DATA STRUCTURE

Based on 0445

Lecture 1

类是数据的计划大纲.

- 类的结构将数据和操作 封装

  Instance data & Instance methods

- 封装允许通过非公开声明进行访问限制(如 data hiding)

  对使用者隐藏实现细节

  类定义者可以明确用户访问的内容 — public, protected, private

- 用户知道数据性质和 public method的规范,但不了解实现细节

  用户不必知道 — 类中使用的数据

  类的方法实现

  ↑↑ Data abstraction

- 通过封装,限制对类的实现细节的访问
- 通过数据抽象,用户使用类而不必知道实现细节

对象是类的实例.

- 每个对象都有类指定的结构与功能.

类定义的关键字

public — 类外可以访问　　private — 类外不可访问

protected — 只能在类,其子类[和同一个包]内访问

static — 类的一部分而不是实例,由所有实例共享

final — 常量 — 不能分配(变量),覆写(方法),子类化

Lecture 2

Java变量大多为引用 →

PRIMITIVE TYPE

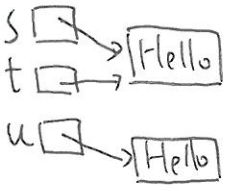
```
int i;

i = 8643;

int j = i;

if (i == j)     true
```

i 8643

j 8643

REFERENCE TYPE

```
String s;

s = new String("Hello");

String t = s;
String u = new String(s);

if (t == s) ...    true
if (u == s) ...    false
```



关于比较(reference & content)

对于reference variable,使用方法 /例.使用equals()方法比较字符串

[我们不能重新定义比较运算符,必须使用命名的方法]

Reference与Pointer指针如何联系?

C语言使用pointer variables ─ 指针是存储其它内存位置地址的变量
                            └ 指针允许间接访问对象中的数据

也存储在指针中的值是一个地址.

(改变一个pointer variable实质改变其的指向)

取消对指针的引用,则可以访问它"指向"的对象

Java中的Reference与指针相似,但限制更大

没有取消引用操作符,=>使用"dot" "."访问数据/方法

Java中没有指针

但aliasing混叠仍可能发生,注意构建新对象&对旧对象的引用

```
StringBuilder S1 = new StringBuilder("Hello");
StringBuilder S2 = S1;
S1.append(" There");
S2.append(" CS 0445 Students");
System.out.println(S1.toString());
```

Hello There CS 0445 Students

S1、S2指向同一个对象

内存使用

使用new运算符动态分配 ─ 对对象的
                          ├ 有效引用,无限期存在
                          └ 对对象无引用,为标记为Garbage Collection

garbage collator于后台运行 ─ 内存不够多时,垃圾收集器回收标记对象
                            └ 内存充足时,垃圾收集器不会运行

Java有很多预定义类,但仍需自己定义

两种方式 ─ Composition (Aggregation)
          └ Inheritance

1° Composition组合 :用先前定义的类中的instance variables组成新类

- 新类的对象"Has a"旧类的对象
- 新类对取来的instance variable没有特殊访问权限
- 新类中的方法通过instance variable对象的方法来实现

(人人先前的成果用)

```
public class CompoClass
{
   private String name;
   private Integer size;
   public CompoClass(String n, int i)
   {
        name = new String(n);
        size = new Integer(i);
   }
   public void setCharAt(int i, char c)
   {
        StringBuilder b = new StringBuilder(name);
        b.setCharAt(i, c);
        name = b.toString();
   }
}
```

无法访问String，其对象不可更改
只能使用StringBuilder

- 一些类包含mutator方法：允许改变对象的内容。
  对象被称为mutable

  例：append()方法加入字符到现有StringBuilder
  setFrame()方法改变Rectangle2D.Double的大小和位置
  add(), remove()方法改变ArrayList

- 一些类不包含mutator方法：这些类里的对象immutable

  例：String
  wrapper objects (Integer, Float ...)

可变对象 VS 不可变对象

1° 不变的复杂性（mutation操作需要很多工作）

例：字符串连接 `String s1 = "Hello";`
`s1 = s1 + "there";` > 创建了两个一个新对象，而不是把String加到现有对象

2° 可变的复杂性

- 对象的集合 把对象加进集合同样可以从外部方法访问该对象
  将对象外部改动集合，破坏不集合属性

例：创建ArrayList的子类SortedArrayList => 必须根据compareTo()方法按从小到大序维护数据
此时

若该子集of StringBuilder，意图改变

? 使原对象不变，将其副本放入集合
为不出错，应当返回对象的副本

另：创建副本：copy constructor & clone() method

- 所有Java类都能用clone()方法（但除了数组，我们仍需要override）
- 任何新Java类都能写copy constructor

许多类都用 composition 来构建（一个对象包含对其它对象的引用，层层套娃）

复制对象时，其内部引用 —— shallow copy，将引用东西之给新对象（两者引用指向相同的嵌套对象）
—— deep copy，嵌套对象也被复制

```
SBArray orig = new SBArray(5);
orig.add(new StringBuilder("One"));
orig.add(new StringBuilder("Two"));
SBArray deep = new SBArray(orig);
orig.set(0, new StringBuilder("Three"));
deep.get(1).append("ish");
```

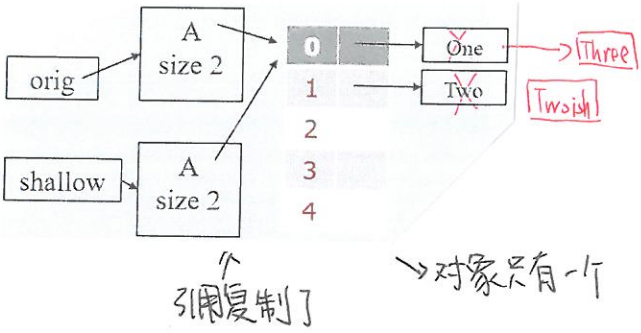
// 5个元素的数组
// 放入2个元素
// 复制
// 第1个元素设为 Three
// 第2个元素加入字符 ish

copy 方式？

StringBuilder 建立新对象

1° Shallow Copy

```
public class SBArray
{
   private StringBuilder [] A;
   private int size;

   // shallow copy
   public SBArray(SBArray old)
   {
      A = old.A;
      size = old.size;
   }
}
```

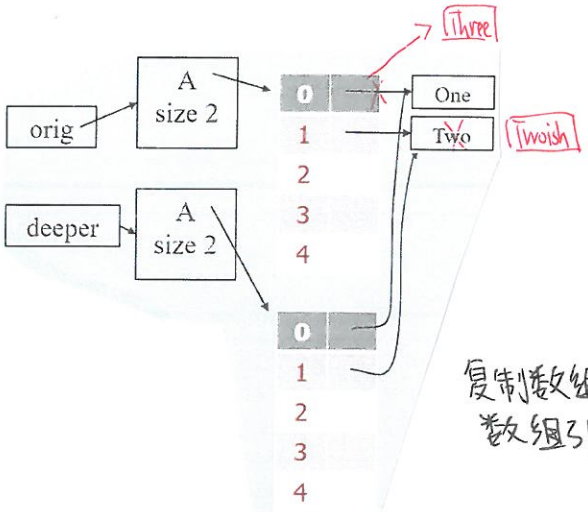


引用复制了

对象只有一个

2° Deeper Copy

```
public class SBArray
{
   private StringBuilder [] A;
   private int size;

   // deeper copy
   public SBArray(SBArray old)
   {
      A = new StringBuilder[
            old.A.length];
      size = old.size
      for (int i=0; i<size; i++)
          A[i] = old.A[i];
   }
}
```

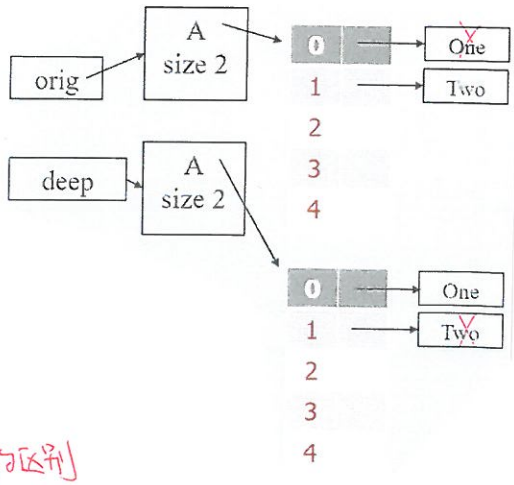


复制数组，数组引用的对象共享

3° Deep Copy

```
public class SBArray
{
   private StringBuilder [] A;
   private int size;

   // deep copy
   public SBArray(SBArray old)
   {
      A = new StringBuilder[
            old.A.length];
      size = old.size
      for (int i=0; i<size; i++)
          A[i] = new StringBuilder(
                     old.A[i]);
   }
}
```

与上面的区别



全盘复制

orig 与 deep 完全分离

通常(true)深层复制很复杂，其涉及数层深的所有引用

以链表为例(对前节点有引用)，深层复制遍历整个列表并复制所有节点及数据

2° Inheritance继承：通过继承已定义的类(父类)创建子类

- 子类有父类所有的属性(数据&方法)
- "Is a" subclass is a superclass；子类对象能当成给父类变量

例：Foo与SubFoo

```
public class Foo
{
        public void foomethod()
        { // implementation here }
}
public class SubFoo extends Foo
{
        public void subfoomethod()
        { // implementation here }
}
```

考虑以下语句

```
Foo f1;
SubFoo s1;
f1 = new Foo(); // √
f1 = new SubFoo(); //√，只能访问父类Foo
                   //最早定义的变量与方法
f1.foomethod(); // √(访问)
f1.subfoomethod(); //X不存在该子类方法
((SubFoo)f1).subfoomethod();
                   //√，可以访问
                   //此时引用指向实际的类
s1 = new SubFoo(); //√
s1.subfoomethod();
s1.foomethod();
s1 = new Foo(); //X[违反"is a"错]
```

一些规则

- 父类引用的数组&集合能用于访问父类对象与子类对象的混合
- 父类和子类中都定义了一个方法(签名相同)，调用时使用与类相对应的版本。
  (方法相似，子类自己定义使其更细化)

```
public class Animal
{
  // omitted decls
  public void move()
  {
     System.out.println("I move");
  }
}

public class Bird extends Animal
{
  // omitted decls
  public void move()
  {
     System.out.println("I fly");
  }
}
```

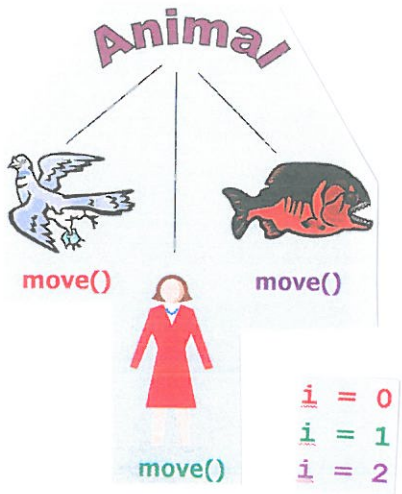
redefine move() 反映鸟会飞的特征

override

```
Animal [] A = new Animal[3];
A[0] = new Bird();
A[1] = new Person();
A[2] = new Fish();
for (int i = 0; i < A.length; i++)
        A[i].move();
```

语法相同

子类以自我方式覆写move()方法



多态

① Method overriding 方法覆写

- (方法签名相同), 在父类中定义过的方法在子类中重定义.
- 由于签名相同, 是 override 而不是 overload (ad hoc多态)

(使用父类引用访问子类对象, 方法定义会替换为子类版本)

② Dynamic binding 动态绑定

- 运行时, 代码与调用绑定
- 执行方法取决于对象类型, 而不是引用类型

/多态有助于访问混合数据类型　　例: 一组不同图形, 有着 draw() 方法
绘制方式不同, draw() 方法不同, 但调用一致

Abstract class 抽象类

- 声明时使用关键字 abstract
- 不能实例化&定义任何对象, 只有被继承才能使用.
  包含方法, 但不实现
  必须由抽象类的子类, 必须实现其所有抽象方法
- 优势: 1. 多态: 使用父类引用访问子类对象 (以Animal类为例, 最好它是abstract类, move()为abstract方法)
  在其中定义通用data&method, 由子类继承, 组织类结构.

Lecture 3

Java中的继承是单一的: 一个子类只能有一个父类
但是, 接口使得多个父类引用访问一个子类对象

接口是一组命名的抽象方法.
视作无实例的抽象类

- 允许
  - static constant
  - default method
  - static method

{ 无实例数字段
实例方法无主体
接口本身无法实例化

任何Java类都能通过实现其中定义的方法来实现接口, 数量不限

```
public interface Laughable
{
   public void laugh();
}

public interface Booable
{
   public void boo();
}
```

例: 实现其中方法就实现接口

```
public class Comedian implements Laughable, Booable
{
   // various methods here (constructor, etc.)
   public void laugh()
   {
       System.out.println("Ha ha ha");
   }
   public void boo()
   {
       System.out.println("You stink!");
   }
}
```

多态适用于接口，使接口充当父类
实现类来实现想要的方法

-接口变量能引用任何实现该接口的对象
⇓ 但接口方法只能被接口引用访问

```
Laughable [] funny = new Laughable[3];
funny[0] = new Comedian();
funny[1] = new SitCom(); // implements Laughable
funny[2] = new Clown();  // implements Laughable
for (int i = 0; i < funny.length; i++)
    funny[i].laugh();
```

Sorting 排序

- selection sort：简单⇒找到最小的，放到位置0；第二小的，放到位置1…
- Sort任意 comparable object
  ⇓ Compare<T>接口 `int compareTo(T r);`
  - <r，返负数
  - =r，返0
  - >r，返正数

  允许任意基础数据

  ⇒变成a[i]与a[j]的比较

```
public static <T extends Comparable<? super T>>
       void selectionSort(T[] a, int n)
{
    for (int index = 0; index < n - 1; index++)
    {
        int indexOfNextSmallest = getIndexOfSmallest(a,index,n-1);
        swap(a, index, indexOfNextSmallest);
      } // end for
} // end selectionSort
private static <T extends Comparable<? super T>>
        int getIndexOfSmallest(T[] a, int first, int last)
{
    T min = a[first];
    int indexOfMin = first;
    for (int index = first + 1; index <= last; index++)
    {
        if (a[index].compareTo(min) < 0)
        {
            min = a[index];
            indexOfMin = index;
        } // end if
      } // end for
      return indexOfMin;
} // end getIndexOfSmallest
```

实参是T的数组

简单一方法

compareTo的实参是类型T
比较对象要兼容
否则导致编译错误
例 C1.compareTo(C2)
同一继承路径！

以Java泛型来模拟数组
- 为一个位置存放一个值
- 从位置检索值
- 分配存数组大小

```
public class MyArray<T>
{
    private T[] theArray
```

//T没有限制，可以是任何reference type

```
MyArray<String> A = new MyArray<String>();
```

MyArray类型并非真正有用
数组的创建：制作array of object，将其转换为T

Lecture 4

Abstract Data Type (ADT)

数据类型可以视为两者的封装 — data本身及在内存中的表示 — 对类是 instance variables
— 操作数据的operation — 对类是 methods

reference type可以实现 & primitive type 不行

以BigInteger为例：数据的性质及可以对数据进行的操作才有用
↓有些运算符无用，想实现必须用method

abstract part？— 不必知道实现细节
— 只用知道它是什么，操作怎么做

ADT is a data type (data + operation)，其功能与实现是分开的
— 不同的实现产生相同的功能（使用者只用知道功能）
（但实施者应当知道实施细节）

- ADT是language-independent数据类型
  — 不同的编程语言以不同方式定义抽象数据类型
- Class是language-specific的允许实现ADT的结构（存在于面向对象语言）

ADT到类没有一一映射

给定的ADT能用不同的类以不同的方式实现，例：Stack, Queue, SortedList
给定的类能用来表示多个ADT，例，类ArrayList数组能表示 ∀（效率不保证）

考虑接口：粗略指定一组行为&能力，但不指定实现方式与依赖数据
适合ADT

文本将接口用作ADT，将类用作ADT实现
— 接口依赖于数据描述而非实际数据
它是不确定的，在实现接口的类中细化，数据特定于实现

例，ADT栈
- 将一个对象推到栈的顶部
- 将一个对象弹出栈的顶部

— 不关心数据实际表达方式，按指示工作

许多ADT表示数据的集合 — 通过由接口用作的ADT指定组织与访问抽象对象
— Class以某种方式完成数据&操作的特定实现

两个常用ADT：Queue & Doule Ended Queue (Deque)

① Queue 队列

想法：数据加到逻辑后端，从逻辑前端移除。（前端删除，后端插入）

操作：offer() 插入一个元素到队尾 // enqueue()

poll() 弹出队头元素 // dequeue()

peek() 查看队头元素，但不移除。 // getFront()

```
public interface QueueInterface<T>   //声明队列接口
{
  public void enqueue(T newEntry);   //加到队尾

  public T dequeue();

  public T getFront();

  public boolean isEmpty();   //队列是否空

  public void clear();   //重设队列到空状态

}
```

实施者需要清楚 & 使用者不必

{ 数据如何存储
{ 操作如何实施

队列管理数据遵循 FIFO（先进先出）

FIFO用于模拟现实生活：银行为柜员设置线路

选1：每个柜员单独一线； 选2：一条单行线，队前客户前往可用柜员

② Deque (Double-Ended Queue) 双端队列

队列操作在任意端都能执行 —— deque的头、尾都能 add & remove，但仍不能访问中间

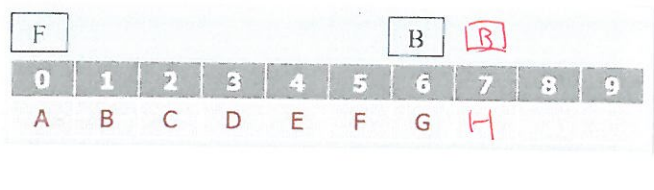
```
public interface DequeInterface<T>   //双端队列接口
{
  public void addToFront(T newEntry);   //队前加入
  public void addToBack(T newEntry);    //队尾加入

  public T removeFront();
  public T removeBack();   //移除

  public T getFront();
  public T getBack();   //查看

  public boolean isEmpty();   //是否空

  public void clear();   //重设
}
```

> 除常规功能外，能移除后返回队头
例：买了优先邮筒，出队后回到队头

？以上两者如何实现
- Queue：一头加入，一头移除
- Dueue：两头都能加入 & 移除

- 使 queue & deque 的 front 位于数组 index 0 的位置
- 随元素增加，使 queue & deque 的 back 位于 back index

因此，在back的操作简单，在front的操作难（要移动其它所有元素）



⇓ 对于queue，我们可以 { 在前面加，在后面删 / 在后面加，在前面删 }

在前端操作时，需要移动数据 to 前端位置定为index 0.

队列实现 & 链表

Lecture 5

另一个ADT：Bag

啥都能放，但放的物品必须同质（在同一继承路径）

```
public interface BagInterface<T>
{
    //接口描绘包对象的操作
    public int getCurrentSize();
    //获取包中当前条目数，返回int数
    public boolean isEmpty();
    //查看是否空包，返回T,F
    public boolean add(T newEntry);
    //添加啥条目  成功返回T  添加的条目为newEntry
    public T remove();
    //删除未指定条目
    删成返回删除项，不成返null
    public boolean remove(T anEntry);
    //删除给定条目的另一次出现  成功返回T  删除对象
    public void clear();
    //移除包中所有条目
    public int getFrequencyOf(T anEntry);
    //计算条目在包中出现的次数  返回次数  条目
    public boolean contains(T anEntry);
    //查询是否包含给定条目  包含返回True  对象
    public T[] toArray();
    //检查包中所有条目
    返回一个含有所有条目的数组 (不常用)
}
```

接口不含的 —— 任何具体细化的数据（接口仅指定行为）

—— 任何方法的实现

例. public boolean add(T newEntry)

考虑 — 正常情况下操作的目的/效果？

— 可能发生什么异常，如何处理？

1° normal case的判断：precondition & postcondition

方法执行前ADT的状态、方法执行后ADT的状态 → 比较判断

以newEntry为例

Pre：包处于有n个项目的有效状态。

Post：包处于有n+1个项目的有效状态。

此时通过add()后确定包内有newEntry

⇓ 用于验证方法正确性