第六章　封装

6.1 类的抽象与封装

- 每个组件都可以看做是一个有属性和方法的对象。
- 内部功能被封装起来，对使用者是隐藏的。

主板

cpu

硬盘

内存

风扇

6.1 类的抽象与封装

- Fan类提供了公开的构造、开启和关闭方法，公开的方法(public)是向外界提供的接口。
- Fan的属性是private，被封装起来，外部不能访问。

风扇

```
//创建电风扇对象gree
Fan gree= new Fan(4.5,400);

//调用开启方法
Gree.turnOn();
```

6.1 类的抽象与封装

- 类的**封装**隐藏了实现细节，通过公开的方法访问数据。为了更好的保护类，类的属性和方法要设置访问控制权限。

- 常用的封装手段有：

  （1）**修改属性的可见性**以达到限制访问的目的。

  （2）**设置对属性进行读取的方法**，以便实现属性的访问。

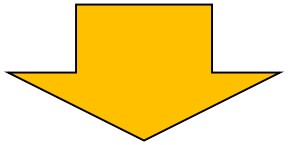
  （3）在读取属性的方法中，**添加对属性读取的限制**。

```
//Fan.java
public class Fan {
    private double size;   //使用private将属性对外隐藏
    private int speed;     //使用private将属性对外隐藏

    //提供公开的方法设置和读取属性
    public void setSize(double size){
        this.size = size;
    }
    public double get size() {
        return size;
    }
    public void setSpeed(double speed){
        this.speed  speed;
    }
    public int getSpeed(){
        return speed;
    }
}
```

6.2 Java中的包

如何解决命名冲突的问题?



Java采用包（Package）来管理类名空间，也是一种课件性限制的机制，提高了安全性

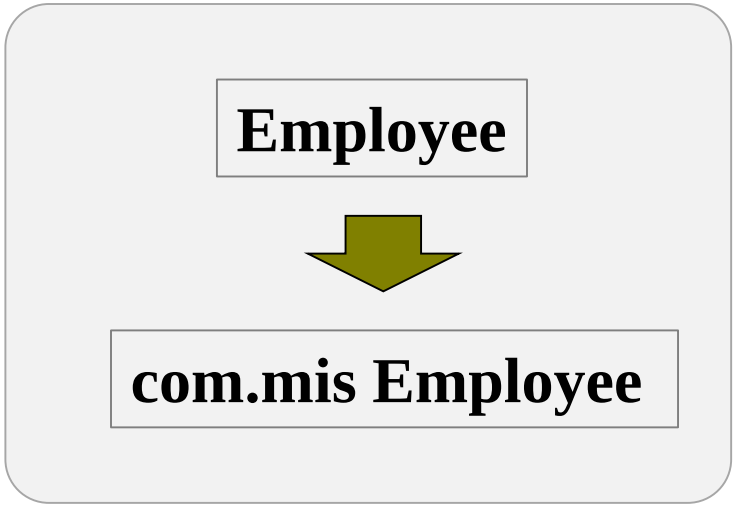
例如：java.lang : Java的基础类

java.io : 所有读和写的类

```
package com.mis;
public class Employee{
        private String number;
        private String name;

        public Employee(String numberIn, String nameIn){
                number=numberIn;
                name=nameIn;
        }
        public void setName(String nameIn){
                name=nameIn;
        }
        public String getNumber(){
                return number;
        }
        public String getName(){
                return name;
        }
}
```

编译后，生成的类文件都放在工作目录的com/mis文件夹中



6.2 Java中的包

- package语句：用package语句，包的层次结构必须与文件目录的层次相同。否则，在编译时可能出现查找不到的问题。

- import语句：如果不在一个包中，访问使用【包名.类名】的形式或就是用关键字import导入所在的包，这样就可以直接用类名访问。

6.2 Java中的包

```
import java.util.Scanner;
// import java.util.*
class testImport{
    public static void main(String ags[]){
        Scanner sc = Scaner(System.in);
        .
        .
        .
    }
}
```

6.2 Java中的包

- **java.lang**：包含一些Java语言的核心类，如String、Math、Integer、System和Thread,提供常用功能。
- **java.awt**：包含了构成抽象窗口工具集(abstract window toolkits)的多个类,这些类被用来构建和管理应用程序的图形用户界面(GUI)。
- **java.applet**：包含applet运行所需的一些类。
- **java.net**：包含执行与网络相关的操作的类。
- **java.io**：包含能提供多种输入/输出功能的类。
- **java.util**：包含一些实用工具类,如定义系统特性、使用与日期日历相关的函数。

Calendar类

- 该类通过调用getInstance()静态方法获取一个Calendar对象，此对象已由当前日期时间初始化，即默认代表当前时间。

```
Calendar c = Calendar.getInstance();
```

Calendar类

- public static Calendar getInstance()

  创建Calendar对象public int get(int field)返回给定日历字段的值。日历类中的每个日历字段都是静态成员变量，且都是int类型

- public void add(int field int amount)

  根据给定的日历字段和对应的时间，来对当前日历进行操作

- public final void set(int year,int month,int date)

  设置当前日历的年月日

- public final Date getTime()

  用来获取Date对象，完成Calendar和Date的转换

- public long getTimeInMillis()

  返回此Calendar的时间值，以毫秒为单位（和Date类的getTime方法类似）

DateFormat类

- SimpleDateFormat类是DateFormat类(抽象类)的子类

  作用：对日期时间进行格式化(如：可以将日期转换为指定格式的文本，也可以将文本转换为日期)

构造方法

- public SimpleDateFormat()

  用默认的模式和默认语言环境的日期格式符号构造

- public SimpleDateFormat(String pattern)

  用给定的模式和默认语言环境的日期格式符号构造

成员方法

- public final String format(Date date)

  Date-String(格式化)（将日期转换为指定文本格式）

- public Date parse(String source)

  String-Date(解析)（将文本转换为日期）

DateFormat类

字母	日期或时间元素	表示	示例
G	Era 标志符	Text	AD
y	年	Year	1996; 96
M	年中的月份	Month	July; Jul; 07
w	年中的周数	Number	27
W	月份中的周数	Number	2
D	年中的天数	Number	189
d	月份中的天数	Number	10
F	月份中的星期	Number	2
E	星期中的天数	Text	Tuesday; Tue
a	Am/pm 标记	Text	PM
H	一天中的小时数（0-23）	Number	0
k	一天中的小时数（1-24）	Number	24
K	am/pm 中的小时数（0-11）	Number	0
h	am/pm 中的小时数（1-12）	Number	12
m	小时中的分钟数	Number	30
s	分钟中的秒数	Number	55
S	毫秒数	Number	978
z	时区	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	时区	RFC 822 time zone	-0800

http://blog.csdn.net/w_linux

6.3 类的成员的访问控制

- **包访问权限**：成员变量或成员方法前不使用任何访问权限修饰符，就是默认的包访问权限。

- 默认的访问权限的访问范围是：
  - 本类的成员方法可以访问
  - 与该类在同一个包中的类也可以访问

6.3 类的成员的访问控制

- **public**：接口访问权限，使用关键字public修饰成员变量或成员方法，就意味着是公开的，任何类的成员方法均可访问。

- **private**：类内部访问权限，表示私有的，被private修饰的成员，仅能被包含该成员的类访问，任何其他类都不能访问，即private成员只能在类的内部使用。

6.3 类的成员的访问控制

- **protected**：继承访问权限，表示受保护的，主要修饰存在继承关系的类。被protected修饰的类的成员，既可以被同一包中的其他类访问，也可以被不同包中的子类访问。可见，protected 比默认访问权限的访问范围要宽。

protected = 默认权限 + 不同包中的子类

6.4 类的访问权限

- 类（内部类除外）的访问权限仅有两个：包访问或是public。类不可以被private和protected修饰，内部类除外。
- 如果希望某个类被任何类都能访问，用public修饰类。这样做时，注意类所在的文件名要与被public修饰的类名完全一样，否则就会编译出错。
- 类名前没有任何修饰时，就是默认的包访问权限，在同一包中的类可以访问。

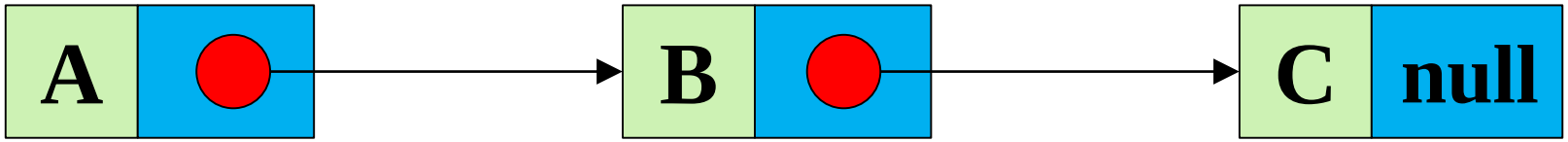
6.5 链表实例

- 链表是一种物理存储单元上的非连接、非顺序的存储结构，数据元素中的逻辑顺序是通过链表中的指针链接次序实现的。
- 链表由一系列结点组成，结点可以在运行时动态生成。
- 单向链表结点包括两个部分：
  - 存储数据元素的数据域
  - 存储下一个结点地址的指针域
- 相比于线性表顺序结构，链表对插入和删除的操作效率更快。

6.5 链表实例

- 链表是一种物理存储单元上的非连接、非顺序的存储结构，数据元素中的逻辑顺序是通过链表中的指针链接次序实现的。
- 链表由一系列结点组成，结点可以在运行时动态生成。
- 单向链表结点包括两个部分：
  - 存储数据元素的数据域
  - 存储下一个结点地址的指针域
- 相比于线性表顺序结构，链表对插入和删除的操作效率更快。

6.5 链表实例

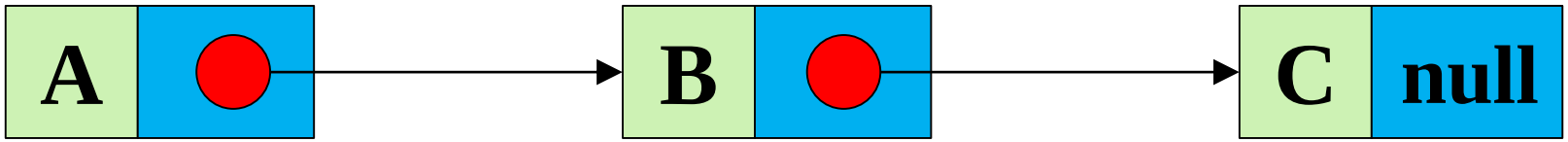


6.5.1 链表结点

```
package linklist;
public class ListCell {
    // 属性 字符型的节点内容和指向下个节点的指针
    char item='\0';
    ListCell next=null;
    // 构造方法
    ListCell(){
    }
    ListCell(char c, ListCell cell){
        item=c;
        next=cell;
    }
    char content(){
        return item;
    }
}
```

每一个结点对象都存储了字符数组和指向下一个结点的指针。

```
ListCell listx = new ListCell('C',null);     // 链表: (C)
listx = new ListCell('B',listx);             // 链表: (B C)
listx = new ListCell('A',listx);             // 链表: (A B C)
```



如果链表由很多结点，都可以采用从链表头插入的方式创建。

6.5.2 链表类

- 链表中最关键的结点是指向第一个结点的头结点，链表的创建、查找、删除等操作都要使用头结点。
- 链表的操作有：
  - 初始化链表
  - 判断链表是否为空
  - 查找结点
  - 删除结点
  - 替换
  - 输出

6.5.1 链表结点

```
package linklist;
public class ListCell {
    // 属性 字符型的节点内容和指向下个节点的指针
    char item='\0';
    ListCell next=null;
    // 构造方法
    ListCell(){
    }
    ListCell(char c, ListCell cell){
        item=c;
        next=cell;
    }
    char content(){
        return item;
    }
}
```

1. ListCell first() 方法

找到链表的第一个结点，返回头结点。头结点就是第一个结点。

```
package linklist;
public class CharList {
        private ListCell head=null;
        public ListCell first (){
                return head;     // gets first cell
        }
}
```

2. Boolean isEmpty() 方法

判断链表是否为空。头结点如果是空，则表示链表为空。方法返回判断boolean型的结果

```
package linklist;
public class CharList {
        private ListCell head=null;
        public boolean isEmpty(){
                return head==null;
        }
}
```

3. ListCell last() 方法

- 找到链表的最后一个结点。
- 最后一个结点的指针域为空，其他节点的指针域都不为空。
- 因此，从头结点开始顺序查找，找到指针域为空的结点就是最后一个结点。

3. ListCell last() 方法

```
package linklist;
public class CharList {
        private ListCell head=null;
        public ListCell last(){
                ListCell p=head;
                while (p !=null && p.next !=null)
                        p=p.next;
                return p;
        }
}
```

4. ListCell find(char c) 方法

- 查找数据域是字符c的结点。
- 每个结点都有数据域item，item是字符型的数据。
- 从链表头开始(head),到链表尾为止 (指针域为null)，结点的数据域依次比较。
- 如果找到则返回当前结点，如果没有则返回null。

4. ListCell find(char c) 方法

```
package linklist;
public class CharList {
        private ListCell head=null;
        public ListCell find(char c){
                for(ListCell p=head; p!=null; p=p.next)
                        if(p.item ==c)
                                return p;
                return null;
        }
}
```

5. boolean substitute(char r, char s) 方法

- 将字符r替换链表中的第一个字符s。
- 首先要找到链表中的字符s，下一步才能替换。
- 查找方法find(char c)可以返回制定数据域内容的结点，因此直接调用该方法。
- 将找到的结点数据域替换为r。

5. boolean substitute(char r, char s) 方法

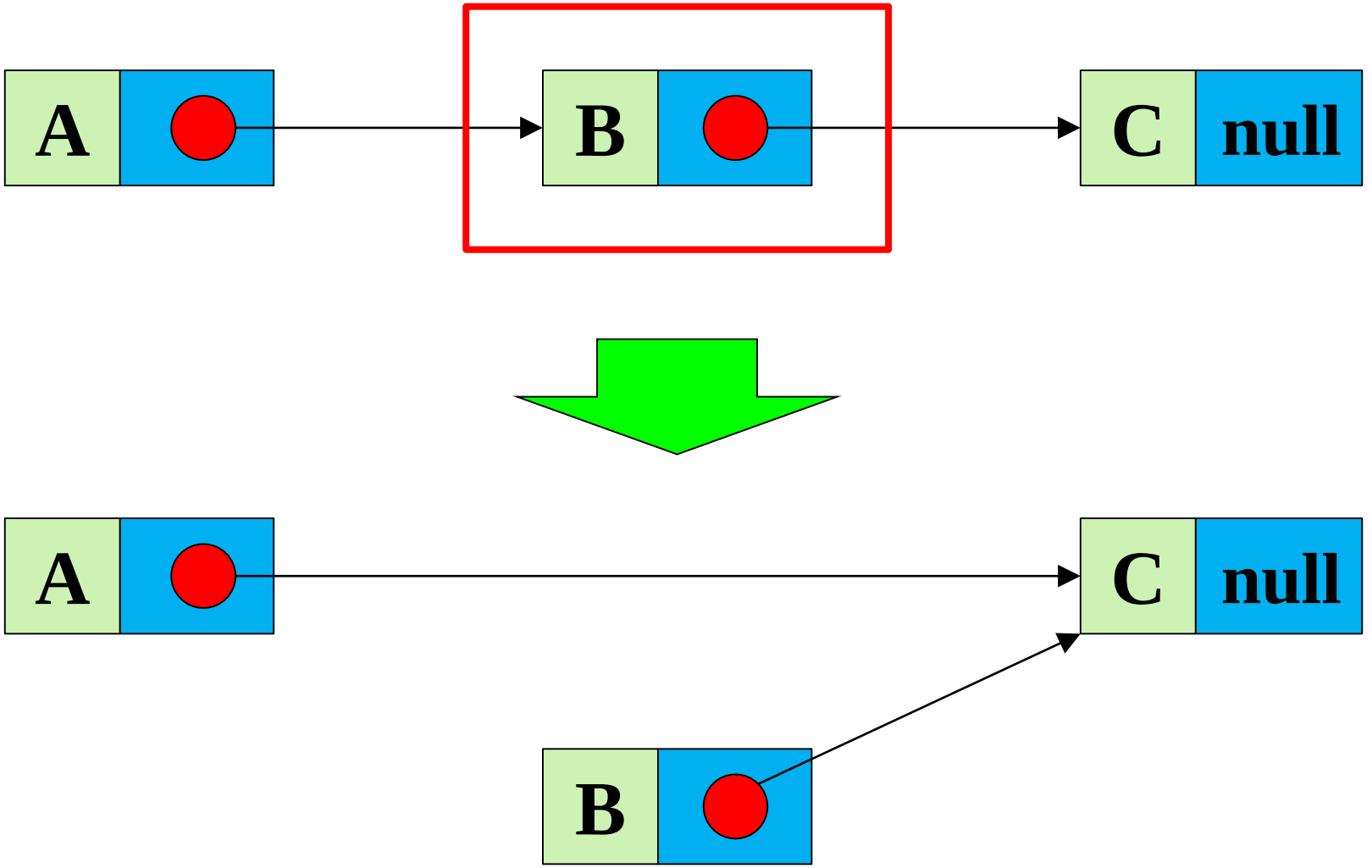
```
package linklist;
public class CharList {
        private ListCell head=null;
        public boolean substitute (char r, char s) {
                ListCell p=find(s);
                if(p==null) return false;     //s not on list
                p.item=r;
                return true;
        }
}
```

6. int remove(char c) 方法

删除链表中数据域为字符c的结点，返回删除结点的个数。

- 由于查找只能找到第一个结点，不符合操作的要求，因此不能使用find方法。
- 要把链表中的所有数据域为c的结点都找到，就要从表头开始，到链表尾为止，依次比较，发现目标时删除结点。
- 删除操作就是把结点从链表中去除，即将目标结点的前一个结点指针域指向目标结点的下一个结点。

6. int remove(char c) 方法



```
// removes c from entire list
public int remove(char c){
        ListCell p=head;
        int count=0;  //number of items removed
        if( p==null)  return count;
        //treat all but head cell
        while(p.next != null){
                if ((p.next).item ==c){
                        count ++;
                        p.next=(p.next).next;
                }else
                        p=p.next;
        }
        // treat head cell
        if (head.item==c)  {
                head=head.next;
                count++;
        }
        return count;
}
```

7. void putOn(char c) 方法

从链表头插入结点，数据域为字符c。

```
package linklist;
public class CharList {
        private ListCell head=null;
        //insert in front
        public void putOn(char c){
                head=new ListCell(c,head);
        }
}
```

8. void insert(char c, ListCell e) 方法

在结点e之后插入数据域为c的字符。类似putOn()方法，需要新建结点。新结点是结点e的后续。

```
package linklist;
public class CharList {
        private ListCell head=null;
        //insert after cell
        public void insert(char c, ListCell e){
                e.next=new ListCell(c,e.next);
        }
}
```

9. void append(char c) 方法

在链表尾追加结点。相当于insert()，插入的结点位置为last()，因此调用insert(c, last())即可。

```
package linklist;
public class CharList {
        private ListCell head=null;
        //insert at end
        public void append(char c){
                insert(c,last());
        }
}
```

10. public String toString() 方法

- 显示链表，希望以此显示链表结点的数据域，以(A B C)的形式展示。

- 该方法是重写父类Object的toString()方法。

```
package linklist;
public class CharList {
        private ListCell head=null;
         //overladed toString methods
        public String toString(){  //for whole list
                return toString(head);
        }
        public String toString(ListCell p){
                String s="(";
                while (p !=null){
                        s=s +p.item;
                        if((p=p.next) !=null)
                        s=s+" ";
                }
                return  (s+ ")");
        }
}
```

6.5.3 测试类

测试类要新建链表，并使用插入、查找、替换、删除等操作。

```
package linklist;
public class TestCharList {
        public static void main(String[] args) {
                CharList a=new CharList('B');
                a.putOn('A');
                a.putOn('D');
                a.putOn('E');
                a.putOn('F');
                System.out.println("a=" + a);
                ListCell lp=a.find('B');
                System.out.println(a.toString(lp));
                a.insert('C', lp);
                System.out.println("after insert C, a=" + a);
                a.remove('E');
               System.out.println("after remove E, a=" + a);
        }
}
```

数组

- 在内存中，数组是一块连续的区域。

- 数组需要预留空间，在使用前要先申请占内存的大小，可能会浪费内存空间。插入数据和删除数据效率低，插入数据时，这个位置后面的数据在内存中都要向后移。删除数据时，这个数据后面的数据都要往前移动。

- 随机读取效率很高。因为数组是连续的，知道每一个数据的内存地址，可以直接找到给地址的数据。并且不利于扩展，数组定义的空间不够时要重新定义数组。

链表

- 在内存中可以存在任何地方，不要求连续。
- 每一个数据都保存了下一个数据的内存地址，通过这个地址找到下一个数据。
- 增加数据和删除数据很容易。
- 查找数据时效率低，因为不具有随机访问性，所以访问某个位置的数据都要从第一个数据开始访问，然后根据第一个数据保存的下一个数据的地址找到第二个数据，以此类推。
- 不指定大小，扩展方便。链表大小不用定义，数据随意增删。

链表和数组的区别

- 数组的优点：
  - 随机访问性强
  - 查询速度快
- 数组的缺点：
  - 增删速度慢
  - 可能浪费内存
  - 内存空间要求高，必须有足够大的连续内存存储空间。
  - 数组的大小固定，不能动态扩展。
- 链表的优点
  - 插入删除速度快
  - 大小不固定，可以动态扩展。
  - 内存利用率高，不会浪费内存
- 链表的缺点：
  - 不能随机查找，必须从第一个开始遍历，查找效率低。