contains both a static-import-on-demand declaration and a type-import-on-demand (§7.5.2) declaration that name the same type, the effect is as if the static member types of that type were imported only once.

It is not a compile-time error to name the current package or `java.lang` in a type-import-on-demand declaration. The type-import-on-demand declaration is ignored in such cases.

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

The example:

```
import java.util.*;
```

causes the simple names of all `public` types declared in the package `java.util` to be available within the class and interface declarations of the compilation unit. Thus, the simple name `Vector` refers to the type `Vector` in the package `java.util` in all places in the compilation unit where that type declaration is not shadowed (§6.3.1) or obscured (§6.3.2). The declaration might be shadowed by a single-type-import declaration of a type whose simple name is `Vector`; by a type named `Vector` and declared in the package to which the compilation unit belongs; or any nested classes or interfaces. The declaration might be obscured by a declaration of a field, parameter, or local variable named `Vector` (It would be unusual for any of these conditions to occur.)

7.5.3 Single Static Import Declaration

A *single-static-import declaration* imports all accessible (§6.6) static members with a given simple name from a type. This makes these static members available under their simple name in the class and interface declarations of the compilation unit in which the single-static import declaration appears.

SingleStaticImportDeclaration:
`import static` *TypeName* `.` *Identifier* `;`

The *TypeName* must be the canonical name of a class or interface type; a compile-time error occurs if the named type does not exist. The named type must be accessible (§6.6) or a compile-time error occurs. The *Identifier* must name at least one static member of the named type; a compile-time error occurs if there is no member of that name or if all of the named members are not accessible.

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*.

Note that it is permissable for one single-static-import declaration to import several fields or types with the same name, or several methods with the same name and signature.

If a compilation unit contains both a single-static-import (§7.5.3) declaration that imports a type whose simple name is *n*, and a single-type-import declaration (§7.5.1) that imports a type whose simple name is *n*, a compile-time error occurs.

If a single-static-import declaration imports a type whose simple name is *n*, and the compilation unit also declares a top level type (§7.6) whose simple name is *n*, a compile-time error occurs.

7.5.4 Static-Import-on-Demand Declaration

A *static-import-on-demand declaration* allows all accessible (§6.6) static members declared in the type named by a canonical name to be imported as needed.

StaticImportOnDemandDeclaration:

```
import static TypeName . * ;
```

It is a compile-time error for a static-import-on-demand declaration to name a type that does not exist or a type that is not accessible. Two or more static-import-

on-demand declarations in the same compilation unit may name the same member; the effect is as if that member were imported only once.

Note that it is permissable for one static-import-on-demand declaration to import several fields or types with the same name, or several methods with the same name and signature.

If a compilation unit contains both a static-import-on-demand declaration and a type-import-on-demand (§7.5.2) declaration that name the same type, the effect is as if the static member types of that type were imported only once.

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

7.5.5 Automatic Imports

Each compilation unit automatically imports all of the `public` type names declared in the predefined package `java.lang`, as if the declaration:

```
import java.lang.*;
```

appeared at the beginning of each compilation unit, immediately following any `package` statement.

7.5.6 A Strange Example

Package names and type names are usually different under the naming conventions described in §6.8. Nevertheless, in a contrived example where there is an unconventionally-named package `Vector`, which declares a `public` class named `Mosquito`:

```
package Vector;
public class Mosquito { int capacity; }
```

and then the compilation unit:

```
package strange.example;
import java.util.Vector;
import Vector.Mosquito;
class Test {
   public static void main(String[] args) {
      System.out.println(new Vector().getClass());
      System.out.println(new Mosquito().getClass());
```

```
    }
}
```

the single-type-import declaration (§7.5.1) importing class `Vector` from package `java.util` does not prevent the package name `Vector` from appearing and being correctly recognized in subsequent `import` declarations. The example compiles and produces the output:

```
class java.util.Vector
class Vector.Mosquito
```

7.6 Top Level Type Declarations

A *top level type declaration* declares a top level class type (§8) or a top level interface type (§9): with signature

TypeDeclaration:
ClassDeclaration
InterfaceDeclaration
;

By default, the top level types declared in a package are accessible only within the compilation units of that package, but a type may be declared to be `public` to grant access to the type from code in other packages (§6.6, §8.1.1, §9.1.1).

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

If a top level type named *T* is declared in a compilation unit of a package whose fully qualified name is *P*, then the fully qualified name of the type is *P*`.`*T*. If the type is declared in an unnamed package (§7.4.2), then the type has the fully qualified name *T*.

Thus in the example:

```
package wnj.points;
class Point { int x, y; }
```

the fully qualified name of class `Point` is `wnj.points.Point`.

An implementation of the Java platform must keep track of types within packages by their binary names (§13.1). Multiple ways of naming a type must be expanded to binary names to make sure that such names are understood as referring to the same type.

For example, if a compilation unit contains the single-type-import declaration (§7.5.1):

```
import java.util.Vector;
```

then within that compilation unit the simple name `Vector` and the fully qualified name `java.util.Vector` refer to the same type.

When packages are stored in a file system (§7.2.1), the host system may choose to enforce the restriction that it is a compile-time error if a type is not found in a file under a name composed of the type name plus an extension (such as `.java` or `.jav`) if either of the following is true:

- The type is referred to by code in other compilation units of the package in which the type is declared.
- The type is declared `public` (and therefore is potentially accessible from code in other packages).

This restriction implies that there must be at most one such type per compilation unit. This restriction makes it easy for a compiler for the Java programming language or an implementation of the Java virtual machine to find a named class within a package; for example, the source code for a `public` type `wet.sprocket.Toad` would be found in a file `Toad.java` in the directory `wet/sprocket`, and the corresponding object code would be found in the file `Toad.class` in the same directory.

DISCUSSION

When packages are stored in a database (§7.2.2), the host system must not impose such restrictions. In practice, many programmers choose to put each class or interface type in its own compilation unit, whether or not it is public or is referred to by code in other compilation units.

A compile-time error occurs if the name of a top level type appears as the name of any other top level class or interface type declared in the same package (§7.6).

A compile-time error occurs if the name of a top level type is also declared as a type by a single-type-import declaration (§7.5.1) in the compilation unit (§7.3) containing the type declaration.

In the example:

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

the class `Point` is declared in a compilation unit with no `package` statement, and thus `Point` is its fully qualified name, whereas in the example:

```
package vista;
class Point { int x, y; }
```

the fully qualified name of the class `Point` is `vista.Point`. (The package name `vista` is suitable for local or personal use; if the package were intended to be widely distributed, it would be better to give it a unique package name (§7.7).)

In the example:

```
package test;

import java.util.Vector;

class Point {
   int x, y;
}

interface Point {                    // compile-time error #1
   int getR();
   int getTheta();
}

class Vector { Point[] pts; }// compile-time error #2
```

the first compile-time error is caused by the duplicate declaration of the name `Point` as both a `class` and an `interface` in the same package. A second error detected at compile time is the attempt to declare the name `Vector` both by a class type declaration and by a single-type-import declaration.

Note, however, that it is not an error for the name of a class to also to name a type that otherwise might be imported by a type-import-on-demand declaration (§7.5.2) in the compilation unit (§7.3) containing the class declaration. In the example:

```
package test;

import java.util.*;

class Vector { Point[] pts; }// not a compile-time error
```

the declaration of the class `Vector` *is permitted even though there is also a class* `java.util.Vector`. Within this compilation unit, the simple name `Vector` refers to the class `test.Vector`, not to `java.util.Vector` (which can still be referred to by code within the compilation unit, but only by its fully qualified name).

As another example, the compilation unit:

```
package points;

class Point {
   int x, y;                         // coordinates
   PointColor color;                 // color of this point
   Point next;                       // next point with this color
   static int nPoints;
}

class PointColor {
   Point first;                      // first point with this color
   PointColor(int color) {
      this.color = color;
   }
```

```
    private int color;          // color components
}
```

defines two classes that use each other in the declarations of their class members. Because the class types `Point` and `PointColor` have all the type declarations in package `points`, including all those in the current compilation unit, as their scope, this example compiles correctly—that is, forward reference is not a problem.

It is a compile-time error if a top level type declaration contains any one of *the following access modifiers:* `protected`, `private` or `static`.

7.7 Unique Package Names

Did I ever tell you that Mrs. McCave
Had twenty-three sons and she named them all "Dave"?
Well, she did. And that wasn't a smart thing to do. . . .
—Dr. Seuss (Theodore Geisel), *Too Many Daves* (1961)

Developers should take steps to avoid the possibility of two published packages having the same name by choosing *unique package names* for packages that are widely distributed. This allows packages to be easily and automatically installed and catalogued. This section specifies a suggested convention for generating such unique package names. Implementations of the Java platform are encouraged to provide automatic support for converting a set of packages from local and casual package names to the unique name format described here.

If unique package names are not used, then package name conflicts may arise far from the point of creation of either of the conflicting packages. This may create a situation that is difficult or impossible for the user or programmer to resolve. The class `ClassLoader` can be used to isolate packages with the same name from each other in those cases where the packages will have constrained interactions, but not in a way that is transparent to a naïve program.

You form a unique package name by first having (or belonging to an organization that has) an Internet domain name, such as `sun.com`. You then reverse this name, component by component, to obtain, in this example, `com.sun`, and use this as a prefix for your package names, using a convention developed within your organization to further administer package names.

In some cases, the internet domain name may not be a valid package name. Here are some suggested conventions for dealing with these situations:

- If the domain name contains a hyphen, or any other special character not allowed in an identifier (§3.8), convert it into an underscore.
- If any of the resulting package name components are keywords (§3.9) then append underscore to them.

If any of the resulting package name components start with a digit, or any other character that is not allowed as an initial character of an identifier, have an underscore prefixed to the component.

Such a convention might specify that certain directory name components be division, department, project, machine, or login names. Some possible examples:

```
com.sun.sunsoft.DOE
com.sun.java.jag.scrabble
com.apple.quicktime.v2
edu.cmu.cs.bovik.cheese
gov.whitehouse.socks.mousefinder
```

The first component of a unique package name is always written in all-lowercase ASCII letters and should be one of the top level domain names, currently `com`, `edu`, `gov`, `mil`, `net`, `org`, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. For more information, refer to the documents stored at `ftp://rs.internic.net/rfc`, for example, `rfc920.txt` and `rfc1032.txt`.

The name of a package is not meant to imply where the package is stored within the Internet; for example, a package named `edu.cmu.cs.bovik.cheese` is not necessarily obtainable from Internet address `cmu.edu` or from `cs.cmu.edu` or from `bovik.cs.cmu.edu`. The suggested convention for generating unique package names is merely a way to piggyback a package naming convention on top of an existing, widely known unique name registry instead of having to create a separate registry for package names.

Brown paper packages tied up with strings,
These are a few of my favorite things.
—Oscar Hammerstein II, *My Favorite Things* (1959)