

Portability and Safety: Java

سید مرتضیٰ رضوی

پنج شنبه، ۱۴ فروردین ۱۳۹۹



حروف چینی شده با X3 Persian

فصل ۱۳

امنیت و قابلیت حمل: جاوا

زبان برنامه نویسی جاوا^۱ توسط آقای جیمز گاسلینگ^۲ و چند تن دیگر در شرکت Sun Microsystems طراحی شد. این زبان از یک پروژه در سال ۱۹۹۰ بوجود آمد که Oak^۳ نام داشت و بر روی دستگاه هایی مورد استفاده قرار گرفت که به عنوان set-top box نیز شناخته می شدند.

set-top box دستگاه کوچک محاسباتی که به یک شبکه متوسط متصل شده و در بالای یک تلویزیون قرار می گیرد. این دستگاه ویژگی های زیادی فراهم می کند. شما می توانید تصور کنید با این فرض که یک مرورگر وب بر روی تلویزیون شما به نمایش درآمده و به جای استفاده از صفحه کلید، شما با استفاده از یک کنترل بر روی آیکون های آن کلیک می کنید. ممکن است شما بخواهید یک برنامه یا فیلم را انتخاب کنید و یا اینکه یک برنامه شبیه ساز کامپیوتر را که توانایی اجرا بر روی یک دستگاه محاسباتی را داشته باشد دانلود و اجرا کنید و آن را به نمایش درآورید. یا حتی یک تبلیغات تلویزیونی در مورد ماشین می تواند با ایجاد یک شبیه سازی منحصر به فرد رانندگی آن ماشین در جاده در قالب یک تور برای هر بیننده ایجاد کند. هرچند که این سناریو می تواند برای شما جذاب باشد، محیط گرافیکی می تواند شامل گرافیک، اجرای برنامه های ساده و ایجاد ارتباط بین یک سایت و یک برنامه که به صورت داخلی اجرا می شود، باشد.

در نقطه از توسعه Oak، مهندسين و مديران در شرکت سان میکرو سیستم متوجه یک نیاز فوری برای ایجاد یک زبان برنامه نویسی تحت مرورگر شدند، یک زبان که می توانست برای نوشتن برنامه های کوچکی که تحت شبکه منتقل می شدند و بر روی هر مرورگر استاندارد در یک پلتفرم استاندارد اجرا می شدند استفاده شود. و از طرفی نیاز برای استاندارد به دلیل خواسته ذاتی اشخاص و کمپانی ها برای

^۱Java.

^۲James Gosling همچنین به عنوان Dr.java شناخته می شوند.

^۳شرکت سان میکرو سیستم سازنده رایانه و نرم افزار است.

^۴/əʊk/ به معنی بلوط.

داشتن بیشترین مخاطب ممکن بود. علاوه بر قابل حمل بودن، همچنین نیاز به امنیت هم وجود داشت بنابراین کسی که برنامه‌ی کوچکی را دانلود می‌کرد، قادر به اجرای آن بدون ترس از ویروس کامپیوتری یا خطرات دیگر بود.

زبان برنامه نویسی Oak با هدف پیاده سازی دوباره زبان برنامه‌نویسی C++ شروع شد. درحالی که، این طراحی زبان هدف انتهایی پروژه نبود، بلکه تبدیل به تمرکز گروه بر روی آن شد. برخی دلایل طراحی یک زبان جدید در این کتاب بیش از حد نمایشی هستند، اما هنوز برخی نقل قول‌های آموزنده از "The Java Saga" توسط دیوید بلانک در Hot Wierd^۵ (در دسامبر ۱۹۹۵) موجود است:

جیمز گاسلینگ

جیمز گاسلینگ مهندس ارشد و طراحی کلیدی زبان برنامه نویسی و پلتفرم جاوا بود. او هم‌اکنون در مرکز تحقیقات شرکت سان بر روی ابزارهای توسعه نرم افزار فعالیت می‌کند. اولین پروژه وی در شرکت سان، سیستم پنجره NeWS^a بود که برای Sun workstation^b در سال ۱۹۸۰ توزیع شده بود. قبل از پیوستن وی به شرکت سان، گاسلینگ، یک نسخه چند پردازنده از یونیکس^c ساخت، همچنین سیستم پنجره اندرو^d به همراه ابزار آن و خیلی از کامپایلرها و سیستم‌های mail. همچنین او در نظر افراد زیادی سازنده ویرایشگر Emacs است.

تکنیکی با حس شوخ طبعی، گاسلینگ در تصویر سمت راست در حال زدن یک کیک به صورت یک بازیگر با ماسک بیل گیتس است، عکسی که در صفحه سایت خود در <http://java.sun.com/people/jag/> که تا حدی سیاسی شده‌است قرار دارد.

جیمز گاسلینگ مدرک ارشد خود را در رشته کامپیوتر از دانشگاه کلگری^e در کانادا دریافت کرد، او همچنین مدرک دکترای خود را از دانشگاه ملون و با پایان نامه‌ای در عنوان "دستکاری جبری محدودیت‌ها"^f دریافت کرد.

^aمخفف Network extensible Window System.

^b/sʌn 'wɜ:k steɪʃən/ به معنی ایستگاه کاری سان.

^cUNIX /'ju:niks/.

^dAndrew.

^eCalgary, Canada.

^fThe Algebraic Manipulation of Constraints.

^۵ سایت <http://www.hotwired.com/>

گاسلین به سرعت متوجه شد که زبان‌های موجود کارایی مورد نیاز را برای کاری که در سر داشت، ندارند. از جهتی زبان ++C برای برنامه نویسان برنامه‌های تخصصی تقریباً یک زبان نزدیک به استاندارد تبدیل شده بود، جایی که سرعت همه چیز بود. اما ++C برای چیزی که گاسلینگ در سر داشت به حد کافی قابل اطمینان نبود. آن سریع بود، اما رابط‌های آن ناسازگار بودند، و برنامه‌های با خطا موجه می‌شدند. درحالی که برای مصرف‌کنندگان وسایل الکتریکی قابلیت اطمینان از سرعت مهم‌تر است. رابط‌های نرم‌افزاری باید به اندازه یک دوشاخه که مناسب پریز برق است، قابل اطمینان باشند. و اینطور شد که گاسلینگ گفت ”من به این نتیجه رسیدم که به زبان برنامه‌نویسی جدید نیاز دارم“.

برای دلایل متنوعی از جمله تلاش عظیم سان میکرو سیستم، جاوا بعد از انتشار آن به عنوان زبان ارتباط اینترنت در میانه سال ۱۹۹۵ به طور شگفت‌آوری موفق شد. قسمت‌های اصلی جاوا عبارت‌اند از:

- زبان برنامه‌نویسی جاوا.
- کامپایلر و سیستم‌های زمان اجرا (ماشین مجازی جاوا)
- کتابخانه گسترده، شامل جعبه ابزار جاوا برای نمایش‌های گرافیکی و کاربردهای دیگر و نمونه‌های برنامه‌های کوچک جاوا.

اگرچه کتابخانه و مجموعه ابزارها به دسترسی سریع‌تر کمک کردند، اما ما در درجه اول به زبان برنامه نویسی، پیاده سازی آن و نحوه تاثیر گذاری طراحی زبان و پیاده سازی آن علاقه‌مند شدیم. گاسلینگ، در زندگی عادی فروتن تر از آن چیزی است که در نقل قول قبلی نشان می‌دهد است، او درمورد زبان‌هایی که بر روی جاوا تاثیر گذار بودند چنین می‌گوید: ”یکی از مهم‌ترین زبان‌های تاثیر گذار بر روی طراحی جاوا زبانی ابتدایی به نام *Simula* بود. آن اولین زبان شی‌گرا بود که من استفاده کردم (بر روی یک CDC 6400!) ... جایی که مفهوم ’کلاس’ ابداع شده بود“.

۱.۱۳ پیشگفتاری بر زبان جاوا

۱.۱.۱۳ اهداف زبان جاوا

- زبان برنامه‌نویسی جاوا و محیط اجرایی آن با اهداف زیر طراحی شدند.
- قابلیت حمل: برنامه‌ها باید به آسانی بر روی شبکه انتقال پیدا کنند و به درستی بر روی محیط دریافت کننده اجرا شوند، بدون در نظر گرفتن سخت‌افزار، سیستم عامل یا حتی مرورگر مورد استفاده.

- قابلیت اطمینان: به دلیل اینکه برنامه‌ها توسط افرادی از راه دور اجرا می‌شوند که کد را ننوشته‌اند، باید در حد امکان از پیام‌های خطا و قفل شدن برنامه‌ها جلوگیری کرد.
 - امنیت: محیط محاسباتی که برنامه را دریافت می‌کند باید نسبت به خطاهای برنامه‌نویسان و همچنین برنامه‌نویسان مخرب محافظت شده باشد.
 - لینک شدن پویا: برنامه‌ها در بخش‌های مختلفی توزیع شده‌اند، و هر بخش به صورت جدا و در زمان مورد نیاز در محیط اجرایی جاوا اجرا می‌شود.
 - اجرای چندنخی: برای اینکه همزمانی برنامه‌ها بر روی سخت افزارهای متنوع اجرا شوند، زبان باید دارای پشتیبانی صریح و رابط استاندارد برای این عمل باشد.
 - سادگی و آشنایی: زبان باید برای یک برنامه‌نویس متوسط وب جذابیت داشته باشد، معمولاً یک برنامه‌نویس زبان C یا یک برنامه‌نویس که حدوداً با C/C++ آشناست.
 - بازدهی: این امر مهم است، ولی ممکن است جزء ملاحظات ثانویه قرار گیرد.
- در حالت کلی، تمرکز کمتر بر روی بازدهی زبان انعطاف بیشتری را برای برنامه‌نویسان جاوا نسبت به C++ ایجاد کرد.

۲.۱.۱۳. تصمیمات طراحی

برخی از اهداف طراحی و تصمیمات طراحی کلی در [جدول ۱.۱۳](#) قرار دارند، که در آن علامت + به معنی این است که آن تصمیم منجر به تاثیر مثبت در آن هدف، علامت - به معنی تاثیر منفی در آن هدف، و +/- نشان دهنده این است که برخی معایب و برخی مزایا در این تصمیم وجود دارد. همچنین برخی خانه‌ها خالی مانده‌اند، که نشان دهنده بی تاثیر یا کم تاثیر بودن آن تصمیم در آن هدف است. به راحتی می‌توانیم اهمیت بازدهی در فرایند طراحی جاوا را با نگاه کردن به راست‌ترین ستون متوجه شویم. اما این بدین معنی نیست که بازدهی کاملاً بی‌نیاز قربانی شده باشد، و فقط نسبت به دیگر اهداف در اولویت اول قرار نداشت.

مفسری بودن. شروع و بیشتر پیاده‌سازی جاوا بر پایه تفسیر بایت کد^۶ است. که در [بخش ۴.۱۳](#) بیشتر در این مورد بحث شده است. به طور خلاصه برنامه‌های جاوا ابتدا به یک زبان ساده و سطح پایین کامپایل می‌شوند. این زبان، بایت کد نام دارد، به این دلیل که معمولاً زمانی مورد استفاده قرار می‌گیرد که برنامه‌های جاوا در بستر شبکه به عنوان قسمتی از صفحه وب ارسال می‌شوند. بایت‌کدهای جاوا توسط یک مفسر به نام ماشین مجازی جاوا (JVM)^۷ اجرا می‌شوند. یکی از مزایای این معماری این است که هنگامی که JVM برای یک سخت افزار و سیستم عامل خاص پیاده سازی می‌شود، تمام برنامه‌های جاوا می‌توانند بدون تغییر در کد در آن اجرا شوند. علاوه بر قابلیت حمل، تفسیر شدن این بایت‌کدها به اجرای ایمن کمک می‌کنند، و درست زمانی فرمانی نقض شود تشخیص‌گر معنایی زبان جاوا قبل از

^۶byte code.

^۷Java virtual machine , /'dʒɑ:və 'və:tʃu(ə)l mə'ʃi:n/.

جدول ۱۰۱۳: تصمیمات طراحی جاوا

قابلیت حمل	امنیت	سادگی	بازدهی
مفسری بودن	+	+	-
نوع ایمنی	+	-/+	-/+
بیشتر مقادیر شی هستند	-/+	-/+	-
اشیاء اشاره گر هستند	+	+	-
زباله روب	+	+	-
پشتیبانی از همزمانی	+	+	+

اجرا آن را تشخیص می‌دهد. برای یک مثال خوب می‌توان به چک شدن حدود آرایه اشاره کرد. چون تشخیص اینکه یک برنامه به خارج از مرز آرایه دسترسی دارد در زمان کامپایل ممکن نیست. اما، JVM تست‌های زمان اجرایی دارد تا مطمئن شود هیچ برنامه‌ای به خارج از مرز آرایه خود دسترسی ندارد.

نوع ایمنی. سه سطح از نوع ایمنی در جاوا وجود دارد. اولین آن چک کردن سورس کد جاوا در زمان کامپایل است. تشخیص نوع جاوا مانند خیلی از تشخیص نوع‌های مرسوم عمل می‌کند (همانند پاسکال^۸، C++ و غیره)، که از کامپایل برنامه‌هایی که از رویه زبان جاوا پیروی نمی‌کنند جلوگیری می‌کند. هیچ عملیات ریاضی بر روی اشاره‌گرها وجود ندارد، هیچ نوع تبدیل صریحی وجود ندارد و زبان زباله روبی می‌شود، برای مثال اثبات شده است که یک درجه از امنیت نوع بیشتر نسبت به C++ را دارد. و سطح دوم از امنیت نوع، چک کردن نوع قبل از زمانی است که برنامه‌های بایت‌کد جاوا اجرا می‌شوند. و سطح سوم چک کردن نوع در زمان اجرا است، مانند چک کردن حدود آرایه که در زیر بخش قبل توضیح داده شد. به علاوه بر امنیت، سیستم نوع جاوا برخی از ساختارهایی را که ممکن است سیستم معنایی و یا اجرایی زبان را پیچیده کنند، ساده سازی و حذف می‌کند.

اشیاء و ارجاعات^۹. در جاوا حدوداً همه چیز شی هستند، ولی نه همه چیز. به صورت خاص، برخی از انواع معین و پایه مثل اعداد صحیح^{۱۰}، نوع بولین^{۱۱} و رشته‌ها شی نیستند. این مقایسه بین سادگی و کارایی است. در بیان جزئی تر، اگر تمامی عملیات مربوط به اعداد صحیح نیازمند مراجعه پویا به حافظه باشند، به طور قابل ملاحظه‌ای باعث کند شدن محاسبات در اعداد صحیح می‌شود. یک تصمیم ساده در مورد اشیاء باعث شد که تمامی آنها از طریق اشاره گر در دسترس باشند و انتصاب با استفاده از اشاره گر تنها نوع انتصابی است که برای تمام اشیاء فراهم دیده شده. این امر باعث سادگی برنامه‌ها در

^۸یک زبان برنامه نویسی کامپایلری.

^۹References.

^{۱۰}Integer.

^{۱۱}Boolean.

مسائل خاص شده است، اما در برخی شرایط کارایی برنامه را کاهش می‌دهد. تمامی پارامترها در متدهای جاوا با استفاده از مقدار فرستاده می‌شوند. زمانی که یک پارامتر یک نوع ارجاعی دارد (شامل تمامی اشیاء و آرایه‌ها)، اگرچه خودش یک نوع ارجاع است، اما کپی شده و با مقدار فرستاده می‌شود. در واقع، این یعنی که مقادیر نوع‌های اولیه با مقدار و اشیاء با ارجاع ارسال می‌شوند.

زباله روب. همانطور که در زیر بخش ۱۰.۲.۶ بحث شد، زباله روب برای کامل کردن ایمنی نوع نیاز است. زباله روب همچنین باعث ساده شدن برنامه نویسی می‌شود، که با حذف آن قسمت از کد که باید تشخیص داده شود که چه لحظه‌ای باید بازپس‌گیری حافظه انجام شود، اما باعث کندی اجرا می‌شود. زباله روب جاوا از مزایای همزمانی در برنامه استفاده می‌کند، به این حالت که در پس زمینه اجرای برنامه و به صورت یک رشته با اولویت پایین پیاده‌سازی شده است، که این اجازه را به زباله روب می‌دهد تا زمانی اتفاق بیافتد که کاربر متوجه افت سرعت نشود.

پیوند پویا. کلاس‌هایی که در جاوا تعریف و استفاده شده‌اند ممکن است به صورت مکرر در JVM بارگزاری شوند، درست زمانی که آنها مورد نیاز برنامه اجرایی باشند. این کار زمان سپری شده بین شروع انتقال برنامه بر بستر شبکه و زمان شروع اجرای آن را کوتاه‌کند، درحالی که برنامه درست قبل از اینکه تمام کد منتقل شود می‌تواند اجرا شود. علاوه بر این در صورتی که برنامه‌ای بدون اینکه به کلای نیاز داشته باشد خاتمه یابد در اینصورت آن کلاس در JVM بارگزاری و حتی انتقال داده نمی‌شود. پیوند پویا تاثیر چندانی بر روی طراحی زبان ندارد، غیر از نیاز برای رابط‌های شفاف که بتوانند قسمت‌هایی از یک برنامه را تحت برخی فرضیات نسبت به قسمتی از کد فراهم شده در بخش دیگر برنامه چک کنند.

پشتیبانی از همزمانی. جاوا یک مدل همزمانی بر پایه رشته‌ها^{۱۲} دارد، که به فرایندهای همزمان وابسته است. این یک قسمت مهم از زبان است، به دو دلیل طراحی اساسی و اهمیت داشتن اصول همزمانی استاندارد به عنوان قسمتی از زبان جاوا است. به طور واضح اگر همزمانی جاوا وابسته به مکانیسم وابسته به سیستم عامل بود، به هیچ عنوان نمی‌توانست بین دستگاه‌های دیگر قابلیت حمل داشته باشد.

سادگی. اگرچه جاوا در طی سال‌ها رشد کرده و ویژگی‌هایی مانند کلاس‌های داخلی و بازتاب به نظر ساده نیستند، اما این زبان هنوز هم خیلی ساده‌تر و کوچک‌تر از دیگر زبان‌های همه منظوره و برپایه کیفیت تولید است. یکی از راه‌هایی که که متوجه بشید زبان جاوا ساده است این است که لیست ویژگی‌هایی از C++ را که در جاوا وجود ندارند را ببینید. که این شامل ویژگی‌های زیر می‌شود:

- ساختارها و اتحادها: ساختارها در رده اشیاء قرار می‌گیرند، و برخی از کاربردهای اتحادها را می‌توان با کلاس‌هایی که از یک کلاس ارث‌بری شده‌اند جایگزین کرد.
- توابع را می‌توان با متدهای ایستا جایگزین کرد.
- ارث‌بری چندگانه پیچیده است و در بیشتر مواقع اگر یک مفهوم رابط ساده‌تر از جاوا استفاده

¹²threads.

شود میتوان از آنها پرهیز کرد.

- **Goto** ضروری نیست.
 - سربارگذاری عملگرهای پیچیده است و غیر ضروری تلقی می‌شود، توابع جاوا می‌توانند سربارگذاری شوند.
 - تبدیل‌های خودکار پیچیده هستند و غیر ضروری تلقی می‌شوند.
 - اشاره‌گرها به طور پیشفرض برای اشیاء وجود دارند و برای بقیه نوع‌ها نیاز نیستند، پس در نتیجه یک نوع اشاره‌گری جدا نیاز نیست.
- برخی از این ویژگی‌ها که در C++ موجود هستند به دلیل سازگاری رو به عقب با C است. و بقیه به دلایل برخی تصمیمات از جاوا حذف شده‌اند، به این دلیل که آنها تصمیم گرفتند که پیچیدگی اضافه کردن آن ویژگی‌ها خیلی بیشتر از عملکرد آنها مورد توجه است. و قابل توجه‌ترین حذف ویژگی مربوط به ارث‌بری چندگانه، تبدیل‌های خودکار، سربارگذاری عملگرها و عملیات اشاره‌گر در C و C++ هستند.

۲.۱۳ کلاس‌های جاوا و ارث‌بری

۱.۲.۱۳ کلاس‌ها و اشیاء

جاوا در به نحو C++ نوشته شده است، پس در نتیجه برنامه‌نویسی آن هم به برنامه نویسان C و C++ نزدیک تر است. این باعث می‌شود که کد نقطه یک بعدی که در زیربخش ۱.۳.۱۲ آمده است، به کدی که به جاوا تبدیل شده است خیلی شبیه‌تر باشد. که در اینجا یک ورژن خلاصه شده از کلاس با حذف متد move را داریم:

```
1 class Point {
2     public int getX() { ...}
3     protected void setX (int x) { ...}
4     private int x;
5     Point(int xval) {x = xval;}
6 };
```

مانند خیلی از زبان‌های دیگر کلاس محور، جاوا هم داده‌ها و توابع را با تمام شی‌هایی که توسط این کلاس ساخته می‌شوند مرتبط می‌کند. و زمانی که یک شی ایجاد می‌شود فضای داده‌های آن اختصاص داده شده و سازنده برای مقدار دهی اولیه مقادیر فراخوانی می‌شود. مانند C++ سازنده نامی برابر با نام کلاس دارد. و همچنین مانند C++ اجزاء در کلاس point دسترسی‌های public، private و protected را دارند. همچنین public، private و protected کلمات کلیدی جاوا هستند، و همانطور که در زیربخش ۲.۲.۱۳ اشاره شده، این خصوصیات دسترسی به معنی شبیه بودن این دو زبان نیست. واژه شناسی جاوا اندکی با Simula، Smalltalk و C++ متفاوت است، که در اینجا یک خلاصه

کوتاه از مهم‌ترین واژه‌های استفاده شده در توصیف جاوا را بررسی می‌کنیم:

- کلاس و شیء، اساساً همانند دیگر زبان‌های برپایه کلاس و شیء‌گرا به یک معنی هستند.
- فیلد،^{۱۳} داده عضو، متد^{۱۴} : توابع عضو، عضو ایستا^{۱۵} : مشابه به فیلد کلاس و یا متد کلاس در Smalltalk، `this`^{۱۶} : مانند `this` در C++ و یا `self` در Smalltalk مشخصه `this` در بدنه متدهای جاوا به معنی شیء است که این متد را فراخوانی کرده است.
- متدهای بومی: متدهایی که در زبان دیگری نوشته می‌شوند، مانند C
- پکیج: یک مجموعه از کلاس‌ها که یک فضای نام را به اشتراک می‌گذارند.

ما خیلی از مشخصات کلاس‌ها در جاوا را شامل فیلدها و متدهای ایستا، سربارگذاری، متدهای پایانی، متد `main`، متد `toString` که برای چاپ یک نما از شیء و همچنین امکان استفاده از کدهای بومی را مشاهده کردیم. ما در ادامه بحث‌هایی حول زمان اجرای جاوا از اشیاء و پیاده‌سازی متد `lookup` در بخش ۴.۱۳ که در رابطه با ایجاد یک ارتباط به جنبه‌های دیگر ساختار زمان اجرای سیستم است، خواهیم داشت.

مقدار دهی اولیه. جاوا تضمین می‌کند، زمانی که یک شیء ساخته می‌شود یک سازنده هم فراخوانی می‌شود. و به دلیل اینکه با ارث بری مسائلی ایجاد می‌کند، از قسمت در **زیربخش ۳.۲.۱۳** مفصل بحث شده است.

متدها و فیلدهای ایستا. متدها و فیلدهای ایستا در جاوا مشابه به متد و فیلد در Smalltalk است. اگر یک فیلد به صورت ایستا تعریف شود، در نتیجه آن فیلد برای کل کلاس مشترک است و نه برای هر یک شیء و اگر متدی ایستا تعریف شود در نتیجه این متد بدون یک شیء از کلاس می‌تواند فراخوانی شود. در حالت کلی، متدهای ایستا می‌توانند حتی قبل از اینکه شیء از کلاس ایجاد شوند، فراخوانی شوند. در نتیجه متدهای ایستا تنها قابلیت دسترسی به فیلدهای ایستا و بقیه متدهای ایستا را دارند. آنها نمی‌توانند به شیء `this` ارجاع کنند زیرا هیچ قسمتی از شیء خاصی از کلاس نیستند. در خارج از یک کلاس عضو ایستا با اسم خود کلاس فراخوانی می‌شود که فراخوانی آن به صورت `class_name.static_method(args)` است و نه از طریق یک شیء از کلاس. فیلدهای استاتیک می‌توانند با یک عبارت یا یک بلوک مقدار دهی ایستا، مقدار دهی شوند. که هر دو در قطعه کد زیر نمایش داده شده‌اند.

¹³field.

¹⁴Method.

¹⁵static member.

¹⁶ اشاره‌گری به شیء جاری

```

1 class ... {
2     /* --- static variable with initial value --- */
3     static int x = initial value;
4     /* --- static initialization block --- */
5     static { /* code to be executed once, when class is loaded */
6     }
7 }

```

همانطور که از کامنت‌های برنامه مشخص است، بلوک مربوط به ایستا در کد فقط یک بار زمانی که کلاس بارگذاری می‌شود اجرا می‌شود. بارگذاری کلاس در بخش ۴.۱۳ در اتصال به JVM مفصل بحث شده است. قوانین مختلفی بر ترتیب اجرای مقداردهی‌های ایستا حاکم است. زمانی که یک کلاس شامل هر دو مقداردهی، هم نوع عبارت و هم بلاک ایستا است. حتی محدودیت‌هایی هم بر مقداردهی ایستا وجود دارد. برای

مثال، یک مقداردهی ایستا نمی‌تواند یک حالت استثنا ایجاد کند، به این دلیل که هیچ اطمینانی وجود ندارد که یک کنترل‌کننده متناظر در هنگام بارگذاری کلاس برای آن ایجاد شود.

سربارگذاری. سربارگذاری در جاوا برحسب امضاء متد است، که شامل اسم متد، تعداد پارامترها و نوع هر پارامتر ورودی است. و همچنین اگر دو متد از یک کلاس (چه هر دو در یک کلاس اعلان شده باشند یا هر دو ارث‌بری شده باشند و یا یکی ارث‌بری شده و دیگری اعلان شده باشد) اسم یکسان ۷ اما امضاء متفاوتی داشته باشند، در نتیجه متد نام آنها سربارگذاری شده است. و همانند دیگر زبان‌ها در زمان کامپایل چک می‌شود.

زباله روب و نهایی کردن. به دلیل اینکه جاوا زباله روبی می‌شود، هیچ نیازی بر آزاد سازی صریح اشیاء نیست. به علاوه، برنامه نویسان نیازی به نگرانی در مورد ارجاع‌های زائدی که توسط بازپس‌گیری حافظه انجام می‌شود ندارند. هرچند زباله روب فقط فضایی که توسط یک شیء ایجاد شده است را باز پس‌گیری می‌کند. اما اگر شیء دسترسی بر نوع خاصی از منبع را نگه دارد، در نتیجه این منبع باید زمانی که این شیء دیگر در دسترس نیست آزاد شود. به همین دلیل، اشیاء در جاوا، متد `finalize` دارند که بر اساس دو شرط فراخوانی می‌شوند، به وسیله زباله روب، زمانی که فضا بازپس‌گیری می‌شود و به به وسیله ماشین مجازی، زمانی که ماشین مجازی خاتمه می‌یابد. یکی از عرف‌های مفید متد `finalize` فراخوانی `super.finalize` است. همانطور که در ادامه نشان می‌دهیم، هر قطعه کد خاتمه‌ای مربوط به کلاس پدر نیز فراخوانی می‌شوند:

```

1 class ... {
2     ...
3     protected void finalize () {
4         super.finalize();
5         close(file);
6     }
7 };

```

همچنین یک بازخورد جالب بین متد `finalize` و استثناء‌ها وجود دارد، بدین‌گونه که در صورت رخ دادن هر استثناء مدیریت نشده‌ای آن استثناء نادیده گرفته می‌شود.

یک مشکل مرتبط با متد `finalize` کنترل نداشتن صریح برنامه‌نویس زمان فراخوانی متد `finalize` است. که این تصمیم‌گیری به سیستم زمان اجرا واگذار شده است. و در صورتی که یک شیء یک منبع مشترک را در دسترس داشته باشد مشکل ساز است، برای مثال، چنانچه که قفل دسترسی ممکن است تا زمانی که زباله روب تشخیص ندهد که برنامه به فضای بیشتری نیاز دارد آزاد نشود. که یک راه‌حل گذاشتن عملگرهایی از جمله آزاد سازی تمامی قفل دسترسی‌ها^{۱۷} یا منابع دیگر در متدی که صریحاً در برنامه فراخوانی می‌شود. این مورد تا زمانی که درستی کار می‌کند که تمامی کاربران استفاده کننده از کلاس اسم متد را بدانند و به یاد داشته باشند که این متد را زمانی که شیء، دیگر به منابع نیاز نداشت فراخوانی کنند.

برخی از دیگر جنبه‌های جذاب اشیاء و کلاس‌های جاوا، متدهای `main` که برای شروع اجرای برنامه استفاده می‌شود، متدهای `toString` که برای تولید یک نمایش چاپی از شیء به کار می‌رود، و امکان تعریف متدهای بومی است:

■ `main` : ک برنامه جاوا از کلاسی فراخوانی می‌شود که اسم آن از اسم برنامه مشتق گرفته شده است. این کلاس باید یک متد `main` داشته باشد که باید دسترسی `public` و در نوع `static`^{۱۸} باشد و نوع `void`^{۱۹} را برگرداند. همچنین یک تک آرگومان از نوع `String[]` نیز دریافت کند. متد `main` در یک آرایه رشته به همراه برنامه فراخوانی شده است. هر کلاسی با یک متد `main` می‌تواند به صورت مستقیم فراخوانی شود مثل اینکه یک برنامه مستقل است، که می‌تواند برای امتحان مناسب باشد.

■ `toString` : یک کلاس ممکن یک متد `toString` تعریف کند، و زمانی فراخوانی می‌شود که یک تبدیل نوع به رشته نیاز باش، مانند چاپ یک شیء.

■ **متدهای بومی** : یک متد بومی، متدی است که در زبان دیگر نوشته می‌شود، مانند C با استفاده از متدهای بومی قابلیت حمل و امنیت برنامه کاهش پیدا می‌کند. کدهای بومی را نمی‌توان بر حسب تقاضا بر روی شبکه منتقل کرد، و کنترل‌های موجود در JVM به دلیل تفسیر نشدن کد با ماشین مجازی، بی‌اثر هستند. دلایم استفاده از کدهای بومی (۱) کارایی کد شیء بومی و (۲) دسترسی به کد و برنامه‌هایی که در حال حاضر در زبان دیگر نوشته شده‌اند، هستند.

¹⁷lock.

^{۱۸}ایستا.

^{۱۹}پوچ.

۲.۲.۱۳ بسته‌ها و رؤیت

جاوا چهار رؤیت متفاوت برای فیلدها و متدها دارد، که سه مورد آنها مربوط به سطوح رؤیت در C++ و چهارمین آن ناشی از وجود بسته‌ها است.

بسته‌های جاوا کپسوله سازی مشابه با فضای نام ^{۲۰} C++ دارند که اجازه می‌دهند، اعلان‌های مرتبط با برخی اعلان‌های دیگر یک دسته شوند و از بسته‌های دیگر مخفی باشند. در یک برنامه جاوا، هر فیلد و متد متعلق به یک کلاس خاص است و هر کلاس بخشی از یک بسته است، همانطور که در شکل ۱۰.۱۳ آمده است، یک کلاس می‌تواند متعلق به یک بسته بی‌نام پیش فرض، یا برخی از بسته‌های دیگر که در فایلی که شامل کلاس است مشخص شده‌اند، باشد.

تمایزات رؤیت ^{۲۱} در جاوا حالت‌های زیر هستند:

- عمومی ^{۲۲} : قابل دسترس در همه جا و کلاس قابل رؤیت است.
- محافظت شده ^{۲۳} : قابل دسترس در متدهای کلاس و هر زیر کلاسی، و همچنین به کلاس‌های دیگر در بسته یکسان.
- خصوصی ^{۲۴} : قابل دسترس فقط در خود کلاس.
- بسته ^{۲۵} : قابل دسترس فقط در کد با بسته یکسان، و در زیرکلاس‌های بسته‌های دیگر قابل رؤیت نیست. اعضای که با یک مشخصه دسترسی خاصی اعلان نشده‌اند، فقط دسترسی بسته را دارند.

به عبارت دیگر، یک متد می‌تواند به عضوی از کلاسی که به آن تعلق دارد، به اعضای غیر خصوصی همه کلاس‌ها در یک بسته، اعضای محافظت شده از کلاس‌های پدر (شامل کلاس‌های پدر در بسته‌های دیگر)، و همچنین اعضای عمومی همه کلاس‌ها در هر بسته قابل رؤیتی، مراجعه کند.

نام‌های که در بسته‌های دیگر اعلان شده‌اند می‌توانند به وسیله `import` قابل دسترس باشند، که اعلان‌های دیگر بسته‌ها را وارد می‌کند، یا اسامی واجد شرایط که در حالت زیر هستند، که نشان می‌دهد، بسته صریحاً شامل اسم است:

```
java.lang.String.substring()
package    class    method
```

²⁰ namespaces.

²¹ visibility.

²² public.

²³ protected.

²⁴ private.

²⁵ package.

۳.۲.۱۳ ارث‌بری

در واژه‌شناسی، یک زیر کلاس از کلاس پدر خود ارث‌بری می‌کند. که مکانیسم ارث‌بری جاوا ذاتاً مشابه ارث برای در Smalltalk، C++، دیگر زبان‌های برپایه کلاس و شیء کرا است. که نحو نگارش آن در ارث‌بری مشابه C++، اما با کلیدواژه `extends` است. همانطور که در این مثال نشان داده شده است، کلاس `ColorPoint` از کلاس `Point` در زیربخش ۱.۲.۱۳ گسترش یافته است:

```
1 class ColorPoint extends Point {
2     // Additional fields and methods
3     private Color c;
4     protected void setC (Color d) {c = d;}
5     public Color getC() {return c;}
6     // Define constructor
7     ColorPoint(int xval, Color cval) {
8         super(xval); // call Point constructor
9         c = cval;
10    } // initialize ColorPoint field
11 };
```

برجسته‌سازی متد^{۲۶} و پنهان سازی فیلدها. همانطور که در زبان‌های دیگر، یک کلاس تمامی فیلدهای کلاس پدر خود را به ارث می‌برد، بجز زمانی که یک فیلد یا متد به همان نام در زیر کلاس مورد نظر اعلان شده باشد. زمانی که یک متد در کلاس فرزند با یک متد در کلاس پدر هم نام باشد، زیر کلاس آن متد را با همان امضاء برجسته می‌کند. در یک برجسته‌سازی نوع بازگشتی متد نباید با متدی که برجسته^{۲۷} می‌شود، با برگشت دادن نوع داده دیگر تداخل داشته باشد. یک متد برجسته شده می‌تواند به وسیله کلمه کلیدی `super` قابل دسترس باشد. برای فیلدها، یک اعلان فیلد در کلاس فرزند با اسم مشابه، تمامی فیلدهای کلاس پدر با همان نام را پنهان می‌کند. یک فیلد می‌تواند با یک نام توصیف شده (اگر نوع آن ایستا باشد) یا با استفاده از یک عبارت دسترسی به فیلد که شامل یک تبدیل به نوع کلاس پدر است و یا با استفاده از کلیدواژه `super` قابل دسترس باشد.

سازنده‌ها. جاوا ضمانت می‌کند، زمانی که یک شیء ایجاد شود سازنده آن نیز فراخوانی می‌شود. در کامپایل شدن سازنده زیر کلاس، کامپایلر چک می‌کند که حتماً سازنده کلاس پدر نیز فراخوانی شود. که اینکار به روشی خاصی که عموماً برنامه‌نویسان در نظر دارند، انجام می‌شود. بخصوص زمانی که در خط ابتدایی یک سازنده فراخوانی سازنده کلاس پدر انجام نشده است، در نتیجه کامپایلر فراخوانی تابع `super()` را در خط اول اضافه می‌کند. البته این حالت همیشه درست عمل نمی‌کند، زیرا اگر کلاس پدر سازنده‌ای با هیچ آرگومان^{۲۸}

^{۲۶} برتر سازی متد overriding.^{۲۷} override /əʊvə'raɪd/.^{۲۸} متغیرهایی که در ورودی تابع تعبیه می‌شوند تا بتوان به هنگام فراخوانی مقادیر را به تابع فرستاد.

This does not always work well, because, if the superclass does not have a constructor with no arguments, the call `super()` will not match a declared constructor, and a compiler error results. An exception to this check occurs if one constructor invokes another. In this case, the first constructor does not need to call the superclass constructor, but the second one must. For example, if the constructor declaration `ColorPoint() ColorPoint(0,blue);` is added to the preceding `ColorPoint` class, then this constructor is compiled without inserting a call to the superclass `Point` constructor. A slight oddity of Java is that the inheritance conventions for `finalize` are different from the conventions for constructors. Although a call to the superclass is required for constructors, the compiler does not force a call to the superclass `finalize` method