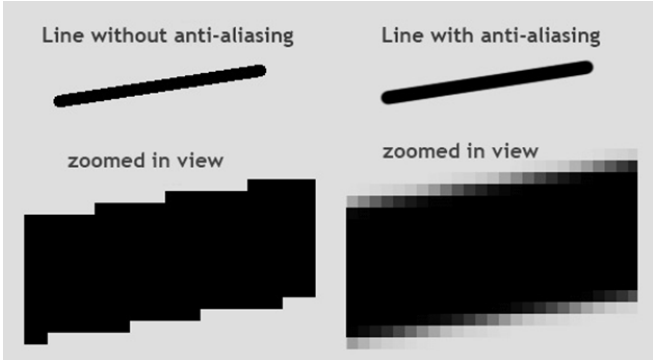
موتور بازی‌سازی

امیرحسین رازلیقی

در ایـن بخـش کدنامـه تلاشـمون اینـه کـه کمـی در مـورد game engines یـا موتورهـای بازی‌سـازی صحبـت کنیـم. اول از همـه، بیایـن در مـورد یـک موضـوع ابتدایـی فکـر کنیـم. ایـن صحنه‌هـای گرافیکـی کـه داریـم روی موبایـل و کامپیوترهامـون می‌بینیـم، چطـور بـه وجـود میـان؟ ایـن سـوال، سرچشـمه‌ی اصلـی سلسـله تغییـرات بنیـادی‌ای بـود کـه منجـر شـد تـا تـوی سـال‌های کمـی، از صفحـه‌ی ترمینـال سـاده‌ی سیسـتم عامـل DOS بـه ایـن GUIهـای جـذاب و متنـوع امـروزی برسـیم! امـا داسـتان بـه ایـن سـادگی‌ها هـم نبـوده! ممکنـه براتـون اصـلا قابـل درک نباشـه ولـی یکـی از سـخت‌ترین کارهایـی کـه در ابتـدا مهندسـین کامپیوتـر می‌خواسـتن انجـام بـدن، رسـم یـک خـط بـود! بلـه، یـک خـط سـاده :) ماجـرا اینـه کـه بـرای رسـم چیـزی مثـل خـط، بایـد یـک الگوریتـم مشـخص رو اجـرا کنیـم: اگـر خـط از نقطـه ی x گـذر می‌کنـه، بایـد پیکسـل متناظـر ایـن نقطـه رو روی صفحـه ی کامپیوتـر، رنگـی کنیـم و اگـر نـه، بذاریـم بـه همـون حالتـی کـه هسـت باقـی بمونـه و تغییـری تـوش ندیـم.

حـالا مشـکل اصلـی چیـه؟ ایـن فرآینـد بـه نظـر ساده‌سـت ولـی مشـکلاتی کـه طـی ایـن فرآینـد پیـش میـان، نیـاز بـه الگوریتم‌هـای خلاقانـه‌ای بـرای حـل دارن! مثـلا تصویـر زیـر رو نـگاه کنیـد. خطـی کـه در بـالای تصویـر، سـمت چـپ رسـم شـده، خطیـه کـه بـا همیـن الگوریتـم سـاده‌ای کـه گفتیـم رسـم شـده. اگـر کمـی دقـت کنیـن، میبینیـن کـه ایـن خـط یکمـی دچـار «اعوجـاج» شـده! دلیلـش هـم مشـخصه. صفحـه‌ی کامپیوتـر ، یـک صفحـه‌ی پیوسـته نیسـت. بلکـه از چندیـن پیکسـل (گسسـته) تشـکیل شـده. ایـن گسسـتگی باعـث میشـه چنیـن مشـکلی پیـش بیـاد (اسـم تخصصـی ایـن اتفـاق هـم aliasing هسـتش!) راه حلـش چیـه؟ (منطقـا اسـمش میشـه anti-aliasing)

بـرای رفـع ایـن مشـکل، الگوریتم‌هـای زیـاد و خلاقانـه‌ای پیشـنهاد شـده کـه یکـی از سـاده ترین هـاش رو بـا هـم بررسـی میکنیـم. بیایـن بـه جـای اینکـه بـه رسـم خـط بـه صـورت ۰ و ۱ ی نـگاه کنیـم (یـا رسـم کنیـم یـا نکنیـم)، بـه شـکل یـک متغیـر پیوسـته نـگاه کنیـم. یعنـی چـی؟ یعنـی اگـر یـه نقطـه ی x داریـم، بـا توجـه بـه فاصلـه‌ش از پیکسـل‌های متناظـرش، بیایـم اونهـا رو رنـگ آمیـزی کنیـم (نزدیـک تریـن پیکسـل میشـه پررنـگ تریـن و دورتریـن میشـه کمرنـگ تریـن). بـه ایـن صـورت، پیکسـل‌های مـا رنگ‌هایـی تـوی طیـف مشـخصی میشـن (مثـلا بیـن سـفید و سـیاه) و ایـن باعـث میشـه مشـکل aliasing کمتـر بشـه. عجیبـه ولـی واقعیـه! بـه تصویـر زیـر (خـط سـمت راسـت) نـگاه کنیـد تـا نتیجـه‌ی ایـن کار رو ببینیـن.



خـب حـالا کـه تونسـتیم کمـی بـا هـم وارد دنیـای «گرافیـک کامپیوتـری» بشـیم، بیایـن و تـوی ایـن تاریخچـه جـذاب، کمـی جلوتـر بریـم! از جایـی بـه بعـد، یکـی از بزرگتریـن صنایـع دنیـا (کـه بـه پشـتوانه علـم گرافیـک کامپیوتـری ایجـاد شـد)، صنعـت بازی‌سـازی شـد. بـه طـوری کـه امـروزه ادعـا میشـه کـه میـزان سـود خالـص بزرگتریـن شـرکت‌های صنعـت بازی‌سـازی از سـود خالـص هالیـوود هـم بیشـتره! درواقـع میشـه گفـت کـه بازی‌سـازی، نـوع متعالـی‌ای از جلـوه گرافیـک کامپیوتریـه :)

جالبـه بدونیـد بـا پیشـرفت صنعـت بازی‌سـازی، چالش‌هـاش هـم پیشـرفت کـردن و از چیـزی کـه دیدیـم بزرگتـر و پیچیده‌تـر شـدن! یکـی از معـروف تریـن چالش‌هـای امـروزه صنعـت گیـم، PhotoRealistic Rendering هسـت. یعنـی بتونـن طـوری بازی‌هـا و asset هـای داخلـش رو بسـازن و عرضـه کننـ کـه اگـر کسـی ببینـه، نتونـه تشـخیص بـده کـه سـاختگیه (و فکـر کنـه کـه ایـن یـه فیلـم ضبـط شـده از یـه اتفـاق تـوی دنیـای واقعیـه!)

حـالا سـوالی کـه احتمـالا تـوی ذهنتونـه اینـه کـه بـا چـه ابزارهایـی، کارهایـی بـه ایـن پیچیدگـی رو تولیـد میکننـ و برنامـه نویسـی میکننـ؟! بـه ابزارهایـی کـه برنامه‌نویس‌هـا (یـا بهتـره بگـم game developer هـا) باهاشـون بـه توسـعه دادن بازی‌هـای خودشـون می‌پـردازن، Game Engine یـا موتـور بازی‌سـازی میگـن. بعضـی از معـروف تریـن موتورهـای بازی‌سـازی کـه امـروزه داره اسـتفاده میشـه رو در تصویـر زیـر می‌بینیـن:

ماجرا اینه که این موتورهای بازی‌سازی ، با ویژگی‌هایی که دارن به برنامه‌نویس‌ها کمک می‌کنن تا بتونن راحت‌تر مشکلات پایه‌ای مربوط به گرافیک کامپیوتری رو حل کنن و بتونن روی مشکلات پیچیده‌تر و جذاب‌تر بازیشون تمرکز کنن و سعی کنن اون‌ها رو حل کنن! دوتا از معروف‌ترین این game engine ها، Unity و Unreal Engine هستن. موتور بازی‌سازی Unity از زبان C# بعنوان زبان اصلی برای ساخت بازی‌هاش استفاده می‌کنه و موتور Unreal Engine از زبان C++ استفاده می‌کنه. البته قابل توجه هست که بگم شرکت‌های بزرگ گرافیک مثل Adobe و شرکت‌های بزرگ بازی‌سازی مثل Ubisoft، برای خودشون به صورت اختصاصی، موتور توسعه دادن تا بتونن کارهای گرافیکی خاص و جذاب خودشون رو با دست باز توسعه بدن. بعنوان نکته نهایی این رو بگم که امروزه این موتورهای معروف (مثل همین Unreal Engine)، دیگه فقط برای ساخت بازی استفاده نمیشن! بلکه برای استفاده‌های پیچیده‌تر مرتبط با گرافیک کامپیوتری (مثل واقعیت مجازی و واقعیت افزوده(Augmented Reality/Virtual Reality) و حتی تست های شبیه‌سازی مربوط به ماشین‌های خودران(self-driving cars)) هم مورد استفاده قرار می‌گیرن! اگر علاقمندین که بیشتر راجع به قدرت این موتورهای بازی‌سازی بدونین، پیشنهاد می‌کنم ویدیوی معرفی Unreal Engine ۵ که سال ۲۰۲۰ معرفی شده رو ببینین! ضمنا اگر در نهایت متوجه شدین که به این حوزه علاقه‌ی زیادی دارین و میخواین راجع به فرآیندهای پشت پرده‌ی گرافیک کامپیوتری بدونین، می‌تونین درس «گرافیک کامپیوتری» که توی دانشکده‌مون هست رو پاس کنین!

لینک‌های مرتبط:

تریلر موتور بازی‌سازی unreal: تریلر اول، تریلر دوم

آشنایی با Sass

سروش شرافت

حتما برای تمرین گرافیک یا پروژه خود تا حالا با CSS دست و پنجه نرم کرده‌اید و از سختی و زبان‌نفهمی آن شکایت دارید. واقعیت این است که هیچ راه‌حل ساده‌ای برای stylesheet پروژه‌ها وجود ندارد، اما بهترین انتخاب شما می‌تواند کار با زبان برنامه‌نویسی Sass باشد که بخش زیادی از کار را خودش بر عهده میگیرد.

Sass چیست؟

زبان Sass یک preprocessor language و یک extension برای CSS است. یعنی کد Sass شما توسط node js و پکیج Sass، به CSS تبدیل می‌شود. دقت کنید Sass تنها در مراحل توسعه برنامه به کار می‌رود و درنهایت CSS حاصل‌از آن در برنامه استفاده می‌شود.

چرا Sass؟

اگر با CSS کار کرده باشید به‌خوبی از معایب آن آگاهید. بسیاری از کدنویسان حرفه‌ای از کار با CSS به‌عنوان یکی از بزرگ‌ترین چالش‌های خود نام می‌برند. با توجه به این‌که Sass درنهایت به CSS تبدیل می‌شود، برای نوشتن کد Sass لازم است که بر CSS تسلط کافی را داشته باشید. اما استفاده از این تکنولوژی، باعث می‌شود maintainability کد و سرعت توسعه به‌شدت افزایش پیدا کند. امکانات Sass مانند ماژولارکردن stylesheet و import از فایل‌های stylesheet مختلف، استفاده از متغیرها و ثابت‌ها، استفاده از فانکشن‌ها، قابلیت extend کلاس‌ها و ... درمقایسه با CSS معمولی، به آن برتری چشم‌گیری می‌بخشد، به‌طوری‌که امروزه در بسیاری از پروژه‌های بزرگ، دیگر کدنویسان خود را درگیر نوشتن CSS نمی‌کنند.

نصب و استفاده

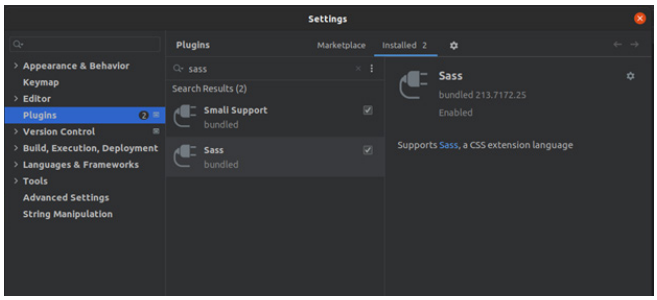
برای کار با Sass ابتدا لازم است که node js را برروی دستگاه خود نصب کنید. بسته‌به سیستم‌عامل و ترجیح شخصی خود، می‌توانید node js را دانلود کنید یا ازطریق package manager خود نصب کنید. پس از نصب node js، با اجرای command زیر در ترمینال، می‌توانید Sass را به‌صورت global نصب کنید:

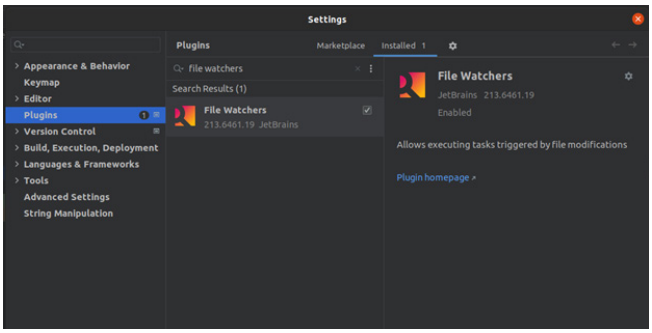
```
npm install -g sass
```

فایل‌های Sass می‌توانند دو فرمت sass. و scss. را داشته باشند. تفاوت اصلی این دو فرمت، در استفاده از براکت برای مشخص‌کردن بلاک‌های کد است. اکثر کدنویسان از فرمت scss. استفاده می‌کنند که استفاده آن از براکت، خوانایی کد را افزایش می‌دهد. ما نیز همین کار را می‌کنیم.

برای تبدیل کد Sass به CSS ابزارهای مختلفی وجود دارد. امکان انجام این کار در ترمینال نیز هست اما برای سهولت کار، پیشنهاد ما استفاده از امکانات IDE/editor مورداستفاده‌تان است. اکثر IDEها و editorهای پیشرفته، ابزاری برای کار با این فایل‌ها دارند، اما ما در این‌جا فقط IntelliJ شرکت جت‌برینز را توضیح می‌دهیم. برای نشان‌دادن چگونگی انجام این کار، CSS استفاده‌شده در کد کارگاه گرافیک سال پیش را با Sass جایگزین می‌کنیم.

ابتدا مطمئن شوید پلاگین‌های Sass و File Watchers روی IntelliJ نصب شده است.

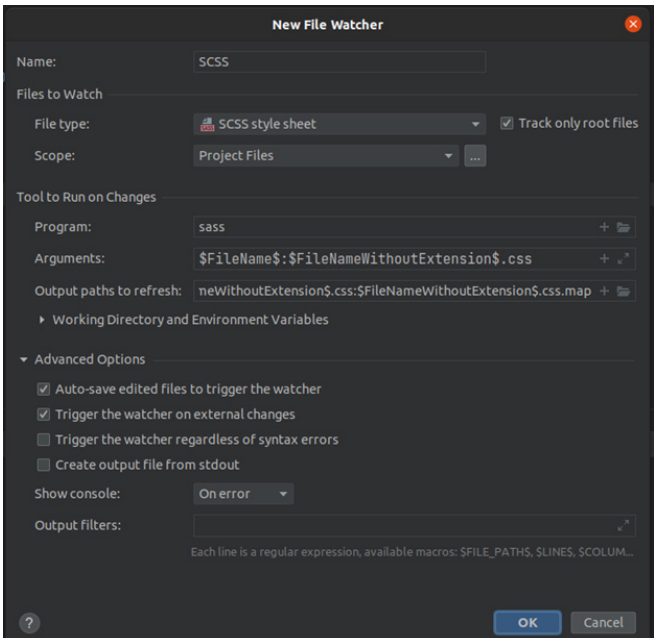




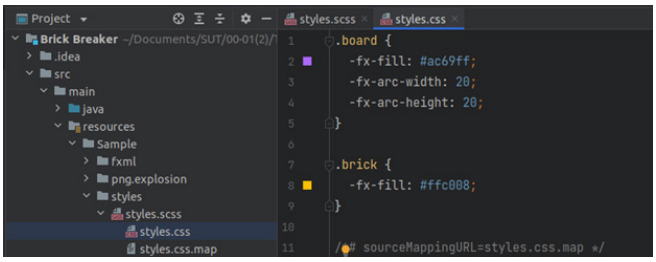
حال فایل main.css را پاک می‌کنیم و محتویات آن را در فولدر جدید styles، در فایل styles.scss کپی می‌کنیم.

احتمالا IntelliJ به شما پیشنهاد ایجاد یک file watcher برای این فایل scss می‌دهد.

درصورتی‌که به شما این پیشنهاد را نداد در settings/tools/file watchers می‌توانید یک file watcher برای scss اضافه کنید. IntelliJ باید بتواند کامپایلر Sass شما که قبلا توسط npm روی دستگاهتان نصب شده را پیدا کند.



پیشنهاد ما این است که به properties پیش‌فرض این file watcher دست نزنید.
بعد از اضافه‌کردن این file watcher باید فرآیند کامپایل فایل scss شما به کدهای CSS به‌طور اتوماتیک انجام شود.



فایل‌های styles.css و styles.css.map داخل فولدر styles توسط کامپایلر Sass ایجاد شده است. حواستان باشد در این فایل‌ها تغییری ندهید. حال می‌توانید کد Sass بزنید و در پروژه خود از styles.css استفاده کنید.

کار با Sass

تا اینجا راه‌اندازی Sass در پروژه خود را یاد گرفتید اما حالا وقت استفاده واقعی از امکانات بی‌نظیر آن است. در این‌جا چند نمونه از قابلیت‌های Sass و چه‌گونگی استفاده از آن‌ها را ذکر می‌کنیم. برای آموزش جامع‌تر می‌توانید به این فیلم یوتیوب یا داکیومنتیشن سایت Sass مراجعه کنید.

متغیرها

یکی از ساده‌ترین قابلیت‌های Sass استفاده‌از متغیرهاست. با استفاده از متغیرها و درنتیجه تایپ‌کردن کدهای کم‌تری به‌صورت دستی، شانس اشتباه پایین می‌رود و ایجاد تغییرات در استایل آسان‌تر می‌شود. CSS 3 از متغیرها پشتیبانی می‌کند اما به‌دلیل سادگی کار با متغیرهای Sass، ترجیح ما استفاده از Sass است. البته دقت کنید که JavaFX از CSS 3 پشتیبانی نمی‌کند و درنتیجه این گزینه برای ما مطرح نیست. به‌عنوان نمونه برای -fx-arc-width و -fx-arc-height یک متغیر تعریف می‌کنیم؛ زیرا می‌خواهیم همواره اندازه برابر داشته باشند و همچنین بعدا درصورت تمایل، بتوانیم به‌سادگی مقدار آن‌ها را عوض کنیم. متغیرها می‌توانند تایپ‌های متفاوتی داشته باشند. مثلا با کمک یک متغیر، اندازه و فونت متن‌های صفحه را تنظیم می‌کنیم. متغیرهای Sass با کاراکتر $ آغاز می‌شوند.

Nesting

یکی از قابلیت‌های بسیار مهم Sass به شمار می‌رود که انگیزه اصلی بسیاری از برنامه‌نویسان برای کار با Sass به‌جای CSS است. احتمالا با مشکلات کلاس‌های تودرتو در CSS آشنا باشید. فرض کنید می‌خواهیم کلاس title-text-box را تعریف کنیم. این text boxها هم در board هم در brick وجود دارند. Text boxهای board و brick هردو alignment مشابهی دارند اما padding و border-color آن‌ها متفاوت است. نوشتن این کلاس در CSS کار چندان سختی به نظر نمی‌آید اما زمانی که پروژه بزرگ‌تر می‌شود و این nestingها پیچیده‌تر می‌شوند کد CSS بسیار گنگ و ناخوانا می‌شود. برای رفع این مشکل ما از Sass استفاده می‌کنیم. ابتدا برای primary color و secondary color که تا پیش از این در stylesheet ما hard code شده‌بودند، دو متغیر تعریف می‌کنیم تا بتوانیم از مقدارشان در border colorهای کلاس خود استفاده کنیم. سپس کلاس title-text-box را می‌نویسیم:

```
$arc-length: 20;
$font-family: sans-serif;
$font-size: 10;
$primary-color: #ac69ff;
$secondary-color: #ffc008;

body {
 -fx-font-family: $font-family;
 -fx-font-size: $font-size;
}

.title-text-box {
 -fx-alignment: center;
}

.board {
 -fx-font-size: $font-size + 6;
 -fx-fill: $primary-color;
 -fx-arc-width: $arc-length;
 -fx-arc-height: $arc-length;

 .title-text-box {
   -fx-padding: 4px;
   -fx-border-color: $secondary-color;
 }
}

.brick {
 -fx-font-size: $font-size * 2;
 -fx-fill: $secondary-color;

 .title-text-box {
   -fx-padding: 2px;
   -fx-border-color: $primary-color;
 }
}
```

این فایل scss. به css پایین تبدیل می‌شود:

```
body {
 -fx-font-family: sans-serif;
 -fx-font-size: 10;
}

.title-text-box {
 -fx-alignment: center;
}

.board {
 -fx-font-size: 16;
 -fx-fill: #ac69ff;
 -fx-arc-width: 20;
 -fx-arc-height: 20;
}
.board .title-text-box {
 -fx-padding: 4px;
 -fx-border-color: #ffc008;
}
.brick {
 -fx-font-size: 20;
 -fx-fill: #ffc008;
}
```

```
.brick .title-text-box {
 -fx-padding: 2px;
 -fx-border-color: #ac69ff;
}
```

می‌توانید تصور کنید نوشتن مستقیم css بالا می‌توانست کار عذاب‌آوری باشد. در پروژه‌های بزرگ‌تر، یکی از بزرگ‌ترین چالش‌های stylesheetها این nesting کلاس‌ها محسوب می‌شود.

Mixin

درحقیقت mixinها همان محتویات کلاس‌های CSSند که می‌توانند reuse شوند. دقت کنید به mixinها می‌توان پارامترهای ورودی داد و به ورودی‌های mixinها می‌توان مقدار دیفالت نسبت داد.

Inheritance

کلاس‌های Sass برخلاف CSS می‌توانند از یک‌دیگر ارث‌بری کنند.

فرض کنید می‌خواهیم به پروژه خود popupهایی را اضافه کنیم تا به کاربر اطلاعاتی مانند هشدارها و پیام‌های باخت و برد بازی را نشان دهیم. برای این کار به سه کلاس warning-popup و win-popup و lost-popup نیاز داریم.

این popupها ویژگی‌های مشترکی دارند که کاملا یکسان نیستند. به‌طور مثال هرکدام تم رنگی مخصوص‌به خود را دارند که در borderها و backgroundشان اعمال می‌شود. برای این بخش از کار، یک mixin به‌نام themed-box می‌نویسیم که رنگی را به‌عنوان ورودی می‌گیرد و یک box با این تم رنگی می‌سازد. درصورتی که رنگی به‌عنوان ورودی داده نشود نیز مقدار دیفالت این رنگ، زرد است.

می‌دانیم popupهای ما ویژگی‌های کاملا مشترکی هم دارند، مانند font sizeشان. برای reusabilty این گروه از ویژگی‌های popupهایمان یک کلاس popup تعریف می‌کنیم که می‌دانیم از آن به‌صورت مستقیم استفاده نخواهد شد، بلکه فقط از ویژگی‌های آن، در کلاس‌های اصلی popupهایمان ارث‌بری خواهد شد. برای این کار، به‌جای این‌که با . کلاسمان را مشخص کنیم، آن را با % مشخص می‌کنیم. کلاس‌هایی که با % نوشته شوند، در فایل CSS نهایی جایی ندارند و تنها درصورتی که ازشان ارث‌بری شود، در کلاس‌های بچه ویژگی‌های خود را نشان می‌دهند. از این نظر می‌توان آن‌ها را با کلاس‌های abstract جاوا مقایسه کرد که به‌صورت مستقیم استفاده نمی‌شوند.

درضمن، popupهای win و lost چون که اهمیت بیشتری دارند، موردتاکید قرار می‌گیرند. برای این کار، یک کلاس emphasized-text-box می‌نویسیم که در آن font weight و دیگر attributeها را طوری تنظیم می‌کنیم که متن text-boxمان بیشتر به چشم بیاید. از این کلاس در بقیه stylesheet خود نیز می‌توانیم استفاده کنیم، به همین دلیل این کلاس را با . به‌جای % تعریف می‌کنیم. در popupهای win و lost، این کلاس را extend می کنیم و به این ترتیب، متن‌های popupهای ما نیز برجسته می‌شود.

```
@mixin themed-box($theme: Yellow) {
 -fx-border-color: $theme;
 -fx-background-color: rgba($theme, .25);
}

%popup {
 -fx-font-size: $font-size;
}
```

```
.emphasized-text-box {
 -fx-text-alignment: centers;
 -fx-padding: $arc-length / 2;
 -fx-font-weight: bold;
}

.warning-popup {
 @extend %popup;
 @include themed-box();
}

.win-popup {
 @extend %popup;
 @include themed-box($theme: Green);
 @extend .emphasized-text-box;

}

.lost-popup {
 @extend %popup;
 @include themed-box($theme: Red);
 @extend .emphasized-text-box;
}
```

خروجی نهایی متن scss بالا، css پایین می‌شود:

```
.lost-popup, .win-popup, .warning-popup {
 -fx-font-size: 10;
}

.emphasized-text-box, .lost-popup, .win-popup {
 -fx-text-alignment: centers;
 -fx-padding: 10;
 -fx-font-weight: bold;
}

.warning-popup {
 -fx-border-color: Yellow;
 -fx-background-color: rgba(255, 255, 0, 0.25);
}

.win-popup {
 -fx-border-color: Green;
 -fx-background-color: rgba(0, 128, 0, 0.25);
}

.lost-popup {
 -fx-border-color: Red;
 -fx-background-color: rgba(255, 0, 0, 0.25);
}
```

در آخر

برای پروژه خود هیچ الزامی به استفاده از Sass ندارید. اما استفاده از تکنولوژی‌های روز و مورداستفاده در صنعت، باعث می‌شود کد تمیزتر، پروژه بهتر و تجربه زیباتری از این ترم خود داشته باشید. در دانشگاه فرصت یادگیری تکنولوژی‌های جدید و به‌کارگیری آن‌ها زیاد پیش نمی‌آید و درس برنامه‌سازی پیشرفته از بهترین فرصت‌ها برای کسب تجربه واقعی محسوب می‌شود. پیشنهاد ما این است که نه‌تنها Sass و stylesheet، بلکه در هر بخش پروژه خود، سعی کنید از بهترین لایبرری‌ها و تکنولوژی‌ها استفاده کنید و کد، پروژه و درنهایت، ترم بهتری را پشت سر بگذارید.

داده‌ساختارها در جاوا

عرفان مجیبی، بنیامین ملکی و مهدی لطفیان

با توجه به تنوع بالای داده‌ساختارها در زبان جاوا، ممکن است موقع انتخاب یک داده‌ساختار مناسب برای ذخیره کردن داده‌ها، دچار سردرگمی شده باشید. بنابراین قصد داریم برخی از مهم‌ترین داده‌ساختارهای آماده جاوا را به شما معرفی کنیم تا با شناخت بهتر، بتوانید راحت‌تر تصمیم بگیرید و راحت‌تر داده‌ساختار مد نظرتان را انتخاب کنید!

در ابتدا پایه‌ای‌ترین واسط توصیف داده‌ساختارها، یعنی Iterable رو معرفی می‌کنیم.

Interface Iterable<T>

Iterable یک واسط است که مجموعه‌ای از رفتارها را در اختیار شما قرار می‌دهد که با پیاده کردن آن‌ها، می‌توانید یک کلاس دلخواه Iterable بسازید. اصلی‌ترین رفتاری که Iterableها در اختیارتان قرار می‌دهند، همانطور که از اسمش نیز پیداست، این است که اجازه‌ی استفاده از کلاس مدنظر را در for-each loop به عنوان شی مورد پیمایش (Iteration) می‌دهد. در حقیقت ویژگی مشترک تمامی کلاس‌های پیاده‌کننده این واسط، این است که می‌توانید آنها را به صورت عنصرعنصر در نظر بگیرید و بر روی تک تک عناصر آن، یک عملیات را انجام دهید.

ممکن است عبارت <T> بعد از اسم این واسط برایتان ناآشنا باشد. به این نوع تعریف یک واسط، تعریف جنریک(Generic) گفته می‌شود (که البته در ادامه درس با آن مفصلا آشنا می‌شوید). به طور خلاصه جنریک این امکان را به ما می‌دهد که بدون اینکه دقیق مشخص کنیم در زمان استفاده از این واسط با چه نوع اشیایی به عنوان عناصر Iterable مواجه هستیم، آن را تعریف کنیم. به این معنا که مهم نیست بعدا عناصر Iterable قرار است چه اشیائی باشند! این مسئله به abstraction در طراحی نرم‌افزار ما کمک زیادی می‌کند.

حال، کمی به رفتارهای موجود در Iterable می‌پردازیم:

هر کلاسی که Iterable را پیاده می‌کند باید حتما تابع زیر را پیاده‌سازی کند:

```
Iterator<T> iterator():
```

این متد، در حقیقت یک پیمایشگر (Iterator) برای پیمایش کردن عناصر موجود در کلاس پیاده‌کننده Iterable بر می‌گرداند. حال، ممکن است که بپرسید، خود Iterator<T> چیست؟ این هم یک واسط دیگر است که لازم است آن را پیاده‌سازی کنید تا بتوانید طریقه پیمایش کردن Iterable خودتان را به جاوا توضیح دهید. برای پیاده‌سازی Iterator باید متدهای next() و hasNext() را در کلاستان پیاده‌سازی کنید که به ترتیب به شما «عنصر بعدی در پیمایش فعلی» و «وجود داشتن عنصر بعدی در پیمایش فعلی» را برمی‌گردانند. فرضا برای ساخت یک آرایه اگر بخواهیم منطق پیمایش را طوری تعریف کنیم که عناصر را به صورت پی در پی مشاهده کنیم (یعنی a[i] درست بعد از a[i-1] ظاهر شود)، متد next به این صورت پیاده می‌شود که نشانگر پیمایش (شی‌ای که نشان‌دهنده آخرین عنصر پیمایش‌شده است) را به عنصر بعدی (عنصری با index+1) در آرایه ببرد و یک مرحله در پیمایش کل آرایه جلو برود. متد hasNext هم اینکه آیا دسته داده‌های ما هنوز عنصری برای پیمایش کردن دارد یا خیر را، با توجه به نشانگر فعلی، به ما می‌گوید. همچنین برای Iteratorها لازم است که نقطه شروع پیمایش را نیز مشخص کنید و در حقیقت نشانگر را برای حالت اولیه مقداردهی کنید.

منابع

- oracle
- geeksforgeeks

Interface Collection<E>

Collection نیز یک واسط است و مجموعه ای از رفتارها را در اختیار استفاده کننده هایش قرار می‌دهد. Collection از Iterable ارث بری می‌کند بنابراین غیر از متد موجود در Iterable توابع دیگری را هم در بر گرفته و کلاس‌هایی که آن را پیاده‌سازی می‌کنند مجبور به تعریف آنها می‌کند. اکثر داده ساختارهایی که در جاوا با آنها سروکار داریم Collection را پیاده‌سازیکرده اند (به جز mapها) در نتیجه اکثر متدهای معروفی که برای داده ساختارهای مختلف از آنها استفاده می‌کنید در این واسط معرفی شده اند.

قابل توجه است که Collection هم همانند Iterable از نوع جنریک تعریف میشود تا بتوان متدها را روی انواع شیءها استفاده کرد.

در ادامه تعریف متدهای مهم Collection آورده شده است:

```
boolean add(E e):
```

شی e را به کالکشن اضافه میکند.

```
boolean addAll(Collection<? Extends E> c):
```

تمامی عناصر داخل کالکشن c را به کالکشن اضافه میکند.

```
void clear():
```

کالکشن را خالی می‌کند.

```
boolean contains(Object o):
```

اگر شی o در کالکشن باشد، True برمیگرداند.

```
boolean containsAll(Collection<?> c):
```

اگر تمام اعضای کالکشن c در کالکشن موجود باشد، True برمیگرداند.

```
boolean remove(Object o):
```

اگر شی o در کالکشن باشد، آن را حذف میکند.

```
boolean removeAll(Collection<?> c):
```

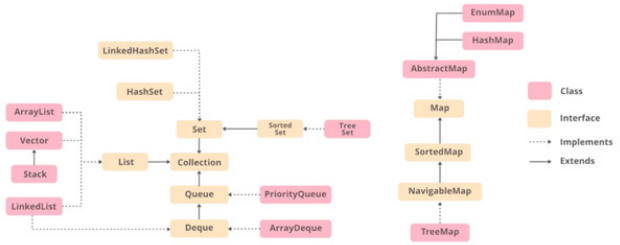
تمام اعضای کالکشن c را در صورت حضور در مجموعه، حذف میکند.

```
int size():
```

تعداد عناصر داخل مجموعه را برمیگرداند.

```
Object[] toArray():
```

آرایه ای تشکیل شده از تمام اعضای داخل کالکشن را برمیگرداند.



Interface List<E>

List یـک داده‌سـاختار دارای ترتیـب و فرزنـد Collection اسـت. در صـورت اسـتفاده از ایـن واسـط،کنترل کاملـی بـر محـل درج شـدن عناصـر داریم. در حقیقـت دسترسـی یافتـن و جسـت‌وجو کـردن عناصـر یـک List، از طریـق ایندکـس عـددی آنهـا ممکـن اسـت.

بـر خـلاف Setهـا، داشـتن عناصـر تکـراری در Listهـا ممکـن اسـت. بـه عبارتـی اگـر دو عنصـر e۱ و e۲ را در اختیـار داشـته باشـیم، بـه طـوری کـه equals(e۲.e۱) برقـرار باشـد، e۱ و e۲ می‌تواننـد همزمـان در یـک داده سـاختار پیاده‌کننـده List حضـور داشـته باشـند.

در یـک List رفتارهایـی، مـازاد بـر رفتارهـای Collectionهـا، وجـود دارنـد کـه لازم اسـت در زمـان پیاده‌سـازی در نظـر گرفتـه شـوند. Listهـا دارای ٤ متـد بـرای دسترسـی جایگاهـی (indexed) بـه عناصـر خودشـان دارنـد. از طرفـی ایـن ایندکس‌هـا ماننـد آرایه‌هـای جـاوا از صفـر شـروع می‌شـوند (zero based). البتـه بسـته بـه داده‌سـاختاری کـه از آن اسـتفاده می‌کنیـم، ممکـن اسـت دسترسـی بـه عناصـر فـورا صـورت نگیـرد. بـه همیـن دلیـل، در مواقعـی کـه از نحـوه پیاده‌سـازی پیمایشـگر (Iterator) لیسـت مـورد نظـر خـود اطلاعـی نداریـم، در پیمایـش آن، بهتـر اسـت کـه ترتیـب دیـده شـدن عناصـر را بـه کمـک ایندکس‌دهـی مناسـب نشـان دهیـم (و کار را بـه پیمایشـگرمربوط بـه آن List نسـپاریم!).

در Listهـا عـلاوه بـر Iterator یـک پیمایشـگر دیگـر بـه اسـم ListIterator نیـز داریـم کـه بـرای درج کـردن، جایگزیـن کـردن و دسترسـی دوطرفـه (هـم آدرس‌دهـی از ابتـدا و هـم آدرس‌دهـی از انتهـا) اسـتفاده می‌شـود. همچنیـن متـدی بـرای دسترسـی بـه خـود ایـن نشـانگر نیـز وجـود دارد. همچنیـن، Listهـا ۲ متـد بـرای جسـت‌وجوی اشـیا، در اختیـار مـا قـرار می‌دهنـد. لازم بـه ذکـر اسـت کـه در بسـیاری از پیاده‌سـازی‌هایی کـه از Listهـا داریـم، لازم اسـت کـه بـا احتیـاط از ایـن توابـع اسـتفاده شـود، چـرا کـه لزومـا بـرای رفتـار «جسـت‌وجو» بهینه‌سـازی نشـده‌اند و ممکـن اسـت بـه زمـان اجـرای برنامـه مـا ضربـه بزننـد.

عـلاوه بـر آن، در Listهـا ۲ متـد بـرای درج و حـذف یـک (یـا چندیـن) عنصـر در یـک آدرس بخصـوص، وجـود دارد.

نکتـه: لازم اسـت کـه بـه یـاد داشـته باشـید اسـتفاده از یـک لیسـت بـه عنـوان عنصـر یـک لیسـت دیگـر، بایـد بـا احتیـاط همـراه باشـد، چـرا کـه توابـع equal و hashcode بـر روی چنیـن داده‌سـاختارهایی خـوش تعریـف نخواهنـد بـود.

متدهای مخصوص List:

```
void add(int index, E e) :
```

شـی e را در آدرس index لیسـت مـا درج می‌کنـد و در صورتـی کـه عناصـری از پیـش در ان آدرس و یـا آدرس‌هـای بعـدی وجـود داشـته باشـند، آنهـا را یـک واحـد شـیفت می‌دهـد.

```
boolean addAll(int index, Collection<?Extends E> c):
```

تمامـی عناصـر داخـل کالکشـن c را در آدرس index لیسـت مـا درج می‌کنـد و مشـابه add دارای آدرس، در صـورت وجـود عناصـر، آنهـا را شـیفت می‌دهـد.

```
E get(int index):
```

شی‌ای که در آدرس index قرار دارد را برمی‌گرداند.

```
int indexOf(Object o):
```

در صـورت وجـود داشـتن رفرنـس داده شـده در لیسـت، اولیـن آدرس مربـوط بـه آن بازگردانـده می‌شـود. در صـورت عـدم وجـود، ۱- بازگردانـده می‌شـود.

```
int lastIndexOf(Object o):
```

در صـورت وجـود داشـتن رفرنـس داده شـده در لیسـت، آخریـن آدرس مربـوط بـه آن بازگردانـده می‌شـود. در صـورت عـدم وجـود، ۱- بازگردانـده می‌شـود

```
ListIterator<E> listIterator():
```

پیمایشـگر عناصـر لیسـت را مطابـق ترتیبـی کـه برایـش مشـخص شـده برمی‌گردانـد.

```
ListIterator<E> listIterator(int index):
```

پیمایشـگر عناصـر لیسـت را بـه ترتیـب مشـخص شـده و از عنصـر pindex و بـه بعـد بـاز می‌گردانـد.

```
E remove(int index):
```

در صـورت وجـود داشـتن عنصـری بـا آدرس index، آن را از لیسـت حـذف کـرده و عناصـر پـس از آن (در صـورت وجـود) را یـک واحـد بـه سـمت چـپ شـیفت می‌دهـد در صـورت وجـود نداشـتن چنیـن عنصـری اکسپشـن IndexOutOfBoundsException را تولیـد می‌کنـد.

```
E set(int index, E element):
```

شـی element را جایگزیـن عنصـر موجـود در آدرس index می‌کنـد. توجـه کنیـد کـه مشـابه تمامـی توابعـی کـه بـا index سـر و کار دارنـد، در صورتـی کـه آدرسـی بـا آن شـماره در لیسـت فعلـی موجـود نباشـد، اکسپشـن IndexOutOfBoundsException تولیـد می‌شـود.

```
default void sort(Comparator<? super  E> c):
```

لیست را طبق مقایسه‌کننده‌ای که به آن داده می‌شود، مرتب می‌کند.

```
List<E> subList(int fromIndex, int toIndex):
```

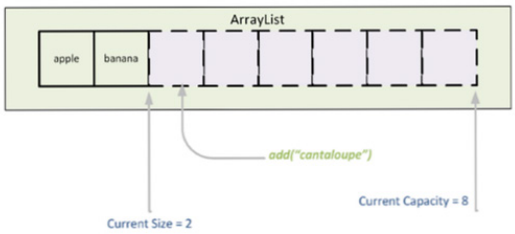
عناصر موجود در آدرس‌های fromIndex تا toIndex را به صورت یک List بازگردانند.

منابع

- oracle

حال به برخی از داده‌ساختارهای built-in جاوا که List را پیاده‌سازی می‌کنند می‌پردازیم:

Class ArrayList<E>



در حقیقت یک آرایه با سایز متغیر است که List و تمامی توابع اختیاری آن را پیاده‌سازی می‌کند.علاوه بر آن، همانطور که گفته شد، این داده‌ساختار متدهایی دارد که اجازه اعمال تغییر در سایز آرایه‌ای که به صورت پنهان برای نگه داشتن عناصر لیست ساخته‌شده است را، می‌دهد. در حقیقت این تغییر سایز آرایه به صورت سرشکن در زمان ثابت انجام می‌شود(با اصطلاح سرشکن مفصلا در درس ساختمان داده آشنا خواهید شد، اما به طور کلی به این معنی است که اگر به ازای سایزهای مختلف آرایه، در تعداد عناصر آرایه تغییر ایجاد کنیم، و میانگین زمان انجام این عملیات را به ازای آن سایزهای مختلف محاسبه کنیم، زمانی ثابت (یا O(۱)) خواهد شد).
در حقیقت هر ArrayList یک متغیر capacity دارد که سایز آرایه پنهان در خود را در هر لحظه نشان می‌دهد که در بسیاری از اوقات بیشتر از تعداد عناصر موجود در ArrayList است .
حال با برخی از توابع مخصوص به ArrayList آشنا می‌شویم:

```
void ensureCapacity(int minCapacity):
```

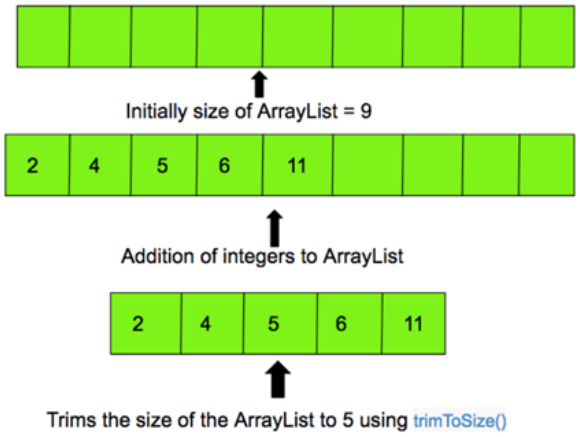
به کمک تابع ensureCapacity می‌توانید در مواقعی که می‌دانید قرار است تعداد داده زیادی در ArrayList خود ذخیره کنید، حافظه آرایه خود را افزایش دهید تا سرعت برنامه‌اتان بهبود پیدا کند.

```
protected void removeRange(int fromIndex, int
toIndex):
```

این تابع، عناصری که آدرس آنها در ArrayList برابر یا بزرگتر از fromIndex و کوچکتر از toIndex باشد را حذف می‌کند و عناصر بعد از این بازه را (در صورت وجود) شیفت می‌دهد.

```
void trimToSize():
```

این تابع، capacity (که همان سایز آرایه پنهان در ArrayList هست) را طوری کاهش می‌دهد که آرایه پنهان، سایزی برابر با تعداد عناصر موجود در ArrayList داشته باشد.

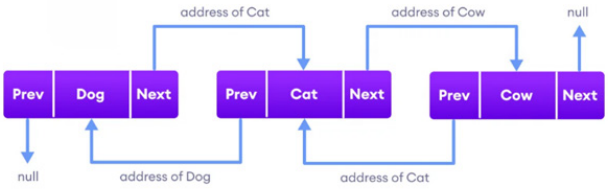


در تصویر بالا نحوه عملکرد ()trimToSize با مثالی آورده شده است.

منابع

- ephesossoftware
- geeksforgeeks
- oracle

Class LinkedList<E>



این داده‌ساختار در حقیقت پیاده‌سازی List و Deque است که یک زنجیره دو طرفه از عناصر در اختیار کاربر قرار می‌دهد.
از آنجایی که عناصر ایندکس مستقیمی ندارند و صرفا به صورت نسبی در کنار یکدیگر قرار می‌گیرند، برای رسیدن به یک عنصر بخصوص باید آنقدر LinkedList خود را پیمایش کنیم تا به شیء مورد نظر خود برسیم (بنابراین زمان دسترسی به عناصر موجود در LinkedList به صورت خطی (یا O(n)) است).
همچنین جالب است بدانید، در توابع رایج LinkedList، تابعی مخصوص به خودش وجود ندارد و صرفا توابع List و Deque را پیاده‌سازی کرده است که با توابع List پیش از این آشنا شده اید و در مورد Deque هم در ادامه خواهید خواند!

منابع

- oracle
- programiz

Class Vector<E>

این داده‌ساختار نیز مشابه ArrayList، واسط List را پیاده‌سازی می‌کند و همچنین یک حافظه پویا است (سایز آرایه درونی‌ش قابل

تغییر است). اما میان این دو داده ساختار تفاوت‌های مهمی وجود دارد که به برخی از آنها می‌پردازیم:

- داده‌ساختار Vector در ورژن‌های اولیه جاوا موجود بوده، در صورتی که ArrayList در JDK ۱٫۲ به جاوا اضافه شده و جدیدتر و بروزتر از Vector است.
- داده‌ساختار ArrayList سریع‌تر از Vector است.
- در هر بار افزایش سایز آرایه درونی این داده‌ساختارها، سایز آرایه Vector به مقدار ۱۰۰ درصد افزایش می‌یابد و ۲ برابر می‌شود در حالی که سایز آرایه Arraylist به مقدار ۵۰ درصد افزایش یافته و ۱٫۵ برابر می‌شود.
- داده‌ساختار ArrayList برای پیمایش عناصر از واسط Iterator استفاده می‌کند، در صورتی که Vector علاوه بر آن از Enumeration نیز، ممکن است، استفاده کند. Enumeration نیز، مشابه Iterator، روشی برای پیمایش یک Collection است با این تفاوت که پس از مشخص کردن enumeration، دیگر آن پیمایش قابل تغییر نیست و به عبارتی read-only خواهد بود و همچنین تنها قابل استفاده بر روی کلاس‌های Legacy (به کلاس‌ها و واسط‌هایی گفته می‌شود که پیش از معرفی شدن Collectionها در JDK۱٫۲، وجود داشتند) است.

با توجه به موارد بالا ممکن است بپرسید، پس چه نیازی به Vector داریم؟

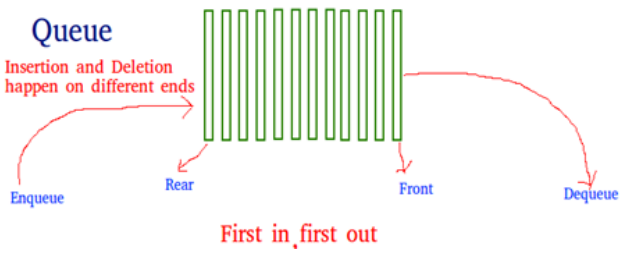
پاسخ این است که در طراحی‌های همروند(multi-thread)، ممکن است، بکار بیاید. اما، به طور کلی توصیه می‌شود که از این داده‌ساختار استفاده نشود و بسیاری از برنامه‌نویسان جاوا، آن را منسوخ(obsolete) می‌دانند. بنابراین، با توجه به سریع‌تر و جدیدتر بودن ArrayList، بهتر است از ArrayList استفاده کنیم.

منابع

- geeksforgeeks
- geeksforgeeks
- edureka
- javaconceptoftheday

Interface Queue<E>

یکی از داده‌ساختارهای برای نگه‌داری عناصر بر اساس اولویت می‌باشد؛ به این صورت که هر عضوی که زودتر در صف قرار گرفته باشد زودتر هم خارج می‌شود(First-In-First-Out).



همانطور که می‌توانید حدس بزنید در مواردی که ترتیب اضافه شدن و خارج شدن از یک دسته داده برای ما مهم باشد، از صف استفاده می‌کنیم.

متدهایی که Queue مضاف بر کالکشن، پدر Queue، فراهم می‌کند شامل موارد زیر است:

```
boolean offer(E e):
```

عملکردی مشابه متد add در کالکشن دارد، با این تفاوت که اگر صف پر باشد(صفتان طول محدودی داشته باشد و پر شده باشد) عضو را اضافه نکرده و مقدار false برمی‌گرداند. عملی معادل با آنچه در تصویر با Enqueue نشان داده شده است. توجه کنید تابع add اگر صف پر باشد استثناء پرتاب می‌کند.

```
E poll():
```

عنصر سر صف را از آن حذف می‌کند و برمی‌گرداند؛ یا اگر صف خالی باشد، null برمی گرداند. عملی معادل با آنچه در تصویر با Dequeue نشان داده شده است.

```
E peek():
```

سر صف را بر می‌گرداند اما حذف نمی کند؛ یا اگر صف خالی باشد، null برمی گرداند.

منابع

- geeksforgeeks
- oracle

Interface Deque<E>

با صف آشنا شدیم که از انتها به آن عضو اضافه می‌شد و از ابتدا اعضای آن خارج می‌شدند. حال اگر قابلیت اضافه کردن و حذف عضو از ابتدا و انتها هر دو وجود داشته باشد، یک صف دوطرفه خواهیم داشت.

Representation of deque

به یک صف دوطرفه، می‌توان به دید یک استک هم نگاه کرد؛ درصورتی که صرفا از انتها به آن عضو اضافه یا از آن حذف کنید. مهم‌ترین متدهای Deque اضافه بر متدهای Queue، شامل موارد زیر است:

```
boolean offerFirst(E e), offerLast(E e)
```

همانطور که از نامگذاری پیداست، این دو متد مانند متد Offer در صف عمل می‌کنند و قابلیت اضافه کردن عضو به ابتدا و انتهای صف را دارند. در صورتی که صف پر نباشد عضو اضافه شده و true برمی‌گردانند. توجه کنید توابع addFirst و addLast نیز به طور مشابه وجود دارند؛ اما تفاوتی که دارند این است که در صورت پر بودن صف استثناء پرتاب می‌کنند.

```
E pollFirst(), pollLast():
```

سر(First) یا ته(Last) صف را برگردانده و از صف حذف می کند، یا اگر صف خالی باشد، null برمی گرداند. توجه کنید توابع removeFirst و removeLast با عملکرد مشابه وجود دارند؛ با این تفاوت که در صورت خالی بودن صف، استثناء پرتاب می‌کنند.

```
E peekFirst(), peekLast():
```

سر(First) یا ته(Last) صف را برمی‌گردانند اما آنها را حذف نمی‌کند؛

اگـر صـف خالـی باشـد null برمی‌گردانـد. توابـع getFirst و getLast بـا عملکـرد مشـابه وجـود دارنـد؛ بـا ایـن تفـاوت کـه در صـورت خالـی بـودن صـف اسـتثناء پرتـاب می‌کننـد.

```
boolean removeFirstOccurrence(Object o), re-
moveLastOccurrence(Object o):
```

ایـن توابـع بـه ترتیـب اولیـن و آخریـن حضـور شـیء o را از صـف دوطرفـه حـذف می‌کننـد. اگـر شـیء مـد نظـر اصـلا در صـف وجـود نداشـت، اتفاقـی نمی‌افتـد.

```
void push(E e), E pop():
```

بـه یـک صـف دوطرفـه، می‌تـوان بـه دیـد یـک اسـتک هـم نـگاه کـرد؛ درصورتـی کـه صرفـا از انتهـا بـه آن عضـو اضافـه یـا حـذف کنیـد. جـاوا متدهـای پـوش و پـاپ کـه بـه ترتیـب معـادل اضافـه کـردن بـه ابتـدای صـف و حـذف عضـو از ابتـدای صـف هسـت را بـه همیـن منظـور در ایـن واسـط قـرار داده اسـت.

منابع

- javatpoint
- oracle

Class PriorityQueue<E>

اگـر در صـف بـرای اعضـای موجـود، اولویت(اولویتـی بجـز ترتیـب ورود!) داشـته باشـیم می‌توانیـم از صـف اولویـت اسـتفاده کنیـم. صـف اولویـت، یـک صـف اسـت کـه در آن هـر آیتمـی دارای یـک اولویـت اسـت و یـک عنصـر بـا اولویـت بالاتـر پیـش از سـایر عناصـر از صـف خـارج می‌شـود. مثـلا یـک صـف اولویـت از اعـداد، بـا توجـه بـه اولویـت بزرگتـر بـودن اعـداد، بـه ایـن شـکل خواهـد بـود کـه در هنـگام برداشـتن عضـو از آن هر بـار بزرگتریـن عضـو صـف را برمی‌گردانـد و آن را حـذف می‌کنـد. پـس از ایـن بـه بعـد دقـت کنیـد اگـر بـه داده‌سـاختاری نیـاز داشـتید کـه هنـگام برداشـتن عضـو از آن، بـه عضـو بـا بالاتریـن اولویت(مثـلا بزرگتریـن) نیـاز داریـد بجـای اسـتفاده از یـک لیسـت کـه هـر بـار از آن ماکزیمـم بگیریـد، از صـف اولویـت اسـتفاده کنیـد! :)

ایـن کلاس در جـاوا واسـط‌های کالکشـن و صـف را پیاده‌سـازی می‌کنـد، بنابرایـن توابـع آنهـا را دارد؛ دقـت کنیـد توابعـی ماننـد peek و poll عضـو بـا بالاتریـن اولویـت را برمی‌گردانـد (در واقـع سـر صـف، عضـو دارای بالاتریـن اولویـت اسـت). همچنیـن اولویـت صـف بـه صـورت پیشـفرض بـر اسـاس ترتیـب طبیعـی اعضایـش اسـت؛ امـا می‌توانیـد بـا اسـتفاده از کانسـتراکتور PriorityQueue(Comparator<? super E> comparator) بـا ورودی دادن یـک comparator، نحـوه مقایسـه و تعییـن اولویـت را مشـخص کنیـد.

منابع

- geeksforgeeks
- oracle

Class ArrayDeque<E>

یـک پیـاده سـازی از صـف دو طرفـه در جـاوا، آرایـه دو طرفـه اسـت. ایـن داده‌سـاختار محدودیـت انـدازه نـدارد و متدهایـی کـه در واسـط صـف دو طرفـه شـرح داده شـد را دارا می‌باشـد و می‌توانیـد از آنهـا اسـتفاده کنیـد. توجـه کنیـد در ایـن داده‌سـاختار نمی‌توانیـد null ذخیـره کنیـد.

نکتـه‌ی آخـری کـه بـد نیسـت بدانیـد، بـه عنـوان یـک صـف دوطرفـه یـا اسـتک، آرایـه دو طرفـه از LinkedList(کلاس دیگـری کـه صـف دو طرفـه را پیاده‌سـازی می‌کنـد) سـریعتر اسـت.

منابع

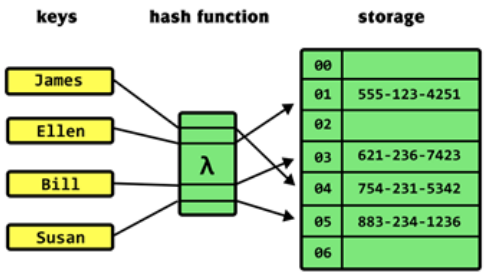
- geeksforgeeks
- oracle

Interface Set<E>

Set یـک داده سـاختار اصلـی در جـاوا اسـت کـه Collection را پیاده‌سـازی می‌کنـد. ویژگـی اصلـی Set کـه آن را از بقیـه داده سـاختارها متمایـز میکنـد، نداشـتن عضـو تکـراری اسـت. در حقیقـت اگـر بـرای هـر دو عضـو e۱ و e۲ در Set، متـد equals(e۲.e۱) را صـدا بزنیـم، نتیجـه False خواهـد شـد. و اگـر بخواهیـم عضـوی را بـه Set اضافـه کنیـم کـه از قبـل وجـود داشـته اسـت، عضـو جدیـد جایگزیـن عضـو قدیمـی شـده و تعـداد اعضـای Set تغییـری نمی‌کنـد. داده سـاختار Set بـر خـلاف Listهـا سـاختار index ی نداشـته و متدهایـی بـرای دسترسـی بـه عناصـر بـر اسـاس index را بـه کاربـر ارائـه نمیدهـد.

Set متـدی مخصـوص خـود نـدارد و همـان متدهـای کالکشـن بـرای آن‌هـا قابـل اسـتفاده اسـت. البتـه قابـل توجـه اسـت کـه متدهـای add, addAll و ... کـه وظیفـه دارنـد عضوهایـی را بـه مجموعـه اضافـه کننـد، ایـن کار را در صـورت تکـراری نبـودن عضـو انجـام میدهنـد و در غیـر ایـن صـورت بسـته بـه پیـاده سـازی ممکـن اسـت هیـچ عملـی انجـام ندهنـد یـا عضـو جدیـد را جایگزیـن قبلـی کننـد.

Class HashSet<E><E>



یکـی از مهم‌تریـن کلاس‌هایـی کـه Set را پیاده‌سـازی کلاس HashSet اسـت. HashSet در واقـع مجموعـه ای از اشـیاء اسـت کـه عضـو تکـراری ندارنـد، بنابرایـن می‌تـوان بـرای پـاک کـردن اعضـای تکـراری در یـک داده سـاختار از تبدیـل آن بـه HashSet اسـتفاده کـرد. همچنیـن HashSet بـه دلیـل سـاختارش کـه بـر پایـه Hash Table اسـت بسـیار سـریع عمـل میکنـد (در درس سـاختمان داده و الگوریتم‌هـا بـه طـور مفصـل و جـدول هـش آشـنا خواهیـد شـد فعـلا کافـی اسـت بدانیـد بـه کمـک یـک تابـع خـاص، بـرای هـر عنصـر یـک کـد هـش، تولیـد شـده و بـر اسـاس آن کـد در یـک جـدول ذخیـره میشـود و تمـام عملیات‌هـا در ادامـه بـه کمـک آن کـد انجـام خواهنـد شـد). تمـام متدهـای اصلـی در HashSet ماننـد add, remove, contains و size در ایـن داده سـاختار در زمـان O(۱) اجـرا خواهنـد شـد کـه یعنـی بـدون توجـه بـه سـایز مجموعـه، بلافاصلـه خواسـته شـما انجـام خواهـد شـد. مشـکل اصلـی کـه HashSetهـا دارنـد ایـن اسـت کـه نمی‌تـوان ترتیـب خاصـی بـرای اشـیا وارد شـده بـه آن در نظـر گرفـت و اعضـا بـا ترتیـب خـاص و نامشـخصی کـه بـرای سـرعت بهتـر بوجـود آمـده در مجموعـه قـرار میگیرنـد.

بنابرایـن اگـر شـما میخواهیـد کـه تمـام عملیـات روی یـک داده سـاختار

کـه عضـو تکـراری نپذیـرد در کمتریـن زمـان ممکـن انجـام شـود و ترتیـب قرارگیـری اعضـا هـم برایتـان اهمیتـی نداشـته باشـد، HashSet بهتریـن انتخـاب اسـت.
متدهـای قابـل اسـتفاده در HashSet همـان متدهـای Set هسـتند و متـدی مخصـوص خـود نـدارد.

Class LinkedHashSet<E>

ایـن کلاس ترکیبـی از سـاختارهای HashSet و LinkedList دو طرفـه اسـت. اعضـا هماننـد کلاس HashSet بـر پایـه Hash Table کنـار هم قـرار میگیرند کـه باعـث میشـود بتـوان عملیـات اصلـی را بـا نهایـت سـرعت انجـام داد. از طرفـی یـک LinkedList دو طرفـه هـم بـه ترتیـب اضافـه شـدن عناصـر بـه LinkedHashSet سـاخته می‌شـود، بنابرایـن می‌تـوان بـا حلقـه هـای for, foreach و ... بـه ترتیـب اضافـه شـدن عناصـر بـه داده سـاختار روی آنهـا پیمایـش کـرد.
قابـل توجـه اسـت کـه LinkedHashSet ماننـد هـر Set دیگـری عضـو تکـراری نمی‌پذیـرد بنابرایـن اگـر دسـتور اضافـه کـردن عضـو تکـراری داده شـد، آن دسـتور انجـام نمیشـود تـا ترتیـب اولیـه برهـم نخـورد.
بنابرایـن اگـر نیـاز بـه انجـام سـریع عملیـات بـر روی یـک مجموعـه بـدون اعضـای تکـراری داریـد و ترتیـب اضافـه شـدن عناصـر در مجموعـه هـم برایتـان اهمیـت دارد LinkedHashSet انتخـاب صحیـح اسـت.

Interface SortedSet<E>

یـک واسـط اسـت کـه از Set ارث بـری میکنـد. هـر عضـوی کـه بـه ایـن داده سـاختار افـزوده میشـود، بـر اسـاس یـک مقایسـه بـه مـکان درسـت اضافـه شـدن را پیـدا کـرده و در آن محـل قـرار می‌گیـرد. بنابرایـن بدیهـی اسـت کـه تمـام اعضـای SortedSet بایـد قابلیـت مقایسـه شـدن داشـته باشـند (اینترفیـس Comparable را پیاده‌سـازی کـرده باشـند). SortedSet هـم ماننـد دیگـر Setهـا عضـو تکـراری نمی‌پذیـرد. همچنیـن عملیاتـی اضافـه بـر Set را نیـز بـه دلیـل مرتـب بـودن ارائـه می‌دهـد کـه در زیـر شـرح داده شـده انـد.

```
Comparator<? super E> comparator():
```

مقایسه گری که عناصر بر اساس آن مرتب شده اند را برمیگرداند.

```
E first():
```

اولین (بر اساس مرتب سازی) عضو مجموعه را برمیگرداند.

```
SortedSet<E> headSet(E toElement):
```

عناصر قبل از toElement را به صورت مرتب برمیگرداند.

```
E last():
```

آخرین (بر اساس مرتب سازی) عضو مجموعه را برمیگرداند.

```
SortedSet<E> tailSet(E fromElement):
```

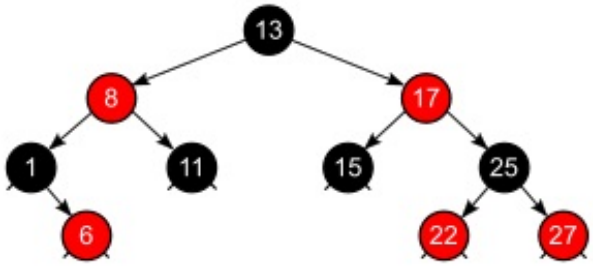
عناصر بعد از fromElement را به صورت مرتب برمیگرداند.

```
SortedSet<E> subSet(E fromElement, E toElement):
```

عناصـر قبـل از toElement و بعـد از fromElement را بـه صـورت مرتـب برمیگردانـد.

Class TreeSet<E>

مهم‌تریـن کلاسـی کـه SortedSet را پیاده‌سـازی می‌کنـد کلاس TreeSet اسـت. در ایـن داده سـاختار، عناصـر بـر اسـاس مقایسـه کننـده داده شـده و بـر پایـه سـاختار درخـت جسـتجوی دودویـی متوازن(بـا ایـن درخـت هـم در درس سـاختمان داده و الگوریتم‌هـا بـه طـول مفصـل آشـنا خواهیـد شـد فعـلا کافـی اسـت بدانیـد کـه در ایـن درخـت هـر راس دو فرزنـد مسـتقیم دارد کـه تمـام راسـهای زیـر درخـت سـمت راسـت از آن راس بزرگتـر و تمـام راس هـای زیردرخـت سـمت چـپ از آن راس کوچکترنـد) بـه مجموعـه اضافـه خواهنـد شـد. تمامـی عملیـات هـای اصلـی هماننـد add, remove, contains, first, last, headSet, tailSet و subSet بـا پیچیدگـی زمانـی O(logn) انجـام می‌شـوند کـه زمـان بسـیار خوبـی اسـت امـا بلافاصلـه نیسـت و بـا زیـاد شـدن اعضـای مجموعـه بـه صـورت لگاریتمـی زمـان عملیـات افزایـش پیـدا میکنـد.



نکتـهٔ مثبـت اصلـی ایـن داده سـاختار مرتـب بـودن همیشـگی اعضـای آن اسـت بنابرایـن اگـر نیـاز داریـد کـه بـا عناصـر غیـر تکـراری بـه صـورت مرتـب کار کنیـد و زمـان زیـادی هـم صـرف نکنیـد TreeSet می‌توانـد گزینـه مناسـبی بـرا ذخیـرهٔ عناصـر باشـد.
TreeSet عـلاوه بـر متدهـای SortedSet چنـد متـد مخصـوص هـم دارد کـه در زیـر شـرح داده شـده انـد.

```
E ceiling(E e):
```

آخریـن عنصـری کـه قبـل یـا هـم مـکان عنصـر e در درخـت قـرار دارد را برمیگردانـد.

```
NavigableSet<E> descendingSet():
```

یه مجموعه درختی با ترتیب برعکس مجموعه کنونی برمیگرداند.

```
E floor(E e):
```

اولیـن عنصـری کـه بعـد یـا هـم مـکان عنصـر e در درخـت قـرار دارد را برمیگردانـد.

```
E higher(E e):
```

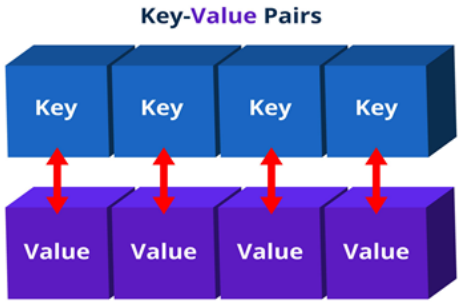
آخرین عنصری که قبل از عنصر e در درخت قرار دارد را برمیگرداند.

```
E lower(E e):
```

اولین عنصری که بعد از عنصر e در درخت قرار دارد را برمیگرداند.

Interface Map<E>

بـر خـلاف تمـام واسـط‌هاییکه تـا کنـون مشـاهده کردیـم، Map از Iterator و Collection ارث بـری نمی‌کنـد. بنابرایـن برخـلاف Iteratorهـا نمی‌تـوان روی آن بـه راحتـی بـا کمـک حلقه‌هـا پیمایـش کـرده و متدهایـش بـا متدهـای Collectionهـا تفاوت‌هـای قابـل توجهـی دارنـد. سـاختار Map بـه طـور کلـی بـه ایـن شـکل اسـت کـه یـک کلیـد و یـک مقـدار را بـه عنـوان یـک عضـو دریافـت کـرده هـر مقـدار را بـا کلیـدش شناسـایی میکنـد. بنابرایـن می‌تـوان کلیدهـای متفـاوت بـا مقدارهـای یکسـان در آن نگـه داشـت امـا نمی‌تـوان چنـد مقـدار را بـه یـک کلیـد یکسـان نسـبت داد و در صـورت انجـام ایـن کار بـه طـور معمـول مقـدار قبلـی از دسـت مـی‌رود و کلیـد بـه مقـدار جدیـد اشـاره می‌کنـد.
بـا وجـود اینکـه حلقه‌هـای عـادی روی Mapهـا قابـل اسـتفاده نیسـتند می‌تـوان بـا کمـک Map.Entry روی اعضـای Map بـدون ترتیـب خاصـی پیمایـش کـرد و از کلیدهـا و مقادیـر آنهـا بهـره بـرد.



متدهای مهم Map شامل موارد زیر هستند:

```
boolean containsKey(Object key):
```

اگر کلید key در Map موجود باشد، True برمیگرداند.

```
boolean containsValue(Object value):
```

اگر مقدار value در Map موجود باشد، True برمیگرداند.

```
Set<Map.Entry<K,V>> entrySet():
```

مجموعه ای از کلیدها و مقادیر موجود را ساخته و برمیگرداند.

```
V get(Object key):
```

مقـدار نسـبت داده شـده بـه کلیـد key را در صـورت وجـود برمیگردانـد. (اگـر وجـود نداشـته باشـد null برمیگردانـد.)

```
default V getOrDefault(Object key, V default-
Value) :
```

مقـدار نسـبت داده شـده بـه کلیـد key را در صـورت وجـود برمیگردانـد. (اگـر وجـود نداشـته باشـد defaultValue را برمیگردانـد.)

```
Set<K> keySet():
```

مجموعه‌ای از کلیدها را برمیگرداند.

```
Collection<V> values():
```

کالکشنی از مقادیر را برمیگرداند.

```
V put(K key, V value):
```

مقـدار value را بـا کلیـد key کـه value بـه آن نسـبت داده شـده اسـت را بـه Map اضافـه می‌کنـد.

```
void putAll(Map<? extends K,? extends V> m):
```

تمام اعضای مپ m را با همان کلید و مقادیر به Map اضافه می‌کند.

```
V remove(Object key):
```

عضـوی ازMap بـا کلیـد key را بـه همـراه مقـدار نسـبت داده شـده بـه key حـذف می‌کنـد.

```
default boolean remove(Object key, Object Val-
ue):
```

عضـوی از مـپ بـا کلیـد key را در صـورت داشـتن مقـدار value حـذف می کنـد .

```
default V replace(K key, V value):
```

مقـدار نسـبت داده شـده بـه key را در صـورت وجـود بـا value عـوض می کنـد .

```
default boolean replace(K key, V oldValue, V
newValue):
```

مقـدار نسـبت داده شـده بـه key را در صـورت وجـود و برابـر بـودن آن مقـدار بـا oldValue بـا newValue عـوض می‌کنـد.

```
int size():
```

تعداد اعضا که برابر با تعداد کلیدها است را برمیگرداند.

Class HashMap<E>

مهـم تریـن کلاسـی کـه Map را پیاده‌سـازی، HashMap اسـت کـه هـر برنامـه نویـس جـاوا قطعـا در پـروژه هـای خـود چنـد بـاری بـه سـراغ آن رفتـه اسـت. در HashMap ماننـد هـر Map دیگـری هـر مقـدار بـه یـک یـا چنـد کلیـد می‌توانـد نسـبت داده شـود و بـه کمـک آن کلیـد در ادامـه یافـت شـود. نکتـه مثبتـی کـه HashMap را بسـیار کاربـردی می‌کنـد سـرعت بسـیار زیـاد عملیـات آن اسـت. تمـام عملیـات اصلـی ماننـد contains, get, put, remove و replace در O(۱) انجـام میشـوند کـه یعنـی بـدون توجـه بـه سـایز Hashmap در کمتریـن زمـان صـورت می‌گیرنـد. ترتیـب قرارگیـری اعضـا در Hashmap مشـخص نیسـت و بـا انجـام عملیـات دچـار تغییـر می‌شـوند، بنابرایـن حتـی بـا وجـود راهـکاری بـرای پیمایـش روی کلیدهـای آن ترتیـب رسـیدن بـه هـر کلیـد مشـخص نیسـت.
بنابرایـن اگـر نیـاز بـه داده سـاختاری بـرای نگـه‌داری کلیدهـا و مقادیرشـان داریـم بـه طـوری کـه بتـوان تمـام عملیـات اصلـی را بـا بیشـترین سـرعت روی آنهـا انجـام داد و ترتیـب آنهـا برایمـان اهمیتـی نداشـته باشـد، HashMap بهتریـن انتخـاب اسـت.

Class TreeMap<E>

از دیگر کلاس‌هایی که Map را آپیاده‌سازی می‌کنندمی‌توان به TreeMap اشاره کرد. در ساختار TreeMap همانند TreeSet از درخت متوازن برای نگه‌داری عناصر استفاده شده است. در این ساختار اعضا بر اساس کلیدشان مرتب شده و در درخت قرار می‌گیرند بنابراین کلیدها باید قابلیت مقایسه شدن را داشته باشند(واسط Comparable را پیاده کرده باشند). همانند TreeSet، تمام عملیات اصلی همانند contains, get, put, remove و replace در O(log n) انجام میشود که زمان بسیار خوبی است؛ هرچند با افزایش اعضای مجموعه این زمان به صورت لگاریتمی افزایش میابد.
بنابراین هرگاه نیاز به ساختاری داشتیم که کلیدها و مقادیر ما را به صورت مرتب نگه داری کند و همه عملیات در بهترین زمان ممکن انجام شوند TreeMap انتخاب درست است.
به دلیل مرتب بودن اعضا، متدهایی برای کار راحت تر با TreeMap علاوه بر متدهای Map تعبیه شده است که در ادامه شرح داده شده ا ند .

```
NavigableSet<K> descendingKeySet():
```

مجموعه ای از کلیدها با ترتیب عکس کنونی ساخته و برمیگرداند

```
NavigableMap<K, V> descendingMap():
```

یک Map با ترتیب عکس کنونی بر اساس کلیدها ساخته و برمیگرداند.

```
Map.Entry<K, V> firstEntry():
```

اولین عضو را به صورت کلید-مقدار برمیگرداند.

```
K firstKey():
```

اولین کلید را برمیگرداند.

```
Map.Entry<K, V> lastEntry():
```

آخرین عضو را به صورت کلید-مقدار برمیگرداند.

```
K lastKey():
```

آخرین کلید را برمیگرداند.

```
Map.Entry<K, V> ceilingEntry(K key):
```

اولین عنصر بعد از key یا برابر با آن را به صورت کلید-مقدار برمیگرداند.

```
K ceilingKey(K key):
```

اولین کلید بعد از key یا برابر با آن را برمیگرداند.

```
Map.Entry<K, V> floorEntry(K key):
```

آخرین عنصر قبل از key یا برابر با آن را به صورت کلید-مقدار برمیگرداند.

```
K floorKey(K key):
```

آخرین کلید قبل از key یا برابر با آن را برمیگرداند.

```
Map.Entry<K, V> higherEntry(K key):
```

اولین عنصر بعد از key را به صورت کلید-مقدار برمیگرداند.

```
K higherKey(K key):
```

اولین کلید بعد از key را برمیگرداند.

```
Map.Entry<K, V> lowerEntry(K key):
```

آخرین عنصر قبل از key را به صورت کلید-مقدار برمیگرداند.

```
K lowerKey(K key):
```

آخرین کلید قبل از key را برمیگرداند.

```
SortedMap<K, V> headMap(K toKey):
```

تمام عناصری که کلیدهایشان از toKey کمتر است را به صورت یک مپ مرتب برمیگرداند.

```
SortedMap<K, V> tailMap(K fromKey):
```

تمام عناصری که کلیدهایشان از fromKey بزرگتر است را به صورت یک مپ مرتب برمیگرداند.

```
SortedMap<K, V> subMap(K fromKey, K toKey):
```

تمام عناصری که کلیدهایشان از fromKey بزرگتر است و از toKey کمتر است را به صورت یک مپ مرتب برمیگرداند.

> Programming is the factoring of a set of requirements into a set of functions and data structures.
>
> *– Douglas Crockford*

کدنامه

شماره: ۵
تاریخ انتشار: ۱۰ خرداد ۱۴۰۱
نویسندگان: سروش شرافت، امیرحسین رازلیقی، عرفان مجیبی، بنیامین ملکی و مهدی لطفیان
طراحی: سجاد سلطانیان و حسام الدین سلیمانی

دستیاران آموزشی دروس مبانی برنامه‌سازی و برنامه‌سازی پیشرفته، طی نیمسال‌های گذشته، مطالب متعدد درسی را در قالب فایل‌های گوناگون منتشر می‌کردند. از سال گذشته، برآن شدیم تا با تشکیل «کدنامه»، تمامی مطالب آموزشی را در این قالب تدوین کنیم تا هم به سرعت بتوان آنها را مطالعه کرد و هم بتوانیم به این بهانه، محتوایر کاربردی‌تری را در اختیار دانشجویان قرار دهیم. توجه شود که «کدنامه»، به هیچ عنوان، یک نشریه نیست و زمان عرضه مشخصی نیز ندارد، بلکه تنها قالبی جدید برای عرضه همان مطالب و محتوای آموزشی است و در آن تنها به مباحث درسی مخصوص درس برنامه‌سازی پیشرفته نیم‌سال جاری پرداخته می‌شود.

مطالعه مطالب «کدنامه»، برای انجام بهتر پروژه و تمرین‌های درس برنامه‌سازی پیشرفته، اکیدا توصیه می‌شود.