Portability and Safety: Java

–

–

فصل ۱۳

امنیت و قابلیت‌حمل: جاوا

زبان برنامه‌نویسی جاوا [۱] توسط آقای جیمز گاسلینگ [۲] و چند تن دیگر در شرکت Sun Microsystems [۳] طراحی شد. این زبان از یک پروژه در سال ۱۹۹۰ بوجود آمد که Oak [۴] نام داشت و بر روی دستگاه‌هایی مورد استفاده قرار گرفت که به عنوان set-top box نیز شناخته می‌شدند.

set-top box دستگاه کوچک محاسباتی که به یک شبکه متوسط متصل شده و در بالای یک تلویزیون قرار می‌گیرد. این دستگاه ویژگی‌های زیادی فراهم می‌کند. شما می‌توانید 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 آیکون‌های آن کلیک می‌کنید. 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 می‌تواند 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 می‌تواند برای شما جذاب باشد، محیط گرافیکی می‌تواند شامل گرافیک، اجرای برنامه‌های ساده و ایجاد ارتباط بین یک سایت و یک برنامه که به صورت داخلی اجرا می‌شود، باشد.

در نقطه از توسعه Oak ،مهندسین و مدیران در شرکت سان میکرو سیستم متوجه یک نیاز فوری برای ایجاد یک زبان برنامه نویسی تحت مرورگر شدند، یک زبان که می‌توانست برای نوشتن برنامه‌های کوچکی که تحت شبکه متنقل می‌شدند و بر روی هر مرورگر استاندارد در یک پلتفرم استاندارد اجرا می‌شدند استفاده شود. و از طرفی نیاز برای استاندارد به دلیل خواسته ذاتی اشخاص و کمپانی‌ها برای

---

[1] Java.

[۲] James Gosling همچنین به عنوان Dr.java شناخته می‌شوند.

[۳] شرکت سان میکرو سیستم سازنده رایانه و نرم‌افزار 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، گاسلینک 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 گاسلینک 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.

---

[a] مخفف Network extensible Window System.

[b] /sʌn ˈwɜːkˌsteɪʃən/ به معنی ایستگاه کاری سان.

[c] UNIX /ˈjuːnɪks/.

[d] Andrew.

[e] Calgary, Canada.

[f] The 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 مصرف کنندگان 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 ازجمله 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 تأثیر 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 تأثیر گذار بودند چنین می‌گوید: " یکی از مهم‌ترین زبان‌های تأثیر گذار بر روی طراحی جاوا زبانی ابتدایی به نام *Simula* بود. آن اولین زبان شئ‌گرا 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 برنامه نویس متوسط وب جذابیت داشته باشد، معمولاً یک برنامه‌نویس زبان C یا یک برنامه نویس که حدوداً با C/C++ آشناست.
- *بازدهی*: این امر مهم است، ولی ممکن است جزء ملاحظات ثانویه قرار گیرد.

درحالت 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 تأثیر مثبت در آن هدف، علامت - به معنی تأثیر 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 تأثیر یا کم تأثیر 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.

جدول ۱.۱۳: تصمیمات طراحی جاوا

	قابلت حمل	امنیت	سادگی	بازدهی
مفسری بودن	+	+		-
نوع ایمنی	+	+	+/-	+/-
بیشتر مقادیر شئ هستند	+/-	+/-	+	-
اشیاء اشاره‌گر هستند	+		+	-
زباله روب	+	+	+	-
پشتیبانی از همزمانی	+	+		

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 اجرا آن را تشخصی 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 مطمعن 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 شئ هستند، ولی نه همه چیز. به صورت خاص، برخی از انواع معین و پایه مثل اعداد صحیح [۱۰] ، نوع بولین [۱۱] و رشته‌ها شئ 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 اشاره‌گر در دسترس باشند و انتصاب با استفاده از اشاره‌گر تنها نوع انتصابی است که برای تمام اشیاء فراهم دیده شده. این امر باعث سادگی برنامه‌ها در مسائل خاص شده است، اما در برخی شرایط کارایی برنامه را کاهش می‌دهد.

---

[6] byte code /baɪt kəʊd/.

[7] Java virtual machine , /ˈdʒɑːvə ˈvəːtʃʊ(ə)l məˈʃiːn/.

[۸] یک زبان برنامه نویسی کامپایلری.

[9] References.

[10] Integer.

[11] Boolean.

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 ۱.۲.۶ 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 بازپس گیری 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 پیاده سازی 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 کوتاه کند، درحالی که برنامه درست قبل از اینکه تمام کد منتقل شود می‌تواند اجرا شود. علاوه براین درصورتی 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 تأثیر 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 همزممانی 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 ارث بری شده‌اند جایگزین کرد.
- *توابع* را می‌توان با متدهای ایستا جایگزین کرد.
- *ارث بری چندگانه* پیچیده است و در بیشتر مواقع اگر یک مفهوم رابط سادهتر از جاوا استفاده شود میتوان از آنها پرهیز کرد.

[12] threads.

- 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 شده اند، به این دلیل که آنها تصمیم گرفتند که پیچیدگی اضافه‌کردن آن ویژگی‌ها خیلی بیشتر از عملکرد آنها مورد توجه است. و قابل توجه ترین حذف ویژگی مربوط به ارث‌بری چندگانه، تبدیل‌های خودکار، سربارگذاری عملگرها و عملیات اشاره‌گر در C و C++ هستند.

۲.۱۳ کلاس‌های جاوا و ارث بری

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 شئ‌هایی که توسط این کلاس ساخه می‌شوند مرتبط می‌کند. و زمانی که یک شئ 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.

اصلاحات جاوا اندکی با Simula، Smalltalk و C++ متفاوت است، که در اینجا یک خلاصه کوتاه از مهم‌ترین اصلاحات استفاده شده در توصیف جاوا را بررسی می‌کنیم:

- *کلاس* و شئ، اساساً همانند دیگر زبان‌های برپایه کلاس و شئ‌گرا 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 جاوابه معنی شئ 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 شئ یه 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 ارتبطا به جنبه‌های دیگر ساختار زمان اجرای سیستم است، خواهیم داشت.

مقدار دهی اولیه. جاوا تضمین می‌کند، زمانی که یک شئ ساخته می‌شود یک سازنده‌هم 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 فیل در Smalltalk است. اگر یک فیلد به صورت ایستا تعریف شود، درنتیجه آن فیل برای کل کلاس مشترک است و نه برای هر یک شئ و اگر متدی ایستا تعریف شود در نتیجه این متد بدون یک شئ از کلا می‌تواند فراخوانی شود. در حالت کلی، متدهای ایستا می‌توانند حتی قبل ز اینکه شئ از کلاس ایجاد شوند، فراخوانی شوند. در نتیجه متدهای ایستا تنها قابلیت دسترسی به فیلدهای ایسستا و بقیه متدهای ایستا را دارند. آنها نمی‌توانند به شئ this ارجاع کنند زیرا هیچ قسمتی از شئ 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 شئ 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.

```
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   }
```

[13] field.

[14] Method.

[15] static member.

[۱۶] اشاره‌گری به شئ جاری

```
7 }
```

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

مثال، یک مقداردهی ایستا نمی‌تواند یک حالت استثنا ایجاد کند، به این دلیل که هیچ اطمینانی وجود ندارد که یک کنترل کننده 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 دوو متد از یک کلاس (چه هردو در یک کلاس اعلان شده باشند یا هر دو ارث بری شده باشند و یا یکی ارث بری شده و دیگری اعلان شده باشد) اسم یکسان۷ اما امضاء متفاوتی داشته باشند، در نتیجه متد نام آنها سربار گذاری شده است. و همانند دیگر زبان‌ها در زمان کامپایل چک می‌شود.

زباله روب و نهایی‌کردن. 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 بازپس گیری حافظه انجام می‌شود ندارند. هرچند زباله روب فقط فضایی که توسط یک شئ ایجاد شده است را باز پس گیری می‌کند. اما اگر شئ دسترسی بر نوع خاصی از منبع را نگه دارد، در نتیجه این منبع باید زمانی که این شئ دیگر در دسترس نیست آزاد شود. به همین دلیل، اشیاء در جاوا، متد `finilize` دارند که بر اساس دو شرط فراخوانی می‌شوند، به وسیله زباله روب، زمانی که فضا بازپس گیری می‌شود و به به وسیله ماشین مجازی، زمانی که ماشین مجازی خاتمه می‌یابد. یکی از عرف‌های مفید متد `finilize` فراخوانی `super.finilize` است. همناطورکه 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 `finilize` و استثناها وجود دارد، بدین‌گونه که در صورت رخ‌دادن هر استثنا مدیریت نشده‌ای آن استثنا نادیده گرفته می‌شود.

یک مشکل مرتبط با متد `finilize` کنترل نداشتن صریح برنامه‌نویس زمان فراخوانی متد `finilize` است. که این تصمیم گیری به سیستم زمان اجرا واگذار شده است. و درصورتی که یک

شئ 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 به 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 شئ، 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 شئ 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 شئ.
- *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 شئ یومی و (۲) دسترسی به کد و برنامه‌هایی که در حال حاظر در زبان دیگر نوشته شده‌اند، هستند.

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.

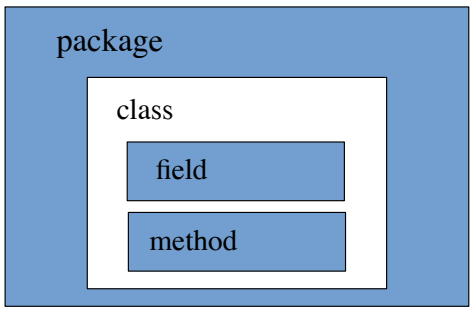
بسته‌های جاوا کپسوله سازی مشابه با فضای نام C++ [۲۰] دارند که اجازه می‌دهند، اعلان‌های مرتبط با برخی اعلان‌های دیگر یک دسته شوند و از بسته‌های دیگر مخفی باشند. در یک برنامه جاوا، هر فیلد و

[17] lock.

[۱۸] ایستا.

[۱۹] پوچ.

[20] namespaces.



شکل ۱.۱۳: بسته‌های جاوا و سطوح رؤیت کلاس.

متد متعلق به یک کلاس خاص است و هر کلاس بخشی از یک بسته است، همانطورکه در شکل ۱.۱۳ 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 همه‌جا 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. اعضایی 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 صریحا شامل اسم است:

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

[21] visibility.
[22] public.
[23] protected.
[24] private.
[25] package.

۳.۲.۱۳ ارث بری

در اصلاح، یک زیر کلاس از کلاس پدر خود ارث بری می‌کند. که مکانیسم ارث بری جاوا ذاتاً مشابه ارث برای در Smalltalk، C++ ، دیگر زبان‌های برپایه کلاس و شئ گرا است. که نحو نگارش آن در ارث بری مشابه C++ ، اما با کلیدواژه `extends` است. همانطور که در این مثال نشان داده‌شده 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 زمانی‌که 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، زیرکلاس 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 شئ ایجاد شود سازنده آن نیز فراخوانی می‌شود. در کامپایل شدن سازنده زیرکلاس، کامپایلر چک می‌کند که حتماً سازنده کلاس پدر نیز فراخوانی شود. که اینکار به روشی خاصی که عموماً برنامه‌نویسان درنظر 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 آرگومانی [۲۸] وجود نداشته باشد. فراخوانی متد `super()` با سازنده‌های اعلان

[۲۶] برتر سازی متد overriding.

[27] override /əʊvəˈrʌɪd/.

[۲۸] متغیره‌هایی که در ورودی تابع تعبیه می‌شوند تا بتوان به هنگام فراخونی مقادیر را به تابع فرستاد.

شده همخوانی نخواهد داشت، و در نتیجه باعث خطای زمان کامپایل می‌شود. یک استثنا در این مورد زمانی است که یک سازنده سازنده دیگری را فراخوانی کند. در این حالت سازنده اول نیازی به فراخوانی سازنده کلاس پدر را ندارد، اما سازنده دوم باید این کار را انجام دهد. برای مثال، اگر چنین سازنده کلاسی `ColorPoint() { ColorPoint(0,blue);}` به کلاس `ColorPoint` که قبلاً معرفی شد اضافه شود. در نتیجه این سازنده بدون اضافه کردن فراخوانی از سازنده کلاس پدر کامپایل می‌شود. یک اعجاب جزئی جاوا متفاوت بودن رفتارهای ارث بری در دو متد finalize[۲۹] و سازنده است، که کامپایلر هیچ اجباری برای فراخوانی متد finalize کلاس پدر در متد finalize زیرکلاس وجود ندارد.

متد نهایی [۳۰] ***و کلاس‌ها.*** جاوا دارای مکانسیم‌های جذابی برای محدود سازی زیرکلاس‌های یک کلاس دارد: به این ترتیب که یک متد یا تمام یک کلاس می‌تواند به صورت final اعلان شود. اگر کلاس final باشد در نتیجه، آن کلاس نمی‌توند هیچ زیر کلاسی داشته باشد. دلیل این ویژگی این است که اگر یک برنامه نویس بخواهد تمام رفتارهای تمامی شیءهای یک نوع را تعریف کند. به دلیل اینکه یک زیرکلاس زیرنوع‌هایی ایجاد می‌کند، همانطور که در بخش ۳.۱۳ بررسی خواهد شد، اینکار نیازمند چند محدودیت بر روی زیرکلاس‌ها است. برای نشان دادن یک مثال مناسب، الگوی یکتایی [۳۱] است که در بخش بخش ۴.۱۰ مورد بررسی قرار گرفت، که نشان داد چطور کلاسی طراحی کنیم که فقط یک شئ از کلاس بتواند ایجاد شود. این الگو سازنده را مخفی می‌کند و فقط یک تابع عمومی ایجاد می‌کند که در حین اجرای برنامه فقط یک بار می‌تواند سازنده را فراخوانی کند، اما فقط اگر هیچ زیرکلاسی متدی عمومی را برای ایجاد بیشتر از یک شئ override نکند. اگر برنامه نویس واقعاً بخواهد الگوی یکتایی اجبار کند، باید راهی برای جلوگیری از برنامه نویسان برای تعریف زیرکلاس از یک کلاس یکتا، باشد.

کلاس جاوا `java.lang.System` یک مثال دیگر برای یک کلاس final است. این کلاس final است، تا برنامه نویسان متدهای سیستم را override نکنند.

به عبارت دیگر، متد final جاوا برعکس توابع مجازی [۳۲] C++ است: متدهای جاوا تا زمانی که به عنوان final معرفی نشده باشند، می‌توانند override شوند، اگر چه توابع عضو C++ فقط زمانی می‌توانند override شوند که به عنوان تابع مجازی در نظر گرفته شوند. این قیاس خیلی دقیق نیست، هرچند که یک تابع عضو C++ نمی‌تواند همزمان در یک کلاس مجازی باشد و در کلاس پدر خود و یا در کلاس مشتق شده آن مجازی نباشد، زیرا، شکل یکسان مورد نیاز جدول مجازی برای کلاس‌های پایه و مشتق شده را نقض می‌کند.

کلاس Object. در اصل، هر کلاس که در جاوا اعلان می‌شود یک گسترش [۳۳] از کلاسی دیگر است، بدین صورت که هر کلاسی که صراحتاً از کلاس دیگری ارث برای نشود به عنوان یک زیر کلاس

---

[۲۹] نهایی سازی.

[30] Final method.

[31] singleton

[32] virtual functions.

[33] extend, inherited

از کلاس Object در نظر گرفته می‌شود. کلاس Object یک کلاس است که هیچ کلاس پدری ندارد. کلاس Object شامل متدهای زیر است که در کلاس‌های مشتق شده override می‌شوند:

- GetClass

  که یک شئ کلاس را که نمایانگر شئ کلاس است را بر می‌گرداند. که می‌تواند برای پی بردن به اسم کاملاً توصیف شده کلاس، اعضای آن کلاس، کلاس پدر بی واسطه آن و رابط‌هایی که کلاس پیاده سازی می‌کند استفاده کرد.
- ToString یک رشته که نمایانگر شئ است برمی‌گرداند.
- equals که یک ارزش از مفهوم شئ تعریف می‌کند، که مرجع مقایسه نیست.
- hasCode که یک مقدار برای ذخیره سازی شئ در جداول درهم [۳۴] را برمی‌گرداند.
- clone که برای ایجاد یک کپی از شئ استفاده می‌شود.
- متدهای wait، notify و notifyAl که در برنامه نویسی همزمانی استفاده می‌شود.
- finalize که درست زمانی که یک شئ از بین برود اجرا می‌شود (که در زیر بخش ۱.۲.۱۳ مرتبط با زباله روب مورد بررسی قرار گرفت.)

به دلیل اینکه تمامی کلاس‌ها از متدهای کلاس Object ارث بری می‌کنند، در نتیجه هر شئی تمامی متدهایی که در بالا ذکر شد را دارد.

۴.۲.۱۳ کلاس‌های انتزاعی و رابط‌ها

مکانیزم کلاس‌های *انتزاعی* [۳۵] زبان جاوا مشابه با زبان C++ است. همانطورکه ما در فصل ۱۲ بررسی کردیم. کلاس انتزاعی کلاس است که تمامی متدهای آن نیاز به پیاده سازی ندارند و از طرفی دیگر هیچ نمونه‌ای نمی‌تواند داشته باشد. جاوا از کلیدواژه abstract به جای نحو "=0" در C++ استفاده می‌کند، که در کد زیر نشان داده شده است:

همچنین جاوا یک قالب از کلاس "انتزاعی خالص" را نیز دارد که رابط [۳۶] نامیده می‌شود. یک رابط مانند یک کلاس تعریف می‌شود، بجز اینکه تمام اعضای یک رابط باید ثابت یا متد انتزاعی باشند. یک رابط هیچ پیاده سازی مستقیمی ندارد، اما کلاس‌ها ممکن است طوری اعلان شوند که یک رابط را implement [۳۷] کنند. به علاوه، یک رابط ممکن است به عنوان یک افزونه برای سایر کلاس‌ها باشد، که یک قالب از ارث برای رابط است.

یکی از دلایل اینکه برنامه نویسان جاوا به جای کلاس‌های انتزاعی خالص از رابط و در زمانی که

[34] hash table.

[35] abstract.

[36] interface.

[۳۷] در اینجا به عنوان یک کلمه کلیدی به کار رفته است.

یک مفهوم تعریف شده اما پیاده سازی نشده است، استفاده می‌کنند، این است که جاوا اجازه می‌دهد که یک کلاس مجزا چندین رابط را implement کند، درحالی که یک کلاس فقط می‌تواند یک کلاس پدر داشته باشد. در کلاس و رابط‌های زیر این حالت به نمایش درآمده است. رابط Shape چند ویژگی از یک شکل هندسی ساده را مشخص می کند، برای مثال هرکدام دارای یکی نقطه مرکزی و یک متد rotate[۳۸] هستند. و همچنین Drawable چند ویژگی از شئ را که می‌تواند بر روی صفحه به نمایش درآید را نشان می‌دهد. اگر دایره یک شکل هندسی است پس همچنین می‌تواند بر روی صفحه به نمایش درآید، در نتیجه همانطور که در ادامه نشان داده شده است، کلاس Circle می‌تواند به نحوی اعلان شود که هر دو Shape و Drawable را implement کند.

```
1 interface Shape {
2      public Point center();
3      public void rotate(float degrees);
4 }
5 interface Drawable {
6   public void setColor(Color c);
7   public void draw();
8 }
9 class Circle implements Shape, Drawable {
10   // does not inherit any implementation
11   // but must define Shape, Drawable methods
12 }
```

برخلاف ارث بری چندگانه در C++ (که در بخش ۵.۱۲ بررسی شد)، هیچ مشکل تصادم اسم در رابط‌های جاوا وجود ندارد. به طور خاص، فرض کنید که دو رابط Shape و Circle تعریف شده در قبل هر دو متد Size را پیاده سازی کرده‌اند. اگر متدهای Size هردو تعداد آرگومان با نوع یکسان داشته باشند. در نتیجه کلاس Circle باید یک متد Size با این تعداد آرگومان و نوع آرگومان موجود در دو رابط پیاده سازی کند. از طرف دیگر، اگر دو متد Size آرگومان‌هایی با تعداد و یا نوع متمایز داشته باشند، در نتیجه آنها به عنوان دو متد متفاوت در نظر گرفته شده و کلاس Circle باید متد Size را برای هر کدام را پیاده سازی کند. به این خاطر که جستجوی متد جاوا از نام و تعداد و نوع آرگومان‌ها برای انتخاب کد متد استفاده می‌کند، دو متد با نام‌های یکسان در زمان جستجوی متدها به صورت مجزا تلقی می‌شوند.

[۳۸] دوران.

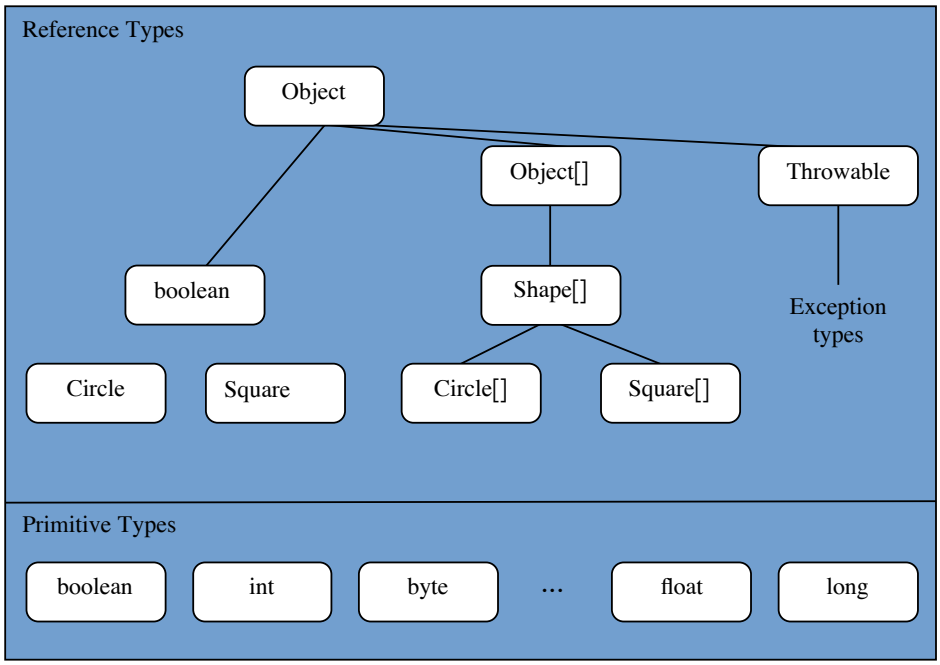
۳.۱۳ نوع‌ها و زیر نوع‌های جاوا

۱.۳.۱۳ طبقه بندی نوع‌ها

نوع‌های جاوا در دو دسته تقسیم می‌شوند: نوع‌های اولیه و نوع‌های ارجاعی. هشت نوع اولیه به نوع `boolean` و هفت نوع دیگر از انواع عددی در فرم `integers byte`، `short` ، `int`، `long`، `char` و نوع‌های اعشاری `float` و `double` هستند. سه نوع در انواع ارجاعی وجود دارند، نوع‌های `class`، `interface` (که اخیراً بحث شد) و نوع آرایه. و همچنین نوع خاص `null` هم موجود است. مقادیر یک نوع ارجاعی یک ارجاع به شئ است (که شامل آرایه نیز می‌شود). تمام اشیاء، شامل آرایه‌ها، متدهای کلاس `Object` را پشتیبانی می‌کنند.

رابطه زیر نوع بین نوع‌های اصلی در شکل ۲.۱۳ به نمایش درآمده‌اند، که شامل رابط‌های `Shape` و کلاس‌های `Circle` و `Square` تا نشان بدهند که چطور کلاس‌ها و رابط‌های تعریف شده توسط کاربر در تصویر گنجانده شده‌اند. مابقی نوع‌های پیش تعریف شده مانند: `Strings`، `ClassLoader` و `Thead` در جایگاهی مانند `Exception` در این شکل دارند. `Object[]` نوع آرایه‌ای از `object` ها است. و همینطور برای نوع‌های آرایه `Shape[]`، `Circle[]` و `Square[]` نیز همینطور است.

همچنین این یک استاندارد از اصلاحات جاوا برای فراخوانی `Object` و نوع‌های ارجاعی زیرنوع آن است، که ممکن است کمی گیج کننده باشد. همچنین C++ نوع `Object` را از `Object *` متمایز می‌کند، که در جاوا ما هیچ نوع اشاره‌گری نداریم. درعوض تفاوت صریحی بین یک اشاره‌گر به شئ و خود شئ، بدر دسترسی به اشاره‌گر ترکیب شده با عمل‌هایی مانند فراخوانی متد و یا دسترسی به فیلد وجود دارد. اگر `T` یک نوع ارجاعی باشد، درنتیجه متغیر `x` از نوع `T` یک ارجاع به اشیاء `T` است، که در C++ متغیر `x` نوع `T *` را دارد. زیرا هیچ راه صریحی برای دسترسی به شئ `x` برای گرفتن مقدار نوع `T` نیست، جاوا هیچ نوع جدایی برای اشیائی که اشاره‌گر نباشند ندارد. به این دلیل که هر کلاس یک



شکل ۲.۱۳: طبقه بندی نوع‌ها در جاوا.

زیرنوع از `Object` است، متغیرهای از نوع `Object` می‌توانند به اشیاء و یا آرایه‌ای از هر شئ اشاره کنند.

۲.۳.۱۳ زیر نوع سازی برای کلاس‌ها و رابط‌ها

زیر نوع بودن برای کلاس‌ها به وسیله کلاس‌های سلسه مراتبی و مکانیسم رابط مشخص می‌شود. اگر کلاس `A` از کلاس `B` ارث بری شود، زیرنوع‌های کلاس `A` زیرنوع‌های کلاس `B` نیز هستند. هیچ راهی برای تعریف یک کلاس از نوع دیگر برای یک کلاس وجود ندارد. و هیچ کلاس خصوصی (مانند C++) اجازه ارث بری بدون زیرنوع بودن را نمی‌دهد.

یک کلاس ممکن است اعلان شده باشد تا یک یا چند رابط را `implement` کند، بدین معنی که هر نمونه از کلاس تمامی متدهای انتزاعی مشخص شده در رابط را پیاده سازی می‌کند. این زیرنوع شدن از رابط(ها) به اشیاء اجازه پشتیبانی از یک (یا چند) نوع رفتار بدون خاص بدون اشتارک گذاری یک پیاده سازی مشترک را می‌دهد.

تبدیل نوع زمان اجرا. جاوا اجازه هیچ نوع تبدیل بررسی نشده‌ای را نمی‌دهد، هرچند اشیاء یک نوع پدر ممکن است با استفاده از یک مکانیسم که شامل آزمون نوع زمان اجرا می‌شود به زیر نوع‌ها تبدیل شوند. اگر شما نیازمند ایجاد یک کلاس لیست در جاوا هستید، شما طوری آن را می‌سازید که

اشیاء نوع Object را نگه دارد. اشیاء هر کلاسی می‌توانند داخل یک لیست قرار بگیرند، اما برای جداسازی و استفاده از آنها به صورت غیر مستقیم، باید به نوع اصلی خود تبدیل شوند (یا برخی نوع پدر). در جاوا، تبدیل نوع در زمان اجرا بررسی می‌شود، و اگر شئ نوع تبدیل شده را نداشته باشد یک استثنا ایجاد می‌شود.

پیاده سازی. جاوا از یک ساختار بایت‌کد متفاوت برای جستجو اعضا بر اساس رابط و کلاس یا زیر کلاس استفاده می‌کند. در کامپایلر با عملکرد بالا، جستجو بر اساس کلاس می‌تواند مانند C++ ، به وسیله جابه‌جایی [۳۹] شناخته شده در زمان کامپایل پیاده سازی شود. هرچند، امکان جستجو بر اساس رابط نمی‌تواند چنین باشد، زیرا یک کلاس ممکن است چندین رابط را پیاده سازی ([۴۰]) کرده باشد و ممکن است رابط‌ها با ترتیب دیگری اعضا را لیست کنند. که در زیر بخش ۴.۴.۱۳ به جزییات پرداخته شده است.

۳.۳.۱۳ آرایه‌ها،

برای هر نوع T ،جاوا یک نوع آرایه T[] از آرایه‌ای که عناصرش از نوع T هستند دارد. همچنین نوع‌های آرایه با کلاس‌ها و رابط‌ها هم سطح هستند. امکان اینکه از یک آرایه ارث بری شود وجود ندارد، آرایه‌ها در اصلاحات جاوا نوع نهایی هستند.

نوع‌های آرایه زیر نوعی از Object هستند، از این رو تمامی متدهای مرتبط با کلاس Object را ساپورت می‌کنند. مانند نوع‌های ارجاعی دیگر، یک متغیر آرایه یک اشاره‌گر به یک آرایه است و می‌تواند null باشد. و زمانی که یک آرایه اعلان می‌شود، مرسوم است که یک شئ از آرایه نیز ایجاد کنیم، مانند:

```
1 Circle[] x = new Circle[array size]
```

هرچند، این امکان وجود دارد که یک آرایه را "ناشناس" ایجاد کنیم، به همان روشی که ما می‌توانیم دیگر اشیاء جاوا را ایجاد کنیم. برای مثال:

```
1 new int[] {1,2,3, ...10}
```

که یک عبارت برای ایجاد یک آرایه از اعداد صحیح به طول ۱۰، با مقادیر ۱، ۲، ۳، ...، ۱۰ است. به این دلیل که یک متغییر از نوع T[] می تواند با یک آرایه از هر اندازه‌ای مقدار دهی شود، طول آرایه جزئی از نوع ایستای آن نیست.

برخی عوارض در طریقه‌ای که نوع‌های آرایه جاوا در سلسله مراتب زیر نوع قرار گرفته است، وجود دارند. قابل توجه‌ترین آن‌ها این است که اگر A <: B [۴۱] درنتیجه بررسی کننده جاوا همچنین از زیر نوع

[39] offset n /ˈɒfˌsɛt/

[40] implement

[۴۱] به معنی اینکه A یک زیرکلاس از B است.

A[] <: B[] نیز برای بررسی استفاده می‌کند. که این مشکلی ایجاد می‌کند که اغلب از آن به عنوان کواریانس آرایه یاد می‌شود. برای مثال، اعلان کلاس و آرایه زیر را درنظر بگیرید:

```
1 class A {...}
2 class B extends A {...}
3 B[] bArray = new B[10]
4 A[] aArray = bArray // considered OK since B[ ] <: A[ ]
5 aArray[0] = new A() // allowed but run-time type error; raises
    ↪ ArrayStoreException
```

در این کد، ما B <: A را داریم، زیرا کلاس B از کلاس A ارث بری می‌کند. آرایه ارجاعی bArray به یک آرایه از اشیاء B اشاره می‌کند، و همه را null مقداردهی می‌کند، و آرایه ارجاعی aArray نیز به همان آرایه اشاره می‌کند. اعلان A[] aArray = bArray به وسیله بررسی کننده نوع جاوا مجاز است (اگرچه از لحاظ معنایی نباید مجاز باشد) به این خاطر که در تصمیمات طراحی جاوا اگر B <: A در نتیجه B[] <: A[]. مشکل اجازه دادن اینکه آرایه ارجاعی A[] به آرایه‌ای از نوع اشیاء B اشاره کند در آخرین دستور نشان داده شده است. انتصاب aArray[0] = new A(); به نظر کاملاً منطقی است: که aArray یک آرایه از نوع ایستای A[] است، نشان می‌دهد که انتصاب یک شئ A در هر اندیس آرایه قابل قبول است. هرچند به خاطر اینکه aArray به یک آرایه از اشیاء B اشاره می‌کند، این انتصاب نوع bArray را نقض می‌کند. به خاطر اینکه این انتصاب باعث ایجاد مشکلات نوع می‌شود، ساختار جاوا اجازه اجرای چنین انتصابی را نمی‌دهد. یک آزمایش زمان اجرا مشخص می‌کند که مقداری که به آرایه از اشیاء B انتصاب داده شده است یک شئ B نیست و استثنا ArrayStoreException رخ می‌دهد.

همچنین طراحان زبان جاوا کواریانس آرایه را برای برخی اهداف مشخص سودمند دانستند (نوشتن برخی روال‌های کپی باینری)، با اینحال کواریانس آرایه در جاوا باعث ایجاد برخی سردرگمی‌ها و آزمون‌های زمان اجرای زیادی می‌شود. که به نظر نمی رسد یک تصمیم طراحی موفق زبان باشد.

۴.۳.۱۳ سلسله مراتب

جاوا ممکن است استثنائات را اعلان، ایجاد، و کنترل کند.

مکانیسم استثناء جاوا دارای ویژگی های کلی است که در بخش ۲.۸ در مورد آنها صحبت کردیم. استثنائات جاوا ممکن است ناشی از دستور throw در برنامه کاربر و یا نتیجه برخی از حالت‌های خطا که توسط ماشین مجازی تشخیص داده شده‌اند باشد، مانند دسترسی خارج از محدوده یک آرایه. در اصطلاحات جاوا، یک استثنا از جایی که رخ داده است پرتاب [۴۲] می شود و باید در جایی که کنترل

[42] thrown

منتقل شده است گرفته [۴۳] شود. مانند زبان‌های دیگر، پرتاب یک استثنا باعث می‌شود که جاوا هر عبارت، دستور، فراخوانی متد یا سازنده، مقدار دهی، و مقدار دهی فیلدهای یک عبارت که اجرای آن شروع شده است اما پایان نیافته است را متوقف کند. این عمل تا زمانی که یک کنترل کننده مطابق با کلاس استثنا اتفاق افتاده پیدا نشود ادامه پیدا می‌کند.

یک جنبه جالب از مکانیسم استثنا جاوا، طریقه ارتباط آن با کلاس و سلسه مراتب نوع است. هر استثنا جاوا نمایانگر یک نمونه از کلاس Throwable یا یکی از زیر کلاس‌های آن است. یک مزیت نمایانگر بودن استثناها به عنوان شئ حمل اطلاعات از نقطه‌ای که پرتاب می‌شوند به نقطه‌ای که یک کنترل کننده آن را می‌گیرد است. زیر نوع بودن همچنین می‌تواند کمک شایانی در کنترل استثنا داشته باشد: به این صورت که زمانی یک کنترل کننده با استثنا مطابقت می‌کند که صراحتاً هم نام کلاسی که استثنا را پرتاب کرده یا همنام با یک کلاس پدر از کلاس استثنا باشد.

مکانیزم استثنا جاوا به گونه‌ای طراحی شده است که در برنامه‌های چند نخی (همزمانی) به خوبی کار کند. هنگامی که یک استثنا پرتاب می‌شود، فقط نخی که استثنا در آن رخ داده، تحت تأثیر قرار می‌گیرد. تأثیر یک استثنا در مکانیسم هماهنگ سازی همزمانی این است که قفل کردن به عنوان دستورات هماهنگ منتشر می‌شود و فراخوانی متدهای هماهنگ کاملاً ناگهانی انجام می‌شود. که ما در فصل ۱۴ به این موضوع باز می‌گردیم.

استثنائات جاوا در ساختاری به نام بلاک try-finally گرفته می‌شوند. و اینجا یک مثال ساختار آورده شده است، که یک بلاک با دو کنترل کننده نشان می‌دهد، و هر کدام با کلیدواژه catch مشخص می‌شوند. واضح است که، یک بلاک try-finally سعی می‌کند تا یک دنباله از دستورات را اجرا کند. اگر دنباله دستورات به طور معمول خاتمه یابد، سپس آنجا پایان نتیجه بلوکه خواهد بود. هرچند اگر یک استثنا رخ دهد، ممکن است داخل بلاک گرفته ود. اگر یک استثنا رخ دهد و گرفته شود، سپس دنباله دستورات کلمه کلیدی finally بعد از اینکه کنترل کننده استثنا به اتمام رسید، اجرا خواهد شد:

```
try {
   <statements>
} catch (<ex-type1>?identifier1>) {
   <statements>
} catch (<ex-type2><identifier2>) {
   <statements>
} finally {
   <statements>
}
```

اگرچه ممکن است به طور واضح مشخص نباشد، اما در JVM مشکلاتی با بلاک‌های try-finally وجود دارد. مخصوصاً، بخش قابل توجهی از پیچیدگی‌های تأیید bytecode جاوا، نتیجه پیاده سازی

[43] catch

قسمت‌های `finally` به عنوان "زیر روال محلی" (به نام `jsr`) در مفسر `bytecode` جاوا است. ما JVM را در بخش ۴.۱۳ مورد بررسی قرار خواهیم داد.

JVM the in complication some is there why، apparent be not may it Although of complexity the of fraction significant a Specifically، blocks. x with sociated as- as implemented are clauses finally way the of result a is verifier bytecode Java the will We interpreter. bytecode Java the in (jsr (called subroutine" "local of form a Figure in shown are exceptions of classes The .۴.۱۳ Section in JVM the discuss Throw- of subclass some of object an tion، defini- by is، exception Every .۳.۱۳ the use can Programs . Object of subclass direct a is Throwable class The . able exception additional define or statements throw in classes exception preexisting one or Throwable of classes sub- be must classes exceptions Additional classes. check- compile-time platform's Java the of advantage take To subclasses. its of as classes tion excep- new most define to typical is it handlers، exception for ing sub- not are that Exception of subclasses are These classes. exception checked "unchecked and exceptions checked phrases The . RuntimeException of classes be may that exceptions of set the of checking time compile- to refer exceptions" The program. Java a in thrown