بسمه تعالی

جزوه ضمیمه درس

اصول طراحی کامپایلرها

« 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 اجـزاء اصلـی Java، Java Virtual Machine یـا JVM اسـت. JVM یک کامپیوتر مجازی است که معمولا به صورت نـرم افـزاری بـر روی یک سیستم سخت افزاری و سیستم عامل آن پیاده سازی می شود و برنامـه های ترجمه شده Java را اجرا می کند. با استفاده از JVM ، برنامه هـایی کـه به زبان Java نوشته مـی شـوند ، بـدون نیـاز بـه هیچگونـه تغییـر بـر روی ماشینها و سیستم عاملـهای مختلـف اجـرا مـی شـوند. ایـن مسـاله می‌توانـد انگیزه‌ای قوی برای نوشتن کامپایلرهایی باشد که کـد خروجـی آنـها نـیز بـا استفاده از JVM قابل اجرا باشد. این موضوع در دانشکده کامپیوتر دانشـگاه علم و صنعت ایران توسط « دکتر سعید پارسا » استاد درس اصول طراحـی کامپایلرها در مقطع کارشناسی و کامپایلرهای پیشرفته در مقطع کارشناسـی ارشد مورد توجه قرار گرفته است. تقریبا از سال ۱۳۷٦ با توجه بـه رویکـرد درس کامپایلرها و استاد این درس به این سـمت، دانشـجویان ایـن دانشـکده پروژه‌های مختلفی در این زمینه انجام داده‌اند که به عنوان نمونه می‌توان بـه پروژه Jacomizer و راهنمای ایجاد Applet های جـاوا اشـاره کـرد. در ایـن مستندات ما تلاش کرده‌ایم تا مبانی علمی لازم بـرای ایـن رویکـرد را فراهـم کنیم. امید است که راهگشای دانش‌پژوهان ایران عزیزمان باشد.

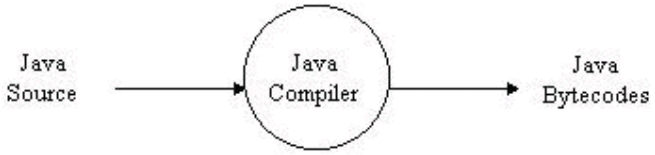
Java Virtual Machine چیست ؟

JVM یک کامپیوتر مجازی است که برنامه های ترجمه شده Java را اجرا می‌کند. تمام برنامه های Java برای اجرا توسط JVM ترجمه می شوند، بنابراین برای اجرای این برنامه‌ها حتما باید JVM روی سکوی کاری پیاده‌سازی شده باشد. حجم کم JVM و قابلیت سوارشدن 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.

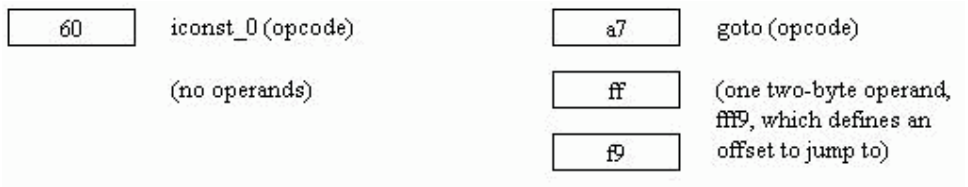
برنامه های ترجمه شده Java
ماشین مجازی Java یا Java Virtual Machine
سکوی سخت افزاری و سیستم عامل

Java bytecode چیست ؟

برنامه های Java به یک قالب خاص که Java bytecode نامیده می‌شود، ترجمه می شوند. JVM ، bytecode ها را اجرا می کند، بنابراین می‌توان bytecode ها را به عنوان زبان ماشین JVM در نظر گرفت. کامپایلر Java ، فایلهای 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 پشت‌سرهم می‌باشد. هر دستورالعمل شامل یک بایت کد عمل (opcode) و تعدادی (صفر یا بیشتر)عملوند است. opcode به JVM می‌گویدکه چه عملی باید انجام شود. اگر این کد عمل دارای عملوند باشد، عملوندهای آن بلافاصله بعد از کد عمل می‌آیند. هر کد عمل با یک دستور شبه اسمبلی نیز مشخص می‌شود. مثلا 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 ۱٦ بیتی علامتدار بعد از opcode می‌آید. کد این دستور 0xA7 و معادل شبه اسمبلی آن goto است.



سخت افزار مجازی JVM

سخت افزار مجازی یک JVM را می‌توان به ٤ بخش تقسیم کرد: رجیسترها، پشته، garbage-collected heap ، و method area . این قسمتها مانند ماشینی که تشکیل می‌دهند، انتزاعی هستند، ولی حتما باید در هر JVM به‌گونه‌ای پیاده‌سازی شده باشند.

آدرس‌ها در JVM به صورت ۳۲ بیتی بیـان می‌شـوند، بنـابراین JVM می تواند تا ٤ گیگابایت فضای حافظه را آدرس‌دهـی کنـد. هـر رجیسـتر یـک آدرس ۳۲ بیتـی را نگـهداری می‌کنـد. پشـــته، garbage-collected heap ، و method area هرکدام جایی درون این فضای حافظه قـابل آدرس‌دهـی قـرار دارنـد. مکـان دقیـق ایـن فضاهـای حافظـه بـه تصمیـم کسـی کـه JVM را پیاده‌سازی می‌کند بستگی دارد.

هر کلمه در JVM ۳۲ بیت است. JVM انواع داده اولیه کمـی دارد کـه عبـارتند از: byte (۸ بیـت) ، short (١٦ بیـت) ، int (۳۲ بیـت) ، long (٦٤ بیـــت) ، float (۳۲ بیت) ، double (٦٤ بیت) ، و char (١٦ بیت) که بـــه جـز نـوع داده char (که یک کـاراکتر بـدون علامـت Unicode اسـت) همـه علامتـدار هستند. این انواع به راحتی با نوع داده های زبـان Java قـابل تطبیـق هسـتند. یک نوع داده ابتدایی دیگر، object handle است که به صورت یـک آدرس ۳۲ بیتی به یک object روی heap اشاره می‌کند.

چون method area محدوده‌ای است که bytecode ها را دربرمی‌گیرد، به صورت بـایتی سـازماندهی شـده اسـت. امـا پشـته و garbage-collected heap از یک ساختار ۳۲ بیتی استفاده می‌کنند.

چون JVM یک سیستم stack-based است (یعنی تمام عملیـات مـورد نیاز در حافظه برای bytecode ها را با یـک پشـته انجـام می‌دهـد) نیـازی بـه رجیسترهایی برای نگهداری مقادیر میـانی نـدارد. بنـابراین در طراحـی JVM فقط یک شمارنده برنامه (program counter) و سه رجیستر برای مدیریـت پشته پیش‌بینی شده است که این مساله (کم بودن تعداد رجیسترها) موجـب کوچک شدن مجموعه دستورات JVM می‌شود.

JVM از رجیسـتر pc بـرای نگهداشـــتن آدرس حافظـــه ای کـــه بـــاید دستورات درون آن اجرا شوند استفاده می‌کند. سه رجیستر دیگر که optop ، frame ، و vars نام دارند به قسمتهای مختلف stack frame متد درحال اجرا

(قسمتی از پشته که به متد درحال اجرا اختصـــاص دارد) اشـاره می‌کننـد. stack frame هر متد وضعیت آن متد (مقادیر متغیرهای محلی، نتایج میـانی، محاسبات و غیره) را برای هر فراخوانی به صورت جداگانه نگهداری می‌کنـد (یعنی هر فراخوانی stack frame مخصوص به خود را دارد).

method area و program counter :

method area جـایی اسـت کـه bytecodeهـا قـرار دارنـد. program counter همواره به آدرسی در حافظه اشاره می‌کنـد کـه بـاید دسـتورات آن آدرس اجرا شوند. پس از اجرای هر دستور bytecode ، اگر این دستور باعث یک پرش (jump) نشود، JVM مقدار program counter را برابـر آدرس دستوری قرار می‌دهد که بلافاصله پس از دستور اجراشده قرار دارد.

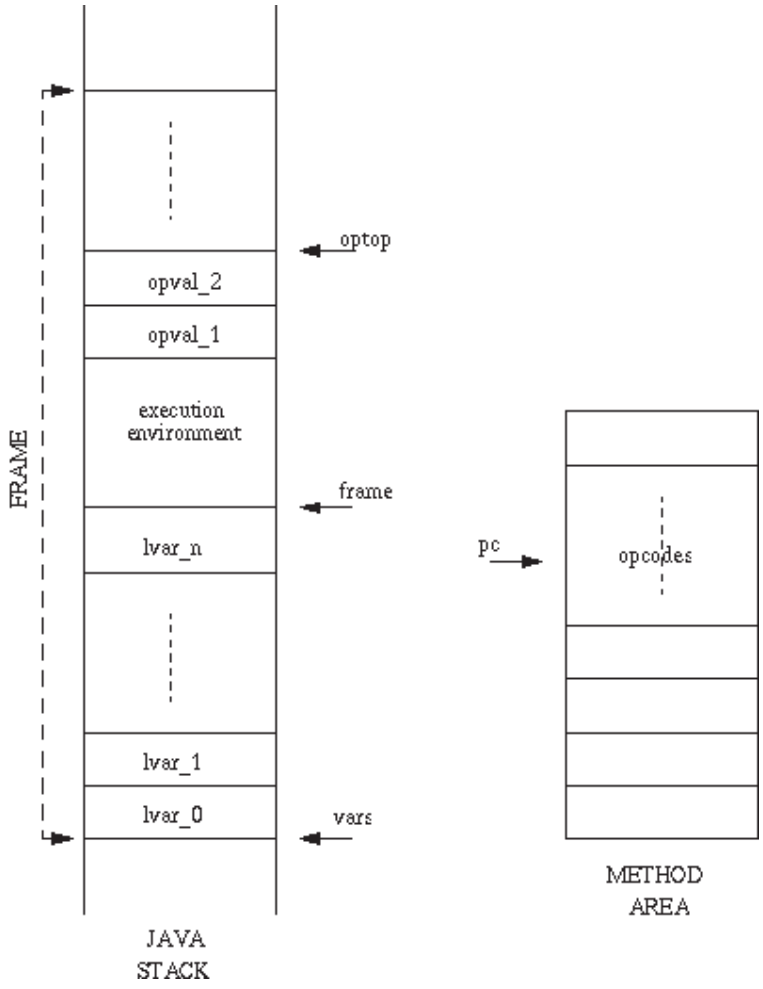
پشته و رجیسترهای مربوط به آن

در Java از پشته برای نگهداری پارامترهای موردنیاز و نتایج حـاصل از دستورات bytecode ، ارسال پارامتر به method ها و بـرگردانـدن مقـادیر بازگشـتی ، و نگـهداری وضعیـت هـر متـد در حـال اجـرا اسـتفاده می‌شـود. رجیسترهای optop ، frame ، و vars هم به قسمتهای مختلـف stack frame جاری اشاره می‌کنند.

در هر stack frame سه بخش مجـزا وجـود دارد: متغیرهـای محلـی (local variables)، محیط اجرا (execution environment)، و پشته عملوندها (operand stack). بخش متغیرهای محلی، تمام متغیرهای محلـی فراخوانـی جاری متد را نگهداری می‌کند. آدرس این بخـش در رجیسـتر vars نگـهداری می‌شود. بخش execution environment بـه نگـهداری عملیـات خـود stack اختصـاص دارد و آدرس آن در رجیسـتر frame نگـهداری می‌شـود. پشـــته

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 می‌کنند. هرزمان که شما با استفاده از دستور new به یک object حافظه اختصاص می‌دهید، این حافظه از heap می‌آید. در زبان Java شما نمی‌توانید حافظه گرفته شده را مستقیما آزاد کنید. به جای این، در زمان اجرا تعداد ارجاع هایی که به یک object روی حافظه وجود دارد کنترل می‌شود و هرزمان ارجاع دیگری به یک object وجود نداشت، فضای حافظه اختصاص یافته به آن آزاد می‌شود. به این پروسه garbage collection می‌گویند.

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 می‌کند و با توجه به syntax ساده‌ای 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 می‌کند و Java نیز به جای رجیسترهای معمول از یک پشته استفاده می‌کند، دیگر نیازی به عملیات Register Management نیست که این هم یکی از مزایای استفاده از کد Jasmin به عنوان خروجی کامپایلر می‌باشد.

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 (مثلاtarget.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 بایتی 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 مسلما 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 قبلا 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 می کند.
dstore	عنصر بالای پشته را در یک متغیر محلی double ذخیره می‌کند.
fload	یک متغیر محلی float را درون پشته push می کند.
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 می‌کند.
lload	یک متغیر محلی long را درون پشته push می کند.
lstore	عنصر بالای پشته را در یک متغیر محلی long ذخیره می‌کند.

دستورات متغیرهای محلی

به عنوان مثال، به این دستورات توجه کنید :

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

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

۲- دستورات bipush ، sipush ، و iinc :

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

```
bipush <int>
sipush <int>
```

البته bipush یک عدد یک بایتی را push می‌کند ولی sipush برای دو byte این کار را انجام می‌دهد.

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 یـا بـه 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 :

goto	یـک پـرش بـدون شـرط انجـام می‌دهـد. در ایـن دسـتور branch offset پرش ۱۶ بیتی است. (آدرس پرش: pc + branch offset)
goto_w	یک پرش بدون شرط انجام می دهد، ولـی بـرای پیـدا کـردن offset مورد نیاز برای پرش از یک branch offset ۳۲ بیتـی استفاده می‌کند. چـون Jasmin خـود در ایـن مـورد تصمیـم می‌گیرد، در Jasmin این دستور با goto معادل است.

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 موفقیت‌آمیز بوده و پرش انجام می شود.
if_icmpge	اگر بین دو عدد int بالای پشته رابطه value2 >= value1 برقرار باشد، مقایسه موفقیت‌آمیز بوده و پرش انجام می‌شود.
if_icmpgt	اگر بین دو عدد int بالای پشته رابطه value2 > value1 برقرار باشد، مقایسه موفقیت‌آمیز بوده و پرش انجام می‌شود.
if_icmple	اگر بین دو عدد int بالای پشته رابطه value2 <= value1 برقرار باشد، مقایسه موفقیت‌آمیز بوده و پرش انجام می‌شود.
if_icmplt	اگر بین دو عدد int بالای پشته رابطه value2 < value1 برقرار باشد، مقایسه موفقیت‌آمیز بوده و پرش انجام می‌شود.
if_icmpne	اگر دو عدد int بالای پشته مساوی نباشند، مقایسه موفقیت‌آمیز بوده و پرش انجام می‌شود.
Ifeq	اگر عدد int بالای پشته صفر باشد، مقایسه موفقیت‌آمیز بوده و پرش انجام می‌شود.

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
                  1
      jsr   PrintMe                          1

                  2
      jsr   PrintMe                          2

                  3
      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
```

درجدول بالا دقت کنید که تمام عناصری که باید مقایسه شـوند، ابتـدا 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.

٤- 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 داده‌شده) می‌تواند نام یک کلاس، interface و یا یک descriptor نوع آرایه (مانند [[Ljava/lang/String;) باشد. در دستور new ، <type> فقط می‌تواند نام یک کلاس باشد. این دستورات عبارتند از:

anewarray	یک آرایه برای نگهداری object reference هایی از نوع داده‌شده ایجاد می‌کند. اندازه این آرایه از بالای پشته pop می‌شود و پس از ایجاد شدن، یک reference به آن در پشته push می‌شود. تمامی اعضای این آرایه در ابتدا null هستند.
checkcast	چک میکند که آیا object reference بالای پشته قابل تبدیل به نوع داده‌شده می‌باشد یا خیر؟ اگر پاسخ مثبت باشد و یا مقدار object reference برابر null باشد، اجرا ادامه پیدا می‌کند، وگرنه پیغام خطای ClassCastException صادر می‌شود. در این دستور، پشته تغییری نمی‌کند.
instanceof	ابتدا object reference از پشته pop می‌شود. اگر <type> یک کلاس باشد، و object reference یک instance از آن کلاس و یا زیرکلاس‌های آن باشد، عدد ۱ (int) و اگر نه عدد صفر در پشته push می‌شود. اگر <type> یک interface باشد، این تست درباره پیاده‌سازی شدن یا نشدن <type> در object reference صورت می‌پذیرد.
new	یک شیء جدید از کلاس داده‌شده می‌سازد و object

	reference آن را در پشته push می‌کند. تمامی field هـــای int و boolean این شیء در ابتدا مساوی صفر قرارمی‌گیرند، امـــا هنوز شیء uninitialized است و برای اســتفاده بـاید یکـی از متدهای init آن را با دستور invokespecial فراخوانی کرد.

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 :

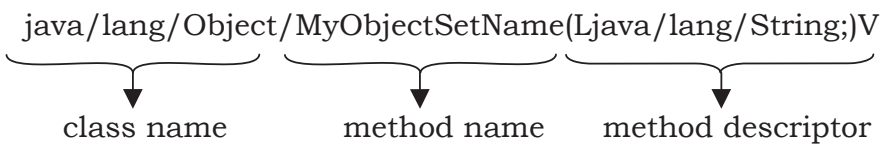
در Jasmin چهار دستور برای فراخوانی متدها پیش‌بینی شده‌اند که عبارتند از:

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

invokespecial گسترش‌یافته دستور invokenonvirtual (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;
```

٦- دستورات کار با field ها :

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

directive <field-specification> <descriptor>

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

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.
alaod_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 کاراکترهـای <field-specification> تا قبل از آخرین '/' نام کلاس و بقیه کاراکترهای بعـد از آخریـن '/' هـم نـام 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 می‌کننـد.

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 زبـان پاسـکال اسـت 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 مقـدار بـالای پشـته بـا مقادیر داده‌شده در دستور مقایسه می‌شود تا با توجـه بـه بـرچسـب‌ها، کـد 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   :   DefaulLabel

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، این‌بار اجرا به Label2 منتقل خواهد شد. تعداد این مقایسه‌ها به تعداد Label هاست که اگر هیچ مقایسه‌ای به نتیجه نرسد، اجرا به DefaultLabel منتقل خواهدشد.

۱۲- دستوراتی که هیچ عملوندی ندارند :

دستورات زیر که بیشتر دستورات زبان Jasmin را تشـــکیل می‌دهنـد، هیچ عملوندی ندارند:

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

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

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	ثابتdouble ، 0.0 را بالای پشته push می کند.
dconst_1	ثابت double ، 1.0 را بالای پشتهpush می کند.
ddiv	دو double را بر هم تقسیم می کند.
dload_0	یک double از متغیر محلی صفر درون پشته push می‌کند.
dload_1	یک double از متغیر محلی ۱ درون پشته push می‌کند.
dload_2	یک double از متغیر محلی ۲ درون پشته push می‌کند.
dload_3	یک double از متغیر محلی ۳ درون پشته push می‌کند.
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 درc میباشد.
dreturn	یک double را از تابع بر می گرداند.
dstore_0	double بالای پشته را در متغیر محلی شماره صفر ذخیره می‌کند.
dstore_1	double بالای پشته را در متغیر محلی شماره ۱ ذخیره می‌کند.

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

fastore	یک float را در آرایه‌ای از float ها ذخیره می کند.
fcmpg	دو float را با هم مقایسه کرده در صورتی کـه عملونـد اول بزرگتر از دوم بود ،۱،در صورتی که عملوند دوم بـزرگـتر بود ،بالا پشته ۱- ودر صورتی که با هم برابـر بودنـد ۰ را بالای پشته push می کند.
fcmpl	دو float را با هم مقایسه کرده در صورتی کـه عملونـد اول بـزرگـتر از دوم بـود، ۱- ، در صورتـی کـه عملونـد دوم بزرگتر بود ،۱ ودر صورتی که با هم برابـر بودنـد 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 می‌کند.
fload_1	یک float از متغیر محلی ۱ درون پشته push می‌کند.
fload_2	یک float از متغیر محلی ۲ درون پشته push می‌کند.
fload_3	یک float از متغیر محلی ۳ درون پشته push می‌کند.
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 پشـته push می‌کند.
freturn	یک float را از تابع برمی گرداند.
fstore_0	float بالای پشته را در متغیر محلـی شـماره صفـر ذخـیره می‌کند.

fstore_1	float بالای پشته را در متغیر محلی شماره ۱ ذخیره می‌کند.
fstore_2	float بالای پشته را در متغیر محلی شماره ۲ ذخیره می‌کند.
fstore_3	float بالای پشته را در متغیر محلی شماره ۳ ذخیره می‌کند.
fsub	دوfloat را از هم تفریق می کند.
i2d	int را به doubl e تبدیل می کند.
i2f	int را به float تبدیل می کند.
i2l	int را به l ong تبدیل می کند.
iadd	دو عدد صحیح بالای پشته را با هم جمع کرده نتیجه را بالای پشته قرار می دهد.
iaload	یک int را از آرایه ای از int ها load می کند.
iand	دو int را با هم and می کند.
iastore	یک int را در آرایه ای ازint ها ذخیره می کند.
iconst_0	عدد صحیح ثابت صفر را در بالای پشته قرار می دهد.
iconst_1	عدد صحیح ثابت یک را در بالای پشته قرار می دهد.
iconst_2	عدد صحیح ثابت ۲ را در بالای پشته قرار می دهد.
iconst_3	عدد صحیح ثابت ۳ را در بالای پشته قرار می دهد.
iconst_4	عدد صحیح ثابت ٤ را در بالای پشته قرار می دهد.
iconst_5	عدد صحیح ثابت ٥ را در بالای پشته قرار می دهد.
iconst_m1	عدد صحیح ثابت ۱- را در بالای پشته قرار می دهد.
idiv	دو عدد صحیح بالای پشته را بر هم تقسیم کرده نتیجه را بالای پشته قرار می دهد.

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

iushr	شیفت منطقی به سمت راست.
ixor	دوint 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	دولong را با هم جمع می‌کند.
laload	یک long را از آرایه ای ازlong 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 کـه عملونـد اول بزرگتر از دوم بود، ۱،در صورتیکـه عملونـد دوم بـزرگـتر بود، ۱- و در صورتی کـه بـا هـم برابـر بودنـد ۰ را بـالای پشته push می کند.
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 می کند.
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	دولong را در هم ضرب می کند.
lneg	علامت long را بر عکس می کند.
lor	دو long را با هم or می کند.
lrem	دو long را بر هم تقسیم کرده، باقیمانده تقسیم را در پشـته

	push می‌کند.
lreturn	یک long را از تابع برمی گرداند.
lshl	یک long را به اندازه مشخص شده در بالای پشته شیفت منطقی به سمت چپ می دهد.
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 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 می کند.
lushr	شیفت منطقی به سمت راست
lxor	دو long را با هم xor می کند.
monitorenter	object reference بالای پشته را pop کرده،object آن را lock می‌کند.
monitorexit	object reference بالای پشته را pop کرده،object آن را unlock می‌کند.
nop	هیچ عملی را انجام نمی دهد. این دستور توسط کمپایلرها برای مقاصدی مانند اشکال‌زدایی ایجاد می‌شود.
pop	عنصر بالایی را از پشته pop می‌کند.
pop2	دوخانه از عناصر بالای پشته را pop میکند. (دو int ، یک

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

برای کسب اطلاعات بیشتر، می‌توانید The Java Virtual Machine Specification را از سایت 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 هرسه 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 linits 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 ٤ 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 قبلا توضیح داده شده‌است.

Main method

پس از initialization method ، متد اصلی برنامه آمده است. ایـن متـد 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. بـا توجـه بـه descriptor ، مشخص می‌شود که متد main آرایه ای از string هـا را بـه عنـوان پـارامتر ورودی می‌گیرد و با توجه به V مشخص می‌شود که این متد مقدار بازگشتی ندارد.

بقیه دستورات این مثال، قبلا توضیح داده شده‌اند.

برنامه Count

این مثال، مقداری پیچیده‌تر از مثال قبل است و هدف آن پیاده‌سازی دسـتور 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 HelloWeb 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 مستقیما با فرمان Java HelloWeb.class قابل اجرا نیست. در مثال زیر ما

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

```
.class public HelloWeb
.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 HelloWeb/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 HelloWeb/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. بـرای مثـال فـرض کنیـد می‌خواهیم descriptor یک آرایه سه‌بعدی از نوع double را بنویسیم. تعریف این آرایه و deecriptor آن به صورت زیر است:

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

۲- method descriptor ها

یـک method descriptor پارامترهـا و مقـدار بـازگشـتی یـک متـد را مشخص می‌کند. method descriptor هـا بـا اسـتفاده از گرامـر زیـر نوشـته می‌شوند:

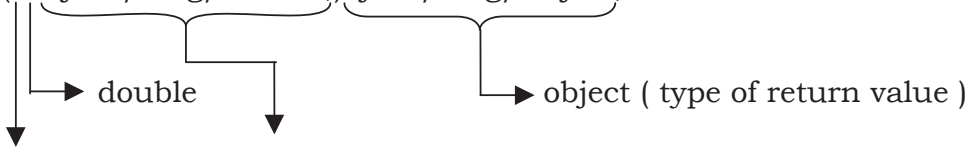
```
MethodDescriptor          ->
        ( ParameterDescriptor * ) ReturnDescriptor
ParameterDescriptor       ->      FieldType
ReturnDescriptor          ->      FieldType
                          |       V
```

در ایـن گرامـر، ParameterDescriptor پارامترهـای ارسـالی بـه یـک متـد را مشخص می‌کند. علامت * در این گرامـر بـه معنـای تکـرار (صفـر مرتبـه یـا بیشتر) می‌باشد. ReturnDescriptor نیز یک رشته کاراکتری است که مقـدار بازگشتی متد را مشخص می‌کند. در ReturnDescriptor ، حـرف V نمـایانگر نوع داده void می‌باشد (یعنی متد هیچ داده‌ای را بازنمی‌گرداند).

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

ضمیمه 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;
  Label1, 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;
```

لطفا هرگونه پیشنهاد یا انتقاد خود در جهت بهبود این مستندات

را با ما در میان بگذارید.

rzakery2001@yahoo.com