

## Mangling for MJI

### TracNav

- [JPFWiki - Welcome Page](#)

[Introduction...](#)

[Installing JPF...](#)

[User Guide...](#)

[Developer Guide](#)

- [Design](#)
- [Choice Generator](#)
- [Partial Order Reduction](#)
- [Attributes](#)
- [Listener](#)

### MJI

- [Mangling for MJI](#)
- [Bytecode Factory](#)
- [Logging](#)
- [Report](#)
- [Embedded](#)
- [JPF tests](#)
- [JPF project layout](#)
- [Create a JPF project](#)
- [Coding Conventions](#)
- [Hosting update site](#)

[Projects...](#)

- [Summer Projects](#)
- [External Projects](#)
- [Change\(B\)log](#)

### About...

- [Events](#)
- [Presentations](#)
- [Papers](#)
- [FAQ](#)
- [History?](#)
- [Support](#)
- [People?](#)
- [Playground](#)
- [Table of Context](#)

## Mangling Methods

Suppose your method looks like

T1 foo(T2, T3, ...)

where Ti are Java types.

If `T1` is a primitive type or "void", then the mangled MJI method looks like

```
public static T1 foo__MT2MT3..._MT1(...)
```

where the `MTi` are the mangled versions of the `Ti`. Mangling is described in [Mangling Types below](#). Note that `T1` appears twice, once not mangled and once mangled. The "`_`" is two consecutive underscores: "`_`" followed by "`_`".

As a not-so-special case, if `foo` has no arguments, then the mangled method will have four consecutive underscores:

```
T1 foo()
```

goes to

```
public static T1 foo____MT1(...)
```

If `T1` is not a primitive type, then the mangled MJI method looks like

```
public static int foo__MT2MT3..._MT1
```

where the `MTi` are as above. Note that `T1` only appears once in this case. The method's return type is `int`. As before, a method with no arguments gets mangled to something with four consecutive underscores.

Also, the use of generics is ignored when mangling names.

## Mangling Constructors

Constructors are treated as methods named `$init` with return type `void`.

## Mangling Types

- Convert primitives and "void" as follows

Java Type	Mangled Type
<code>boolean</code>	<code>Z</code>
<code>byte</code>	<code>B</code>
<code>char</code>	<code>C</code>
<code>short</code>	<code>S</code>
<code>int</code>	<code>I</code>
<code>long</code>	<code>J</code>
<code>float</code>	<code>F</code>
<code>double</code>	<code>D</code>
<code>void</code>	<code>V</code>

Convert a non-array reference type `T` in package `x.y` (e.g. `java.lang.String`) as follows

- `x.y.T --> Lx_y_T_2`
- Example: `java.lang.String -->Ljava_lang_String_2`

Convert an array of primitive type `T` (e.g. `byte[]`) as follows:

- `T[] --> _3MT` where `MT` is the mangled version of `T` (e.g. for `T=byte`, `MT=B`)
- Example: `byte[] --> _3B`

Convert an array of reference type `T` in package `x.y` (e.g. `java.lang.String[]`) as follows:

- `x.y.T[] --> _3Lx_y_T_2`
- Example: `java.lang.String[] --> _3Ljava_lang_String_2`

## Method Examples

### Void return type, single primitive argument

```

public static void resetCounter(int id)
-->
public static final void resetCounter__I__V(MJIEnv env, int objref, int id)

```

#### Primitive return type, no arguments

```

public native boolean isArray()
-->
public static boolean isArray____Z(MJIEnv env, int objref)

```

#### Primitive return type, single primitive argument

```

public static double abs(double a)
-->
public static double abs__D__D(MJIEnv env, int clsObjRef, double a)

```

#### Primitive return type, two primitive arguments

```

public static long min(long a, long b)
-->
public static long min__JJ__J(MJIEnv env, int clsObjRef, long a, long b)

```

#### void return type, arguments include an array of a primitive type

```

public native void write (byte[] buf, int off, int len);
-->
public static void write____3BII__V(MJIEnv env, int objref,
int bufRef, int off, int len)

```

#### void return type, argument is an array of a reference type

```

public static void print(String s)
-->
public static void print____3Ljava_lang_String_2__V(MJIEnv env, int clsRef, int argsRef)

```

#### Array of reference types returned, no arguments

```

public native Annotation[] getAnnotations()
-->
public static int getAnnotations____3Ljava_lang_annotation_Annotation_2(MJIEnv env, int robj)

```

Notice there are 5 underscores before the "3L": two marking the arguments, two marking the return type, and one from the "\_3" signalling an array.

#### Array of reference types using generics returned, no arguments

```

public native Class<?>[] getParameterTypes()
-->
public static int getParameterTypes____3Ljava_lang_Class_2(MJIEnv env, int objref)

```

Note: the use of generics is ignored in the mangling.

### Constructor Examples

Constructors are treated as though they were methods named \$init returning void, so the method examples above should be helpful. Here are a few more examples.

In the class ConsoleOutputStream:

```
public ConsoleOutputStream()  
-->  
public static void $init___V(MJIEnv env, int objref)
```

In the class AtomicLongFieldUpdater:

```
protected AtomicLongFieldUpdater(Class<T> objClass, String fieldName)  
-->  
public static void $init__Ljava_lang_Class_2Ljava_lang_String_2__V  
    (MJIEnv env, int objRef,  
     int tClsObjRef, int fNameRef)
```