

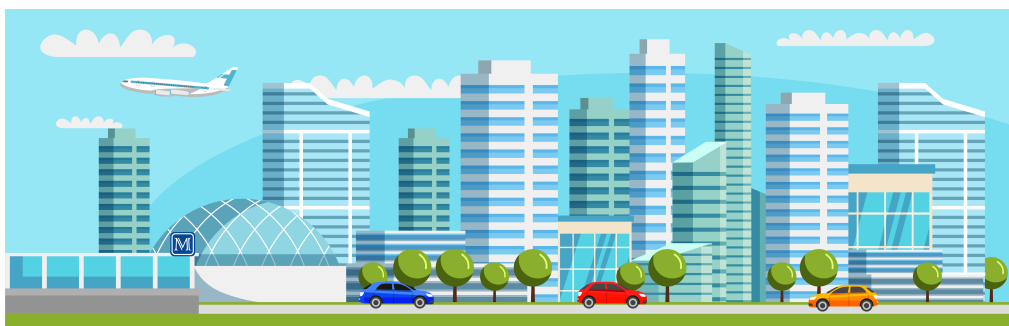
# Linear List

<b>1</b>	<b>环境作用域和闭包</b>	<b>3</b>
1.1	环境作用域和通俗理解	3
1.2	函数的环境和作用域原理	3
1.3	延伸函数环境的生命周期	4
1.4	构造函数中作用域的使用形态	5
1.5	let 块作用域	6
1.6	let const var 在 for 循环中的使用差异	6
1.7	var 的伪造块作用域	7
1.8	什么是闭包?	7
1.9	闭包的内存泄漏问题	10
<b>2</b>	<b>模块化编程</b>	<b>13</b>
2.1	为什么使用模块化编程	13
2.2	开发一个模块化管理工具	13
2.3	模块的基本使用	15
2.4	模块的导入导出	16
2.5	按需动态加载模块	18
<b>3</b>	<b>Array</b>	<b>19</b>
3.1	Overview of Array	19
3.2	Practice: To create an Array Class with Java	19
<b>4</b>	<b>LinkedList</b>	<b>23</b>
4.1	Overview of LinkedList	23
4.2	LinkedList In Java	23
4.3	MikeLinkedList 的搭建 1: 完成 Node Class 的实现与 MikeLinkedList 的基础	24
4.4	MikeLinkedList 的搭建 2: 实现具体的方法	24
4.5	练习题 1: 反转链表	28
4.6	练习题 2: 寻找倒数第 K 个链表节点	29
<b>5</b>	<b>Stack</b>	<b>31</b>
5.1	Overview of Stack	31
5.2	Stack in Java	31
5.3	MikeStack 的实现	32
5.4	练习题 1: 反转字符串	33
5.5	练习题 2: 判断表达式的平衡	34



# 1. 环境作用域和闭包

## 1.1 环境作用域和通俗理解



可以把环境作用域理解成一个城市，城市里有公园，也有居民楼，这些也构成了更小的环境作用域。对于函数声明来说，我们可以理解成一个建筑的蓝图。比如市政府打算明年建造一个超级商场，但是并没有建造，所以这片环境作用域是不存在的。而当函数执行调用之后，就相当于这个商场已经建造完毕，并且投入使用了，自然这个环境作用域就存在。这个例子非常生动形象，在作用域链中不懂的概念可以联系这个例子进行理解。

## 1.2 函数的环境和作用域原理

在 JS 语言中，父集作用域的变量子集作用域中是可以进行访问的。(PHP 中不可以)。

```
let title="houdunren"
function test(){
    //可以访问到 title 变量
    console.log(title)
}
test()//houdunren
```

**R** 在这个例子中，一共有两个环境，一个是全局环境，另外一个是由函数生成的环境作用域。可以看出子集的环境作用域可以渗透访问得到父集作用域中的变量。

```
function show(){
  let url="https://qiaoyangedu.top"
  console.log(url)
}
show()
show()
show()
```

函数每调用一次，都会生成一块作用域，里面存放了 url 变量，虽然看起来数据都一样，但是访问的内容是不同的。可以将 url 类比为公园里的亭子，而 show 函数则是创建公园的蓝图。每次调用一次函数都会创建一座公园，公园里有一座凉亭。虽然凉亭长的都一样，但是并不是同一个亭子。

### 1.3 延伸函数环境的生命周期

每次函数执行完毕后，这块环境作用域就会销毁。对于 JS 来说，如果一个变量被引用，那么它所在的这块作用域就不会被销毁，反之如果没有被引用，则使用完之后就会被销毁。所以延长函数环境的生命周期方法自然是一直引这个环境作用域中的内容。

```
function hd() {
  let n = 1
  return function sum() {
    console.log(++n)
  }
}
let a=hd()//使用 a 引用了环境作用域中的内容
a()
a()
a()
let b=hd() //输出的 n=1
```

由于我们引用了环境作用域中的内容，存在于这个环境作用域中的 n 自然被保存了下来，所以 a 打印的 n 值是在持续进行叠加的。但是当第二次我们使用 let b=hd() 的时候，相当于重新开辟了一块环境，所以又会重新开始。

```
function hd2() {  
    return function sum() {  
        let m = 1  
        return function show() {  
            console.log(++m)  
        }  
        show()  
    }  
}  
let c = hd2()()  
c()  
c()
```

```
function hd3() {  
    let n = 1  
    return function sum() {  
        let m = 1  
        function show() {  
            console.log(++n)  
        }  
        show()  
    }  
}  
let p = hd3()  
p()  
p()  
p()
```

## 1.4 构造函数中作用域的使用形态

构造函数的原理主要就是新建了一个对象，将构造函数中的 `this` 指向了这个对象，这种指向也可以理解成是一种引用，和 `return` 效果是一样的，所以也可以起到延伸函数环境生命周期的作用。

```
function Hd() {  
    let n = 1  
    this.sum = function () {  
        console.log(++n)  
    }  
}  
  
let a = new Hd()  
a.sum()//2  
a.sum()//3
```

## 1.5 let 块作用域

let 是 ES6 引入的新的概念，它是支持块作用域的，而 var 是不支持的

```
{  
    let a=1;  
    console.log(a)//1  
}  
console.log(a)//undefined  
  
{  
    var m=1;  
}  
console.log(m);
```

## 1.6 let const var 在 for 循环中的使用差异

虽然形式上看起来不像，但是for 循环的每一次执行，都会形成一个块级作用域。

```
for (var i = 1; i <= 3; i++) {  
    console.log(i) //1,2,3  
}  
console.log(i)//4  
  
for (let m = 1; m <= 3; m++) {  
    console.log(m)  
}  
console.log(m)//undefined
```

这个例子实际上和上一节例子中是一个道理，只是块级作用域换成了 for 循环而已，

而当 setTimeout 引入的时候，情况又发生了变化。由于 setTimeout 延迟了一秒执行，var 没有块级作用域，变量 q 已经受到了污染改为了 4，所以输出结果 3 次均为 4。而 let 的 for 循环每循环一次都产生了一块新的环境作用域，所以输出内容正常，分别为 1，2，3。

```
for (let p = 1; p <= 3; p++) {
  setTimeout(() => {
    console.log(i)
  }, 1000)
}

for (let q = 1; q <= 3; q++) {
  setTimeout(() => {
    console.log(q) // 4, 4, 4, 4,
  }, 1000)
}
```

## 1.7 var 的伪块级作用域

虽然不建议使用 var 变量进行声明，但是在维护老项目的时候仍然会遇到，我们应该如何让 var 具有块级作用域？最好的方式是构建一个立即执行函数，利用函数作用域来构造一个伪装的块级作用域。

```
//var 虽然没有块级作用域，但是有函数作用域
for (var i = 1; i <= 3; i++) {
  (function (a) {
    setTimeout(() => {
      console.log(a)
    }, 1000)
  })(i)
}
```

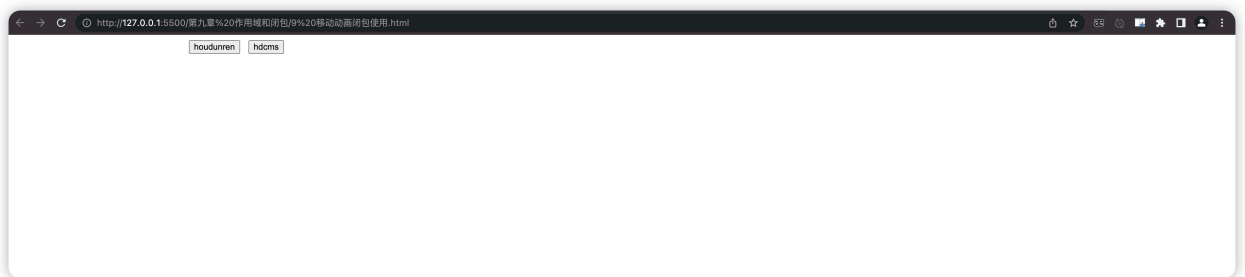
## 1.8 什么是闭包？

闭包指子函数可以访问外部作用域变量的函数特性，即使在子函数作用域外也可以访问。如果没有闭包那么在处理事件绑定，异步请求时都会变得困难。接下来让我们来看三个闭包的例子。

### 1.8.1 使用闭包获取区间商品

```
let arr = [1, 23, 4, 5, 6, 8, 21, 10]
let hd = arr.filter(function (v) {
    return v >= 2 && v <= 9
})
function between(a, b) {
    return function (v) {
        //闭包访问父级函数的变量 a,b
        return v >= a && v <= b
    }
}
console.log(arr.filter(between(3, 9)))
```

### 1.8.2 Button 移动动画



```
let btns = document.querySelectorAll("button")
btns.forEach(function (item) {
    let bind = false
    item.addEventListener("click", function () {
        if (!bind) {
            let left = 1
            bind = setInterval(function () {
                //闭包访问 item
                item.style.left = left++ + "px"
            }, 100)
        }
    })
});
```



```
<body>
<style>
button {
    position: absolute;
}
</style>
<button message=" 后盾人">houdunren</button>
<button message="hdcms">hdcms</button>

</body>
```

### 1.8.3 利用闭包根据字段排序商品

```
let lessons = [
  {
    title: "vue.js 学习手册",
    click: 29,
    price: 300
  },
  {
    title: "css 学习",
    click: 21,
    price: 200
  },
  {
    title: "d3.js",
    click: 211,
    price: 15
  }
]

let hd = lessons.sort((function (a, b) {
  return a.price > b.price ? 1 : -1
})))

function order(field) {
  return function (a, b) {
    return a[field] > b[field] ? 1 : -1
  }
}

let click = lessons.sort(order("click"))
console.table(click)
```


## 1.9 闭包的内存泄漏问题

闭包固然有很多好处，但是使用不当也有危险，因为闭包将某个环境作用域长久保存下来，如果不使用又不进行清理会导致内存泄漏。我们可以来看一个例子，在这个例子中，我们合理解决了闭包泄漏的问题，在这个引用对象不使用的時候，将其设置为 null 清除。

```
<body>
  <div desc="houdunren"> 在线学习 </div>
  <div desc="hdcms"> 开源产品 </div>
</body>
<script>
  let divs = document.querySelectorAll("div")
  divs.forEach(function (item) {
    let desc = item.getAttribute("desc")
    item.addEventListener("click", function () {
      //item 一直存储在内存中，我们如果只需要 desc，这是一种浪费
      console.log(desc)
      item = null
    })

  })
</script>
```





## 2. 模块化编程

### 2.1 为什么使用模块化编程

在项目体积变大的过程中，需要把不同的业务分割成多个文件，这就是模块的思想，模块是比函数和对象更大的单元，使用模块组织程序能够更好的维护程序和进行扩展。

在生产环境中，一般使用打包工具如webpack,vite 等进行构建，这些工具提供了更多的功能。

- 模块就是一个独立的文件，里面是函数或者类库
- 虽然 JS 没有命名空间的概念，使用模块可以解决全局变量冲突
- 模块需要隐藏内部实现，只对外开放需要使用的接口
- 模块可以避免全局变量的滥用，造成代码不可控
- 模块可以被不同的应用使用，从而提高编码效率

### 2.2 开发一个模块化管理工具

在这里使用了立即执行函数来模拟模块 (type="module"), 因为一个函数都是一个独立的作用域，我们封装了模块内部的具体实现，只开放了模块的定义方法 `define`，用于定义模块。只需要提供模块的名称，依赖的模块单元，以及对应的功能函数即可。

```
let module = (function () {
  const moduleList = {}
  function define(name, modules, action) {
    modules.map((m, i) => {
      //将 moduleList 的真正模块传入 modules
      modules[i] = moduleList[m]
    })
    //将 modules 传入作为参数传入 action, 并且放置到 moduleList 中去
    moduleList[name] = action.apply(null, modules)
  }
  return { define }
})()
```

只需要在 Action 中按顺序使用参数即可获取对应的依赖模块, 从而可以使用对应的模块依赖。

```
//定义第一个模块
module.define("hd", [],
function() {
  return {
    first(arr) {
      return arr[0]
    },
    max(arr, key) {
      return arr.sort((a, b) => b[key] - a[key])[0]
    }
  }
})
//定义第二个模块

module.define("lesson", ["hd"], function (hd) {
  let data = [
    { name: "js", price: 199 },
    { name: "mysql", price: 78 },
  ]
  console.log(hd.max(data, "price"))
})
```

## 2.3 模块的基本使用

在 html 中只需要在 script 标签中声明 `type='module'` 即可使用模块, 使用 `export` 可以开放接口, 使用 `import` 引入需要使用的内容。

```
console.log("init...")
let title = " 后盾人"
let url = "houdunren.com"
function show() {
    console.log(title + url)
}
export { title, show }
```

```
<html>
<head>
  <script type="module">
    import { title, show } from "./hd.js"
    console.log(title)
    show()
  </script>
</head>
<body></body>
</html>
```

模块最主要有四个特性:

- 延迟解析
- 严格模式
- 块作用域
- 预解析

严格模式非常好理解, 在声明 script 为模块之后, 整个环境都是严格模式, 比如直接输出 this, 显示的结果为 undefined。而延迟解析的含义是, 无论模块声明的位置在哪儿, 都会在 html 标签加载完成之后才进行加载。而块作用域特性的含义是每一个模块都是一个顶级作用域, 不同模块之间不能进行互相访问。预解析的特性是指模块无论导入多少次, 之后的导入都不会再执行模块中的代码, 而是共享第一次执行的结果。

```

<!--每一个模块都是一个顶级作用域，模块之间不能访问-->
<script type="module">
  let hd = "houdunren.com";
</script>

<script type="module">
  alert(hd); // Error
</script>

```

```

//这是一个模块
console.log("init.....")
let show={()=>{console.log(1)}}

```

```

<script type="module" src="module.js"></script>
<script type="module" src="module.js"></script>
<script type="module" src="module.js"></script>
<script>
  import "./module.js"
  import "./module.js"
  import "./module.js"
  //无论引入多少次，都只会输出一一次 init.....
</script>

```

## 2.4 模块的导入导出

模块导入导出的方法多种多样，如具名导入导出，批量导入，使用别名进行导入导出，默认导入和导出以及混合导入和导出，在本节中将一一进行讲解。

### 2.4.1 具名导入和导出

具名导入和导出指的是精准开放需要使用的变量或者函数。

```

let a=1;
let b=2;
let c={()=>{console.log(1)}}
export {a,b} //精准导出，导入情况和导出一样

```



### 2.4.2 批量导入和使用别名

批量导入使用 `*` 进行导入, 比如 `import * from './db.js'`, 为了使用方便, 我们可以使用 `import * as api from './db.js'`

```
let a=1;
let b=2;
let c=3;
export {a,b,c}
//当然在 export 中也可以使用别名 export {a as balabala}
```

```
<script type="module">
import * as api from './db.js'
console.log(api.a)//1
console.log(api.b)//2
console.log(api.c)//3
</script>
```

**R** 特别注意, 一般情况下不需要使用批量导入, 因为使用 webpack 的时候, 使用批量导入, 打包工具会默认会使用所有的模块内的函数, 每一个都会被打包进来, 从而会增大项目的体积。

### 2.4.3 默认导入和导出

默认导入和导出有些方面是不同的, 需要分别进行讲解。而且默认导出和导入只有一个

```
//方式 1 直接使用 export default 关键字
export default class Test{
  static const version=1.0
}
//方式 2 as default
class Test{
  static const version=1.0
}
export {Test as default}
```

进行导入的时候可以使用任意名字进行接收, 因为默认导出的内容只有一个, 但是建议使用更加合理相关的名称进行接收, 增加可读性。

```
import md from "./db.js"//默认导出
//也可以结合别名进行使用
import md as markdown from "./db.js"
```

#### 2.4.4 混合导入和导出

由于默认导出和具名导出是互不影响的，所以可以一起使用，使用任意一个名称来接受默认导出的内容，使用 {} 来接收具名导出的内容。

```
//导出
export default class Test{
    static const version=1.0
}
let a=1;
let b=2;
export{a,b}
//导入 test 接收默认导出的内容
import test,{a,b} from "./db.js"
```

### 2.5 按需动态加载模块

使用 import 函数可以不在 script 的头部进行模块加载，而是进行动态加载，按需加载。这个函数是一个 Promise，使用 import 传入路径，使用 then 可以接收到加载得到的模块。

```
function importSth(path) {
    import(path).then((module) =>{
        console.log(module)
    })
}
importSth("./m13.js")
```



## 3. Array

### 3.1 Overview of Array

在本节中，我们首先来思考一下数组是什么？

3	1	4	5	6	7
---	---	---	---	---	---

这是一个数组:[3,1,4,5,6,7], 它本质上就是在一段连续的内存片段上存储的一串数据, 由于 int 占了四个字节, 所以每一个数组元素的地址相差四位。

接下来分析数组中常见的操作的算法复杂度:

- Lookup  $O(1)$  根据索引去查找元素, 由于数组中可以通过中括号访问符进行访问, 所以不论访问哪一个元素, 算法复杂度均为  $O(1)$
- Insert  $O(n)$  进行元素插入, 如果在末尾进行元素插入, 算法复杂度为  $O(1)$ , 但是在第一位插入, 我们就需要将每一位都往后挪动一位, 所以复杂度为  $O(n)$ , 通常我们都考虑最坏的情况, 所以算法复杂度为  $O(n)$
- Delete  $O(n)$  无论删除哪一个元素, 其他元素都需要往前移动一位, 所以最终算法复杂度为  $O(n)$

### 3.2 Practice: To create an Array Class with Java

■ **Example 3.1** 在 Java 中, 创建一个 MikeArray 类型, 要求要满足以下几点, 1. 具有 insert 方法, 可以在数组末尾插入元素, 2. 具有 removeAt(int index) 方法, 在 index 处删除对应的元素, 3. 实现 print 方法, 打印出数组中所有的元素 4. indexOf 方法, 返回目标元素第一次出现在数组中的索引, 如果没有出现元素, 则返回索引-1, 4. 动态扩充, 在使用 insert 方法的时候如果超出了数组本身的长度, 扩容到原数组容量的 2 倍, 5. 构造方法是给出数组的长度 length 进行数组构造的。 ■

```
public class MArray {  
    private int[] items;  
    private int count=0;  
    public MArray(int length){  
        items=new int[length];  
    }  
  
    public void print(){  
        for(int i=0;i<count;i++){  
            System.out.println(items[i]);  
        }  
    }  
  
    public void insert(int element){  
        //if the array is full  
        if(items.length==count){  
            //Create a new array (twice the size)  
            int array[]=new int[count*2];  
            //Copy all the items into new Array  
            for(int i=0;i<count;i++){  
                array[i]=items[i];  
            }  
            //set the items to the new array  
            items=array;  
        }  
        items[count++]=element;  
    }  
}
```

```
        public void removeAt(int index){
            if(index<0||index>=count){
                throw new IllegalArgumentException();
            }
            else{
                //[10,20,30,40]
                // index:1
                for(int i=index;i<count;i++){
                    items[i]=items[i+1];
                }
                count--;
            }
        }

        public int indexOf(int item){
            //获取元素的位置
            //肯定是有可能遍历完整个都没有存在的
            for(int i=0;i<count;i++){
                if(items[i]==item){
                    return i;
                }
            }
            return -1;
        }
    }
```

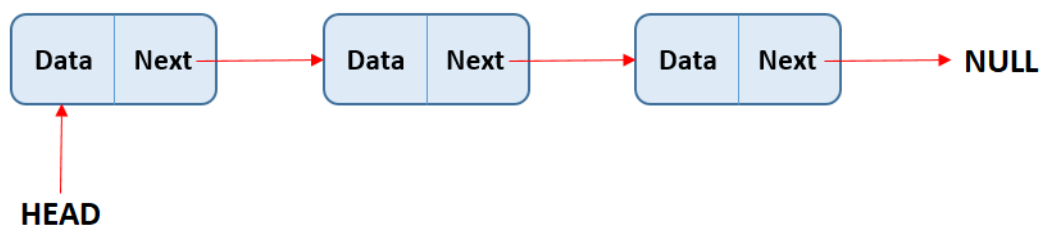




## 4. LinkedList

### 4.1 Overview of LinkedList

链表是由一串节点 (Node) 连接而成, 这个节点由两部分组成, 一部分是数据 Data, 另外一部分是指针 Pointer, 每一个 Node 的指针都指向了下一个节点。一般的链表具有头节点, 当然也有的具有尾节点, 除了一般的单向链表, 还有一种双向链表, 在双向链表中每一个节点都具有两个指针, 一个指向之前的节点, 一个指向了之后的节点。但是总体上来说, 链表的结构都大同小异。



### 4.2 LinkedList In Java

在 Java 中就有 LinkedList, 常用的方法有:

- addFirst 在链表首添加元素
- addLast 在链表尾部添加元素
- contains 是否包含元素
- isEmpty 判断链表是否为空
- indexOf 返回链表中某一元素的位置

这些方法都是由 Java 内部进行实现的, 我们直接引入 LinkedList 这个类即可直接使用, 当然在下一节中, 我们将开始写属于我们自己的链表 (MikeLinkedList)。

```
LinkedList<Integer>linkedList=new LinkedList<>();
linkedList.addFirst(10);
linkedList.addFirst(10);
linkedList.addFirst(10);
System.out.println(linkedList.indexOf(10));
System.out.println(linkedList.contains(10));
System.out.println(linkedList.isEmpty());
linkedList.addLast(29);
System.out.println(linkedList);
```

### 4.3 MikeLinkedList 的搭建 1: 完成 Node Class 的实现与 MikeLinkedList 的基础

在第一节中我们说明了链表是由一个一个节点组成的，所以首先实现一个 Node Class(Integer 整型)，使用整数进行示范，不设置为泛型。

```
public class Node {
    public Node(int value){
        this.value=value;
    }

    public int value;//当前节点的值
    public Node next;//指向下一个节点的指针
}
```

然后完成 MikeLinkedList 的搭建，这个链表首先要两个节点，一个是首节点，一个是尾节点，还需要设置一个 size 属性，记录链表的长度。

```
public class MikeLinkedList{
    //头节点 p
    private Node First;
    private Node Last;
    private int Size;
}
```

### 4.4 MikeLinkedList 的搭建 2: 实现具体的方法

在实现过程中，要考虑极限情况，没有元素的时候，只有一个元素的时候。



#### 4.4.1 addFirst

实现方法很简单，新建一个节点，将它标记为头节点，然后把它的 next 指针指向原本的头节点即可，算法复杂度  $O_n = 1$

```
public void addFirst(int value) {  
    var node = new Node(value);  
    //当前链表是否为空  
    if (isEmpty()) {  
        First = Last = node;  
    } else {  
        var temp = First;  
        First = node;  
        First.next = temp;  
        size++;  
    }  
}
```

#### 4.4.2 addLast

addLast 方法也非常简单，只需要设置将最后一个节点的 next 指向新节点，然后将 Last 标记为最后一个节点即可，算法复杂度为  $O_n = 1$

```
public void addLast(int value) {  
    var node = new Node(value);  
    if (isEmpty()) {  
        First = Last = node;  
    } else {  
        Last.next = node;  
        Last = node;  
    }  
    size++;  
}
```

#### 4.4.3 isEmpty

isEmpty 只需要判断头节点是否为空，如果为空的话就说明这个链表为空，非常简单

```
private boolean isEmpty() {  
    return First == null;  
}
```

#### 4.4.4 indexOf

实现 indexOf 方法只需要从链表头开始遍历，直到发现某个节点的 value 和目标节点的 value 是一样的，如果一样就返回对应的 index，如果遍历完整个链表都没有发现目标，则返回-1。由于在最坏的情况需要完整遍历一遍链表，所以算法复杂度为  $O_n = n$ 。

```
public int indexOf(int value) {  
    if (isEmpty()) {  
        return -1;  
    } else {  
        var cur = First;  
        var index = 0;  
        while (cur != null) {  
            if (cur.value == value) {  
                return index;  
            }  
            index++;  
            cur = cur.next;  
        }  
    }  
    return -1;  
}
```

#### 4.4.5 contains

contains 方法的实现在 indexOf 写过之后就变得非常简单，只需要执行一次 indexOf 方法，如果返回的是-1，则表示不包含，返回 false，否则返回 true

```
public boolean contains(int value) {  
    return indexOf(value) != -1;  
}
```

#### 4.4.6 removeFirst

removeFirst 方法的核心是将第二个节点设置为 First，然后将第一个节点解除关联。

```
public void removeFirst() {  
    if (isEmpty()) {  
        throw new NoSuchElementException();  
    }  
    if (First.next == null) {  
        First = Last = null;  
        return;  
    } else {  
        var second = First.next;  
        First.next = null;  
        First = second;  
    }  
    size--;  
}
```

#### 4.4.7 removeLast

要实现 removeLast 稍微比 removeFirst 难一些，因为我们的思路应该还是找到 Last 之前的那个节点，然后把最后一个节点引用解除，设置倒数第二个节点为 Last 即可。但是如何获取倒数第二个节点？我们可以先写一个 getPrevious 的函数，然后再写这个函数

```
public Node getPrevious(Node node) {  
    var current = First;  
    while (current != null) {  
        if (current.next == node) {  
            return current;  
        }  
        current = current.next;  
    }  
    return null;  
}
```

```
public void removeLast() {
    if (isEmpty()) {
        throw new NoSuchElementException();
    } else if (First == Last) {
        First = Last = null;
    } else {
        var previous = getPrevious(Last);
        Last = previous;
        Last.next = null;
    }
    size--;
}
```

#### 4.4.8 toArray

将链表转为数组，方法很简单，遍历一遍，把链表的元素装数组里即可

```
public int[] toArray() {
    int[] array = new int[size];
    var cur = First;
    var index = 0;
    while (cur != null) {
        array[index] = cur.value;
        cur = cur.next;
        index++;
    }
    return array;
}
```

### 4.5 练习题 1: 反转链表

■ **Example 4.1** 将链表反转，比如 10→20→30→40→50 反转为 50→40→30→20→10。仔细思考，该题并不难，我们发现其实就是将链表的箭头顺序反转了，本质上就是 10←20←30←40←50，每一个指针都指向了之前的指针，所以我们只需要构造一个 previous 指针指向前一个节点即可

```
        public void reverse() {  
            //first second  
            //previous current  
            var previous = First;  
            var current = First.next;  
            Last = First;  
            Last.next = null;  
            while (current != null) {  
                var next = current.next;  
                current.next = previous;  
                previous = current;  
                current = next;  
            }  
            First = previous;  
        }  
    }
```

■

## 4.6 练习题 2: 寻找倒数第 K 个链表节点

■ **Example 4.2** 题目要求不允许直接遍历一整遍链表，然后获取对应的节点，我们使用快慢双指针来解决问题：快指针在一开始的时候就进行向前推进，当推进了 K 步之后慢指针开始运行，当快指针遍历完链表的时候，慢指针指向的节点就是对应的倒数第 K 个节点。

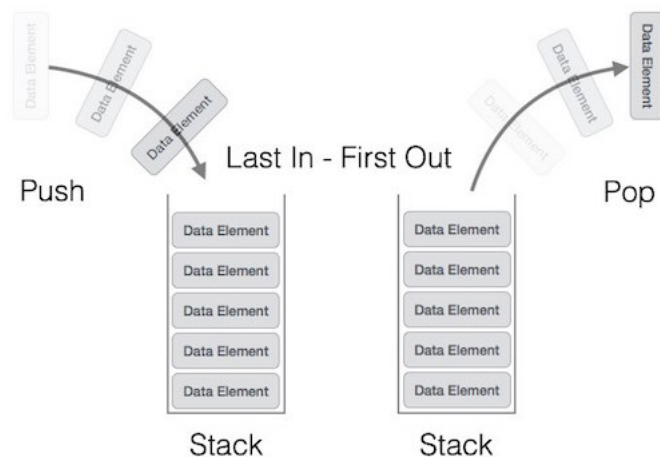
■

```
public int getKthFromTheEnd(int k) {  
    if (isEmpty()) {  
        throw new IllegalStateException();  
    }  
    if (k > size || k <= 0) {  
        throw new IllegalArgumentException();  
    }  
    //声明快指针  
    var a = First;  
    //声明慢指针  
    var b = First;  
    for (int i = 0; i < k - 1; i++) {  
        //先让 a 走 k 步  
        a = a.next;  
    }  
    while (a != Last) {  
        a = a.next;  
        b = b.next;  
    }  
    return b.value;  
}
```

## 5. Stack

### 5.1 Overview of Stack

栈是一种很常见的数据结构, 它遵循先入后出 (FILO) 的原则, 先入栈的元素反而最后出栈, 所以对于一些需要进行反转数据顺序的操作使用栈是非常适合的。



举一个非常通俗易懂的例子: 吃薯片。我们按照顺序向薯片口袋装入薯片, 先吃掉的反而是最后装入的薯片。还有 web 开发中的路由跳转, 每次跳转路由, 我们都将对应的路由压入栈中, 当我们要进行路由回退操作的时候, 回退的都是最后压入的路由

### 5.2 Stack in Java

在 Java 中有对应的 Stack 类型, 它主要的方法是 push 入栈, pop 出栈, peek 方法用于查看位于栈顶的元素, 但是不会弹出, empty 方法用于判断这个 stack 是否为空

```
Stack<Integer> stack = new Stack<>();
stack.push(1);
stack.push(2);
stack.push(3);
System.out.println(stack); //[1,2,3]
System.out.println(stack.pop()); //3
System.out.println(stack.peek()); //2
System.out.println(stack.empty()); //false
```

### 5.3 MikeStack 的实现

MikeStack 的内部，我们使用数组来实现，但是使用数组来实现难免会出现一些问题。问题一：Java 实现的 Stack 不会出现栈满的问题，但是我们使用数组来实现，必然会会遇到数组元素占满，为了简便不对数组进行复制然后扩充，而是抛出一个异常状态。问题二：使用数组实现，在数组没有填满的时候，比如容量为 5，填充三个元素的时候，输出必然为 [1,2,3,0,0]，这样是不像 java 中栈的实现的，所以我们使用一个 counter 计数器，只输出真正有值的部分。gg 对于 push,pop,peek 方法都需要进行临界情况的判定，如果 count==0 的时候，我们需要抛出一个 `IllegalStateException` 异常。

```
public class MikeStack {
    private final int[] array=new int[5];
    private int pointer=0;
}
```



```
public void push(int item){
    if(pointer>=array.length){
        throw new IllegalStateException();
    }
    array[pointer++]=item;
}
public int pop(){
    if(pointer==0){
        throw new IllegalStateException();
    }
    return array[--pointer];
}
public int peek(){
    if(pointer==0){
        throw new IllegalStateException();
    }
    return array[pointer-1];
}
public String toArray(){
    return Arrays.toString(Arrays.copyOfRange(array,0,pointer));
}
```

## 5.4 练习题 1: 反转字符串

■ **Example 5.1** 现在有一个字符串 Hello,World, 我们需要写一个方法来反转这个字符串。由于在前文中提过, 由于 Stack 的特性, 很容易进行一些反转操作, 所以我们使用 Stack 来完成这一题

```
public static String reverseString(String str){
    if(str==null){
        throw new IllegalArgumentException();
    }
    var stack=new Stack<Character>();
    for(char c :str.toCharArray()){
        stack.push(c);
    }
    StringBuilder stringBuilder=new StringBuilder("");
    while(!stack.empty()){
        stringBuilder.append(stack.pop());
    }
    return stringBuilder.toString();
}
```

从入栈到出栈就会导致数据以相反的顺序输出出来，所以我们只需要将数据压入栈中，最后再使用 pop 依次出栈即可。 ■

## 5.5 练习题 2: 判断表达式的平衡

■ **Example 5.2** 该题相较于上题来说，比较复杂。表达式中的符号有 {,(,<,[ 一共四种符号。

- {1+2+3} 平衡
- >1231< 不平衡
- <{1+2+3}> 平衡
- <{1+2}} 不平衡

一次性考虑所有的情况，比较复杂，所以我们首先考虑只有 () 的情况，遇到开始符号入栈，遇到结尾符号出栈，最后栈里是空的，那么就平衡。在考虑清楚单一情况之后，我们考虑所有的情况，可以将这些符号分别录入两个数组，但是一定要注意顺序对应，两个数组 AB,A[1] 和 B[1] 才是完全对应的。

```
public class Expression {
    private List<Character> leftBracket =
        Arrays.asList('(', '<', '[', '{');
    private List<Character> rightBracket =
        Arrays.asList(')', '>', ']', '}');
    String str;

    public Expression(String str) {
        this.str = str;
    }
    private boolean isLeftBracket(char c){
        return leftBracket.contains(c);
    }
    private boolean matchBracket(char left, char top){
        return leftBracket.indexOf(left)==rightBracket.indexOf(top);
    }
    private boolean isRightBracket(char c){
        return rightBracket.contains(c);
    }
    public boolean bBalancedEquation() {
        var stack = new Stack<Character>();
        for (char c : str.toCharArray()) {
            if (isLeftBracket(c)) {
                stack.push(c);
            }
            if (isRightBracket(c)) {
                if (stack.empty()) return false;
                char top = stack.pop();
                if (matchBracket(c,top)) {
                    return false;
                }
            }
        }
        return true;
    }
}
```





# Part One Title

<b>6</b>	<b>Sectioning Examples</b>	<b>39</b>
6.1	Section Title	39
<b>7</b>	<b>In-text Element Examples</b>	<b>43</b>
7.1	Referencing Publications	43
7.2	Link Examples	43
7.3	Lists	43
7.4	International Support	44
7.5	Ligatures	44





## 6. Sectioning Examples

### 6.1 Section Title

Lorem ipsum dolor sit amet, consectetur adipiscing elit<sup>1</sup>. Praesent porttitor arcu luctus, imperdiet urna iaculis, mattis eros. Pellentesque iaculis odio vel nisl ullamcorper, nec faucibus ipsum molestie. Sed dictum nisl non aliquet porttitor. Etiam vulputate arcu dignissim, finibus sem et, viverra nisl. Aenean luctus congue massa, ut laoreet metus ornare in. Nunc fermentum nisi imperdiet lectus tincidunt vestibulum at ac elit. Nulla mattis nisl eu malesuada suscipit.

Aliquam arcu turpis, ultrices sed luctus ac, vehicula id metus. Morbi eu feugiat velit, et tempus augue. Proin ac mattis tortor. Donec tincidunt, ante rhoncus luctus semper, arcu lorem lobortis justo, nec convallis ante quam quis lectus. Aenean tincidunt sodales massa, et hendrerit tellus mattis ac. Sed non pretium nibh. Donec cursus maximus luctus. Vivamus lobortis eros et massa porta porttitor.

#### 6.1.1 Subsection Title

Fusce varius orci ac magna dapibus porttitor. In tempor leo a neque bibendum sollicitudin. Nulla pretium fermentum nisi, eget sodales magna facilisis eu. Praesent aliquet nulla ut bibendum lacinia. Donec vel mauris vulputate, commodo ligula ut, egestas orci. Suspendisse commodo odio sed hendrerit lobortis. Donec finibus eros erat, vel ornare enim mattis et. Donec finibus dolor quis dolor tempus consequat. Mauris fringilla dui id libero egestas, ut mattis neque ornare. Ut condimentum urna pharetra ipsum consequat, eu interdum elit cursus. Vivamus scelerisque tortor et nunc ultricies, id tincidunt libero pharetra. Aliquam eu imperdiet leo. Morbi a massa

---

<sup>1</sup>Footnote example text...Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent porttitor arcu luctus, imperdiet urna iaculis, mattis eros. Pellentesque iaculis odio vel nisl ullamcorper, nec faucibus ipsum molestie.

volutpat velit condimentum convallis et facilisis dolor.

In hac habitasse platea dictumst. Curabitur mattis elit sit amet justo luctus vestibulum. In hac habitasse platea dictumst. Pellentesque lobortis justo enim, a condimentum massa tempor eu. Ut quis nulla a quam pretium eleifend nec eu nisl. Nam cursus porttitor eros, sed luctus ligula convallis quis. Nam convallis, ligula in auctor euismod, ligula mauris fringilla tellus, et egestas mauris odio eget diam. Praesent sodales in ipsum eu dictum. Mauris interdum porttitor fringilla. Proin tincidunt sodales leo at ornare. Donec tempus magna non mauris gravida luctus. Cras vitae arcu vitae mauris eleifend scelerisque. Nam sem sapien, vulputate nec felis eu, blandit convallis risus. Pellentesque sollicitudin venenatis tincidunt. In et ipsum libero.

#### 6.1.1.1 Subsubsection Title

Maecenas consectetur metus at tellus finibus condimentum. Proin arcu lectus, ultrices non tincidunt et, tincidunt ut quam. Integer luctus posuere est, non maximus ante dignissim quis. Nunc a cursus erat. Curabitur suscipit nibh in tincidunt sagittis. Nam malesuada vestibulum quam id gravida. Proin ut dapibus velit. Vestibulum eget quam quis ipsum semper convallis. Duis consectetur nibh ac diam dignissim, id condimentum enim dictum. Nam aliquet ligula eu magna pellentesque, nec sagittis leo lobortis. Aenean tincidunt dignissim egestas. Morbi efficitur risus ante, id tincidunt odio pulvinar vitae.

**Paragraph Title** Nullam mollis tellus lorem, sed congue ipsum euismod a. Donec pulvinar neque sed ligula ornare sodales. Nulla sagittis vel lectus nec laoreet. Nulla volutpat malesuada turpis at ultricies. Ut luctus velit odio, sagittis volutpat erat aliquet vel. Donec ac neque eget neque volutpat mollis. Vestibulum viverra ligula et sapien bibendum, vel vulputate ex euismod. Curabitur nec velit velit. Aliquam vulputate lorem elit, id tempus nisl finibus sit amet. Curabitur ex turpis, consequat at lectus id, imperdiet molestie augue. Curabitur eu eros molestie purus commodo hendrerit. Quisque auctor ipsum nec mauris malesuada, non fringilla nibh viverra. Quisque gravida, metus quis semper pulvinar, dolor nisl suscipit leo, vestibulum volutpat ante justo ultrices diam. Sed id facilisis turpis, et aliquet eros.

In malesuada ullamcorper urna, sed dapibus diam sollicitudin non. Donec elit odio, accumsan ac nisl a, tempor imperdiet eros. Donec porta tortor eu risus consequat, a pharetra tortor tristique. Morbi sit amet laoreet erat. Morbi et luctus diam, quis porta ipsum. Quisque libero dolor, suscipit id facilisis eget, sodales volutpat dolor. Nullam vulputate interdum aliquam. Mauris id convallis erat, ut vehicula neque. Sed auctor nibh et elit fringilla, nec ultricies dui sollicitudin. Vestibulum vestibulum luctus metus venenatis facilisis. Suspendisse iaculis augue at vehicula ornare. Sed vel eros ut velit fermentum porttitor sed sed massa. Fusce venenatis, metus a rutrum sagittis, enim ex maximus velit, id semper nisi velit eu purus.




**Unnumbered Section**

**Unnumbered Subsection**

**Unnumbered Subsubsection**





## 7. In-text Element Examples

### 7.1 Referencing Publications

This statement requires citation [Smith:2022jd]; this one is more specific [Smith:2021qr].

### 7.2 Link Examples

This is a URL link: [LaTeX Templates](#). This is an email link: [example@example.com](mailto:example@example.com).  
This is a monospaced URL link: `https://www.LaTeXTemplates.com`.

### 7.3 Lists

Lists are useful to present information in a concise and/or ordered way.

#### 7.3.1 Numbered List

1. First numbered item
  - a. First indented numbered item
  - b. Second indented numbered item
    - i. First second-level indented numbered item
2. Second numbered item
3. Third numbered item

#### 7.3.2 Bullet Point List

- First bullet point item
  - First indented bullet point item
  - Second indented bullet point item
    - First second-level indented bullet point item
- Second bullet point item

- Third bullet point item

### 7.3.3 Descriptions and Definitions

**Name** Description

**Word** Definition

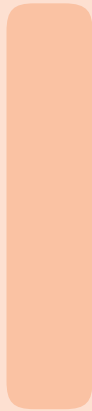
**Comment** Elaboration

## 7.4 International Support

àáâãäåæéêëìíîïðóôõöøùúûüýÿñçšž  
 ÀÁÂÃÄÅÈÉÊËÌÍÎÏÐÓÔÕÖØÙÚÛÜÝŸÑ  
 ßÇÆĖČŠŽ

## 7.5 Ligatures

fi fj fl ffl ff Ty



# Part Two Title

<b>8</b>	<b>Mathematics</b>	<b>47</b>
8.1	Theorems	47
8.2	Definitions	47
8.3	Notations	48
8.4	Remarks	48
8.5	Corollaries	48
8.6	Propositions	48
8.7	Examples	48
8.8	Exercises	49
8.9	Problems	49
8.10	Vocabulary	49
<b>9</b>	<b>Presenting Information and Results with a Long Chapter Title</b>	<b>51</b>
9.1	Table	51
9.2	Figure	51





## 8. Mathematics

### 8.1 Theorems

#### 8.1.1 Several equations

This is a theorem consisting of several equations.

**Theorem 8.1** — **Name of the theorem.** In  $E = \mathbb{R}^n$  all norms are equivalent. It has the properties:

$$||\mathbf{x}|| - ||\mathbf{y}|| \leq ||\mathbf{x} - \mathbf{y}|| \quad (8.1)$$

$$||\sum_{i=1}^n \mathbf{x}_i|| \leq \sum_{i=1}^n ||\mathbf{x}_i|| \quad \text{where } n \text{ is a finite integer} \quad (8.2)$$

#### 8.1.2 Single Line

This is a theorem consisting of just one line.

**Theorem 8.2** A set  $\mathcal{D}(G)$  is dense in  $L^2(G)$ ,  $|\cdot|_0$ .

### 8.2 Definitions

A definition can be mathematical or it could define a concept.

**Definition 8.1** — **Definition name.** Given a vector space  $E$ , a norm on  $E$  is an application, denoted  $||\cdot||$ ,  $E$  in  $\mathbb{R}^+ = [0, +\infty[$  such that:

$$||\mathbf{x}|| = 0 \Rightarrow \mathbf{x} = \mathbf{0} \quad (8.3)$$

$$||\lambda \mathbf{x}|| = |\lambda| \cdot ||\mathbf{x}|| \quad (8.4)$$

$$||\mathbf{x} + \mathbf{y}|| \leq ||\mathbf{x}|| + ||\mathbf{y}|| \quad (8.5)$$

### 8.3 Notations

■ **Notation 8.1** Given an open subset  $G$  of  $\mathbb{R}^n$ , the set of functions  $\varphi$  are:

1. Bounded support  $G$ ;
2. Infinitely differentiable;

a vector space is denoted by  $\mathcal{D}(G)$ .

### 8.4 Remarks

This is an example of a remark.



The concepts presented here are now in conventional employment in mathematics. Vector spaces are taken over the field  $\mathbb{K} = \mathbb{R}$ , however, established properties are easily extended to  $\mathbb{K} = \mathbb{C}$ .

### 8.5 Corollaries

**Corollary 8.1 — Corollary name.** The concepts presented here are now in conventional employment in mathematics. Vector spaces are taken over the field  $\mathbb{K} = \mathbb{R}$ , however, established properties are easily extended to  $\mathbb{K} = \mathbb{C}$ .

### 8.6 Propositions

#### 8.6.1 Several equations

**Proposition 8.1 — Proposition name.** It has the properties:

$$||\mathbf{x}| - |\mathbf{y}|| \leq |\mathbf{x} - \mathbf{y}| \quad (8.6)$$

$$||\sum_{i=1}^n \mathbf{x}_i|| \leq \sum_{i=1}^n ||\mathbf{x}_i|| \quad \text{where } n \text{ is a finite integer} \quad (8.7)$$

#### 8.6.2 Single Line

**Proposition 8.2** Let  $f, g \in L^2(G)$ ; if  $\forall \varphi \in \mathcal{D}(G)$ ,  $(f, \varphi)_0 = (g, \varphi)_0$  then  $f = g$ .

### 8.7 Examples

#### 8.7.1 Equation Example

■ **Example 8.1** Let  $G = \{x \in \mathbb{R}^2 : |x| < 3\}$  and denoted by:  $x^0 = (1, 1)$ ; consider the function:

$$f(x) = \begin{cases} e^{|x|} & \text{si } |x - x^0| \leq 1/2 \\ 0 & \text{si } |x - x^0| > 1/2 \end{cases} \quad (8.8)$$

The function  $f$  has bounded support, we can take  $A = \{x \in \mathbb{R}^2 : |x - x^0| \leq 1/2 + \varepsilon\}$  for all  $\varepsilon \in ]0; 5/2 - \sqrt{2}[$ . ■



### 8.7.2 Text Example

■ **Example 8.2 — Example name.** Aliquam arcu turpis, ultrices sed luctus ac, vehicula id metus. Morbi eu feugiat velit, et tempus augue. Proin ac mattis tortor. Donec tincidunt, ante rhoncus luctus semper, arcu lorem lobortis justo, nec convallis ante quam quis lectus. Aenean tincidunt sodales massa, et hendrerit tellus mattis ac. Sed non pretium nibh. Donec cursus maximus luctus. Vivamus lobortis eros et massa porta porttitor. ■

## 8.8 Exercises

**Exercise 8.1** This is a good place to ask a question to test learning progress or further cement ideas into students' minds. ■

## 8.9 Problems

**Problem 8.1** What is the average airspeed velocity of an unladen swallow?

## 8.10 Vocabulary

Define a word to improve a students' vocabulary.

■ **Vocabulary 8.1 — Word.** Definition of word.





## 9. Presenting Information and Results with a Long Chapter Title

### 9.1 Table

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent porttitor arcu luctus, imperdiet urna iaculis, mattis eros. Pellentesque iaculis odio vel nisl ullamcorper, nec faucibus ipsum molestie. Sed dictum nisl non aliquet porttitor. Etiam vulputate arcu dignissim, finibus sem et, viverra nisl. Aenean luctus congue massa, ut laoreet metus ornare in. Nunc fermentum nisi imperdiet lectus tincidunt vestibulum at ac elit. Nulla mattis nisl eu malesuada suscipit.

Treatments	Response 1	Response 2
Treatment 1	0.0003262	0.562
Treatment 2	0.0015681	0.910
Treatment 3	0.0009271	0.296

表 9.1: Table caption.

Referencing Table 9.1 in-text using its label.

### 9.2 Figure

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent porttitor arcu luctus, imperdiet urna iaculis, mattis eros. Pellentesque iaculis odio vel nisl ullamcorper, nec faucibus ipsum molestie. Sed dictum nisl non aliquet porttitor. Etiam vulputate arcu dignissim, finibus sem et, viverra nisl. Aenean luctus congue massa, ut laoreet metus ornare in. Nunc fermentum nisi imperdiet lectus tincidunt vestibulum at ac elit. Nulla mattis nisl eu malesuada suscipit.

Treatments	Response 1	Response 2
Treatment 1	0.0003262	0.562
Treatment 2	0.0015681	0.910
Treatment 3	0.0009271	0.296

表 9.2: Floating table.



图 9.1: Figure caption.

Referencing Figure 9.1 in-text using its label.



图 9.2: Floating figure.

# Bibliography

Articles

Books





## A. Appendix Chapter Title

### A.1 Appendix Section Title

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam auctor mi risus, quis tempor libero hendrerit at. Duis hendrerit placerat quam et semper. Nam ultricies metus vehicula arcu viverra, vel ullamcorper justo elementum. Pellentesque vel mi ac lectus cursus posuere et nec ex. Fusce quis mauris egestas lacus commodo venenatis. Ut at arcu lectus. Donec et urna nunc. Morbi eu nisl cursus sapien eleifend tincidunt quis quis est. Donec ut orci ex. Praesent ligula enim, ullamcorper non lorem a, ultrices volutpat dolor. Nullam at imperdiet urna. Pellentesque nec velit eget est euismod pretium.







## B. Appendix Chapter Title

### B.1 Appendix Section Title

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam auctor mi risus, quis tempor libero hendrerit at. Duis hendrerit placerat quam et semper. Nam ultricies metus vehicula arcu viverra, vel ullamcorper justo elementum. Pellentesque vel mi ac lectus cursus posuere et nec ex. Fusce quis mauris egestas lacus commodo venenatis. Ut at arcu lectus. Donec et urna nunc. Morbi eu nisl cursus sapien eleifend tincidunt quis quis est. Donec ut orci ex. Praesent ligula enim, ullamcorper non lorem a, ultrices volutpat dolor. Nullam at imperdiet urna. Pellentesque nec velit eget est euismod pretium.