Though there are similarities between C++ and Java, there are certain striking differences. In terms of mangling, the differences that matter are the following:

- Java does not support templates or similar mechanisms for generic programming
- Packages provide a more flexible visibility control mechanism
- There are no global variables
- There are no pointers; all nonprimitive types are handled by references
- A new construct called interfaces is available in Java
- Anonymous classes may be defined; these do not have any name

The scheme outlined here is based on that described in Appendix 1, with appropriate differences.

**Case 1 - Class Data Members:**

```java
class X {
    int a; => X::inta
    float a[2]; => X::float[]_a
    Y y; => X::Y_y
    static bool b; => X::bool_b
}
```

Element qualifiers such as final, static, volatile, etc. are not directly used in the mangling scheme.
Case 2 - Package:

Packages provide a scoping mechanism similar to namespaces and classes in C++. Hence the scope operator "::" is used in this case. For example, java.lang package will be mangled as java::lang. There is always a default package in Java when an explicit package name is not given. Since the language specification does not prescribe a default name, in this thesis, the string default is used. As an example, if the class X defined above does not belong to any named package, its fully qualified name will be default::X. As a simplification, this package name may be omitted when the context is clear.

Case 3 - Methods:

class Y {
    final int foo() {
        // ...
    }
}

The member function is encoded as Y::int_foo_void. Access control mechanism (private, public and protected) have no role to play in the mangling process. The final keyword is ignored since it is not part of the method signature (unlike const in C++).

Case 4 - Local Variables:

void bar(double darg)  => void_bar_double::double_darg$1
{
    int ivalue;  => void_bar_double::int_ivalue$1
    {
        float fvalue;  => void_bar_double::float_fvalue$2
        int ivalue;  => void_bar_double::int_ivalue$2
    }
    char cref;  => void_bar_double::char_cref$1
}

When mangling local variables, an extra suffix indicating the scope level (block nesting level) is used. All function arguments are at the topmost level, i.e., level
1. So are variables explicitly defined at that level. The level number increases as nesting depth increases.

**Case 5 – Variables in Static and Instance Initializers:**

A Java class allows definition of static and instance initializer blocks. The mangling strategy is shown below.

```java
class A {
    private int i;
    static int j;
    // This is the instance initializer. Any number of
    // such blocks are permitted.
    {
        int a; => A::int_a$1
        a = 0; i = 88;
        float b; => A::float_b$2
    }
    // This is the static initializer block. Any number of
    // such blocks are permitted.
    static {
        int k; => A::int_k$3
        j = 9;
        float b; => A::float_b$4
    }
    // Other elements
}
```

Variables defined in these initializers are given an incrementing number as a suffix to the otherwise normally encoded name.

**Case 7 – Local Class Members:**

```java
void foo() { => void_foo_void
    class A { => void_foo_void::A
        int value; => void_foo_void::A::int_value
        int gg() { => void_foo_void::A::int_gg_void
            // ...
        }
    }
}
This scheme is the same as the one used in C++.

**Case 8 – Inner Class Members:**
```java
class A {
char p;
    => A::char_p
class B {
    float f;
    => A::B::float_f
double hh(float f1, int i1)
    => A::B::double_hh_float_int
    { /* ... */ }
}
```

**Case 9 – Anonymous Class Members:**
```java
class A {
    X foo() {
        return new X () {
            int ii;
            => A::X_foo_void$_1::int_ii
            void ff()
            { /* ... */ }
        }; => A::X_foo_void$_1::void_ff_void
    }
}
```

A unique number is generated for each anonymous class in a method and is used as a suffix.

**Case 10 – Interface Elements:**

Interfaces are considered identical to classes as far as mangling is considered.
```java
interface A {
    int a;
    => A::int_a
    void ff(); => A::void_ff_void
}
```
Although methods are not defined in the interface (they are only declared), they are given a mangled representation for uniformity.