



دانشگاه صنعتی امیرکبیر  
(پلی تکنیک تهران)

**دانشکده ریاضی و علوم کامپیوتر**

**درس برنامه‌سازی پیشرفته و کارگاه**

**داده‌ساختارها و کالکشن جاوا**

استاد درس

**دکتر مهدی قطعی**

استاد دوم

**بهنام یوسفی مهر**

نگارش

**پدرام اسماعیل پور و هومن حمیدی پور**

**بهار ۱۴۰۳**

## فهرست

۴	مقدمه
۵	فریم ورک کالکشن های جاوا (JCF)
۵	اینترفیس های کالکشن
۷	پیاده سازی ها
۸	الگوریتم ها
۹	آرایه؛ ساده ترین داده ساختار
۹	آرایه ها در جاوا
۱۰	زمان استفاده از آرایه
۱۱	آرایه پویا و اینترفیس List
۱۲	اینترفیس List
۱۳	ArrayList در جاوا
۱۷	Iterator؛ ابزاری برای پیمودن کالکشن ها
۱۷	اینترفیس Iterator
۱۹	حلقه for-each
۲۰	اینترفیس ListIterator
۲۲	لیست پیوندی
۲۴	پیاده سازی لیست پیوندی
۲۷	LinkedList در جاوا
۲۷	استفاده از Iterator در لیست پیوندی
۳۰	پشته و صف
۳۲	اینترفیس Queue
۳۳	اینترفیس Deque
۳۴	پیاده سازی پشته

---

۳۶.....	پیاده سازی صف
۳۷.....	Iterator در پشته و صف
۳۸.....	پیاده سازی iterator() در پشته
۳۸.....	پیاده سازی iterator() در صف
۴۰.....	<b>هش مپ</b>
۴۰.....	اینترفیس Map
۴۱.....	پیاده سازی هش مپ
۴۳.....	Iterator در هش مپ
۴۵.....	<b>چه چیزی یاد گرفتیم؟</b>

## مقدمه

فرض کنیم می‌خواهیم یک کتاب‌خونه رو مرتب کنیم. می‌تونیم کتاب‌ها رو بدون هیچ نظم‌ی روی هم بریزیم، اما این‌طوری پیدا کردن یک کتاب جدید زمان‌بر می‌شه. می‌تونیم از قفسه‌ها استفاده کنیم تا کتاب‌ها رو طوری بچینیم که پیدا کردن یک کتاب خاص آسون‌تر بشه. مشابه همین سناریو، در دنیای برنامه‌نویسی هم داده‌ساختارها رو داریم. هر داده‌ساختار یک روش برای مرتب کردن و ذخیره‌سازی داده‌ها در کامپیوتر است، به‌طوری که در موقعیت‌های مختلف بتوانیم از عملیات‌هایی مثل اضافه کردن، حذف کردن، و جست‌وجو به بهینگی استفاده کنیم. در این هفته می‌خوایم با داده‌ساختارهای مقدماتی، نوع کارکرد، و نحوه پیاده‌سازی آن‌ها در زبان جاوا آشنا بشیم.

داده‌ساختارها زمان (سرعت انجام عملیات‌ها) و فضا (حافظه مورد استفاده) رو بهینه می‌کنن. از طرفی می‌تونند پیچیدگی‌ها را در رابط‌های ساده پنهان کنن و به عملیات‌های ساده سرعت ببخشن. همچنین در خیلی از الگوریتم‌ها، ما به داده‌ساختارهای خاصی نیازمندیم تا پیاده‌سازی الگوریتم را برای ما ممکن کنن.

## فریم‌ورک کالکشن‌های جاوا<sup>۱</sup> (JCF)

قبل از این‌که با داده‌ساختارها آشنا بشیم، می‌خوایم با یکی از ابزارهای مهم زبان جاوا یعنی فریم‌ورک کالکشن‌های جاوا آشنا بشیم. این ابزار مجموعه‌ای از اینترفیس‌ها و کلاس‌های مختلفه، و یک استاندارد کلی برای پیاده‌سازی داده‌ساختارهای مختلف برای ما فراهم می‌کنه. علاوه‌بر اینترفیس‌ها، برخی داده‌ساختارها به‌طور پیش‌فرض پیاده‌سازی شدن و برخی الگوریتم‌ها (مثل مرتب‌سازی و جست‌وجو) برای انجام عملیات‌ها در یک داده‌ساختار نیز قرار گرفتن. این اینترفیس‌ها همگی از کلاس جنریک استفاده می‌کنن و شما می‌تونین آبجکت‌هایی با هر تایپ در اونا ذخیره کنین.

### اینترفیس‌های کالکشن

اینترفیس‌ها مشخص می‌کنن که داده‌ساختارای که قراره بسازیم از چه عملیات‌هایی پشتیبانی می‌کنه. اینترفیس `Collection` ریشه‌ای‌ترین اینترفیس فریم‌ورک کالکشنه، و به شکل یک محل نگه‌داری برای آبجکت‌های متفاوت عمل می‌کنه. کالکشن سه اینترفیس فرزند هم داره؛ `Set` کالکشنیه که امکان نگه‌داری آبجکت‌های تکراری در اون وجود نداره. اینترفیس `List` برای داده‌ساختارهایی که آبجکت‌هاش ترتیب مشخص دارن. اینترفیس `Queue` هم کالکشنیه که آبجکت‌های آن ترتیب اضافه شدن خاصی رو حفظ می‌کنن.

اگه به سورس‌کد جاوا برین و اینترفیس `Collection` رو باز کنین با چنین چیزی روبه‌رو می‌شین؛

```
public interface Collection<E>
    extends Iterable<E> {
        boolean add(E e);

        boolean remove(Object o);

        boolean contains(Object o);

        int size();

        boolean isEmpty();

        Iterator<E> iterator();
    }
```

<sup>۱</sup> Java Collections Framework

تایپ E تایپ جنریکه. در این جا جنریک به ما این قابلیت رو می ده که بتونیم تایپ اعضایی که در کالکشنمون قرار می گیره رو کاملاً کنترل کنیم.

همون طور که می بینین Collection هم یک اینترفیس عادیه. البته متدهای دیگه ای هم داره که این جا ازشون صرف نظر کردیم، اما ساختار کلی و مهم ترین متدهاش همینان. طبیعتاً سه اینترفیس فرندز Collection هم تمام این متدها رو به ارث می برن.

- `public boolean add(E element)`: آبجکت element رو به کالکشن اضافه می کنه. در صورت موفقیت، این متد مقدار `true` رو برمی گردونه. در صورتی که امکان اضافه کردن این آبجکت به کالکشن نباشه، مقدار `false` رو برمی گردونه. (برخی کالکشن ها فقط برای خواندن هستن. در اون کالکشن ها، هنگام صدا شدن این متد `UnsupportedOperationException` ترو می شود.)

- `public boolean remove(Object element)`: آبجکت element رو از این کالکشن حذف می کنه. مشابه متد `add()`، در صورتی که موفقیت آمیز باشه مقدار `true` و در صورتی که آبجکت element در کالکشن وجود نداشته باشه مقدار `false` رو برمی گردونه. (مشابه متد `add()` در یک کالکشن خواندنی، با صدا شدن این متد `UnsupportedOperationException` ترو می شود.)

- `public boolean contains(Object element)`: در صورتی که کالکشن دارای آبجکت مشخص شده باشه، مقدار `true` را برمی گرداند.

- `public int size()`: تعداد اجزای موجود در کالکشن را خروجی می دهد.

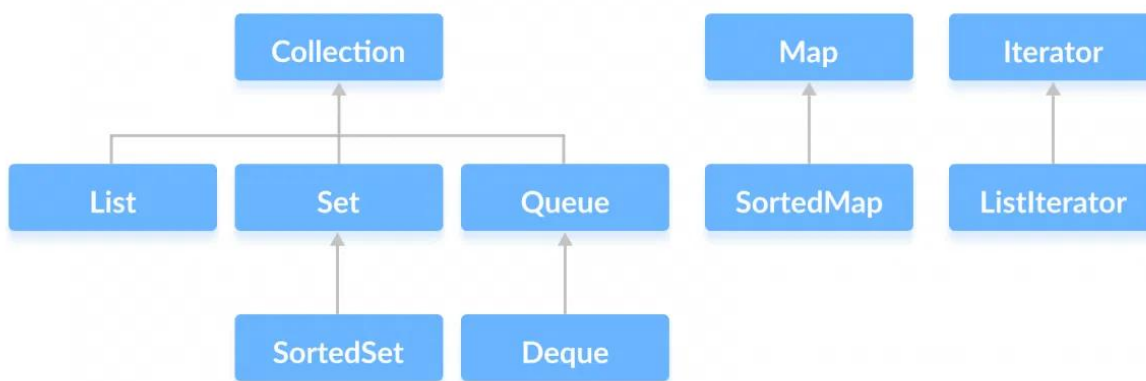
- `public boolean isEmpty()`: در صورتی که کالکشن هیچ عضوی نداشته باشه، مقدار `true` را خروجی می دهد.

- `public Iterator iterator()`: تمام اجزای موجود در کالکشن را بررسی می کند. این متد یک آبجکت از جنس `Iterator` خروجی می دهد که به ما کمک می کند اجزای یک کالکشن رو دونه دونه طی کنیم. در ادامه درباره `iterator` ها بیش تر حرف می زنیم و با دقت بررسیشون می کنیم.

همون‌طور که گفتیم، سه اینترفیس `Set` و `List` و `Queue` فرزندان اینترفیس `Collection` هستند و به همین دلیل متدهای اون رو به ارث می‌برن. همچنین و ممکنه متدهای دیگه‌ای هم داشته باشن که در آینده هر کدوم رو با جزئیات معرفی می‌کنیم.

علاوه‌بر اینترفیس `Collection` و `Queue`، یک اینترفیس دیگه به اسم `Map` هم داریم. اینترفیس `Map` با وجود این‌که جزوی از فریم‌ورک جاواست، فرزند مستقیم اینترفیس `Collection` نیست. در آخر این داکيومنت به این اینترفیس هم می‌پردازیم.

## Java Collections Framework



## پیاده‌سازی‌ها

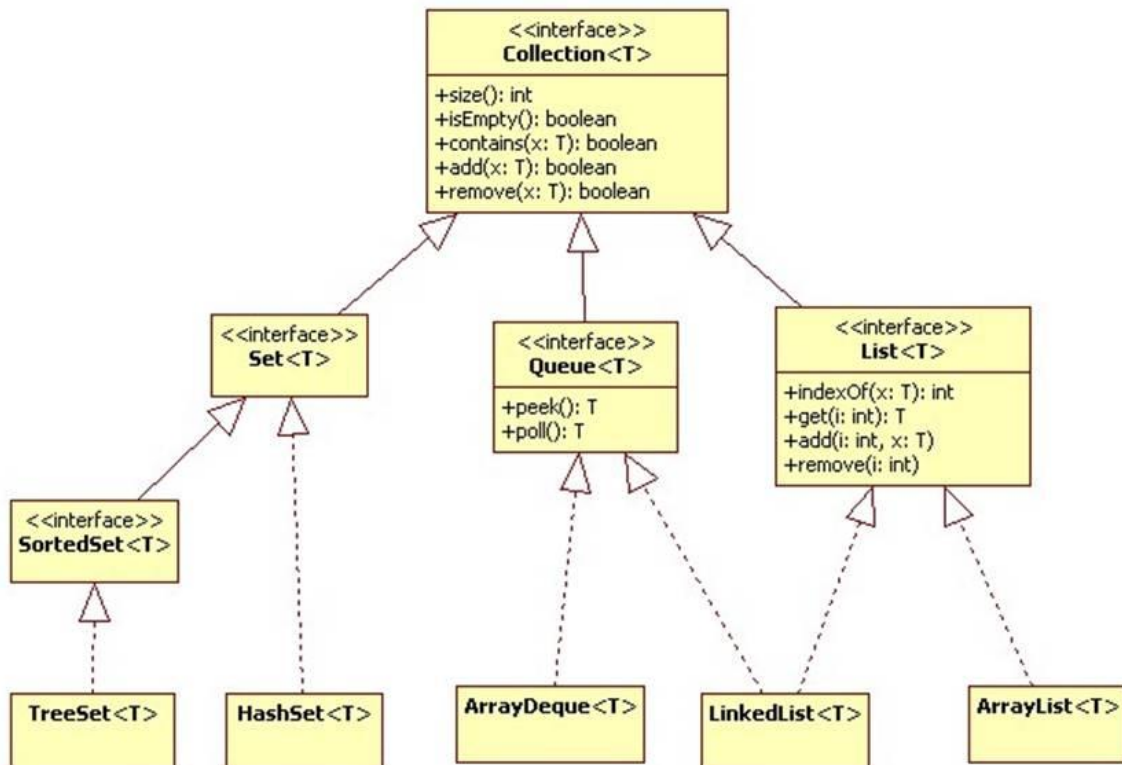
فریم‌ورک کالکشن‌های جاوا شامل پیاده‌سازی‌هایی که با ارث‌بری از اینترفیس‌های کالکشن، مشخص می‌کنن عملیات‌های مختلف چه‌طور انجام می‌گیرن. این پیاده‌سازی‌ها کلاس‌های پیش‌فرضی هستند که استفاده از داده‌ساختارهای مهم رو برای ما ساده‌تر می‌کنن. شما تا الان با کلاس `ArrayList` کار کردین. این داده‌ساختار یک آرایه پویاست (در ادامه با آرایه‌ها و آرایه‌های پویا آشنا می‌شین). که پیاده‌سازی مستقیم اینترفیس `List` هست. از اون‌جایی که `ArrayList` مستقیماً با پیاده‌سازی اینترفیس `Collection` آماده شده، می‌تونین موقع ساختن یک `ArrayList` از دستور `Collection` استفاده کنین.

```
Collection<String> names = new ArrayList<>();
names.add("Arman");
names.add("Pedram");
```

```
System.out.println(names);
```

کد بالا رو خودتون اجرا کنین. می‌بینین که یک `ArrayList` ساخته می‌شه و می‌تونین بهش اعضای جدید اضافه کنین.

کلاس‌های مختلفی از اینترفیس‌های کالکشن پیاده‌سازی شدن که توی نمودار زیر می‌تونین مهم‌ترین‌اشون رو ببینین.



## الگوریتم‌ها

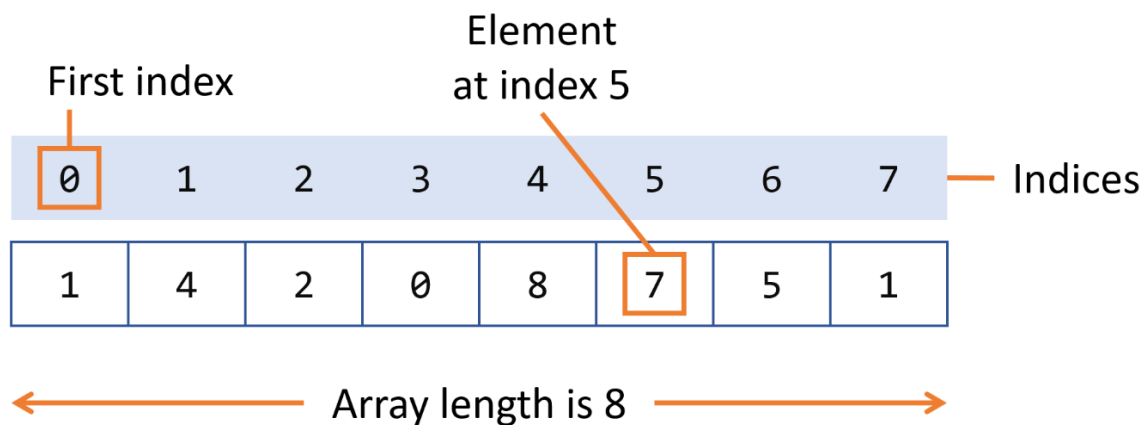
متدهایی مثل `sort()` و `shuffle()` که به راحتی قابل دوباره استفاده شدن هستن در اینترفیس‌ها قرار گرفتن و با توجه به جنریک بودن تایپ‌های اینترفیس‌ها، می‌تونیم ازشون در پیاده‌سازی داده‌ساختارهای پیچیده‌تر استفاده کنیم.



## آرایه؛ ساده‌ترین داده‌ساختار

یکی از قدیمی‌ترین و ساده‌ترین داده‌ساختارهاست. از قبل با آرایه آشنایی و می‌دونیم که مثل یک قفسه کتابه که برای هر کتاب یک جای مشخص داره. این جایگاه‌ها با اعدادی به اسم ایندکس مشخص شدن.

داده‌های ذخیره‌شده در یک آرایه **تعداد مشخص و غیرقابل‌تغییری دارن و همگی از یک تایپ هستن**. به دلیل این ویژگی‌ها، می‌تونیم برای اعضای آرایه ایندکس مشخصی قرار بدیم. این باعث می‌شه که **بتونیم در کوتاه‌ترین زمان ممکن در کامپیوتر به مقدار یکی از اعضای آرایه دسترسی پیدا کنیم**، فقط کافیه که بدونیم ایندکس عضو دل‌خواه‌مون چنده. به این ویژگی Random Access هم می‌گن. از **طرفی داده‌های آرایه به شکل بلاک‌های متوالی در حافظه** قرار می‌گیرن، در نتیجه برای ذخیره‌سازی اعضای آرایه در حافظه کامپیوتر نیازی به اطلاعات زیادی نداریم و از لحاظ حافظه مصرفی بهینه است. (در ادامه با داده‌ساختارهایی آشنا می‌شیم که اجزای متفاوت رو در بلاک‌های متفاوت حافظه قرار می‌دن.) حافظه متوالی باعث می‌شه که پردازش آرایه برای CPU هم سریع‌تر بشه.



## آرایه‌ها در جاوا

در جاوا آرایه‌ها برای ساخت و استفاده از آرایه‌ها سینتکس مخصوصی داریم که کاملاً باهش آشنا هستین.

آرایه‌های جاوا اندازه ثابت دارن، ایندکس‌هاشون از صفر شروع می‌شن، و می‌تونن هم داده‌های primitive (مثل int و char)، و هم آبجکت‌ها رو توی خودشون جا بدن.

اگر یک درصد فراموش کردین که آرایه‌های جاوا چه ظاهری دارن، این‌جا دو نمونه از ساخت و مقداردهی آرایه‌ها رو می‌بینین:

```
int[] numbers;  
numbers = new int[5];  
  
String[] weekdays = {"Mon", "Tue", "Wed", "Thu", "Fri"};
```

## زمان استفاده از آرایه

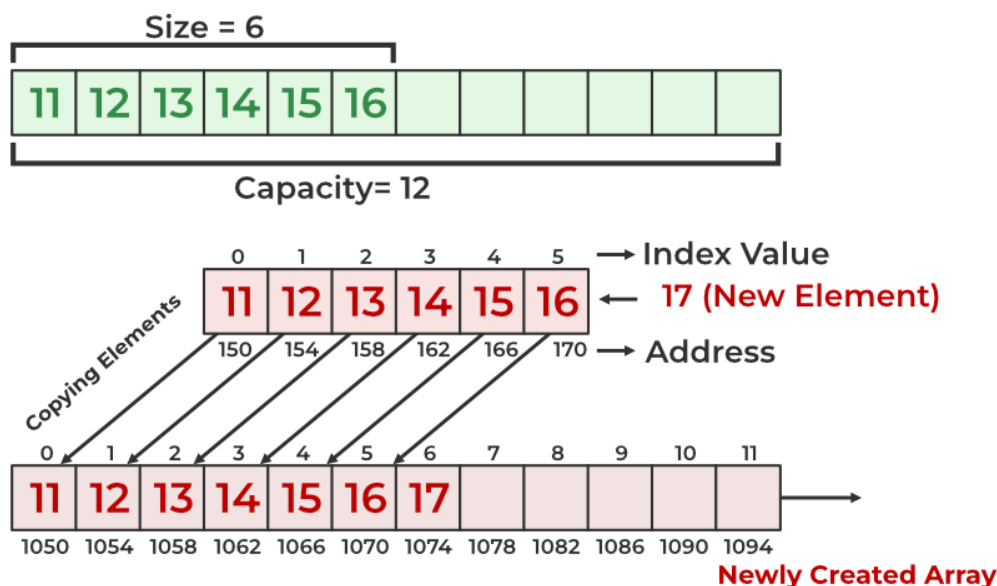
دسترسی به اعضای یک آرایه سرعت زیادی داره و حافظه کم‌تری اشغال می‌کنه. اما از طرفی اندازه ثابتش باعث می‌شه هنگام بیش‌تر شدن اعضا مجبور به کپی کردن کل اعضا باشیم. و آرایه‌های جاوا متدی برای اضافه یا حذف کردن ندارن و در صورتی که بهشون نیاز داشته باشیم باید خودمون پیاده‌سازیشون کنیم. بنابراین استفاده از آرایه تنها زمانی منطقیه که تعداد اعضامون از قبل مشخص باشه، عملکرد برنامه حیاتی باشه، و یا زمانی که با کدهای Low-Level (مثل بافر برای عملیات‌های ورودی و خروجی فایل) سروکار داریم.

در صورتی که به تغییر اندازه پس از آغاز کار، و عملیات‌های پیشرفته‌تری مثل مرتب‌سازی و جست‌وجو نیاز داشته باشیم، باید به سراغ داده‌ساختارهای پیچیده‌تر بریم.

## آرایه پویا<sup>۲</sup> و اینترفیس List

آرایه پویا یک داده‌ساختار دیگه است که با اضافه و حذف شدن اعضا، به‌طور خودکار اندازه‌ش رو بزرگ‌تر یا کوچک‌تر می‌کنه. بر خلاف آرایه‌های ثابت، آرایه‌های پویا مدیریت حافظه رو به‌طور داخلی انجام می‌دن و همزمان ویژگی Random Access رو هم حفظ می‌کنن. (می‌تونیم با داشتن ایندکس یک عضو، مستقیماً به مقدار آن دسترسی پیدا کنیم.)

در واقع هر آرایه پویا با مدیریت و ساخت آرایه‌های ساده کار خودش رو انجام می‌ده. تمام اطلاعات یک آرایه پویا در یک آرایه ساده ذخیره شده، و زمانی که یک آرایه پویا پر می‌شه (یعنی تعداد اعضا با طول تعیین‌شده‌ش برابر می‌شه)، آرایه پویا به‌طور خودکار یک آرایه ساده بزرگ‌تر رو درست می‌کنه و اعضای آرایه‌ی ساده اولیه رو تو آرایه‌ی ساده جدید کپی می‌کنه. بعد از این، آرایه‌ی ساده جدید جایگزین قبلی می‌شه. معمولاً اندازه آرایه ساده جدید دو برابر اندازه آرایه ساده اولیه انتخاب می‌شه. همچنین هنگام کم شدن اعضای آرایه پویا، اندازه آرایه کوچک‌تر می‌شه. معمولاً زمانی که تعداد اعضای موجود آرایه‌ی پویا از یک‌چهارم اندازه آن کم‌تر بشن اعضا رو به یک آرایه ساده جدید با اندازه نصف قبلی منتقل می‌کنیم. این عملیات‌ها همگی به‌طور پیش‌فرض در متدهای حذف و اضافه اتفاق می‌افتن و بعد از پیاده‌سازی یک آرایه پویا و زمانی که باهاش کار می‌کنیم لازم نیست با این جزئیات دست‌وپنجه نرم کنیم.



<sup>2</sup> Dynamic Array

آرایه پویا همچنان یک مشکل داره و اون هم زمانیه که برای اضافه کردن و حذف کردن اعضا تلف می‌شه. اگه بخوایم عنصری رو به آخر آرایه اضافه کنیم مشکلی نیست، اما در خیلی از مواقع ما می‌خوایم به وسط و یا ابتدای آرایه عنصر جدیدی اضافه کنیم. برای این کار آرایه باید تمام عناصری که بعد از عنصر جدید وجود دارن رو به اندازه یک جایگاه جابه‌جا کنه. یا اگه عنصر اول یک آرایه رو حذف کنیم، باید تمامی عناصر رو یک جایگاه به عقب بیاریم. این موضوع شاید وقتی به یک آرایه با اندازه کوچیک فکر می‌کنین چندان مشکل بزرگی نباشه، اما وقتی که بخوایم یک برنامه بزرگ طراحی کنیم می‌تونه باعث کند شدن کارمون بشه.

## اینترفیس List

قبل از این‌که به سراغ پیاده‌سازی آرایه پویا بریم، وقتشه که به‌طور کامل با اینترفیس List در فریم‌ورک کالکشن جاوا آشنا بشیم. اینترفیس List یکی از فرزندان اینترفیس Collection ه و تنها شرط اضافه‌ش نسبت به خود Collection اینه که اعضای اون ترتیب مشخص دارن. بنابراین تمام متدهاش رو به ارث می‌بره. در ادامه متدهای اینترفیس List رو می‌بینیم.

- `public boolean add(E element)`: آبجکت element رو به انتهای لیست اضافه می‌کنه.

- `public void add(int index, E element)`: آبجکت element رو در ایندکس index به لیست اضافه می‌کنه. در صورتی که index کم‌تر از صفر یا بزرگ‌تر از طول لیست باشه، اکسپشن `IndexOutOfBoundsException` ترو می‌شه. در صورتی که از قبل یک آبجکت در ایندکس داده‌شده باشه، ایندکس اون آبجکت و تمام آبجکت‌های بعدی یکی بیش‌تر می‌شن.

- `public void remove(int index)`: آبجکتی که ایندکس برابر با index دارد حذف می‌شود.

- `public E get(int index)`: آبجکتی که در جایگاه برابر با index قرار داره رو برمی‌گردونه اما تغییری در لیست ایجاد نمی‌کنه.

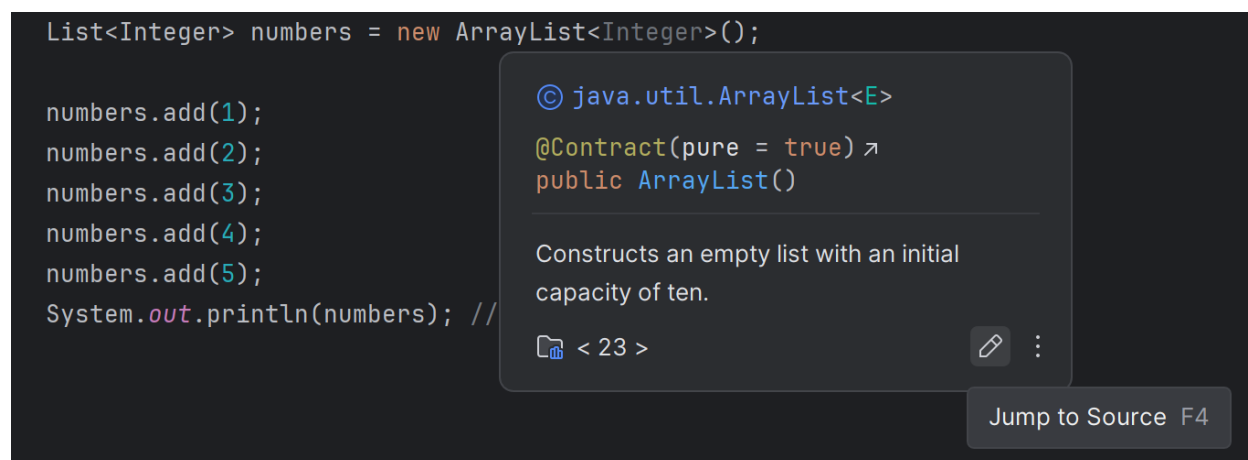
- `public Object set(int index, E element)`: آبجکتی که در جایگاه index قرار داره رو با آبجکت element جایگزین می‌کنه. هیچ عضو دیگه‌ای از لیست تحت تاثیر قرار

نمی‌گیره. در صورتی که آجکتی در جایگاه index موجود نباشد، `IndexOutOfBoundsException` ترو می‌شود.

سایر متدهای `Collection` مثل `size()` بدون تغییر در این اینترفیس هم وجود دارن. تایپ `E` در این متدها به تایپ جنریک کلاس `List` اشاره داره.

## ArrayList در جاوا

کلاس `ArrayList` که از قبل با اون آشنایی داریم، پیاده‌سازی آرایه پویا از طریق اینترفیس `List` a. متدهای کلاس `ArrayList` که در هفته اول باهاشون آشنا شدین همگی پیاده‌سازی متدهای اینترفیس `List` هستن. در صورتی که در محیط اینتلی‌جی روی تعریف یک `ArrayList` نگه داریم می‌تونیم با کلیک روی آیکون مداد، سورس‌کد `ArrayList` در زبان جاوا رو ببینیم.



(می‌تونیم بعد از کلیک روی `ArrayList` دکمه `F4` رو هم بزنیم تا سورس‌کدش براتون باز بشه.) توی صفحه سورس‌کد می‌بینیم که کلاس `ArrayList` از اینترفیس `List` (و چند اینترفیس دیگه) پیاده‌سازی شده.

با خوندن متدهای متفاوت `ArrayList` می‌تونیم بین ابزارهای که تا الان ازش استفاده می‌کردین دقیقاً چه طوری پیاده‌سازی شده. داده‌های `ArrayList` توی یک آرایه ساده به اسم `elementsData` ذخیره شده. یکی از فیلدهای کلاس `ArrayList` هست.

این کلاس علاوه‌بر متدهای اصلی `List` و `Collection` که بهشون اشاره کردیم متدهای کمکی دیگه‌ای هم داره. مثلاً متد `grow()` که با گرفتن ورودی `minCapacity` اندازه آرایه رو طوری افزایش

می‌ده که تعداد اعضای قابل‌ذخیره در آرایه حداقل به `minCapacity` برسه. اگه می‌خواین دقیق‌تر متوجه بشین که این کار چه‌طور انجام می‌شه می‌تونین سورس‌کد این متد رو بخونین:

```
private Object[] grow(int minCapacity) {
    int oldCapacity = elementData.length;
    if (oldCapacity > 0 || elementData != DEFAULTCAPACITY_EMPTY_ELEMENTDATA)
    {
        int newCapacity = ArraysSupport.newLength(oldCapacity,
            minCapacity - oldCapacity, /* minimum growth */
            oldCapacity >> 1           /* preferred growth */);
        return elementData = Arrays.copyOf(elementData, newCapacity);
    } else {
        return elementData = new Object[Math.max(DEFAULT_CAPACITY,
minCapacity)];
    }
}
```

اگه همه‌چیز برای تغییر اندازه درست پیش بره، به خط هشتم کد بالا می‌رسیم. می‌بینین که این متد در نهایت `elementsData` رو با یک آرایه جدید جایگزین می‌کنه و اون رو برمی‌گردونه. `elementsData` جدید مقادیرش رو از خود `elementsData` کپی می‌کنه ولی اندازه‌ش رو به `newCapacity` تغییر داده. این‌که مقدار `newCapacity` از کجا به دست اومده رو می‌تونین توی خط‌های ۵ تا ۷ ببینین.

در ادامه می‌تونین پیاده‌سازی متد `add(int index, E element)` رو هم ببینین. این‌که این متد چی‌کار می‌کنه رو توی متدهای اینترفیس `List` فهمیدین و حالا می‌تونین ببینین چه‌طوری اون کارها برای `ArrayList` پیاده‌سازی شده.

```
public void add(int index, E element) {
    rangeCheckForAdd(index);
    modCount++;
    final int s;
    Object[] elementData;
    if ((s = size) == (elementData = this.elementData).length)
        elementData = grow();
    System.arraycopy(elementData, index,
        elementData, index + 1,
        s - index);
    elementData[index] = element;
    size = s + 1;
}
```

خوندن سورس‌کد کلاس‌ها و متدهای پیش‌فرض یک زبان می‌تونه توی پروسه دیباگ کردن و رفع اشکالات کدتون خیلی کمک کنه. توی نوشتن برنامه‌های پیشرفته‌تر به مشکلاتی برخورد می‌کنین که اگه بدونین متدها و کلاس‌هایی که ازشون استفاده می‌کنین دقیقاً چه‌طور پیاده‌سازی شدن، خودتون می‌تونین حلشون کنین. طبیعتاً کسی نمی‌تونه تمام پیاده‌سازی‌های یک زبان و پکیج‌هاش رو بلد باشه،

اما مهارت خوندن سورس‌کد بهتون کمک می‌کنه هر وقت به مشکلی برخورد کردین خودتون علتش رو پیدا کنین. برای همین توصیه می‌شه به پیاده‌سازی‌های ArrayList نگاهی بندازین. توی تمریناتتون قراره این داده‌ساختارها رو پیاده‌سازی کنین و خوندن پیاده‌سازی‌های کامل و تست‌شده رسمی می‌تونه ایده‌های خوبی بهتون بده.

## هزینه زمانی تغییر اندازه ArrayList

مطالب این بخش خارج از مباحث درس «برنامه‌سازی پیشرفته» است و خوندن این بخش اختیاریه.

آرایه پویا یکی از مشکلات اساسی آرایه رو رفع کرد و حالا می‌تونیم اندازه داده‌ساختارمون رو حتی بعد از مقداردی اولیه تغییر بدیم. اما برای رفع این مشکل، اجرای عملیات‌هایی مثل اضافه و حذف کردن بیش‌تر شد. برای این‌که بهتر به این موضوع پی ببرین به کد زیر دقت کنین.

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        int totalElements = 1_000_000;

        long startTime1 = System.currentTimeMillis();
        List<Integer> listWithSize = new ArrayList<>(totalElements);
        for (int i = 1; i <= totalElements; i++) {
            listWithSize.add(i);
        }
        long endTime1 = System.currentTimeMillis();
        System.out.println("Pre-sized ArrayList time: " + (endTime1 -
            startTime1) + " ms");
    }
}
```

توی این کد می‌بینین که یک ArrayList با اندازه totalElements (که در این‌جا برابر با یک میلیونه) ساخته شده، و بعدش با یک حلقه for اعداد ۱ تا ۱ میلیون به ترتیب به این لیست اضافه شدن. قبل و بعد از این فرآیند لحظه دقیق شروع و اتمام رو با متد currentTimeMillis() ثبت کردیم و با کم کردن این دو مقدار می‌تونیم ببینیم کل این فرآیند چند میلی ثانیه طول کشیده. یک بار این کد رو کپی و اجرا کنین و نتیجه رو ببینین.

حالا کد زیر رو ببینین.

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
```

```

int totalElements = 1_000_000;

long startTime2 = System.currentTimeMillis();
List<Integer> listDefaultSize = new ArrayList<>(10);
for (int i = 1; i <= totalElements; i++) {
    listDefaultSize.add(i);
}
long endTime2 = System.currentTimeMillis();
System.out.println("Default-sized ArrayList time: " + (endTime2 -
startTime2) + " ms");
}
}

```

این دقیقاً مثل کد قبلیه، تنها تفاوت اینه که این دفعه ArrayList با اندازه اولیه ۱۰ ساخته شده. بنابراین در حلقه for و با اضافه کردن اعضای جدید، هر بار که اعضا از اندازه فعلی بیش‌تر بشه مجبوره تا سایز ArrayList رو افزایش بده. قاعدتاً این کار باید هزینه زمانی ما رو بیش‌تر کنه، نه؟

با اجرای این کد به نتیجه‌ای مشابه با نتیجه زیر می‌رسین:

```

Pre-sized ArrayList time: 30 ms
Default-sized ArrayList time: 75 ms

```

می‌بینین که توی روش دوم زمان اجرا از دو برابر روش اول هم بیش‌تره. اما اگه تعداد اعضای این لیست بیش‌تر بشه چی؟ مقدار totalElements رو از یک میلیون به صد میلیون تغییر بدین و کد رو دوباره اجرا کنین. نتیجه اجرای دوباره کدهای قبلی با مقدار جدید به این شکله؛

```

Pre-sized ArrayList time: 3987 ms
Default-sized ArrayList time: 5953 ms

```

می‌بینین که اختلاف این دو به نسبت اندازه‌شون کم‌تر شده. در واقع، تغییر اندازه ArrayList برای ما هزینه زیادی نداره. این‌که علتش چیه در درس‌های ترم‌های آینده‌تون توضیح داده می‌شه.<sup>۳</sup>

<sup>۳</sup> علت این موضوع در تحلیل زمانی الگوریتم‌ها و Amortized Analysis مشخص می‌شه. به این مبحث در فصل ۱۷ کتاب Introduction to Algorithms کاملاً پرداخته شده. این کتاب مرجع اصلی دروس «ساختمان داده‌ها و الگوریتم‌ها» و «طراحی و تحلیل الگوریتم‌ها» است. می‌تونید از این [لینک](#) نسخه سوم این کتاب رو دانلود کنین.



## Iterator؛ ابزاری برای پیمودن کالکشن‌ها

قبل از این‌که به سراغ داده‌ساختار بعدی بریم، می‌خوایم یاد بگیریم که چه‌طور ایمن و بهینه بین اعضای یک کالکشن پیمایش انجام بدیم. پیمایش یا `traverse` در این‌جا به معنای اینه که در هر مرحله سراغ یکی از اعضای کالکشن بریم و دونه‌دونه اون‌ها رو بخونیم و یا تغییر بدیم. ساده‌ترین روش می‌تونه یک حلقه `for` باشه که از راه‌های مختلف (مثل `ایندکس`) بین اعضای یک کالکشن پیمایش انجام می‌ده، این روش وقتی می‌خوایم اعضای یک کالکشن رو فقط بخونیم مشکلی نداره، اما زمانی که بخوایم اعضای کالکشن رو حین پیمایش تغییر بدیم، و یا عضوی اضافه و کم کنیم احتمالاً به مشکلاتی برمی‌خوریم. مثلاً `ConcurrentModificationException` شدن یکی از مشکلاتی است که ممکنه سراغمون بیاد.

`Iterator` یکی از ابزارهای فریم‌ورک کالکشن‌های جاواست که با جدا کردن منطق پیمایش از اساس داده‌ساختار، یک روش استاندارد برای دسترسی به اعضا به ما می‌ده، و همزمان جزئیات پیاده‌سازی رو پنهان می‌کنه.

### اینترفیس `Iterator`

یک `iterator` آبجکتیه که به ما اجازه می‌ده بین یک دنباله از مقادیر پیمایش انجام بدیم. این عملیات یک روش استاندارد داره، اینترفیس `Iterator`. این اینترفیس سه متد اصلی داره؛

- `public boolean hasNext()`: در صورتی که هنوز تمامی اعضای کالکشن رو نپیموده باشیم، این متد مقدار `true` برمی‌گردونه. یعنی زمانی که هنوز اعضای بیش‌تری برای پیمایش باقی مونده.

- `public E next()`: این متد عضو بعدی کالکشن (یک آبجکت با تایپ جنریک `E`) رو برمی‌گردونه.

- `public void remove()`: این متد آخرین آبجکتی که با متد `next()` برگردونده شده رو از کالکشن حذف می‌کنه.

مثال زیر نشون می‌ده که چه‌طور از یک `iterator` استفاده کنیم تا تمام اعضای یک کالکشن رو چاپ کنیم.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Iterator;

public class Main {
    public static void main(String[] args) {
        List<String> tas = new ArrayList<>( List.of(
            "Sania", "Pariya", "Pedram", "Nastaran", "Hooman"
        ));

        Iterator<String> it = tas.iterator();

        while (it.hasNext()) {
            String ta = it.next();
            System.out.println(ta);
        }
    }
}
```

خروجی این کد به شکل زیر خواهد بود.

```
Sania
Pariya
Pedram
Nastaran
Hooman
```

بعد از استفاده کردن متد `next()` برای برگردوندن عضو بعدی، می‌تونیم متد `remove()` رو صدا بزنیم تا اون عضو رو حذف کنیم.

حالا به مثال زیر دقت کنین:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Iterator;

public class Main {
    public static void main(String[] args) {
        List<String> tas = new ArrayList<>( List.of(
            "Sania", "Pariya", "Pedram", "Nastaran", "Hooman"
        ));

        Iterator<String> it = tas.iterator();

        while (it.hasNext()) {
            String ta = it.next();
            if (ta.equals("Pedram")) {
                it.remove();
            }
        }
        System.out.println(tas);
    }
}
```

توی این مثال بعد از هر مرحله چک می‌کنیم که آخرین عضوی که `next()` خروجی داده برابر با استرینگ "Pedram" بوده یا نه، و اگر بوده اون رو حذف می‌کنیم.

```
[Sania, Pariya, Nastaran, Hooman]
```

در نهایت خروجی این کد نشون می‌ده که این استرینگ از بین اعضای `ArrayList` ای که در ابتدا ساختیم حذف می‌شه

## حلقه for-each

در جاوا، می‌تونیم به جای استفاده مستقیم از `Iterator`، از حلقه‌های `for-each` استفاده کنیم که به صورت داخلی از `iterator` ها استفاده می‌کنن. (به این نوع حلقه `enhanced for loop` هم گفته می‌شه.) در واقع این حلقه به ما کمک می‌کنه که بدون نیاز به ایندکس و یا `iterator` توی اعضای یک کالکشن پیمایش انجام بدیم. سینتکس حلقه‌های `for-each` به شکل زیرن.

```
for (E element : collection) {
    // use element
}
```

کد بالا با این کد معادل است:

```
Iterator<E> it = collection.iterator();
while (it.hasNext()) {
    String element = it.next();
}
```

اگر بخوایم مثال قبلی رو با حلقه `for-each` اجرا کنیم چنین کدی خواهیم داشت.

```
public static void main(String[] args) {
    List<String> tas = new ArrayList<>( List.of(
        "Sania", "Pariya", "Pedram", "Nastaran", "Hooman"
    ));

    for (String ta : tas) {
        System.out.println(ta);
    }
}
```

کد بالا خروجی‌ای مشابه استفاده از `iterator` داره. حالا اگر بخوایم عملیات حذف کردن یکی از اعضا رو هم با `for-each` انجام بدیم به این کد می‌رسیم:

```
public static void main(String[] args) {
    List<String> tas = new ArrayList<>( List.of(
        "Sania", "Pariya", "Pedram", "Nastaran", "Hooman"
    ));

    for (String ta : tas) {
        if (ta.equals("Pedram")) {
            tas.remove(ta);
        }
    }
}
```

```

    }
}
}

```

این کد رو اجرا کنین. با اجرای این کد اکسپشن `ConcurrentModificationException` ترو می شه. پس متوجه می شیم که با حلقه `for-each` نمی تونیم عضوی رو تغییر بدیم. یکی دیگه از محدودیت های حلقه `for-each` اینه که نمی تونیم ایندکس آجکت ها رو پیدا کنیم. بنابراین `for-each` برای زمانی مناسبه که می خوایم تمام اعضا رو بدون تغییر دادن بخونیم و نیازی به ایندکسشون نداریم.

## اینترفیس `ListIterator`

همون طور که از اسمش مشخصه، این اینترفیس یک نسخه اکستندشده از اینترفیس `Iterator` ه که برای `List` ها طراحی شده. اینترفیس `ListIterator` از **پیمایش دوطرفه** پشتیبانی می کنه. متدهای مهم این اینترفیس به شرح زیره:

- `public boolean hasPrevious()` در صورتی که عنصر قبلی ای وجود داشته باشه، مقدار `true` رو برمی گردونه.

- `public E previous()`: آجکت عنصر قبلی رو برمی گردونه.

- `public void add(E element)`: آجکت `element` رو در قبل از محل فعلی `iterator` اضافه می کنه.

- `public void set(E element)`: آجکت `element` رو جایگزین آخرین عنصری که برگردونده شده می کنه.

```

import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

public class Main {
    public static void main(String[] args) {
        List<String> tas = new ArrayList<>(List.of(
            "Sania", "Pariya", "Arman", "Nastaran", "Hooman"));

        ListIterator<String> lit = tas.listIterator();

        while (lit.hasNext()) {
            System.out.println(lit.next());
        }

        while (lit.hasPrevious()) {
            String ta = lit.previous();
            if (ta.equals("Arman")) {

```

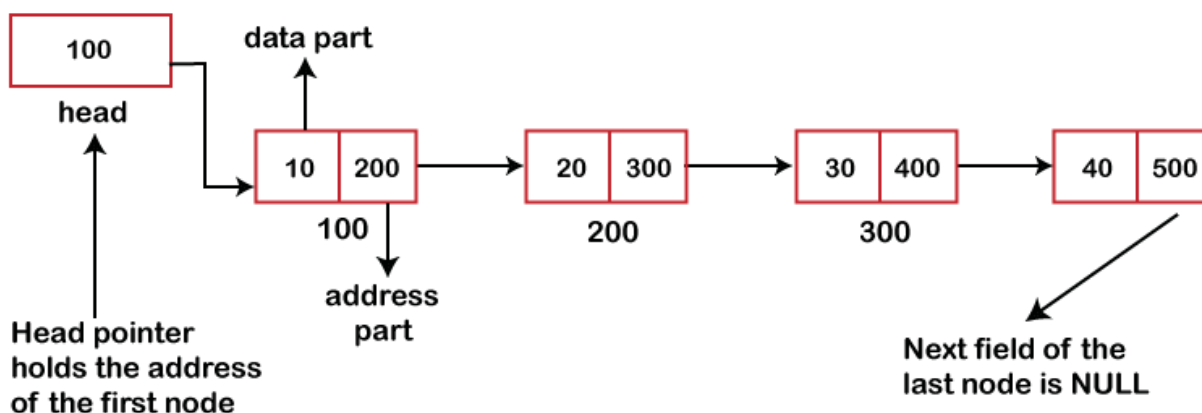
```
        lit.set("Pedram");
    }
    System.out.println(tas);
}
}
```

در این کد ابتدا با یک حلقه `while` تمامی اعضای لیست به ترتیب چاپ می‌شن، بعدش با یک پیمایش معکوس روی اعضای این لیست بررسی می‌کنیم که اگر عضوی برابر با "Arman" بود اون رو به "Pedram" تغییر بده و در نهایت لیست نهایی رو چاپ می‌کنیم. اگه این کد رو اجرا کنیم می‌بینیم که در انتها یکی از اعضای این لیست تغییر کرده.

## لیست پیوندی

در این بخش به یکی دیگه از داده ساختارهای پایه ای می پردازیم که با استفاده از راس، حذف و اضافه کردن عنصر جدید رو بهینه می کنه. بر خلاف آرایه ها، لیست های پیوندی نیازی به بلاک های حافظه متوالی ندارن. این ویژگی باعث می شه که برای داده هایی که اندازه شون دائماً در حال تغییره عالی باشن.

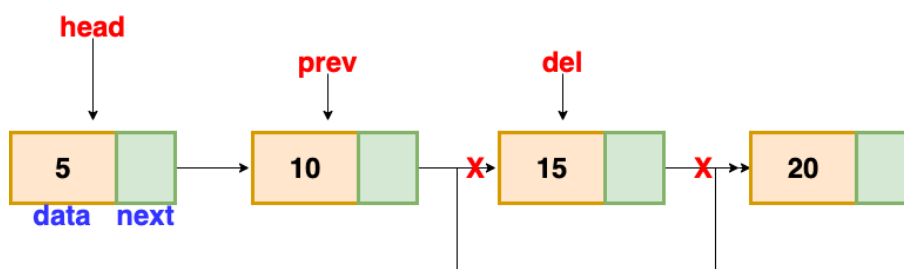
لیست پیوندی از راس های مختلف تشکیل شده. این راس ها هر کدوم شامل دو بخش: **داده و رفرنس**. داده در این جا به معنای مقداریه که توی لیست پیوندی ذخیره می کنیم. رفرنس (یا پوینتر) هم آدرس حافظه مربوط به یک راس دیگه از لیسته. در واقع داده های مختلف یک لیست پیوندی در موقعیت های متفاوتی در حافظه ذخیره شدن و هر کدوم از اعضا ش می دونن که عضو بعدی کجاست.



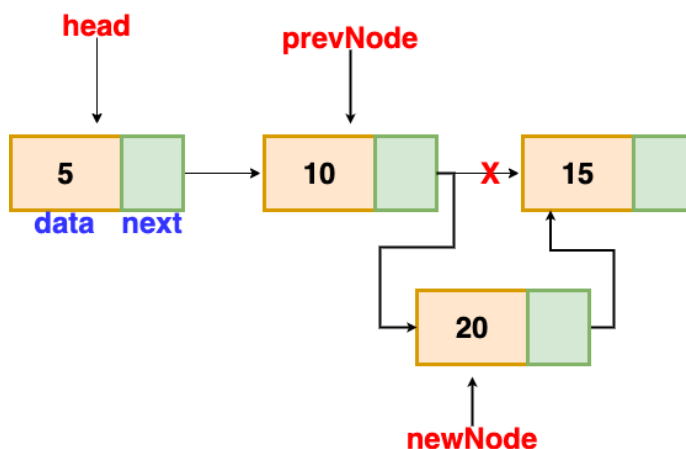
لیست های پیوندی معمولاً یک راس خالی به عنوان head دارن. راس سر اولین راسه که رفرنس اون برای شناسایی کل لیست پیوندی استفاده می شه. تنها مقدار راس سر، آدرس راس بعدیه. راس های بعدی هر کدوم یک مقدار و یک آدرس ذخیره می کنن. در نهایت به راسی می رسیم که بخش رفرنس اون خالیه و این یعنی به انتهای لیست پیوندی رسیدیم.

**اضافه کردن عضو به لیست پیوندی هیچ محدودیتی نداره.** همون طور که دیدین در آرایه پویا با این که مشکل ثابت بودن اندازه رفع شده بود، اما برای این کار هزینه زمانی زیادی پرداخت می کردیم و برای داده هایی که اندازه شون دائم در حال تغییره مناسب نبودن. اما توی لیست پیوندی این مشکل رفع شده. از طرفی یکی دیگه از مشکلات آرایه ها این بود که برای حذف یا اضافه کردن عنصری که در انتهای

آرایه نبود، باید زمان زیادی صرف می شد. توی لیست پیوندی حذف و اضافه کردن عناصر هزینه کمی دارد، چون فقط کافیه رفرنس قبلی رو تغییر بدیم.



Deleting a Node in Linked List



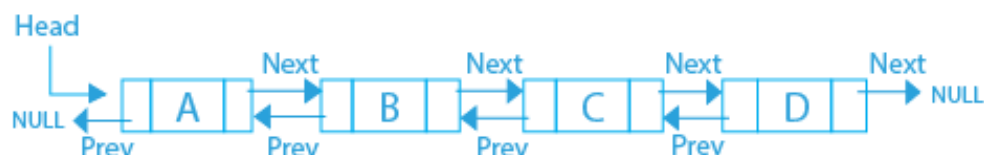
Insertion after a given node

همون طور که قبل تر گفتیم، آرایه مثل یک قفسه کتابه که برای هر کتاب یک جایگاه مخصوص داره. اما لیست پیوندی مثل یک زنجیر طولانیه که هر راس اون در یک جای حافظه قرار گرفته.

یکی از مشکلاتی که توی لیست پیوندی بهش برخورد می کنیم اینه که نمی تونیم پیمایش معکوس انجام بدیم. فرض کنین راس انتهایی (یا یکی از راس های میانی) لیست پیوندی رو داریم. می تونیم راس head رو پیدا کنیم؟ نه. حتی به راس های قبل از اون هم دسترسی نداریم. برای حل این مشکل یک داده ساختار دیگه به اسم «لیست پیوندی دوطرفه»<sup>۴</sup> طراحی شده.

<sup>4</sup> Doubly Linked List

## Doubly Linked List :



هر راس لیست پیوندی دوطرفه شامل سه بخشه؛ **داده**، **رفرنس به راس بعدی**، و **رفرنس به راس قبلی**. تنها تفاوت لیست پیوندی دوطرفه با لیست پیوندی ساده اینه که هر راس می‌دونه راس قبل از خودش کجاست. این ویژگی پیمایش معکوس در لیست پیوندی رو ممکن می‌کنه و به ما اجازه می‌ده تا از یک راس، به تمامی راس‌های لیستمون دسترسی داشته باشیم. وقتی بخوایم در عمل از لیست پیوندی استفاده کنیم، معمولاً از لیست پیوندی دوطرفه استفاده می‌کنیم.

اما چی باعث می‌شه که از اول سراغ لیست پیوندی نرفتیم؟ فرض کنیم می‌خوایم مقدار سومین عنصر یک لیست پیوندی رو بخونیم. برای این کار باید از راس سر شروع کنیم و سه بار به راس بعدی بریم. این به نظر مشکل‌زا نیست، اما وقتی بخوایم عنصر صدم رو پیدا کنیم چه‌طور؟ در آرایه می‌تونستیم به راحتی با ایندکس، به عنصر مورد نظر برسیم. اما این‌جا نمی‌تونیم.

در واقع همون‌طور که حذف و اضافه کردن عنصر به میانه آرایه هزینه زیادی داشت، خوندن یکی از عناصر میانه لیست پیوندی هم هزینه زیادی داره. بنابراین در خیلی از سناریوها، استفاده از لیست پیوندی باعث می‌شه برنامه ما کندتر بشه. ما باید موقعیت مناسب برای استفاده از هر داده‌ساختار رو با توجه به کاربرد داده‌های برنامه‌مون پیدا کنیم.

## پیاده‌سازی لیست پیوندی

برای این‌که لیست پیوندی رو با جاوا پیاده‌سازی کنیم، منطقی‌ترین گزینه اینه که از اینترفیس `List` استفاده کنیم. این اینترفیس برای داده‌ساختارهایی که ترتیب دارن و می‌تونن اعضای تکراری هم بپذیرن. اما برای این‌که بتونیم خود لیست پیوندی رو پیاده‌سازی کنیم، اول باید مشخص کنیم هر کدوم از راس‌های لیستمون از چه تایپین.

برای راس‌های لیست پیوندی کلاس `Node` رو می‌سازیم. این کلاس شامل دو تا فیلده، فیلد `data` و فیلد `next`. تایپ فیلد `data` جنریکه و می‌تونیم موقع ساخت اینستنس از این کلاس تعیین کنیم که



آبجکتی در اون ذخیره می‌شه. تایپ فیلد next هم از جنس خود Node هست و در واقع رفرنس به یک آبجکت دیگه (راس بعدی) از کلاس Node ه.

```
private static class Node<E> {
    E data;
    Node<E> next;

    Node(E data) {
        this.data = data;
        this.next = null;
    }
}
```

حالا با پیاده‌سازی کلاس لیست پیوندی از اینترفیس List می‌تونیم کل لیست رو بسازیم. همون‌طور که گفتیم تنها چیزی که برای معرفی لیست پیوندی نیاز داریم، راس head است. این راس شامل رفرنس به راس بعدیه و همین‌طوری زنجیروار می‌تونیم تا انتهای لیست پیش بریم.

```
public class SimpleLinkedList<E>
    implements List<E> {
    private Node<E> head;
    private int size;

    @Override
    public boolean add(E element) {
        Node<E> newNode = new Node<>(element);
        if (head == null) {
            head = newNode;
        } else {
            Node<E> current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
        size++;
        return true;
    }

    @Override
    public E get(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }
        Node<E> current = head;
        for (int i = 0; i < index; i++) {
            current = current.next;
        }
        return current.data;
    }
}
```

در کد بالا فقط دو متد add() و get() رو پیاده‌سازی کردیم.

متد `add()` یک عضو از آبجکتی که قراره در راس های این لیست ذخیره کنیم رو به عنوان ورودی می گیره. بعدش با اون ورودی یک اینستنس از کلاس `Node` می سازیم و اسمش رو `newBode` می ذاریم. حالا باید ببینیم این راس جدید رو کجای لیست می ذاریم. پس بررسی می کنیم که راس `head` داریم یا نه. اگه نداشتیم، راسی `newNode` رو به عنوان `head` قرار می دیم، مقدار `size` رو یکی بیش تر می کنیم و از مقدار `true` رو برمی گردونیم.

در صورتی که راس `head` داشته باشیم، باید از `head` شروع کنیم و دونه دونه راس های لیست پیوندی رو پیمایش کنیم تا به انتهای لیست برسیم. پس یک اینستنس به اسم `current` از کلاس `Node` می سازیم و با یک حلقه `while`، تمام راس های لیست رو طی می کنیم و هر دفعه بررسی می کنیم که راس فعلی مون `next` داره یا نه. هر وقت مقدار `next` راس `current` برابر `null` باشه، یعنی راس انتهاییه. پس `next` راس انتهایی رو برابر `newNode` قرار می دیم، مقدار `size` رو یکی بیش تر می کنیم و از مقدار `true` رو برمی گردونیم.

متد `get(int index)` با گرفتن یک عدد، آبجکت ذخیره شده در اون شماره رو به عنوان جواب برمی گردونه. گفتیم که توی لیست پیوندی ایندکس نداریم، پس برای پیدا کردن خروجی دادن یک راس با ایندکس مشخص باید از راس `head` شروع کنیم و به تعداد `index` جلو بریم تا به راس دل خواه برسیم. در کد بالا ابتدا چک می کنیم که `index` ورودی از صفر بزرگ تر و `size` کوچک تر باشه. اگه نبود، اکسپشن `IndexOutOfBoundsException` رو ترو می کنیم. اگه ورودی مشکلی نداشت، یک اینستنس از کلاس `Node` به اسم `current` می سازیم و دونه دونه اعضای لیست رو طی می کنیم و وقتی به عدد خواسته شده رسیدیم اون رو برمی گردونیم.

کدهای بالا رو کپی کنین و توی اینتلی جی اضافه شون کنین. دقت کنین که چون کلاس `SimpleLinkedList` از اینترفیس `List` پیاده سازی شده، باید حتماً تمام متدهای اون رو اورراید کنه. لازم نیست توی این متدها چیز خاصی بفرستین و می تونن فقط `null` یا `false` خروجی بدن. برای راحت تر شدن کارتون موس رو ببرین روی اسم کلاس، و گزینه `Implement methods` رو بزنین. توی پنجره ای که باز می شه `OK` رو بزنین و خودش تمام متدها رو اضافه می کنه. حالا کد زیر رو به کلاس `Main` اضافه کنین و اجراش کنین.

```
public static void main(String[] args) {
    SimpleLinkedList<String> names = new SimpleLinkedList<>();

    names.add("Arman");
    names.add("Pedram");
}
```

```
System.out.println(names.get(1));
}
```

در این جا با ساختن یک اینستنس از کلاس SimpleLinkedList، دو تا استرینگ بهش اضافه کردیم. بعدش مقدار ذخیره‌شده در راس دوم رو با استفاده از متد get() به دست آوردیم و چاپ کردیم. این کدها رو توی دستگاه خودتون اجرا کنین و نتیجه رو ببینین.

## LinkedList در جاوا

مشابه ArrayList، داده‌ساختار لیست پیوندی نیز به طور پیش فرض در فریم‌ورک کالکشن پیاده‌سازی شده. این پیاده‌سازی هم مشابه نمونه ساده‌ای که بالا دیدیم از اینترفیس List پیاده‌سازی شده. همون طور که توی بخش‌های قبلی دیدیم، می‌تونین به سورس‌کد این پیاده‌سازی هم دسترسی پیدا کنین.

```
List<String> linkedList = new LinkedList<>();

linkedList.add("Nastaran");
linkedList.addFirst(e: "Arman");
linkedList.addLast(e: "Hooman");

Iterator<String> it = linkedList.
while (it.hasNext()) {
    System.out.println(it.next());
}
```

© java.util.LinkedList<E>  
@Contract(pure = true) >  
public LinkedList()  
Constructs an empty list.  
< 23 >  
Jump to Source F4

این پیاده‌سازی شامل بیش‌تر از ۱۵۰۰ خط کد ه و کلی متد مختلف داره. پیشنهاد می‌کنم به کدهای متدهای add(), linkLast(), و remove() نگاه کنین. شما قراقره توی تمریناتتون داده‌ساختار لیست پیوندی رو پیاده‌سازی کنین، و دیدن پیاده‌سازی‌های خود جاوا برای این داده‌ساختار می‌تونه خیلی کمکتون کنه.

## استفاده از Iterator در لیست پیوندی

در لیست پیوندی حذف و اضافه کردن عنصر به میانه عناصر قبلی سریع‌تر از آرایه‌هاست. اما از طرفی برخلاف آرایه‌ها که می‌تونستیم به سرعت یک عنصر رو پیدا کنیم، در لیست پیوندی باید هزینه زمانی زیادی رو برای پیدا کردن یک عنصر پرداخت کنیم.

فرض کنیم می‌خواهیم تمام عناصر یک داده‌ساختار با  $n$  عنصر رو پیمایش کنیم. در آرایه می‌تونستیم در هر مرحله عنصر  $i$  ام رو طی کنیم. اما در لیست پیوندی ایندکس نداریم، بنابراین در هر مرحله باید از راس سر شروع کنیم و به اندازه  $i$  بار به راس بعدی برویم.

پس برای پیمایش  $n$  عنصر در آرایه، به  $n$  عملیات نیاز داریم و برای پیمایش  $n$  عنصر در لیست پیوندی به  $\frac{n(n+1)}{2}$  عملیات نیاز داریم.

```
LinkedList<Integer> linkedList = new LinkedList<>();

for (int i = 0; i < 100_000; i++) {
    linkedList.add(i);
}

long start = System.currentTimeMillis();
for (int i = 0; i < linkedList.size(); i++) {
    int value = linkedList.get(i);
}
long end = System.currentTimeMillis();
System.out.println("Time taken: " + (end - start) + " ms");
```

کد بالا رو برای خودتون اجرا کنید. توی این کد با کلاس `LinkedList` یک لیست پیوندی ساختیم و اعداد ۱ تا ۱۰۰,۰۰۰ رو بهش اضافه کردیم. بعدش با یک حلقه `for`، مقدار هر عضو رو گرفتیم و در یک آبجکت موقع ذخیره کردیم. با اجرای کد بالا می‌تونین زمان اجرای این برنامه رو ببینین که بسته به شرایط مختلف می‌تونه متغیر باشه، اما در اجرای من چیزی بین ۸,۰۰۰ الی ۹,۰۰۰ میلی ثانیه طول کشید.

حالا چه طور می‌تونیم این روند رو تسهیل کنیم؟ از لحاظ تئوری، راه بهتری برای رسیدن به راسی در میانه لیست پیوندی وجود نداره. اما می‌تونیم آبجکت هر راس رو نگه داریم تا برای پیدا کردن راس بعدی، به جای شروع از راس سر، از اون راس شروع کنیم. در بخش قبلی با `Iterator` ها آشنا شدیم و می‌دونیم که بهترین راه برای پیمایش عناصر یک داده‌ساختار، استفاده از یک `Iterator`، و به طور خودکار خودش این کار رو می‌کنه.

```
LinkedList<Integer> linkedList = new LinkedList<>();

for (int i = 0; i < 100_000; i++) {
    linkedList.add(i);
}

long start = System.currentTimeMillis();

Iterator<Integer> it = linkedList.iterator();
while (it.hasNext()) {
    int value = it.next();
}
```

```
long end = System.currentTimeMillis();  
System.out.println("Time taken: " + (end - start) + " ms");
```

کد بالا رو اجرا کنین و زمان اجراش رو با کد قبلی مقایسه کنین. این کد همون کاری که قبلاً کردیم رو با استفاده از Iterator انجام می‌ده، و زمان اجراش یه چیزی بین ۱ تا ۱۰ میلی ثانیه است.

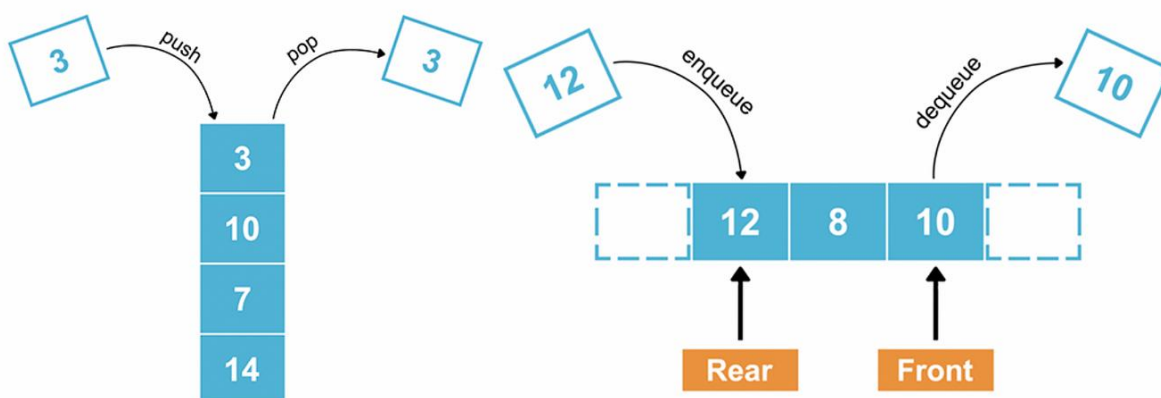
می‌تونین تعداد اعضای لیست رو بیشتر هم بکنین و زمان اجرا رو مقایسه کنین. می‌بینین که هرچه قدر تعداد اعضا باشه، تفاوت بین این دو روش بیشتر می‌شه. مثل همه بخش‌های دیگه برنامه‌نویسی، اگه هزینه زمانی یک برنامه رو کاهش بدیم، توی ورودی‌های بزرگ‌تر خودش رو بیشتر نشون می‌ده.

## پشته<sup>۵</sup> و صف<sup>۶</sup>

پشته و صف دو داده ساختار اساسین که از قوانین مخصوصی در حذف و اضافه کردن عنصر پیروی می کنند.

در پشته ما فقط آخرین عنصر اضافه شده رو می بینیم، اگر بخوایم عنصر جدیدی اضافه کنیم روی آخرین عنصر قرار می گیره، و در صورتی که بخوایم عنصری رو حذف کنیم باید آخرین عنصری که از قبل اضافه کردیم (تنها عنصری که می تونیم ببینیم) رو حذف کنیم. ایده پشته مشابه قرار دادن چند تا کاسه روی هم دیگه است. در پشته، اضافه کردن عنصر به push و حذف کردن از پشته به pop معروفه.

در صف فقط می تونیم به پشت صف عنصری اضافه کنیم، اما برخلاف پشته، برای حذف عنصر باید حتماً آخرین از جلوی صف یک عنصر را برداریم. در صف هم به اضافه کردن enqueue می گیم و حذف کردن رو هم dequeue می نامیم. به تصویر زیر دقت کنید.



در بخش چپ می تونیم تصویر یک پشته رو ببینیم، و در بخش راست تصویر یک صف.

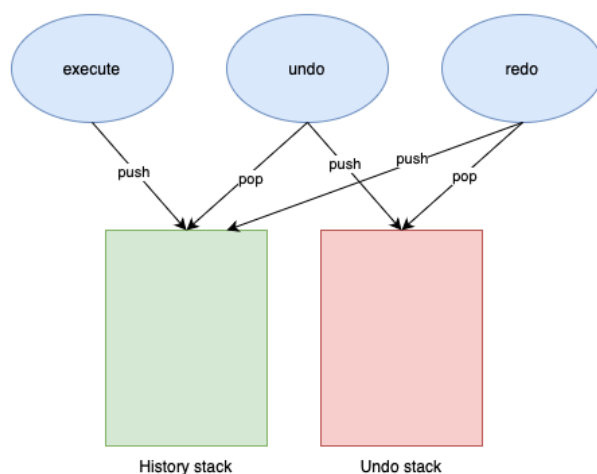
پشته ها از سیاست LIFO استفاده می کنند. LIFO مخفف "Last-in, first-out" است و مطابق اسمش یعنی آخرین عنصری که اضافه کردیم، اولین عنصریه که خارج می شه. صف هم از سیاست FIFO استفاده می کنه که مخفف "First-in, first-out" است، یعنی اولین عنصری که اضافه می کنیم، اولین عنصریه که خارج می کنیم.

<sup>5</sup> Stack

<sup>6</sup> Queue

شاید براتون سوال بشه که چنین داده‌ساختارهای ساده‌ای اصلاً به چه دردی می‌خورن. با داده‌ساختارهایی طرفیم که قابلیت‌های کم‌تری نسبت به آرایه و لیست پیوندی دارن، اما صف و پشته به واسطه سادگی‌شون و قابلیت‌های کم‌ترشون، توی برنامه‌های سطح پایین خیلی بیش‌تر استفاده می‌شن. مثلاً وقتی که داریم اساس یک زبان یا سیستم عامل رو پیاده‌سازی می‌کنیم، ممکنه با مسائلی طرف باشیم که راه حل ساده‌ای می‌خوان، اما قراره به وفور استفاده بشن و برای همین مهمه که بتونیم عملیات‌هاشون رو به سادگی و با کم‌ترین هزینه ممکن انجام بدیم.<sup>۷</sup>

یکی از مشهورترین کاربردهای پشته در ذخیره‌سازی عملیات‌های Undo و Redo در برنامه‌های کامپیوتریه.

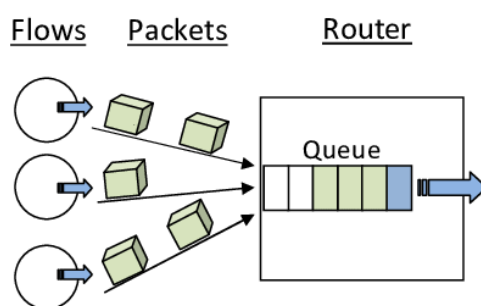


مطابق تصویر بالا، تمام برنامه‌هایی که از Undo و Redo استفاده می‌کنن دو تا پشته دارن که عملیات‌های کاربر رو توشون ذخیره می‌کنن. بعد از اجرای هر عملیات، اون عملیات به پشته اول پوش می‌شه. اگه در بین کارهامون تصمیم بگیریم آخرین عملیاتی که انجام دادیم رو کنسل کنیم، undo رو می‌زنیم و به یک مرحله قبل برمی‌گردیم، و یک عملیات از پشته اول پاپ می‌شه و به پشته دوم پوش می‌شه. حالا اگه بخوایم عملیاتی که کنسل کردیم رو برگردونیم، دکمه redo رو می‌زنیم، و یک عملیات از پشته دوم پاپ می‌شه و به پشته اول پوش می‌شه، و دوباره به حالت اولیه برمی‌گردیم.

<sup>۷</sup> به چنین داده‌ساختارهایی Abstract data type گفته می‌شه. توضیحات بیش‌تر درباره‌شون از این درس خارجه، اما می‌تونین خودتون با جست‌وجو چیزهای جالبی راجع بهشون یاد بگیرین.

از داده ساختار صف توی زمان بندی وظایف کامپیوتر خیلی استفاده می شه. مثل زمان بندی وظایف CPU، پردازش عملیات های چاپ یک پرینتر، و یا مدیریت ترافیک اینترنت.

مثلاً وقتی چند دستگاه به یک مودم اینترنت متصلن و همزمان دارن اطلاعاتی رو آپلود می کنن، با این که سرعت اینترنت بی نهایت نیست تمام این اطلاعات به درستی و کاملی منتقل می شن. توی چنین سناریویی مودم یک پشته داره و بسته های اطلاعاتی که بهش فرستاده می شه توی اون پشته انکیو می شن، و هر وقت سرعت و ترافیک اینترنت اجازه بده، اون بسته ها دیکیو می شن. این طوری هیچ وقت اطلاعاتی گم نمی شه و جا نمی مونه.



در درس های آینده با الگوریتم هایی مواجه می شین که به پشته یا صف در تعریفشون نیازمندن، مثل جست و جوی DFS و BFS.

## اینترفیس Queue

سومین و آخرین فرزند مهم اینترفیس Collection که بهش اشاره کرده بودیم Queue هست. این اینترفیس برای داده ساختارهایی مناسبه که توی اضافه و حذف کردن عناصرشون قوانین خاصی داریم، یعنی دقیقاً پشته و صف. لیست متدهای مهم اینترفیس Queue:

- `boolean offer(E e)`: آبجکت `e` رو به انتهای `queue` اضافه می کنه. در صورتی که `queue` پر باشه مقدار `false` رو برمی گردونه.
- `E poll()`: آبجکت موجود در جلوی `queue` رو حذف می کنه و برمی گردونه. در صورتی که `queue` خالی باشه مقدار `null` رو برمی گردونه.
- `E peek()`: آبجکت موجود در جلوی `queue` رو برمی گردونه اما اون رو حذف نمی کنه. در صورتی که `queue` خالی باشه مقدار `null` رو برمی گردونه.



اینترفیس `Queue` برای داده‌ساختارهایی که سیاست `FIFO` (First-in First-out) دارند استفاده می‌شود. اما با توجه به این‌که متدی برای حذف آبجکت‌های انتهایی ندارد، نمی‌توانیم از آن برای پیاده‌سازی پشته استفاده کنیم.

## اینترفیس `Deque`

اینترفیس `Deque` خودش یک فرزند از اینترفیس `Queue` هستش و اون رو `extend` می‌کنه. کلمه `Deque` به معنای `Double-Ended Queue` و خودش یک داده‌ساختار پیچیده‌تر از صفه. داده‌ساختار `Deque` صفیه که علاوه بر حذف از جلو و اضافه کردن به انتها، حذف از انتها و اضافه کردن به جلو رو هم داره.

بنابراین اینترفیس `Deque` هم این ویژگی رو داره. از اون جایی که اینترفیس `Deque` فرزند اینترفیس `Queue` هست، متدهای اون رو هم به ارث می‌بره.

## لیست متدهای اینترفیس `Deque` (برای مطالعه بیشتر)

- `void addFirst(E e)`: آبجکت `e` رو به جلو اضافه می‌کنه. در صورت پر بودن اکسپشن `ترو` می‌کنه.
- `void addLast(E e)`: آبجکت `e` رو به انتها اضافه می‌کنه. در صورت پر بودن اکسپشن `ترو` می‌کنه.
- `boolean offerFirst(E e)`: آبجکت `e` رو به جلو اضافه می‌کنه. در صورت پر بودن مقدار `false` رو برمی‌گردونه.
- `boolean offerLast(E e)`: آبجکت `e` رو به انتها اضافه می‌کنه. در صورت پر بودن مقدار `false` رو برمی‌گردونه.
- `E removeFirst()`: آبجکت جلویی رو حذف و برمی‌گردونه. در صورت خالی بودن اکسپشن `ترو` می‌کنه.
- `E removeLast()`: آبجکت انتهایی رو حذف و برمی‌گردونه. در صورت خالی بودن اکسپشن `ترو` می‌کنه.

- `E pollFirst()`: آبجکت جلویی رو حذف و برمی‌گردونه. در صورت خالی بودن مقدار `null` رو برمی‌گردونه.
- `E pollLast()`: آبجکت انتهایی رو حذف و برمی‌گردونه. در صورت خالی بودن مقدار `null` رو برمی‌گردونه.
- `E getFirst()`: آبجکت جلویی رو بدون حذف کردن برمی‌گردونه. در صورت خالی بودن اکسپشن ترو می‌کنه.
- `E getLast()`: آبجکت انتهایی رو بدون حذف کردن برمی‌گردونه. در صورت خالی بودن اکسپشن ترو می‌کنه.
- `E peekFirst()`: آبجکت جلویی رو بدون حذف کردن برمی‌گردونه. در صورت خالی بودن مقدار `null` رو برمی‌گردونه.
- `E peekLast()`: آبجکت انتهایی رو بدون حذف کردن برمی‌گردونه. در صورت خالی بودن مقدار `null` رو برمی‌گردونه.
- `void push(E e)`: آبجکت `e` رو به جلو اضافه می‌کنه. (مناسب پشته)
- `E pop()`: آبجکت جلویی رو حذف و برمی‌گردونه. (مناسب پشته)

با توجه به تمام متدهایی که این اینترفیس داره، می‌تونیم حدس بزنیم که می‌تونیم باهاش داده‌ساختارهایی هم با سیاست FIFO و هم داده‌ساختارهایی با سیاست LIFO رو پیاده‌سازی کنیم.

## پیاده‌سازی پشته

در پشته از قاعده خاصی برای نوع ذخیره‌سازی داده‌ها استفاده نمی‌کنیم و خواص این داده‌ساختار فقط مربوط به پوش و پاپ کردن ارزش می‌شه. بنابراین می‌شه در پیاده‌سازی هم با ایندکس و حافظه مشخص (مثل آرایه) و هم بدون ایندکس و به‌صورت بلوک‌های پخش در حافظه (مثل لیست پیوندی) جلو ببریم. توی هر کدوم از این روش‌ها عیب و فایده خودش رو داره، و باید در زمان و جای درست از هر کدوم استفاده کنیم. اما در واقع وقتی می‌خوایم پشته رو پیاده‌سازی کنیم باید از ایده‌های یک داده‌ساختار دیگه مثل لیست پیوندی و یا آرایه استفاده کنیم.

برای پیاده‌سازی پشته در جاوا، می‌تونیم از اینترفیس `Deque` استفاده کنیم.

```

public class Stack <E> implements Deque<E> {
    private Node<E> top;
    private int size;

    private static class Node<E> {
        E data;
        Node<E> next;
        Node(E data) { this.data = data; }
    }

    @Override
    public void push(E e) {
        Node<E> newNode = new Node<>(e);
        newNode.next = top;
        top = newNode;
        size++;
    }

    @Override
    public E pop() {
        if (isEmpty()) throw new NoSuchElementException("Stack is empty!");
        E data = top.data;
        top = top.next;
        size--;
        return data;
    }

    @Override
    public E peek() {
        if (isEmpty()) throw new NoSuchElementException("Stack is empty!");
        return top.data;
    }

    @Override
    public boolean isEmpty() {
        return size == 0;
    }

    @Override
    public int size() {
        return size;
    }
}

```

در پیاده‌سازی نمونه بالا از ساختاری مشابه لیست پیوندی برای پیاده‌سازی پشته استفاده کردیم. کد بالا رو توی اینتلجی کپی کنین و خودتون اجراش کنین.<sup>۸</sup>

می‌تونین از کد زیر برای تست کردن این داده‌ساختار پیاده‌سازی شده استفاده کنین.

---

<sup>۸</sup> دقت کنین که سایر متودهای Deque رو هم باید اورراید کنین. لازم نیست این متودها پیاده‌سازی بشن و می‌تونن فقط یک مقدار null یا false رو برگردونن تا به اروری بر نخورین. اینتلجی خودش این کار رو براتون انجام می‌ده.

```
Stack<String> stack = new Stack<>();
stack.push("Pedram");
stack.push("Hooman");
stack.push("Arman");

System.out.println("Top element: " + stack.peek());

while (!stack.isEmpty()) {
    System.out.println(stack.pop());
}
```

با اجرای این مثال می‌تونین متوجه بشین که پوش و پاپ کردن از پشته چه‌طور کار می‌کنه.

## پیاده‌سازی صف

چیزهایی که درباره پیاده‌سازی پشته گفتیم، درباره صف هم صدق می‌کنه. شما هم می‌تونین از یک آرایه برای پیاده‌سازی صفتون استفاده کنین و فقط برای اضافه کردن به انتها و حذف کردن از ابتدا متد بنویسین. همچنین می‌تونین مثل یک لیست پیوندی از آبجکتهای راس برای ذخیره‌سازی اعضای یک صف استفاده کنین.

```
public class MyQueue<E> implements Queue<E> {
    private Node<E> head;
    private Node<E> tail;
    private int size;

    private static class Node<E> {
        E data;
        Node<E> next;
        Node(E data) { this.data = data; }
    }

    @Override
    public boolean offer(E e) {
        Node<E> newNode = new Node<>(e);
        if (isEmpty()) {
            head = tail = newNode;
        } else {
            tail.next = newNode;
            tail = newNode;
        }
        size++;
        return true;
    }

    @Override
    public E poll() {
        if (isEmpty()) return null;
        E data = head.data;
        head = head.next;
        if (head == null) tail = null;
        size--;
        return data;
    }
}
```

```

    }

    @Override
    public E peek() {
        if (isEmpty()) return null;
        return head.data;
    }

    @Override
    public int size() {
        return size;
    }

    @Override
    public boolean isEmpty() {
        return size == 0;
    }
}

```

در این پیاده‌سازی نیز از ساختاری مشابه لیست پیوندی استفاده کردیم. می‌تونین کد بالا رو توی اینتلیجی کپی کنین و اجراش کنین.<sup>۹</sup>

می‌تونین از کد زیر برای تست کردن این داده‌ساختار پیاده‌سازی‌شده استفاده کنین.

```

MyQueue<String> queue = new MyQueue<>();
queue.offer("Pedram");
queue.offer("Hooman");
queue.offer("Arman");

System.out.println("Front element: " + queue.peek());

while (!queue.isEmpty()) {
    System.out.println(queue.poll());
}

```

با اجرای این مثال می‌تونین متوجه بشین که انکیو (offer()) و دیکیو کردن (poll()) در صف چه طور کار می‌کنه.

## Iterator در پشته و صف

قبلاً هم گفتیم که برای پیمایش امن در اعضای یک داده‌ساختار، باید از Iterator استفاده کنیم. اگه به متدهای اینترفیس Collection مراجعه کنین، می‌بینین که یک متد به اسم iterator() هم داریم که آبجکت‌هایی با جنس Iterator رو برمی‌گردونه. این متد توی تمام کلاس‌هایی که با ایمپلمنت از

<sup>۹</sup> دقت کنین که سایر متودهای Queue رو هم باید اورراید کنین. لازم نیست این متودها پیاده‌سازی بشن و می‌تونن فقط یک مقدار null یا false رو برگردونن تا به اروری برخوردن. اینتلیجی خودش این کار رو براتون انجام می‌ده.

اینترفیس Collection (و فرزندان اون) ساخته می‌شن وجود داره و مشخص می‌کنه که توی یک پیاده سازی خاص از یک اینترفیس، Iterator چه طور باید در اون کلاس پیمایش انجام بده.

پیاده سازی هایی که در بخش های قبل برای پشته و صف فراهم کردیم متد `iterator()` رو کامل نکردن. اول کمی فکر کنین و سعی کنین خودتون این متد رو پیاده سازی کنین. چه ساختاری برای پیاده سازیش به ذهنتون می‌رسه؟

## پیاده سازی `iterator()` در پشته

```
@Override
public Iterator<E> iterator() {
    return new Iterator<>() {
        private Node<E> current = top;

        public boolean hasNext() {
            return current != null;
        }

        public E next() {
            if (!hasNext()) throw new NoSuchElementException();
            E data = current.data;
            current = current.next;
            return data;
        }
    };
}
```

متد بالا رو به کد قبلی پشته اضافه کنین. حالا در `main`، کد زیر رو جایگزین حلقه `while` قبلی کنین.

```
Iterator<String> it = stack.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

حالا دقیقاً کاری که قبلاً داشتین به طور دستی انجام می‌دادین رو با کمک `Iterator` انجام دادین.

## پیاده سازی `iterator()` در صف

```
@Override
public Iterator<E> iterator() {
    return new Iterator<>() {
        private Node<E> current = head;

        public boolean hasNext() {
            return current != null;
        }

        public E next() {
            if (!hasNext()) throw new NoSuchElementException();
            E data = current.data;
            current = current.next;
            return data;
        }
    };
}
```

```
        return data;
    }
};
}
```

متد بالا رو به کد قبلی پشته اضافه کنین. حالا در main، کد زیر رو جایگزین حلقه while قبلی کنین.

```
Iterator<String> it = queue.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

این جا هم مشابه پشته، همون کاری که از قبل انجام می‌دادین رو با کمک Iterator انجام دادین. این توی کدهای پیچیده‌تر و پیاده‌سازی‌هایی از صف و پشته که متدهای بیش‌تری داشته باشن بهتون کمک می‌کنه که پیمایش امن‌تری در اعضا انجام بدین.

## هش‌مپ

یادتونه می‌گفتیم آرایه مثل یه قفسه کتابه و هر جایگاهی با یه عدد مشخص شده؟ حالا فرض کنین یه قفسه کتاب خیلی هوشمند داریم، که به ازای هر کتابی، از روی اسم کتاب میدونه اون کتاب رو دقیقا کجا بذاره، تا دفعه بعدی که خواست پیداش کنه فقط از روی اسمش جاشو پیدا کنه. ((=))

دیدیم که توی آرایه ها، ما میتونستیم به ازای هر خونه‌ای از آرایه که میخواستیم، به مقدار اون خونه دسترسی داشته باشیم و آرایه برامون این کار رو خیلی خیلی سریع انجام میداد. پس اگه قرار باشه به هر عددی یه آبجکت اختصاص بدیم، میتونیم خیلی راحت این کار رو با یه آرایه‌ای از اون آبجکت انجام بدیم. ولی اگه میخواستیم این آبجکت هارو به چیزی غیر از اعداد طبیعی اختصاص بدیم چی؟ مثلا بخوایم آبجکت هامون رو به یه سری رشته اختصاص بدیم به و به ازای هر رشته مثل "ap" بتونیم به آبجکت اختصاص داده شده بهش به سرعت دسترسی پیدا کنیم.

هش‌مپ‌ها این کارو برای ما خیلی آسون میکنن! درواقع به ما اجازه میدن زوج مرتب‌های <key, value> داخل هش‌مپمون ذخیره کنیم، و هر موقع خواستیم بتونیم به مقدار اختصاص داده شده به کلید یا همون key هامون دسترسی داشته باشیم. برای مثال واسه همون کتابخونه، بجای اینکه آرایه‌ای از کتاب‌ها داشته باشیم، میشه هش‌مپی از جنس <String, Book> تعریف کرد، که به ازای هر رشته (اسم کتاب‌ها) به ما آبجکت اون کتاب رو برمیگردونه.

مثلا توی کد زیر یه هش‌مپی که به رشته‌ها عدد اختصاص میده داریم، و اگه کد رو اجرا کنین میبینین که به ازای رشته‌های "Pedram" و "Hooman" مقادیر ۲۵ و ۱۲ که بهشون اختصاص داده شده رو با موفقیت ذخیره و بعد خروجی میدیم، و از اونجایی که چیزی به رشته‌ای مثل "Sania" اختصاص داده نشده بود، خروجی هش‌مپ به ازای اون رشته null خواهد بود.

```
HashMap<String, Integer> map = new HashMap<>();
map.put("Hooman", 12);
map.put("Pedram", 25);

System.out.println("Hooman: " + map.get("Hooman"));
System.out.println("Pedram: " + map.get("Pedram"));
System.out.println(map.get("Sania"));
```

## اینترفیس Map

همون‌طور که در ابتدا گفتیم، فریم‌ورک Collections علاوه بر اینترفیس Collection و فرزندهاش، یک اینترفیس دیگه هم به اسم Map داره.



Map برای نگه‌داری زوج‌های کلید-مقدار<sup>۱۰</sup> استفاده می‌شود. وقتی یک آبجکت رو در یک Map ذخیره می‌کنیم، به همراه اون آبجکت یک کلید (معمولاً یک مقدار عددی) هم ساخته می‌شود و وقتی به دنبال یک آبجکت خاص در Map باشیم، با کمک اون کلید پیدا می‌کنیم. به چنین حافظه‌ای‌هایی «دیکشنری» هم گفته می‌شود. اینترفیس Map هم متدهای خودش رو دارد. در این جا V به تایپ مقدار ما اشاره دارد، یعنی آبجکت‌هایی که در مپ ذخیره می‌شن، و K هم تایپ کلید مقدار ذخیره شده.

- `public V put(K key, V value)`: آبجکت value رو با کلید key به مپ اضافه می‌کند. در صورتی که مپ از قبل دارای آبجکتی با کلید key باشد، آبجکت جدید جایگزین می‌شود و به عنوان خروجی برمی‌گردد.

- `public V get(K key)`: آبجکت مربوط به کلید key را برمی‌گرداند.

- `public V remove(K key)`: آبجکت مربوط به کلید key را از مپ پاک می‌کند و آن را برمی‌گرداند.

- `public int size()`: تعداد زوج‌های مقدار-کلید را برمی‌گرداند.

تمام متدهای اینترفیس Collection برای اینترفیس Map هم معنادار است، به جز متد `iterator()`. یک Map برای هر عضو خود هم مقدار و هم کلید رو نگه‌داری می‌کند و برای هر کدام `iterator()` خودش رو دارد. برای همین اینترفیس Map از Collection ایمپلمنت نمی‌شود، با این حال، این اینترفیس هم یکی از بخش‌های فریم‌ورک Collection جاواست.

## پیاده‌سازی هش‌مپ

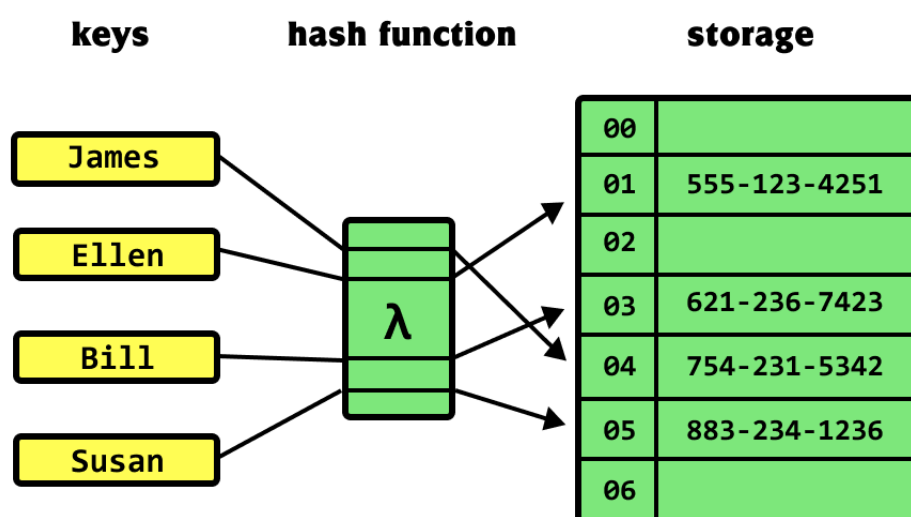
مطالب این بخش خارج از مباحث درس «برنامه‌سازی پیشرفته» است و خواندن این بخش اختیاریه.

از اونجایی که پیاده‌سازی هش‌مپ ها یکم پیچیده تر از بقیه داده‌ساختارهایی که توی این داکيومنت دیدیم، قرار نیست خیلی داخلش عمیق بشیم و کدشو بنویسیم. ولی بیاین باهم ببینیم چطوری هش‌مپ میتونه value ها رو به key ها اختصاص بده.

فرض کنیم یه تابع وجود دارد، که شما بهش یه آبجکت میدین و اون تابع بهتون یه عدد طبیعی برمی‌گردونه. حالا فرض کنیم این تابع یک‌به‌یک باشد، یعنی به ازای دوتا آبجکت مختلف به شما دوتا

<sup>10</sup> Name-Value pairs (or Key-Value pairs)

خروجی مختلف بده. هش‌مپ برای ذخیره کردن value هاش از به آرایه استفاده می‌کند، و با استفاده از یک Hash Function یا همون تابع که وظیفه هاش کردن آبجکت هامون رو داره، به هر key یک عدد اختصاص میده و value مربوط به اون key رو، توی خونه عددی که Hash Function بهمون خروجی داده قرار میده. انگار داریم از به آرایه عادی استفاده می‌کنیم، با این تفاوت که دیگه لازم نیست ایندکس هامون عدد باشه، میتونیم از هر آبجکتی به عنوان key استفاده کنیم، و هاش‌فانکشنمون اون آبجکت رو تبدیل به عدد می‌کند.



حالا چند تا مشکل اصلی داریم. اول اینکه هش‌مپ ما پویاست و لیمیتی برای تعداد اعضا نداره و از اونجایی که اندازه آرایه ها از قبل مشخصه، باید مثل آرایه پویا این موضوع مدیریت بشه.

مورد بعدی اینه که چطوری این تابع‌هرو تعریف کنیم. فرض کنیم کلید یا همون key های ما از جنس رشته‌ن، و ما می‌خوایم به این رشته ها آبجکت های کتاب اختصاص بدیم. اگه ما کلا ۳ تا رشته داشته باشیم و بر فرض تابع هش‌مون به استرینگ "ap" عدد ۱۰۰۰۰۰۰۰ رو اختصاص بده، ما مجبوریم به آرایه خیلی خیلی بزرگ تعریف کنیم، که فقط ۳ تا از خونه‌هاش پر شده. پس نحوه تعریف تابع هش‌مون هم خیلی مهمه.

و در نهایت توابع هاش هیچوقت یک به یک نیستن. ما کل فرض رو بر این چیدیم که این تابع هاش یک به یک، پس هیچوقت این تابع یک عدد رو واسه دوتا آبجکت متفاوت خروجی نمیده، چون اگه این اتفاق بیفته انگار value دوتا key مختلف سعی میکنن که توی یه خونه از آرایه نوشته بشن که به این اتفاق collision میگیم. ولی حالا که توابع هاش مون یک به یک نیستن چی؟ ما سعی میکنیم تابع مون رو جواری تعریف کنیم که تعداد collision ها نسبتا کم باشه، و جاهایی که collision اتفاق میفته رو به کمک روش های مختلفی هندل میکنیم.

همه اینا نکاتی که توی پیاده سازی هاش مپ باید بهشون توجه کنیم، اگه علاقه دارین بیشتر راجع به خود پیاده سازیش بدونین میتونین توی IDE اتون نحوه پیاده سازی هاش مپ توسط خود جاوا رو ببینید.

## Iterator در هاش مپ

همونطور که توی دیتا استراکچر های دیگه که دیدین برای پیمایش کردن روی اعضای کالکشن یک Iterator داشتیم، توی هاش مپ هم برای پیمایش امن روی اعضا Iterator داریم، با این تفاوت که برای پیمایش روی value ها، روی key ها و روی زوج مرتب های <value, key> ۳ تا Iterator جدا داریم و میتونیم با توجه به نیازمون از هرکدوم که خواستیم استفاده کنیم:

```
HashMap<String, Integer> map = new HashMap<>();
map.put("Hooman", 12);
map.put("Pedram", 25);

Iterator<HashMap.Entry<String, Integer>> entryIterator =
map.entrySet().iterator();
while (entryIterator.hasNext()) {
    Map.Entry<String, Integer> entry = entryIterator.next();
    System.out.println("Key: " + entry.getKey() + ", Value: " +
entry.getValue());
}

Iterator<String> keyIterator = map.keySet().iterator();
while (keyIterator.hasNext()) {
    String key = keyIterator.next();
    System.out.println("Key: " + key);
}

Iterator<Integer> valueIterator = map.values().iterator();
while (valueIterator.hasNext()) {
    Integer value = valueIterator.next();
    System.out.println("Value: " + value);
}
```

همونطور که میبینیم، یکبار به کمک آبجکت `entryIterator` روی زوج مرتب هامون پیمایش انجام میدیم و آبجکت `Iterator` هر بار `Entry` بعدی داخل هاش مپ رو بهمون میده، یکبار با `keyIterator` روی کلید ها، یا همون آبجکت هایی که مقادیر بهشون اختصاص داده میشه پیمایش انجام میدیم، و یکبار با `valueIterator` روی خود مقادیر اختصاص داده شده.

## چه چیزی یاد گرفتیم؟

- داده‌ساختار چیه و چرا بهش نیاز داریم.
- داده‌ساختار معروف زیر را بشناسین، بدونین هر کدوم چه طور کار می‌کنه و چه طور می‌تونین با استفاده از جاوا پیاده‌سازیشون کنین.
  ۱. آرایه و آرایه پویا
  ۲. لیست پیوندی
  ۳. پشته
  ۴. صف
  ۵. هاشمپ
- با فریم‌ورک کالکشن‌های جاوا آشنا باشین، اینترفیس‌های اصلی و پیاده‌سازی‌های موجود در این فریم‌ورک رو بشناسین و بدونین چه طور ازشون استفاده کنین.
- با Iterator در جاوا آشنا باشین و یاد گرفته باشین چه طور باید ازشون استفاده کنیم. همچنین اهمیت و علت استفاده ازش رو بدونین.