

بسمه تعالی

جزوه ضمیمه درس

# اصول طراحی کامپیالرها

« دکتر سعید پارسا »

# JASMIN

تهیه و تنظیم:

محمد رضا ذاکری نسب

مرضیه بیگ محمدی

پاییز ۱۳۸۰

## آغاز سخن :

خوشحالیم که توانستیم شاید یکی از تعداد قلیل دانشگاه‌های معتبر در سطح دنیا باشیم که توانسته‌ایم در درس کامپایلرها علم کامپایلرنویسی برای سطح وب را به صورت عملی ارائه نماییم. کامپایلرها برای برنامه‌نویسی وب گذری از برنامه‌نویسی پیچیده و غیرمانوس زبانهای اسکریپتی به سمت زبانهای سطح بالا می‌باشد. به خصوص مشاهده می‌کنیم که حتی Microsoft نیز متوجه این نکته شده و ASP را تبدیل به زبان سطح بالای ASP++ نموده است. پس از ارائه تکنیک‌های کامپایلرنویسی به صورت عملی برای زبان HTML که توسط آقای امینایی ارائه شده و کامپایلر برای زبان XML که توسط آقای باقریان ارائه شده است، کامپایلری برای بایت‌کد و به صورت Applet های Java در این بخش ارائه شده است. امیدواریم که گامی به سوی تولید محصولات ملی و آغازی نوین برای صدور نرم‌افزار بر خلاف تلاش آنان که در راه نابودی علم کامپیوتر تلاش می‌کنند، باشیم.

**دکتر سعید پارسا**

**دانشکده کامپیوتر دانشگاه علم و صنعت ایران**

**زمستان ۱۳۸۰**

# JVM

## مقدمه :

یکی از اجزاء اصلی Java، Java Virtual Machine، یا JVM است. JVM یک کامپیوتر مجازی است که معمولاً به صورت نرم افزاری بر روی یک سیستم سخت افزاری و سیستم عامل آن پیاده سازی می شود و برنامه های ترجمه شده Java را اجرا می کند. با استفاده از JVM، برنامه هایی که به زبان Java نوشته می شوند، بدون نیاز به هیچگونه تغییر بر روی ماشینها و سیستم عاملهای مختلف اجرا می شوند. این مساله می تواند انگیزه ای قوی برای نوشتن کامپایلرهای باشد که کد خروجی آنها نیز با استفاده از JVM قابل اجرا باشد. این موضوع در دانشکده کامپیوتر دانشگاه علم و صنعت ایران توسط « دکتر سعید پارسا » استاد درس اصول طراحی کامپایلرها در مقطع کارشناسی و کامپایلرهای پیشرفته در مقطع کارشناسی ارشد مورد توجه قرار گرفته است. تقریباً از سال ۱۳۷۶ با توجه به رویکرد درس کامپایلرها و استاد این درس به این سمت، دانشجویان این دانشکده پروژه های مختلفی در این زمینه انجام داده اند که به عنوان نمونه می توان به [پروژه Jacomizer](#) و [راهنمای ایجاد Applet های جاوا](#) اشاره کرد. در این مستندات ما تلاش کرده ایم تا مبانی علمی لازم برای این رویکرد را فراهم کنیم. امید است که راهگشای دانش پژوهان ایران عزیزمان باشد.

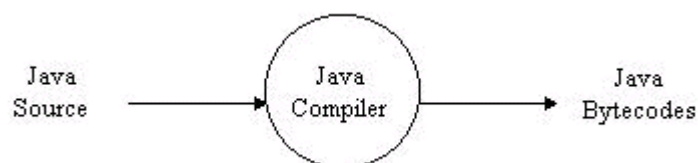
## Java Virtual Machine چیست ؟

JVM یک کامپیوتر مجازی است که برنامه های ترجمه شده Java را اجرا می کند. تمام برنامه های Java برای اجرا توسط JVM ترجمه می شوند، بنابراین برای اجرای این برنامه ها حتما باید JVM روی سکوی کاری پیاده سازی شده باشد. حجم کم JVM و قابلیت سوار شدن روی هر سیستم سخت افزاری و حتی سیستم های کوچکی مانند تلفن های سلولی ، به JVM در تحقق شعار Ubiquity ( حضور در یک زمان در همه جا ) کمک می کند.

برنامه های ترجمه شده Java
ماشین مجازی Java یا Java Virtual Machine
سکوی سخت افزاری و سیستم عامل

## Java bytecode چیست ؟

برنامه های Java به یک قالب خاص که Java bytecode نامیده می شود، ترجمه می شوند. JVM ، bytecode ها را اجرا می کند، بنابراین می توان bytecode ها را به عنوان زبان ماشین JVM در نظر گرفت. کامپایلر Java ، فایل های source با پسوند java. را می خواند و آنها را به صورت bytecode درآورده در یک فایل با پسوند class. ذخیره می کند. در این پروسه به ازاء هر کلاسی که در source وجود دارد، یک فایل class. ایجاد می شود.



از نظر JVM هر رشته ای از bytecode ، یک ترتیب از دستورالعملهای پشت سرهم می باشد. هر دستورالعمل شامل یک بایت کد عمل ( opcode ) و تعدادی ( صفر یا بیشتر) عملوند است. opcode به JVM می گوید که چه عملی باید انجام شود. اگر این کد عمل دارای عملوند باشد، عملوندهای آن بلافاصله بعد از کد عمل می آیند. هر کد عمل با یک دستور شبه اسمبلی نیز مشخص می شود. مثلاً دستوری وجود دارد که به JVM می گوید یک صفر در پشته بگذارد. کد این دستور 0x60 و دستور شبه اسمبلی آن `iconst_0` است. این دستور هیچ عملوندی ندارد. به عنوان یک مثال دیگر، دستوری وجود دارد که به JVM می گوید از مکان حافظه کنونی به کدام مکان حافظه برود. این دستور یک عملوند دارد که به صورت یک offset ۱۶ بیتی علامتدار بعد از opcode می آید. کد این دستور 0xA7 و معادل شبه اسمبلی آن `goto` است.

60	<code>iconst_0 (opcode)</code>	a7	<code>goto (opcode)</code>
	(no operands)	ff	(one two-byte operand, fff9, which defines an offset to jump to)
		f9	

## سخت افزار مجازی JVM

سخت افزار مجازی یک JVM را می توان به ۴ بخش تقسیم کرد: رجیسترها، پشته، garbage-collected heap ، و method area . این قسمتها مانند ماشینی که تشکیل می دهند، انتزاعی هستند، ولی حتما باید در هر JVM به گونه ای پیاده سازی شده باشند.

آدرس‌ها در JVM به صورت ۳۲ بیتی بیان می‌شوند، بنابراین JVM می‌تواند تا ۴ گیگابایت فضای حافظه را آدرس‌دهی کند. هر رجیستر یک آدرس ۳۲ بیتی را نگهداری می‌کند. پشته، garbage-collected heap، و method area هرکدام جایی درون این فضای حافظه قابل آدرس‌دهی قرار دارند. مکان دقیق این فضاها به تصمیم کسی که JVM را پیاده‌سازی می‌کند بستگی دارد.

هر کلمه در JVM ۳۲ بیت است. JVM انواع داده اولیه کمی دارد که عبارتند از: byte ( ۸ بیت ) ، short ( ۱۶ بیت ) ، int ( ۳۲ بیت ) ، long ( ۶۴ بیت ) ، float ( ۳۲ بیت ) ، double ( ۶۴ بیت ) ، و char ( ۱۶ بیت ) که به جز نوع داده char ( که یک کاراکتر بدون علامت Unicode است ) همه علامتدار هستند. این انواع به راحتی با نوع داده های زبان Java قابل تطبیق هستند. یک نوع داده ابتدایی دیگر، object handle است که به صورت یک آدرس ۳۲ بیتی به یک object روی heap اشاره می‌کند.

چون method area محدوده‌ای است که bytecode ها را دربرمی‌گیرد، به صورت بایستی سازماندهی شده است. اما پشته و garbage-collected heap از یک ساختار ۳۲ بیتی استفاده می‌کنند.

چون JVM یک سیستم stack-based است ( یعنی تمام عملیات مورد نیاز در حافظه برای bytecode ها را با یک پشته انجام می‌دهد ) نیازی به رجیسترهایی برای نگهداری مقادیر میانی ندارد. بنابراین در طراحی JVM فقط یک شمارنده برنامه ( program counter ) و سه رجیستر برای مدیریت پشته پیش‌بینی شده است که این مساله ( کم بودن تعداد رجیسترها ) موجب کوچک شدن مجموعه دستورات JVM می‌شود.

JVM از رجیستر pc برای نگهداشتن آدرس حافظه ای که باید دستورات درون آن اجرا شوند استفاده می‌کند. سه رجیستر دیگر که optop ، frame ، و vars نام دارند به قسمت‌های مختلف stack frame متد در حال اجرا

( قسمتی از پشته که به متد در حال اجرا اختصاص دارد ) اشاره می‌کنند. stack frame هر متد وضعیت آن متد ( مقادیر متغیرهای محلی، نتایج میانی، محاسبات و غیره) را برای هر فراخوانی به صورت جداگانه نگهداری می‌کند (یعنی هر فراخوانی stack frame مخصوص به خود را دارد).

method area و program counter :

method area جایی است که bytecode ها قرار دارند. program counter همواره به آدرسی در حافظه اشاره می‌کند که باید دستورات آن آدرس اجرا شوند. پس از اجرای هر دستور bytecode ، اگر این دستور باعث یک پرش ( jump ) نشود، JVM مقدار program counter را برابر آدرس دستوری قرار می‌دهد که بلافاصله پس از دستور اجرا شده قرار دارد.

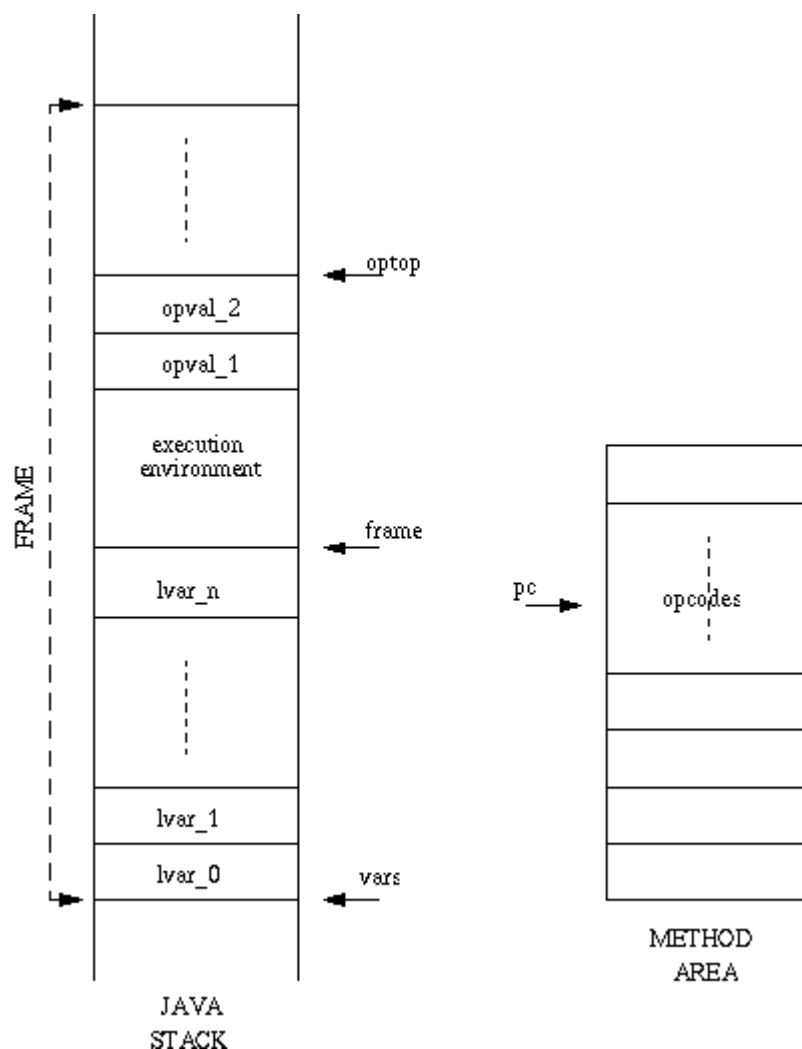
## پشته و رجیسترهای مربوط به آن

در Java از پشته برای نگهداری پارامترهای مورد نیاز و نتایج حاصل از دستورات bytecode ، ارسال پارامتر به method ها و برگرداندن مقادیر بازگشتی ، و نگهداری وضعیت هر متد در حال اجرا استفاده می‌شود. رجیسترهای optop ، frame ، و vars هم به قسمت‌های مختلف stack frame جاری اشاره می‌کنند.

در هر stack frame سه بخش مجزا وجود دارد: متغیرهای محلی ( local variables )، محیط اجرا ( execution environment )، و پشته عملوندها ( operand stack ). بخش متغیرهای محلی، تمام متغیرهای محلی فراخوانی جاری متد را نگهداری می‌کند. آدرس این بخش در رجیستر vars نگهداری می‌شود. بخش execution environment به نگهداری عملیات خود stack اختصاص دارد و آدرس آن در رجیستر frame نگهداری می‌شود. پشته

عملوندها نیز به عنوان فضای حافظه‌ای که دستورات bytecode از آن برای اجرا استفاده می‌کنند عمل می‌کند. دستورات bytecode عملوندهای خود را از این پشته برمی‌دارند و نتایج را نیز در این پشته می‌گذارند. رجیستر optop به بالای پشته عملوندها اشاره می‌کند.

قسمت execution environment همواره بین متغیرهای محلی و پشته عملوندها قرار دارد. پشته عملوندهای متد در حال اجرا همواره بالاترین قسمت پشته را تشکیل می‌دهد، بنابراین رجیستر optop همیشه به بالای تمام پشته Java اشاره می‌کند.



ساختار پشته در Java



## Garbage-collected heap

heap مکانی از حافظه است که تمام object های یک برنامه Java در آن زندگی می کنند. هر زمان که شما با استفاده از دستور new به یک object حافظه اختصاص می دهید، این حافظه از heap می آید. در زبان Java شما نمی توانید حافظه گرفته شده را مستقیماً آزاد کنید. به جای این، در زمان اجرا تعداد ارجاع هایی که به یک object روی حافظه وجود دارد کنترل می شود و هر زمان ارجاع دیگری به یک object وجود نداشت، فضای حافظه اختصاص یافته به آن آزاد می شود. به این پروسه garbage collection می گویند.

# JASMIN

## Jasmin چیست ؟

به زبان ساده، Jasmin یک Java Assembler است که توسط آقای Jon Meyer نوشته شده است. Jasmin از دستورات ساده شبه اسمبلی JVM به عنوان مجموعه دستورات زبان استفاده می کند و با توجه به syntax ساده ای که دارد، برای هدف ما ( ایجاد کامپایلری که بتواند کد اجرایی ایجاد کند ) بسیار مناسب است. با توجه به اینکه Jasmin بر اساس Java کار می کند و Java نیز به جای رجیسترهای معمول از یک پشته استفاده می کند، دیگر نیازی به عملیات Register Management نیست که این هم یکی از مزایای استفاده از کد Jasmin به عنوان خروجی کامپایلر می باشد.

در این روش به جای تولید کد اسمبلی و سپس اجرای آن، ابتدا برنامه source.txt به یک فایل Jasmin (مثلا target.jas) ترجمه می‌شود و سپس این فایل توسط کامپایلر Jasmin به یک فایل اجرایی Java (مثلا target.class) تبدیل می‌شود.

البته ایجاد فایل اجرایی Java (یک class file) به صورت مستقیم و با استفاده از bytecode ها نیز امکان‌پذیر است، اما همانطور که در قسمت‌های گذشته دیدیم، چون bytecode ها در اصل مقادیری بایستی هستند، این روش بسیار دشوار خواهد بود. در مقام مقایسه (گرچه این مقایسه خیلی صحیح نیست) می‌توانید تولید مستقیم فایل class را مانند تولید مستقیم یک فایل exe. فرض کنید که مسلماً این کار بسیار دشوارتر از نوشتن یک برنامه به زبان اسمبلی است. علاوه بر این مساله، مزیت بزرگ دیگر Jasmin این است که برای نوشتن کد Jasmin نیازی به دانستن دقیق مفاهیمی مانند constant pool و یا stack frame نیست، گرچه آشنایی با این مفاهیم در درک چگونگی کار زبان بسیار مفید خواهد بود.

ما در این قسمت ابتدا به بیان دستورات زبان Jasmin می‌پردازیم و سپس چند برنامه نمونه را مورد بررسی قرار می‌دهیم.

## دستورات زبان Jasmin:

دستورات زبان Jasmin به دو دسته پارامتردار و بدون پارامتر تقسیم می‌شوند که ما ابتدا دستورات پارامتردار و سپس دستورات بدون پارامتر را بررسی می‌کنیم. (تقسیم‌بندی زیر از Jasmin Instruction Syntax تهیه شده توسط Jon Meyer اقتباس شده است.)

## ۱- دستورات متغیرهای محلی

همانطور که قبلاً هم گفته شد، فضای متغیرهای محلی در بالاترین قسمت پشته قرار دارد، بنابراین توابعی که برای دسترسی به متغیرهای محلی پیش‌بینی شده‌اند در اصل به فضای بالای پشته دسترسی دارند. این توابع با استفاده از یک شماره ( که در اصل شماره مکان حافظه نگهدارنده مقدار متغیر است ) از این فضا استفاده می‌کنند. قالب کلی این دستورات به صورت زیر است :

```
directive <var-num> ;  
for example : fstore 3;  
               lstore 5;
```

این دستورات در جدول زیر آورده شده‌اند:

ret	از زیرروال خارج شده، مقدار ذخیره شده در آدرس <var-num> ( که از نوع returnAddress می‌باشد ) را در رجیستر pc ذخیره می‌کند.
aload	یک متغیر محلی از نوع refrence را درون پشته push می‌کند.
astore	عنصر بالای پشته را در یک متغیر محلی از نوع refrence ذخیره می‌کند.
dload	یک متغیر محلی double را درون پشته push می‌کند.
dstore	عنصر بالای پشته را در یک متغیر محلی double ذخیره می‌کند.
float	یک متغیر محلی float را درون پشته push می‌کند.
fstore	عنصر بالای پشته را در یک متغیر محلی float ذخیره می‌کند.

iload	یک متغیر محلی int را درون پشته push می کند.
istore	عنصر بالای پشته را در یک متغیر محلی int ذخیره می کند.
lload	یک متغیر محلی long را درون پشته push می کند.
lstore	عنصر بالای پشته را در یک متغیر محلی long ذخیره می کند.

### دستورات متغیرهای محلی

به عنوان مثال، به این دستورات توجه کنید :

aload 1 : مقدار متغیر شماره ۱ را درون پشته push می کند. ( در اصل، مقدار ذخیره شده در خانه شماره ۱ بخش متغیرهای محلی stack frame جاری را درون پشته push می کند. )

ret 2 : از زیرروال خارج شده، آدرس ذخیره شده در متغیر شماره ۲ را در رجیستر pc ذخیره می کند.

### ۲- دستورات bipush ، sipush ، و iinc :

دستورات bipush و sipush یک عدد integer را به عنوان پارامتر دریافت می کنند و آن را درون پشته push می کنند. شکل کلی این دستورات به صورت زیر است :

bipush <int>  
sipush <int>

البته bipush یک عدد یک بایتی را push می کند ولی sipush برای دو byte این کار را انجام می دهد.

دستور `iinc` از دو عملوند استفاده می‌کند. عملوند اول شماره متغیر را مشخص می‌کند و عملوند دوم مقداری را که از آن متغیر باید کم شود یا به آن اضافه شود. شکل کلی این دستور به صورت زیر است :

`iinc <var-num> <amount>`

به عنوان مثال، به دستورات زیر توجه کنید :

`bipush 100` : عدد ۱۰۰ را درون پشته `push` می‌کند.  
`iinc 5-12` : از مقدار متغیر شماره ۵، عدد ۱۲ را کم می‌کند.

### ۳- دستورات پرش :

این دستورات از یک `label` به عنوان عملوند استفاده می‌کنند. قالب کلی این دستورات به شکل زیر است :

`<directive> <label>`  
 for example: `Label1:`  
                   `goto Label1` ; an infinite loop !

این دستورات عبارتند از :

یک پرش بدون شرط انجام می‌دهد. در این دستور <code>branch offset</code> پرش ۱۶ بیتی است. ( آدرس پرش: <code>pc + branch offset</code> )	<code>goto</code>
یک پرش بدون شرط انجام می‌دهد، ولی برای پیدا کردن <code>offset</code> مورد نیاز برای پرش از یک <code>branch offset</code> ۳۲ بیتی استفاده می‌کند. چون <code>Jasmin</code> خود در این مورد تصمیم می‌گیرد، در <code>Jasmin</code> این دستور با <code>goto</code> معادل است.	<code>goto_w</code>

if_acmpeq	اگر دو مقدار بالای پشته ( از نوع object reference ) مساوی باشند ( یعنی هردو به یک شیء اشاره کنند)، مقایسه موفقیت آمیز بوده و پرش انجام می شود.
if_acmpne	اگر دو مقدار بالای پشته ( از نوع object reference ) مساوی نباشند ( یعنی هردو به یک شیء اشاره نکنند)، مقایسه موفقیت آمیز بوده و پرش انجام می شود.
if_icmpeq	اگر دو عدد int بالای پشته مساوی باشند، مقایسه موفقیت آمیز بوده و پرش انجام می شود.
if_icmpge	اگر بین دو عدد int بالای پشته رابطه $value2 \geq value1$ برقرار باشد، مقایسه موفقیت آمیز بوده و پرش انجام می شود.
if_icmpgt	اگر بین دو عدد int بالای پشته رابطه $value2 > value1$ برقرار باشد، مقایسه موفقیت آمیز بوده و پرش انجام می شود.
if_icmple	اگر بین دو عدد int بالای پشته رابطه $value2 \leq value1$ برقرار باشد، مقایسه موفقیت آمیز بوده و پرش انجام می شود.
if_icmplt	اگر بین دو عدد int بالای پشته رابطه $value2 < value1$ برقرار باشد، مقایسه موفقیت آمیز بوده و پرش انجام می شود.
if_icmpne	اگر دو عدد int بالای پشته مساوی نباشند، مقایسه موفقیت آمیز بوده و پرش انجام می شود.
Ifeq	اگر عدد int بالای پشته صفر باشد، مقایسه موفقیت آمیز بوده و پرش انجام می شود.

Ifge	اگر عدد int بالای پشته بزرگتر یا مساوی صفر باشد، مقایسه موفقیت‌آمیز بوده و پرش انجام می‌شود.
ifgt	اگر عدد int بالای پشته بزرگتر از صفر باشد، مقایسه موفقیت‌آمیز بوده و پرش انجام می‌شود.
ifle	اگر عدد int بالای پشته کوچکتر یا مساوی صفر باشد، مقایسه موفقیت‌آمیز بوده و پرش انجام می‌شود.
iflt	اگر عدد int بالای پشته کوچکتر از صفر باشد، مقایسه موفقیت‌آمیز بوده و پرش انجام می‌شود.
ifne	اگر عدد int بالای پشته صفر نباشد، مقایسه موفقیت‌آمیز بوده و پرش انجام می‌شود.
ifnonnull	اگر object refrence بالای پشته، null نباشد، مقایسه موفقیت‌آمیز بوده و پرش انجام می‌شود.
ifnull	اگر object refrence بالای پشته، null باشد، مقایسه موفقیت‌آمیز بوده و پرش انجام می‌شود.
jsr	یک پرش به زیرروال محلی تعریف شده در بدنه یک متد انجام می‌دهد. ( مثال * )
jsr_w	یک پرش به زیرروال محلی تعریف شده در بدنه یک متد انجام می‌دهد، ولی برای پیدا کردن offset مورد نیاز برای پرش از یک branch offset ۳۲ بیتی استفاده می‌کند. چون Jasmin خود در این مورد تصمیم می‌گیرد، در Jasmin این دستور با jsr معادل است.

مثال \* ( دستور jsr و jsr\_w ) :

( این مثال از کتاب Java Virtual Machine نوشته Jon Meyer و Troy

Downing اقتباس شده است.)

```
; This example method uses a PrintMe subroutine to invoke the
; System.out.println() method.
```

```
.method usingSubroutine()V
```

```
    jsr  PrintMe      1
```

```
    jsr  PrintMe      2
```

```
    jsr  PrintMe      3
```

```
    return           ; now return from usingSubroutine
```

```
; define the PrintMe subroutine . . .
```

```
PrintMe:
```

```
    astore_1         ;store return-address in local variable 1
```

```
    ; call System.out.println()
    getstatic java/lang/System/out Ljava/io/PrintStream;
    invokevirtual
        java/io/PrintStream/println(Ljava/lang/String;)V
```

```
    ret 1            ; return to the return-address in local
                    ; variable 1
```

```
.end method
```

در جدول بالا دقت کنید که تمام عناصری که باید مقایسه شوند، ابتدا از پشته pop می‌شوند، بنابراین بعد از مقایسه دسترسی به آنها ممکن نیست. بنابراین اگر به هر دلیلی مقادیر این عناصر هنوز مورد نیاز است، باید قبل از مقایسه آنها را در متغیرهای محلی ذخیره کرد و یا با استفاده از دستور dup آنها را دوباره در پشته push کرد.

## ۴- دستورات مربوط به کلاسها و اشیاء



شکل کلی این دستورات به صورت زیر است:

<directive> <type>

for example: new java/lang/String ; create a new String object

در این دستورات ( به غیر از new )، <type> ( نوع داده شده ) می تواند نام یک کلاس، interface و یا یک descriptor نوع آرایه ( مانند [[Ljava/lang/String; ) باشد. در دستور new ، <type> فقط می تواند نام یک کلاس باشد. این دستورات عبارتند از:

<p>یک آرایه برای نگهداری object reference هایی از نوع داده شده ایجاد می کند. اندازه این آرایه از بالای پشته pop می شود و پس از ایجاد شدن، یک reference به آن در پشته push می شود. تمامی اعضای این آرایه در ابتدا null هستند.</p>	anewarray
<p>چک میکند که آیا object reference بالای پشته قابل تبدیل به نوع داده شده می باشد یا خیر؟ اگر پاسخ مثبت باشد و یا مقدار object reference برابر null باشد، اجرا ادامه پیدا می کند، وگرنه پیغام خطای ClassCastException صادر می شود. در این دستور، پشته تغییری نمی کند.</p>	checkcast
<p>ابتدا object reference از پشته pop می شود. اگر &lt;type&gt; یک کلاس باشد، و object reference یک instance از آن کلاس و یا زیرکلاس های آن باشد، عدد ۱ ( int ) و اگر نه عدد صفر در پشته push می شود. اگر &lt;type&gt; یک interface باشد، این تست درباره پیاده سازی شدن یا نشدن &lt;type&gt; در object reference صورت می پذیرد.</p>	instanceof
<p>یک شیء جدید از کلاس داده شده می سازد و object</p>	new

<p>reference آن را در پشته push می‌کند. تمامی field های int و boolean این شیء در ابتدا مساوی صفر قرار می‌گیرند، اما هنوز شیء uninitialized است و برای استفاده باید یکی از متدهای init آن را با دستور invokespecial فراخوانی کرد.</p>	
--	--

برای درک بهتر این دستورات، به مثال زیر توجه کنید:

```

;;;;;;;;;   anewarray

; allocate a 10-element array for holding strings.
bipush 10
anewarray [Ljava/lang/String;
; store the new array in local variable 1
astore_1

;;;;;;;;;   checkcast

aload_1    ; push object in local variable 1 onto stack
checkcast [Ljava/lang/String;
; because the object on the stack is an array of
; Strings, the execution reaches here. ( it could
; be null too. )

;;;;;;;;;   new

; create a new StringBuffer object.
new java/lang/StringBuffer

; because invokespecial removes the object reference,
; we must make an extra copy of it.
dup    ; duplicates top of stack

; now we must initialize the object.
; we call the default init method.
invokespecial java/lang/StringBuffer/<init>()V

; assign object reference on stack to a local variable.
astore_1
;;;;;;;;;   instanceof

; using instanceof to test for a String
; push object reference in local variable 1 onto stack
aload_1

; test if item on stack is a string

```

```

instanceof java/Lang/String

; if so, goto HaveString
ifne HaveString

; otherwise, return
return

HaveString:
; if this point is reached, local variable 1 holds a string.

```

## ۵- دستورات فراخوانی متدها :

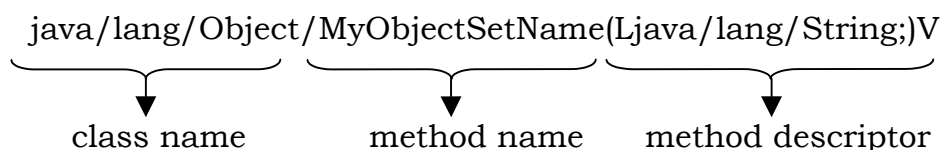
در Jasmin چهار دستور برای فراخوانی متدها پیش‌بینی شده‌اند که عبارتند از:

invokestatic	<method-specification>
invokespecial	<method-specification>
invokevirtual	<method-specification>
invokeinterface	<method-specification> <num-args>

invokespecial گسترش‌یافته دستور invokevirtual ( که در JDK 1.0.2 و پایین‌تر موجود است ) می‌باشد.

هر method specification از سه بخش تشکیل شده است : تمام کاراکترهای قبل از آخرین '/' نام کلاس دارای متد را تشکیل می‌دهند ( در مثال بالا نام کلاس java.io.PrintStream می‌باشد). کاراکترهای بین آخرین '/' و '(' نام متد را تشکیل می‌دهند و بقیه رشته هم method descriptor است. ( برای مطالعه بیشتر درباره descriptor ها به ضمیمه ۱ مراجعه فرمایید )

به مثال زیر توجه کنید :



از دستور `invokestatic` برای فراخوانی متدهای `static` (یا همان متدهای معمولی کلاس) استفاده می‌شود. دستور `invokespecial` هم برای موارد خاصی از فراخوانی‌ها مورد استفاده قرار می‌گیرد که عبارتند از: فراخوانی متد مقداردهی اولیه یک `instance` (`<init>`)، فراخوانی یک متد `private` از شیء `this`، و یا فراخوانی یک متد از `superclass` شیء `this`. البته در بیشتر موارد استفاده `invokespecial` در همان مورد اول خلاصه می‌شود. (به مثال دستور `new` مراجعه کنید). `invokeinterface` هم مخصوص فراخوانی متدهای `interface` ها می‌باشد که بعد از یک مثال، بررسی خواهد شد. در سایر موارد هم از دستور `invokevirtual` استفاده می‌شود. در تمامی این دستورات یک `object reference` و پارامترهای متد از پشته `pop` می‌شود و پس از انجام فراخوانی، نتیجه در پشته `push` می‌شود. به ساختار پشته قبل و بعد از اجرای این دستورات توجه کنید:

before:                  [bottom] ... objectref arg1 arg2 ... arg n  
after:                   [bottom] ... [result]

مثال‌های زیر شیوه استفاده از این دستورها را بهتر نشان می‌دهند:

```
;;;;;;;;; invokestatic  
  
; calls the exit(1) method from System class  
; push 1 onto stack.  
iconst_1  
; now call System.exit(1)  
invokestatic java/lang/System/exit(I)V
```

```

;;;;;;;;;; invokespecial

; create a new StringBuffer object.
new java/lang/StringBuffer

; because invokespecial removes the object reference,
; we must make an extra copy of it.
dup    ; duplicates top of stack

; now we must initialize the object.
invokespecial java/lang/StringBuffer/<init>()V
; now stack contains an initialized StringBuffer.

;;;;;;;;;; invokevirtual

; push local variable 1 (i.e., 'x') onto stack
; x is defined before as a Java object.
aload_1;
; push the string "Jasmin" onto stack,
; which will be used as parameter.
ldc "Jasmin"
; invoke some method, for example equals here.
Invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
; now the boolean result is on the stack.

```

در دستور `invokeinterface`، عملوند دوم تعداد پارامترهای متد فراخوانی شده را بیان می‌کند. به عنوان مثال به فراخوانی زیر توجه کنید :

```

;;;;;;;;;; invokeinterface

; push local variable 1 (i.e., the enum object) onto the
; stack
aload_1

; call hasMoreElements, which has one parameter( the enum
; object ) and returns a boolean value ( which will be pushed
; onto stack ).
invokeinterface  java/util/Enumeration/hasMoreElements()Z 1

; stores the result in a local variable.
istore_2;

```

۶- دستورات کار با field ها :

در Jasmin چهار دستور برای کار با field ها وجود دارد که این دستورات به فرم زیر هستند:

directive      <field-specification> <descriptor>

یک object reference را از بالای پشته pop کرده ، مقدار field مشخص شده را در شیء بازیابی می کند و در پشته push می کند.	Getfield
عنصر static خواسته شده را از کلاس داده شده بر می گرداند. در این دستور، <descriptor> نوع داده عضو استاتیک را مشخص می کند.	Getstatic
یک field را در شیء مقدار می دهد.	Putfield
یک static field را در کلاس مقدار می دهد.	Putstatic

به مثال زیر توجه کنید:

```

;;;;;;;;; getfield

; assume that p is an object from class xyz.point and
; has an integer field named xCoord.
; now we want to retrieve it's xCoord value.

; push object in local variable 1 (i.e., p) onto the stack.
aload_1
; get the value of p.xCoord which is an int.
getfield xyz/point/xCoord I
; store the int value in local variable 2

;;;;;;;;; getstatic

; assume that we want to have an object from type
; System.out
getstatic java/lang/System/out Ljava/io/PrintStream;
; store the object reference result in local variable 1
astore_1
; now each time you use the object stored in local variable 1
; it's equal to using System.out

```

```

;;;;;;;;;; putfield

; assume that we want to set the value of p.xCoord,
; such as p.xCoord = 10.
; push object in local variable 1 (i.e., p) onto the stack
aload_1
; push the integer 10 onto the stack
bipush 10
; set the value of the integer field p.xCoord to 10
putfield xyz/point/xCoord I

;;;;;;;;;; putstatic

; assume that we want to replace System.out with our method
; that is stored in local variable 1.
; push object reference in local variable 1 (i.e., MyMethod)
; onto the stack.
aload_1
; now use putstatic to assign it to System.out
putstatic java/lang/System/out Ljava/lang/PrintStream;

```

در این دستورات، هر <field-specification> از دو قسمت تشکیل شده است: یک نام کلاس و یک نام field. تمام کاراکترهای <field-specification> تا قبل از آخرین '/' نام کلاس و بقیه کاراکترهای بعد از آخرین '/' هم نام field را تشکیل می‌دهند. به عنوان نمونه به مثال زیر توجه کنید:

```

; get java.lang.System.out, which is a PrintStream
getstatic java/lang/System/out Ljava/io/PrintStream;
           {           }
           class name field name

```

۷- دستور newarray :

این دستور به فرم کلی `<array-type> newarray` می‌باشد و عملوند `<array-type>` نوع داده را مشخص می‌کند. به مثال زیر توجه کنید:

```
; push the size of array onto the stack
bipush 10
; make an array of 10 integers and push the arrayref onto
; the stack
newarray int
```

## ۸- دستور `multianewarray`:

این دستور برای ایجاد آرایه‌های چند بعدی پیش‌بینی شده‌است و دو عملوند دارد. عملوند اول نوع داده آرایه و عملوند دوم چند بعدی بودن آرایه را مشخص می‌کند. فرم کلی این دستور به صورت زیر است:

`multianewarray <array-descriptor> <num-dimensions>`

به مثال زیر توجه کنید:

```
; to allocate an array like:
;   new String[2][5]
; push array dimensions
bipush 2
bipush 5
; make the array
multianewarray [[Ljava/lang/String; 2
; stack now hold a reference to the new two dimensional array
```

## ۹- دستورات `ldc` و `ldc_w`:

این دستورات دارای یک عملوند `constant` می‌باشند که این عملوند می‌تواند یک عدد صحیح، یک عدد با ممیز شناور و یا یک `string` باشد. فرم کلی این دستورات به شکل زیر است:

```
ldc <constant>
ldc_w <constant>
```



این دستورات مقدار constant داده‌شده را درون پشته push می‌کنند.  
به مثال‌های زیر توجه کنید:

```
ldc 1.2          ; push a float
ldc 10           ; push an int
ldc "Hello World" ; push a String
ldc_w 3.141592654 ; push PI as a double
```

## ۱۰ - دستور lookupswitch :

دستور lookupswitch معادل دستور case در زبان پاسکال است و دارای ساختاری به شکل زیر است :

```
lookupswitch
  <int1> : <label1>
  <int2> : <label2>
  ...
  default : <default-label>
```

به مثال زیر توجه کنید:

```
lookupswitch
    3 : Label1
    5 : Label2
    default : DefaultLabel

Label1:
    ..... ; got 3
Label2:
    ..... ; got 5
DefaultLabel:
    ..... ; got something else
```

در این کد با رسیدن به دستور lookupswitch مقدار بالای پشته با مقادیر داده‌شده در دستور مقایسه می‌شود تا با توجه به برچسب‌ها، کد اجرایی بعدی انتخاب شود.

## ۱۱ - دستور tableswitch :

شکل کلی این دستور به صورت زیر است:

```
tableswitch <low>
    <label1>
    <label2>
    .....
    default : <default-label>
```

این دستور هم همان کار دستور case را می‌کند ولی با این تفاوت که مقدار بالای پشته را با یک مقدار اولیه پایینی که با <low> مشخص می‌شود، مقایسه می‌کند و این مقایسه را با یک ترتیب صعودی ادامه می‌دهد. به مثال زیر توجه کنید:

```
tableswitch 0    ; comparison starts with zero
    LabelA
    LabelB
    LabelC
    default : DefaultLabel

LabelA:
    ; got 0

LabelB:
    ; got 1

LabelC:
    ; got 2

DefaultLabel:
    ; got something else
```

در این کد با رسیدن به دستور tableswitch مقدار عددی بالای پشته با عدد صفر مقایسه می‌شود و در صورت تساوی اجرا به Label1 منتقل می‌شود. در غیر این صورت مقایسه با عدد ۱ انجام می‌شود که در صورت موفقیت، این بار اجرا به Label2 منتقل خواهد شد. تعداد این مقایسه‌ها به تعداد Label هاست که اگر هیچ مقایسه‌ای به نتیجه نرسد، اجرا به DefaultLabel منتقل خواهد شد.

## ۱۲- دستوراتی که هیچ عملوندی ندارند :

دستورات زیر که بیشتر دستورات زبان Jasmin را تشکیل می‌دهند، هیچ عملوندی ندارند:

یک refrence را از آرایه ای از refrence ها load می کند.	aaload
یک refrence را در آرایه ای از refrence ها ذخیره می کند.	aastore
null را push می کند.	aconst_null
یک object reference از متغیر محلی صفر درون پشته push می کند.	aload_0
یک object reference از متغیر محلی ۱ درون پشته push می کند.	aload_1
یک object reference از متغیر محلی ۲ درون پشته push می کند.	aload_2
یک object reference از متغیر محلی ۳ درون پشته push می کند.	aload_3
Refrence را از تابع برمی گرداند.	areturn
طول آرایه را بدست می آورد.	arraylength

object reference بالای پشته را در متغیر محلی شماره صفر ذخیره می‌کند.	astore_0
object reference بالای پشته را در متغیر محلی شماره ۱ ذخیره می‌کند.	astore_1
object reference بالای پشته را در متغیر محلی شماره ۲ ذخیره می‌کند.	astore_2
object reference بالای پشته را در متغیر محلی شماره ۳ ذخیره می‌کند.	astore_3
object reference بالای پشته ( که از کلاس throwable و یا یکی از زیرکلاس‌های آن ایجاد شده‌است ) را pop کرده و exception پیاده‌سازی شده توسط آن را صادر می‌کند.	athrow
یک byte را از آرایه ای از byte ها load می‌کند.	baload
یک byte را در آرایه ای از byte ها ذخیره می‌کند.	bastore
این دستور اجرا را متوقف می‌کند.	breakpoint
یک char را از آرایه ای از char ها load می‌کند.	caload
یک char را در آرایه ای از char ها ذخیره می‌کند.	castore
double بالای پشته را به float تبدیل می‌کند.	d2f
double بالای پشته را به int تبدیل می‌کند.	d2i
double بالای پشته را به long تبدیل می‌کند.	d2l
دو double را با هم جمع می‌کند.	dadd
یک double را از آرایه ای از double ها load می‌کند.	daload
یک double را در آرایه ای از double ها ذخیره می‌کند.	dastore

dcmpg	دو double را با هم مقایسه کرده در صورتی که عملوند اول بزرگتر از دوم بود، ۱، در صورتیکه عملوند دوم بزرگتر بود، -۱، و در صورتی که با هم برابر بودند ۰ را بالای پشته push می کند.
dcmpl	دو double را با هم مقایسه کرده در صورتی که عملوند اول بزرگتر از دوم بود، -۱، در صورتیکه عملوند دوم بزرگتر بود، ۱، و در صورتی که با هم برابر بودند ۰ را بالای پشته push می کند.
dconst_0	ثابت double، 0.0 را بالای پشته push می کند.
dconst_1	ثابت double، 1.0 را بالای پشته push می کند.
ddiv	دو double را بر هم تقسیم می کند.
dload_0	یک double از متغیر محلی صفر درون پشته push می کند.
dload_1	یک double از متغیر محلی ۱ درون پشته push می کند.
dload_2	یک double از متغیر محلی ۲ درون پشته push می کند.
dload_3	یک double از متغیر محلی ۳ درون پشته push می کند.
dmul	دو double را در هم ضرب می کند.
dneg	Double بالای پشته را در -۱ ضرب می کند.
drem	باقیمانده صحیح تقسیم یک double را بر double برمیگرداند. معادل تابع fmod در c میباشد.
dreturn	یک double را از تابع بر می گرداند.
dstore_0	double بالای پشته را در متغیر محلی شماره صفر ذخیره می کند.
dstore_1	double بالای پشته را در متغیر محلی شماره ۱ ذخیره می کند.

double بالای پشته را در متغیر محلی شماره ۲ ذخیره می‌کند.	dstore_2	
double بالای پشته را در متغیر محلی شماره ۳ ذخیره می‌کند.	dstore_3	
عنصر بالای پشته را یک بار دیگر push می‌کند.	dup	
stack before	stack after	dup_x1
1.0	1.0	
2.0	2.0	
...	1.0	
1.0	1.0	dup_x2
2.0	2.0	
3.0	3.0	
...	1.0	
1.0	1.0	dup2
2.0	2.0	
...	1.0	
	2.0	
1.0	1.0	dup2_x1
2.0	2.0	
100	100	
...	1.0	
	2.0	dup2_x2
1.0	1.0	
2.0	2.0	
100	100	
200	200	
...	1.0	
	2.0	
float را به double تبدیل می‌کند.		f2d
float را به int تبدیل می‌کند.		f2i
float را به long تبدیل می‌کند.		f2l
دو float را با هم جمع می‌کند.		fadd
یک float را از آرایه‌ای از float ها load می‌کند.		faload

fastore	یک float را در آرایه‌ای از float ها ذخیره می کند.
fcmpg	دو float را با هم مقایسه کرده در صورتی که عملوند اول بزرگتر از دوم بود، ۱، در صورتی که عملوند دوم بزرگتر بود، بالا پشته ۱- و در صورتی که با هم برابر بودند ۰ را بالای پشته push می کند.
fcmpl	دو float را با هم مقایسه کرده در صورتی که عملوند اول بزرگتر از دوم بود، ۱-، در صورتی که عملوند دوم بزرگتر بود، ۱ و در صورتی که با هم برابر بودند ۰ را در پشته push می کند.
fconst_0	ثابت float، 0.0 را در پشته push می کند.
fconst_1	ثابت float، 1.0 را در پشته push می کند.
fconst_2	ثابت float، 2.0 را در پشته push می کند.
fdiv	دو float را بر هم تقسیم می کند.
fload_0	یک float از متغیر محلی صفر درون پشته push می کند.
fload_1	یک float از متغیر محلی ۱ درون پشته push می کند.
fload_2	یک float از متغیر محلی ۲ درون پشته push می کند.
fload_3	یک float از متغیر محلی ۳ درون پشته push می کند.
fmul	دو float را در هم ضرب می کند.
fneg	علامت float بالای پشته را برعکس می کند.
frem	دو float را بر هم تقسیم کرده، باقیمانده تقسیم را در پشته push می کند.
freturn	یک float را از تابع برمی گرداند.
fstore_0	float بالای پشته را در متغیر محلی شماره صفر ذخیره می کند.

float بالای پشته را در متغیر محلی شماره ۱ ذخیره می‌کند.	fstore_1
float بالای پشته را در متغیر محلی شماره ۲ ذخیره می‌کند.	fstore_2
float بالای پشته را در متغیر محلی شماره ۳ ذخیره می‌کند.	fstore_3
دو float را از هم تفریق می‌کند.	fsub
int را به double تبدیل می‌کند.	i2d
int را به float تبدیل می‌کند.	i2f
int را به long تبدیل می‌کند.	i2l
دو عدد صحیح بالای پشته را با هم جمع کرده نتیجه را بالای پشته قرار می‌دهد.	iadd
یک int را از آرایه ای از int ها load می‌کند.	iaload
دو int را با هم and می‌کند.	iand
یک int را در آرایه ای از int ها ذخیره می‌کند.	istore
عدد صحیح ثابت صفر را در بالای پشته قرار می‌دهد.	iconst_0
عدد صحیح ثابت یک را در بالای پشته قرار می‌دهد.	iconst_1
عدد صحیح ثابت ۲ را در بالای پشته قرار می‌دهد.	iconst_2
عدد صحیح ثابت ۳ را در بالای پشته قرار می‌دهد.	iconst_3
عدد صحیح ثابت ۴ را در بالای پشته قرار می‌دهد.	iconst_4
عدد صحیح ثابت ۵ را در بالای پشته قرار می‌دهد.	iconst_5
عدد صحیح ثابت -۱ را در بالای پشته قرار می‌دهد.	iconst_m1
دو عدد صحیح بالای پشته را بر هم تقسیم کرده نتیجه را بالای پشته قرار می‌دهد.	idiv



یک از متغیر محلی صفر درون پشته push می‌کند.	iload_0
یک از متغیر محلی ۱ درون پشته push می‌کند.	iload_1
یک از متغیر محلی ۲ درون پشته push می‌کند.	iload_2
یک از متغیر محلی ۳ درون پشته push می‌کند.	iload_3
دو عدد صحیح بالای پشته را در هم ضرب کرده و نتیجه را در بالای پشته قرار می‌دهد.	imul
علامت int بالای پشته را بر عکس می‌کند.	ineg
int را به byte تبدیل می‌کند.	int2byte
int را به char تبدیل می‌کند.	int2char
int را به short تبدیل می‌کند.	int2short
دو int را با هم or می‌کند.	ior
دو int را بر هم تقسیم کرده، باقیمانده تقسیم را در پشته push می‌کند.	irem
یک int را از تابع بر می‌گرداند.	ireturn
یک int را به اندازه مشخص شده در بالای پشته شیفت منطقی به سمت چپ می‌دهد.	ishl
یک int را به اندازه مشخص شده در بالای پشته شیفت منطقی به سمت راست می‌دهد.	ishr
Int بالای پشته را در متغیر محلی شماره صفر ذخیره می‌کند.	istore_0
int بالای پشته را در متغیر محلی شماره ۱ ذخیره می‌کند.	istore_1
int بالای پشته را در متغیر محلی شماره ۲ ذخیره می‌کند.	istore_2
int بالای پشته را در متغیر محلی شماره ۳ ذخیره می‌کند.	istore_3
دو int را از هم تفریق می‌کند.	isub

iushr	شیفت منطقی به سمت راست.
ixor	دو int را با هم xor می‌کند.
l2d	long را به double تبدیل می‌کند.
l2f	long را به float تبدیل می‌کند.
l2i	long را به int تبدیل می‌کند.
ladd	دو long را با هم جمع می‌کند.
laload	یک long را از آرایه ای از long ها load می‌کند.
land	دو long را با هم and می‌کند.
lastore	یک long را در آرایه ای از long ها ذخیره می‌کند.
lcmp	دو long را با هم مقایسه کرده در صورتی که عملوند اول بزرگتر از دوم بود، ۱، در صورتیکه عملوند دوم بزرگتر بود، -۱ و در صورتی که با هم برابر بودند ۰ را بالای پشته push می‌کند.
lconst_0	ثابت long ، 0 را در پشته push میکند.
lconst_1	ثابت long ، 1 را در پشته push میکند.
ldiv	دو long را بر هم تقسیم می‌کند.
lload_0	یک long از متغیر محلی صفر درون پشته push می‌کند.
lload_1	یک long از متغیر محلی ۱ درون پشته push می‌کند.
lload_2	یک long از متغیر محلی ۲ درون پشته push می‌کند.
lload_3	یک long از متغیر محلی ۳ درون پشته push می‌کند.
lmul	دو long را در هم ضرب می‌کند.
lneg	علامت long را بر عکس می‌کند.
lor	دو long را با هم or می‌کند.
lrem	دو long را بر هم تقسیم کرده، باقیمانده تقسیم را در پشته

	push می‌کند.
lreturn	یک long را از تابع برمی‌گرداند.
lshl	یک long را به اندازه مشخص شده در بالای پشته شیفت منطقی به سمت چپ می‌دهد.
lshr	یک long را به اندازه مشخص شده در بالای پشته شیفت منطقی به سمت راست می‌دهد.
lstore_0	long بالای پشته را در متغیر محلی شماره ۰ صفر ذخیره می‌کند.
lstore_1	long بالای پشته را در متغیر محلی شماره ۱ ذخیره می‌کند.
lstore_2	long بالای پشته را در متغیر محلی شماره ۲ ذخیره می‌کند.
lstore_3	long بالای پشته را در متغیر محلی شماره ۳ ذخیره می‌کند.
lsub	دو long را از هم تفریق می‌کند.
lushr	شیفت منطقی به سمت راست
lxor	دو long را با هم xor می‌کند.
monitorenter	object reference بالای پشته را pop کرده، آن را lock می‌کند.
monitorexit	object reference بالای پشته را pop کرده، آن را unlock می‌کند.
nop	هیچ عملی را انجام نمی‌دهد. این دستور توسط کمپایلرها برای مقاصدی مانند اشکال‌زدایی ایجاد می‌شود.
pop	عنصر بالایی را از پشته pop می‌کند.
pop2	دو خانه از عناصر بالای پشته را pop می‌کند. ( دو int ، یک

int و یک object reference ، یک double ، ... )	
یک void را از تابع بر می گرداند.	return
یک short را از آرایه‌ای از short ها load می کند.	saload
یک short را در آرایه‌ای از short ها ذخیره می کند.	sastore
دو عنصر بالای پشته را جابجا می کند.	swap

برای کسب اطلاعات بیشتر، می‌توانید The Java Virtual Machine Specification را از سایت [www.sun.com](http://www.sun.com) ، download نموده و توضیح مفصل هر یک از این دستورات را مطالعه نمایید.

# چگونه برنامه‌ای به زبان Jasmin بنویسیم؟

در این قسمت ما به بررسی ساختار کلی یک برنامه Jasmin می‌پردازیم و به عنوان نمونه چند برنامه ساده Jasmin را بررسی می‌کنیم.  
شایان ذکر است که هر سه برنامه از کتاب Java Virtual Machine نوشته Jon Meyer و Troy Downing اقتباس شده‌اند.

## برنامه HelloWorld

```
.class public HelloWorld
.super java/lang/Object

; specify the constructor method for the example class

.method public <init>()V
    ; just calls object constructor
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method

.method public static main([Ljava/lang/String;)V
    ; set limits used by this method
    .limit stack 2

    ; onto the stack, then invoke the println method
    getstatic java/lang/System/out Ljava/io/PrintStream;

    invokevirtual java/io/PrintStream/println(Ljava/lang/
        String;)V

    return
.end method
```

### بررسی برنامه:

```
.class public HelloWorld
```

این دستور در Jasmin نام کلاس و سطوح دستیابی آن را مشخص می‌کند. برای مثال در این برنامه نام کلاس ما HelloWorld است و این کلاس از سایر کلاس‌ها و حتی سایر package ها قابل دستیابی است.

```
.super java/lang/Object
```

این دستور مشخص می‌کند که کلاس HelloWorld یک زیرکلاس از کلاس java/lang/Object است، بنابراین field ها و متدهای آن را به ارث می‌برد. کلاس Object ابتدایی‌ترین کلاس Java است.

#### Initialization method

اولین متد کلاس HelloWorld، متد <init> است. نام این متد یک reserved word است و در Java به طور خودکار هنگام دیدن دستور new مورد فراخوانی قرار می‌گیرد. البته چون ما در سطح JVM کار می‌کنیم، باید با استفاده از دستور invokespecial این متد را فراخوانی کنیم. مورد قابل توجه بعدی در این متد، public است. این کلمه کلیدی به JVM می‌گوید که این متد از تمام کلاس‌ها و package ها قابل دستیابی است. در Java ؛ کلمه کلیدی برای تعیین حوزه دید وجود دارد که عبارتند از:

- public : قابل رؤیت برای تمام کلاس‌ها
- private : قابل رؤیت فقط درون کلاس
- protected : قابل رؤیت در کلاس، زیرکلاس‌های آن، و package کلاس
- private protected : قابل رؤیت در کلاس و زیرکلاس‌های آن

پس از تعیین حوزه دید، با استفاده از دستور invokeinterface متد <init> از کلاس java/lang/Object فراخوانی شده‌است که این دستور قبلاً توضیح داده شده‌است.

#### Main method

پس از initialization method ، متد اصلی برنامه آمده است. این متد به صورت public static تعریف شده است. static به Java می‌گوید که برای فراخوانی این متد نیازی به pop کردن یک object reference از پشته نیست. مورد بعدی، ([Ljava/java/String;)V است. با توجه به descriptor ، مشخص می‌شود که متد main آرایه ای از string ها را به عنوان پارامتر ورودی می‌گیرد و با توجه به V مشخص می‌شود که این متد مقدار بازگشتی ندارد.

بقیه دستورات این مثال، قبلاً توضیح داده شده‌اند.

## برنامه Count

این مثال، مقداری پیچیده‌تر از مثال قبل است و هدف آن پیاده‌سازی دستور for با استفاده از دستورات Jasmin می‌باشد. خروجی برنامه معادل خروجی کد پاسکال زیر است:

```
var
  i: integer;
begin
  for i := 0 to 9 do
    Writeln(i);
end;
```

```
.class public count
.super java/lang/Object

; the instance initialization method ( as for HelloWorld )
.method public <init>()V
  ; just calls object constructor
  aload_0
  invokespecial java/lang/Object/<init>()V
```



```

        return
    .end method

.method public static main([Ljava/lang/String;)V

    ; set limits used by this method
    .limit stack 3
    .limit locals 4

    ; setup local variables

    ; 1- the PrintStream object held in java.lang.System.out
    getstatic java/lang/System/out Ljava/io/PrintStream;
    astore_1

    ; 2- the integer 10
    bipush 10
    istore 2

    ; now loop 10 times printing out a number
Loop:
    ; compute local variable 2, convert this integer to a
    ; string, and store the result in local variable 3
    bipush 10
    iload_2
    isub
    invokestatic
        java/lang/String/ValueOf(I)Ljava/lang/String;
    astore_3

    ; now print the string in local variable 3
    aload_1          ; push the PrintStream object
    aload_3          ; push the string
    invokevirtual
        java/io/PrintStream/println(Ljava/lang/String;)V

    ; decrement the counter and loop
    iinc 2 1
    iload_2
    ifne Loop
    return

.end method

```

## برنامه HelloWorld Applet

در این مثال، ما به بررسی یک applet ساده می‌پردازیم. قبل از هر چیز دقت کنید که یک applet یک برنامه استاندارد نیست، بنابراین تابع main نیز ندارد و مستقیماً با فرمان Java HelloWorld.class قابل اجرا نیست. در مثال زیر ما

با استفاده از override کردن متدهای `init()` و `paint()` به مرورگر می‌گوییم که چگونه سطح `applet` را رسم کند. این `applet` فقط یک پیغام `Hello Web` را با فونت `Helvetica` چاپ می‌کند.

```
.class public HelloWorld
.super java/applet/Applet

; declare a private field called font, holding
; a java.awt.Font object
.field private font Ljava/awt/Font;

.method public init()V
    .limit stack 6
    aload_0      ; this

    ; new font: Helvetica, Bold, point 48
    new java/awt/Font
    dup
        ; font name
    iconst_1      ; font Bold flag
    bipush 48      ; font size
    invokespecial
        java/awt/Font/<init>(Ljava/lang/String;II)V

    ; stack currently contains this, font
    ; now use putfield to assign the font item to this.font
    putfield HelloWorld/font Ljava/awt/Font;
    return
.end method

.method public paint(Ljava/awt/Graphics;)V
    .limit stack 4
    ; two locals ( 0 = this, 1 = Graphics Object )
    .limit locals 2

    aload_1      ; Graphics object
    aload_0      ; this
    getfield HelloWorld/font Ljava/awt/Font;      ;this.font
    invokevirtual
        java/awt/Graphics/setFont(Ljava/awt/Font;)V

    aload_1      ; Graphics object

    bipush 25
    bipush 50
    invokevirtual
        java/awt/Graphics/drawString(Ljava/lang/String;II)
    v
    return
.end method

; standar
.method public <init>()V
```

```

        aload_0
        invokespecial java/applet/Applet/<init>()V
        return
    .end method

```

برای دیدن این applet باید آن را در یک صفحه html بارگذاری کنید. به عنوان نمونه می‌توانید از کد زیر استفاده کنید:

```

<HTML>
<HEAD>
<TITLE>The Hello Applet</TITLE>
</HEAD>
<BODY>


300 height=200>


</applet>
</BODY>
</HTML>

```

ضمیمه ۱ :

## Descriptors

۱ - Field Descriptor ها:

یک field descriptor، نوع کلاس، instance، و یا متغیر محلی را مشخص می‌کند. هر field descriptor از تعدادی کاراکتر تشکیل می‌شود که این کاراکترها توسط گرامر زیر ایجاد می‌شوند ( جملات پایانی یا terminalها به صورت **bold** آمده‌اند):

FieldDescriptor -> FieldType

```

ComponentType -> FieldType
FieldType     -> BaseType
               | ObjectType
               | ArrayType
BaseType      -> B
               | C
               | D
               | F
               | I
               | J
               | S
               | Z
ObjectType    -> L <classname> ;
ArrayType     -> [ ComponentType

```

در این گرامر، ترم‌های پایانی به صورت زیر تفسیر می‌شوند:

ترم پایانی	نوع داده	توضیح
B	Byte	-----
C	Char	یک کاراکتر Unicode
D	Double	-----
F	Float	-----
I	Int	-----
J	Long	-----
L <classname>;	Reference	یک instance از <classname>
S	Short	-----
Z	Boolean	-----
[	Reference	یک بعد آرایه

برای مثال، descriptor یک متغیر از نوع int به سادگی یک I است و یا descriptor یک متغیر از نوع object به صورت Ljava/lang/Object; می‌باشد. دقت کنید که از فرم داخلی نام کلاس Object استفاده شده است.

در مورد آرایه‌ها نیز به همین صورت عمل می‌کنیم. برای مثال فرض کنید می‌خواهیم descriptor یک آرایه سه‌بعدی از نوع double را بنویسیم. تعریف این آرایه و decriptor آن به صورت زیر است:

```
double d[ ] [ ] [ ]; // array definition
[ [ [ D // array descriptor
```

## ۲- method descriptor ها

یک method descriptor پارامترها و مقدار بازگشتی یک متد را مشخص می‌کند. method descriptor ها با استفاده از گرامر زیر نوشته می‌شوند:

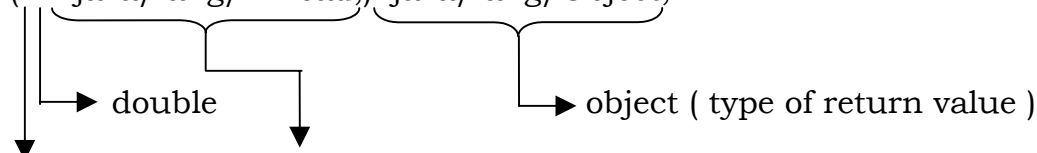
```
MethodDescriptor ->
    ( ParameterDescriptor * ) ReturnDescriptor
ParameterDescriptor -> FieldType
ReturnDescriptor -> FieldType
                    |
                    V
```

در این گرامر، ParameterDescriptor پارامترهای ارسالی به یک متد را مشخص می‌کند. علامت \* در این گرامر به معنای تکرار ( صفر مرتبه یا بیشتر ) می‌باشد. ReturnDescriptor نیز یک رشته کاراکتری است که مقدار بازگشتی متد را مشخص می‌کند. در ReturnDescriptor ، حرف V نمایانگر نوع داده void می‌باشد ( یعنی متد هیچ داده‌ای را باز نمی‌گرداند ).

برای مثال به متد زیر و method descriptor آن توجه کنید:

```
Object mymethod ( int I, double d, Thread t )
```

```
(IDLjava/lang/Thread;)Ljava/lang/Object;
```



int

Thread

ضمیمہ ۲ :

## Jasmin Code Generator for Expressions Grammar

Expressions LL1 Grammar:

```
Expr      -> term Expr2
Expr2     -> ( + | - | or ) term Expr2 | λ
term      -> factor term2
term2     -> ( * | / | and ) factor term2 | λ
factor    -> ( + | - | not ) factor2 | factor2
factor2
```

```
/* Expr -> term Expr2 */
procedure Expr(Stop: Set of Symbols);
Begin
  term(Stop + First_Expr2 );
  Expr2(Stop);
```

```

End;

/* Expr2 -> ( + | - | or ) term Expr2 | λ */
procedure Expr2(Stop: Set of Symbols);
var
  CSymbol: Symbols;
begin
  if not (CurrentSymbol in Stop) then
    begin
      CSymbol := CurrentSymbol;
      NextSymbol;
      term(Stop + First_Expr2);
      Case CSymbol of
        S_Plus: Emitln('iadd');
        S_Minus: Emitln('isub');
        S_Or: Emitln('ior');
        else: SyntaxError(S_Plus, Stop);
      end;
      Expr2(Stop);
    end;
end;

/* term -> factor term2 */
procedure term(Stop: Set of Symbols);
begin
  factor(Stop + First_term2);
  term2(Stop);
end;

/* term2 -> ( * | / | and ) factor term2 | λ */
procedure term2(Stop: Set of Symbols);
var
  CSymbol: Symbols;
begin
  if not (CurrentSymbol in Stop) then
    begin
      CSymbol := CurrentSymbol;
      NextSymbol;
      factor(Stop + First_term2 );
      Case CSymbol of
        S_Mul: Emitln('imul');
        S_Div: Emitln('idiv');
        S_And: Emitln('iand');
        else: SyntaxError(S_Mul, Stop);
      end;
      term2(Stop);
    end;
end;

/* factor -> ( + | - | not ) factor2 | factor2 */
procedure factor(Stop: Set of Symbols );
var
  Csymbol: Symbols;
  Labell, Label2: string;
begin
  if CurrentSymbol in [S_Plus, S_Minus, S_Not] then
    begin

```

```

CSymbol := CurrentSymbol;
factor2(Stop);
case CSymbol of
  S_Plus:           // do nothing
  S_Minus:
  S_Not:      begin
                  Label1 := NewLabel;
                  Label2 := NewLabel;
                  Emitln('ifne ' + Label1);
                  Emitln('iconst_1');
                  Emitln('goto ' + Label2);
                  Emitln(Label1 + ':');
                  Emitln('iconst_0');
                  Emitln(Label2 + ':');
                end;
            end; // end of case
end // end of if
else
  factor2(Stop);
end;

/* factor -> id | num | '('Expr')' */
procedure factor(Stop: Set of Symbols );
begin
  if CurrentSymbol in [S_Id,S_Num,S_OpenPar] then
    case CurrentSymbol of
      S_Id:      Emitln('getstatic /' + CurrentToken.Lexeme +
                        TypeToStr(IdType)); // getstatic /a
I
      S_Num:      Emitln('ldc ' + CurrentToken.Lexeme);
      S_OpenPar: begin
                    NextSymbol;
                    Expr(Stop + [S_ClosePar]);
                    Expect(S_ClosePar, Stop);
                  end;
    end //end of case
  else
    SyntaxError(S_Id, Stop);
  end;
end;

```

لطفا هرگونه پیشنهاد یا انتقاد خود در جهت بهبود این مستندات  
را با ما در میان بگذارید.



[rzakery2001@yahoo.com](mailto:rzakery2001@yahoo.com)