> 由 KiOii (*_EM_Cpper*)整理。 (KiOii (*_EM_Cpper*) makes this note.)

Inheritance

In this chapter

- **5.1 Classes, Superclasses,and Subclasses** (*类，超类，子类*)
- **5.2 *Object:* The Cosmic Superclass** (*Object 是宇宙超类 万类起源*)
- **5.3 Generic Array Lists** (*通用组件之数组列表*)
- **5.4 Object Wrappers and Autoboxing** (*对象包装器和自动装箱器*)
- **5.5 Methods with a Variable Number of Parameters** (*形参数量可变的方法*)
- **5.6 Enumeration Classes** (*枚举类*)
- **5.7 Reflection** (*反射*)
- **5.8 Design Hint for Inheritance** (*关于继承设计方面的提示*)

> The idea behind inheritance is that you can create new classes that are built on existing classes

> - inherit methods
> - add new methods and fields

> ***reflection***
>
> - **find out more about** classes and their properties in a **running** program
> - insterest to **tool builders**

5.1 Classes, Superclasses, and Subclasses

Manager and Employee

> Every manager is an employee

- *"is-a" :* inheritance

5.1.1 Defining Subclasses

- *keyword:* **extends**
- *such as:*

```
public class Manager extends Employee
{
    //...
}
/**
*   Employee : superclass, base class, parent class
*   Manager  : subclass
*/
```

> 和C++的继承对比来看，Java的继承都是public方式的
>
> 一般我们用subclass和superclass来描述

- *wraning:* place the most general methods in the superclass

> superclass设计为具备常用方法
>
> subclass设计更加专门化

5.1.2 Overriding Methods

> Some of the superclass method are not appropiate for the subclass.

- *define:* supply a new method to override the supperclass method
- *use superclass method:* **super**

> 1、子类是不能直接访问超类字段的
>
> 2、由于是override，因此方法名是一样的，因此，我们需要用关键字super来访问超类方法

- *such as:*

```
public class Manager extends Employee
{
    public double getSalary()
    {
        double baseSalary = super.getSalary(); //
        return baseSalary + bous;
    }
}
```

> super和this并不一样，this是reference to object （object variable），而super只是一个关键字，让我们使用superclass方法，它是不能够赋值给其他object variable的
>
> C++使用基类和域操作：：来调用基类成员函数

5.1.3 Subclass Constructors

```
public Manager(String name,double salary,int year,int day)
{
    super(name,salary,year,month,day); // 必须放第一行
    bous = 0;
}
```

> 如果没有指定super去调用superclass的构造函数，则会默认调用superclass默认的构造函数

- *wraning:* **this** and **super**
  - this: **two meanings**
    - call another constructor in the same class

      ```
      this( arguments )
      ```

    - reference to object
  - super: **two meanings**
    - call superclass constructor

      ```
      super( arguments )
      ```

    - call superclass methods

      ```
      super.setSalary( arguments )
      ```

example

```
Manager boss = new Manager("Carl Cracker",80000,1987,12,15);
Employ staff = boss;
System.out.println(staff.getName() + " " + staff.getSalary);
// Carl Cracker 85000.0
```

> The **declared type** of staff is Employee,but the **actual type** of the object thar staff refer to can be either Employee or Manager.
>
> 当指向superclass时，执行supperclass的method
>
> 当指向subclass时，执行subclass的method
>
> 这称之为多态性

ploymorphism 站在对象引用角度 定义多态

> 多态性

- *define:* object variable can refer to multiple actual type

dynamic binding 站在方法调用角度 定义多态

- *define:* automatically selecting the appropriate method at runtime

  > 运行时自动选择合适的方法
  >
  > 在C++中，想实现动态绑定则需要virtual关键字，而Java中，virtual是默认的，想剔除，需要使用final关键字

5.1.4 Inheritance Hierarchies

> 继承的层次结构
>
> 我们可以由Employee扩展出Manager，Secretary，programmer
>
> 可以由Manager扩展出Executive等等
>
> Java不支持C++的多继承机制，但是有interfaces（Section 6）

5.1.5 Ploymorphism

> The "is-a" rule states that every object of the subclass is an object of the superclass
>
> 反之则不然

```
Employee e;
e = new Employee(...);
e = new Manager(...);
```

- *wraning:* In the Java,object variables are ploymorphic

  > 在Java语言中，所有object variables都是多态类型

- *wraning:* you cannot assign a superclass reference to a subclass variable

5.1.6 Understanding Method Calls

x.flag(args)

> 假设现在有这么一个调用 x.flag(args)

1. Looks at the declared type of the object and the method name

   > 从当前类中找出所有名字为f的方法，从supclass类中找出所有可以被访问的名字为f的方法
   >
   > private的superclass方法不可访问

2. determines the types of the arguments that are supplied in the method call

   > 现在，编译器已经知道了 the name and parameter types of the method that need to be called.
   >
   > 注意：如果在subclass中具有和superclass中同签名的方法则重写superclass中的那个方法
   >
   > 进行重写，需要保存返回类型的兼容性，也就是说，你可以使用subclass作为返回值来替代superclass作为返回值

3. private，static，final等方法，constructor等

   > 这种是静态绑定 （static binding）（无法动态绑定）

   depends on the actual type of the implicit parameter

   > 这种是动态绑定，在运行时，通过implicit parameter（object variable）来识别

4. JVM 为每个类预先计算出一个方法表，列出所有的方法签名和要调用的实际方法。

   > 本类的方法和基类的方法 这样的话，识别出implicit parameter就能到该对象的method list中寻找
   >
   > method list中存储了签名(method signatures)

- *wraning:* If you override a method，the subclass method must be a least as visible as the superclass method.

  > 通常，superclass的方法是public的，则subclass重写的方法也必须是public
  >
  > 返回类型不一样的话也必须兼容
  >
  > public Super get() {...}
  >
  > public Sub get(){...}

5.1.7 Preventing Inheritance:Final Classes and Methods

> 防止继承

final class

- *define:* Classes that cannot be extends

final method

- *define:* subclass can not override

  > 我们知道，方法是可以被继承重写的
  >
  > 但是，一旦我们使用了final这个表示常量的关键字，方法名字也被限定了重写。
  >
  > 但是，方法被重载是可以的，因为重载是在本类，而final是针对subclass作出的限定

- *wraning:* All method in a final class are automatically final

  > final类中的所有方法都被设置为final，即，所有方法都不允许被重写

```
public final class A extends B
{
    //...
}
```

> A 不能被继承

> final 字段不可被修改（之前讲过）

inline

- method is not overridden,and it is short

  > 一旦满足这两个条件，编译器就会处理为内联
  >
  > 例如：当进行e.getName()时，用e.name来直接访问
  >
  > 但是如果被重写，则无法inline，因为编译器不知道重写的代码做了什么
  >
  > 在Java中，inline or no inline取决于聪明的编译器

5.1.8 Casting

- use **instanceof** : find out wether a cast will succeed

```
if(x instanceof C)
{
    xx = (C)x;
}
// if x is null, also return false
// x : sub
// C : super
```

- *wraning:* 使用类型转换的警告

  1. You can cast only within an inheritance hierarchy
  2. Use **instanceof** to check before casting from a superclass to subclass

> 进行Casting时，最好采取捕获异常的方式 （ClassCastException）
>
> 最好是不进行Casting，考虑类的设计缺陷

5.1.9 Abstract Classes

> 随着类继承层次的提升，我们可以将super类变得更加抽象

- *use:* **abstract**
  - *wraning:* A class with one or more abstract methods must itselft be declared abstract

    > 如果类拥有abstract method，则类也必须是abstract
  - *such as:*

```
public abstract class Person
{
    ...
    // no implementation required
    public abstract String getDescription();

}
```

- *wraning:* abstract class can have fields and concrete methods

```
public abstract class Person
{
    private String name;
    public Person(String name)
    {
        this.name = name;
    }
    public abstract String getDescription();
    public String getName()
    {
        return name;
    }
}
```

> 始终将公共的fields and methods（不管是不是abstract） 放入 superclass（不管是不是abstract） 中

- *wraning:* two choice in extend an abstract class
  - leave some or all of the abstract methods underfined，subclass must tag abstract

    > 保留abstract methods 但是！subclass必须标记为abstract
  - define all abstract methods

    > 定义abstract methods，subclass不再是abstract
- *wraning:* Abstact classes **cannot be instantiated**

  > 虽然不能实例化对象，但是我们可以定义object variable refer to nonabstract subclass

5.1.10 Protected Access

- *use:* **protected** （subclass can access that in the same package ）
  - *define:* use protected in superclass, subclass can access privated fields or methods of superclass directly

    > protected method 比 protected field更有意义
    >
    > protected field某种意义上失去了OOP精神
    >
    > protected method means that the subclass can use protectd method of supercalss correctly
    >
    > 也就是说，只有subclass可以正确的使用superclass的方法时，该方法的访问权限设置为protectd，例如：Object class提供的的clone method

Access Modifiers

- Visible to the class only : private

  > 只在当前类内可见

- Visible to the world : public

  > 无条件可见

- Visible to the package and all subclasses : protected

  > 在当前包内的subclass可见

- Visible to the package (nothing)

  > 不使用访问修饰符，默认在当前包被可见

5.2 Object : The Cosmic Superclass

- *define:* The ultimate ancestor
- *wraning:* You never have to write : `public class Employee extends Object {}`

  > 默认继承了 Object

- *wraning:* You can use a variable of type Object to refer to object of any type

  > `Object obj = new Employee("Hello",35000);`

5.2.1 The equals Method --- Object method

- *define:* whether two object reference are identical

  > equals 在Object class 实现，被用来判定object reference是否相同
  >
  > 注意，Object实现的equals返回true当且仅当object reference相同（位置相同）
  >
  > 其他类可以重写Object的equals，例如String重写了equals

- *override equals method:*

```
public class Employeee
{
    ...
    public boolean equals(Object otherObject)
    {
        if(this == otherObject)return true;
        if(otherObject == null)return false;
        if(getClass() != otherObject.getClass()) return false;

        Employee other = (Employee)otherObject;
        return Objects.equals(name,other.name) && salary == other.salary &&
    Objects.equals(hireDay,other.hireDay);
    }
}
```

  > getClass return the class of the object

> Objects.equals（a,b）和a.equals(b)的差别：
>
> 前者中，a，b同时为null则返回true，equals则不然（没有这一判定）
>
> Objects.equals是static method，在java.util package中
>
> 一般我们可以使用Objects.equals,除非null和null不能看成相同

- *wraning:* When you define the equals metgid for a subclass,first call equals on the superclass

> 由于在subclass重写，所以，优先级高于superclass中的equals（Object也是superclass）

```
public class Manager extends Employee
{
    ...
    poblic boolen equals(Object otherObject)
    {
        if(!super.equals(otherObject))return false;
        Manager other = (Manager)otherObject;
        return bonus = other.bonus;
    }
}
```

> 问题一?
>
> super.equals做了什么
>
> 问题二?
>
> getClass在 subclass和superclass中使用有区别吗
>
> 答：super.getClass()和this.getClass()得到一样，继承就是"is-a"

5.2.2 Equality Testing and Inheritance

perfect equals method

1. explicit parameter------**otherObject**
2. check for identity-------`if(this == otherObject)return true;`
3. check for null----------`if(otherObject == null)return false;`
4. *two ways:*
    1. `if(getClass() != otherObject.getClass())return false;`
    2. `if(!(otherObject instanceof ClassName))return false;`

```
public class Test
{
    public static void testInstanceof(Object x)
    {
        System.out.println("x instanceof Parent:  "+(x instanceof Parent));
        System.out.println("x instanceof Child:  "+(x instanceof Child));
        System.out.println("x getClass Parent:  "+(x.getClass() == Parent.class));
        System.out.println("x getClass Child:  "+(x.getClass() == Child.class));

    }
```

```
    public static void main(String[] args) {
        testInstanceof(new Parent());
        System.out.println("--------------------------");
        testInstanceof(new Child());
    }
}
class Parent {

}
class Child extends Parent {

}
/*
输出:
x instanceof Parent:  true
x instanceof Child:  false
x getClass Parent:  true
x getClass Child:  false
--------------------------
x instanceof Parent:  true
x instanceof Child:  true
x getClass Parent:  false
x getClass Child:  true
*/
```

> 关于getClass、class、instanceof
>
> 在子类内部，this.getClass() 和 super.getClass()是一样的，getClass()不区分继承关系
>
> class 是唯一的，Sub.class和Super.class不一样
>
> instanceof比较特殊，单向关系
>
> sub instanceof Super : true
>
> sup instanceof Sub : false (sup带指Super对象)
>
> 记住，“is-a"

5. Cast otherObject to your class

   `ClassName other = (ClassName)otherObject;`

6. Compare fields: == 用于基本类型、Objects.equals用于类类型

> 如果有使用到Arrays，可以使用Arrays.equals
>
> 如果你在subclass中重写equals，要包括调用super.equals(other)
>
> Objects.equals会调用类的equals而且有适应null、判定null与null相等机制，所以用Objects.equals

- *explicit override:* **@override**
  - *such as:* `@override public boolean equals(Employee e);`

> 在这边，我们指定了@override，因此，我们告诉了编译器
>
> 这个方法是用于重写的，如果没有覆盖任何方法的话，那么编译器就会告诉你，你写错了，并没有重写，而是重新定义了新方法

5.2.3 The hashCode Method --- Object method

- *define:* get the hashCode of object (object's memory address)

  > Object的hashCode实现为根据对象的内存地址计算得到

  - The hashCode method is defined in the Object class

    > 注意Stirng的hashCode重写版本：
    >
    > String s = "OK";
    >
    > String t = new Stirng("OK");
    >
    > s和t的hashCode返回值是一样的，String将hashCode重写为与elements相关

- *wraning:* no hashCode method has been defined for the StringBuilder class

Example

```
public class Employee
{
    @Override
    public int hashCode()
    {
        return 7*name.hashCode() + 11* new Double(salary).hashCode +
13*hireDay.hashCode();
    }
}
```

> 这边有一个特点：处理基本类型时，不能直接调用 salary.hashCode 基本类型不是class type

```
public class Employee
{
    @Override
    public int hash Code()
    {
        retunr 7*Objects.hashCode(name) + 11*Double.hashCode(salary) +
13*Objects.hashCode(hireDay);
    }
}
```

> 这边有其他特点：类类型使用Objects.hashCode 、基本类型使用特定类型(首字母大写 ， 如Double)
>
> Objects.hashCode(s) : return 0 if s is null or s.hashCode() otherwise

```
@Override
public int hashCode()
{
    retrun Objects.hash(name,salary,hireDay);
}
```

> hash是用于combine multiple hash values

- *wraning:* compatible (兼容的)

- If `x.equals(y)` is true, `x.hashCode()` must return the same value as `y.hashCode()`

  > hash value的相等判定是根据equals来判定的，兼容

5.2.4 The toString Method --- Object method

- *define:* returns a string representing the value of this object

  > 如果我们使用对象的引用当值使用，则值就是该字符串，例如输出（System.out.print）

  > 大多数 `toString` 的重写遵守这样一条规则：
  >
  > "ClassName[field1,field2,,,]"

- *such as:*

```
public String toString()
{
    return "Employee[name="+ name + ",salary" + salary + ",hireDay" + hireDay       +"]";

}
```

  > 使用 getClass 简化

```
public String toString()
{
    return getClass().getName() + "[name....
}
```

  > 在继承中使用 super.toString

```
public String toString()
{
    return super.toString + "[bonus...
}
```

  - *wraning:* Instread of writing `x.toString()` ,you can write `""+x`

    > 当编译器看到 + 的操作是连接字符串时，便调用x.toString()

- *wraning:* The Object class defines the toString method to print the class name and the hash code of the object.
  - *such as:* `System.out.println(System.out);`

    > 输出：java.io.PrintStream@2f6684
    >
    > 理由是 System.out没有对toString进行重写

- *wraning:* arrays inherit the toString method for Object
  - *such as:*

```
int[] arr = {1,2,3};
String s = "" + arr;  // arr.toString()
// print: I@1a46e30
```

- *wraning:* static `Arrays.toString` method instread
  - *such as:*

```
String s = Arrays.toString(arr);
// prints: [1,2,3]
```

toString for logging

```
System.out.println("Current Position" + currPostion);
Logger.global.info("Current Position" + currPostion);  // use:Logger class(Chapter7)
```

> 我们最好为每个类都重写 toString method

5.3 Generic Array Lists

- *set the size of an array at tuntime:*

```
int actualSize=...
Employee[] staff = new Employee[actualSize];
```

ArrayList class

> automatically adjusts its capacity

- *define:* generic class with a type parameter

> define own generic class (Chapter 8)

- *such as:* `ArrayList<Employee> staff = new ArrayList<Employee>();`

  > As of Java SE7 : `ArrayList<Employee> staff = new ArrayList<>();`

- **wraning:** before Java SE5.0，no generic class

  > SE5.0之前没有generic class，但是有一个ArrayList类型，没有<>后缀，元素类型是Object

add method

- add new elements
  - *such as:* `staff.add( new Employee("Harry Hacker",... ) );`
  - *wraning:* The array list manager an internal array of object references

  > automatically creates a bigger array and copies all the object reference from the smaller to the bigger array.

ensureCapacity method

- *use:* call the ensureCapacity method before filling the array list
- *such as:* `staff.ensureCapacity(100);`

  > 一次性初始分配100个object variables

constructor

- *intialize the capacity:* `ArrayList<Employee> staff = new ArrayList<>(100);`

size method

- *define:* returns the actual number of elements in the array list
- *such as:* `staff.size();`

  > The same to `a.length` for an arrary a

trimToSize

- *define:* adjust the size of memory block to use exactly as much storage space as is required to hold the current number of elements

  > 一旦你确定了当前大小是最终的大小，那么调用该方法你可以回收剩余的空间，使得空间利用刚刚好

wraning

- Arraylist class is similar to the C++ vector template
- assignmet operator

  > C++ 的vector<**Type**>对象a，b，执行a = b，则创建新的对象a是b的拷贝，元素也拷贝
  >
  > Java的ArrayList<**Type**>对象a,b，执行a=b，a和b都指向相同的对象

5.3.1 Accessing Array List Elements

- *use:* Instead of the pleasant [] syntax,use **`get` and `set`** method
- *such as:* `staff.set(i , harry);`

  > That is equivalent to : `a[i] = harry;` for an array a
  >
  > index : **0 ~ staff.length**

- *wraning:* Do not call `list.set(i,x)` until the size of the array list is larger than i
  - *such as:*

    ```
    ArrayList<Employee> list = new ArrayList<>(100);
    list.set(0,x); // no element 0 yet
    ```

    > 使用set (mutator)进行赋值，使用add进行添加，使用get进行获取(accessor)

toArray method

- *define:* copy the elements into an array
- *such as:*

```
X[] a = new X[list.size()];
list.toArray(a);
```

add method with an index parameter

- *define:* add elements in the middle of an array list
- *use:* `staff.add(n,e);`

  > The elements at locations n and above are shifted up to make room for the new entry

remove method

- *define:* remove an element form the middle of an array list
- *use:* `Employee e = staff.remove(n);`

  > size太大，效率会低，可以换用linked list (Chapter 9)

"for each" loop

```
for (Employee e : staff)
    do something with e
```

5.3.2 Compatibility between Typed and Raw Array Lists

> interoperate with legacy code that does not use type parameters
>
> 与遗留代码的交互

- *such as:*

```
publi class EmployeeDB
{
    public void update(ArrayList list) { ... }
    public ArrayList find(String query) { ... }
}
// you can do
ArrayList<Employee> staff = new ...
employeeDB.update(staff);                // without any casts
```

- *wraning:* compiler check

```
ArrayList<Employee> result = (ArrayList<Employee>) employeeDB.find(query);
```

> 编译器会进行检查并警告，然后将在运行时擦除type parameter，这样ArrayList和ArrayList<**Employee**>一样的运行时

> 如果你认为你的操作是安全的，换句话说你能够控制，那么你可以抑制编译器不去检查

  - *use :* **@SuppressWranings("unchecked")**

```
@SuppressWranings("unchecked")
ArrayList<Employee>  result = (ArrayList<Employee>) employeeDB.find(query);
```

5.4 Object Wrappers and Autoboxing

Wrappers

- *define:* Like `Integer,Double,...`

  > convert primitive type to a object : int to Integer

Wrappers classes look

- *such as:* `Integer,Long,Float,Double,Short,Byte` inherit from the common superclass `Number`
- *wraning:*
  - The wrapper classes are **immutable**

    > 之前讲过，不提供修改本身的方法，而是返回修改后的新对象

  - The wrapper classes are **final**

    > cannot subclass them

Array List of Integers

- `ArrayList<int>` : type parameter cannot be a primitive type

  > 类型参数不能是基本类型

- `ArrayList<Integer>` : OK , Integer is class type,have object reference
  - *such as:* `ArrayList<Integer> list = new ArrayList<>();`
- *wraning:* less effcient than an `int[]` array

  > 因为都被包装起来，所以效率低于 `int[]` ,你可以使用这种包装器用于构建比较小的集合

***add* method**

- *such as:* `list.add(3);`

  > automatically translated to `list.add(Integer.valueOf(3));`

Autoboxing (example :ArrayList <Integer>)

- ***autoboxing:*** `list.add(3) ------> list.add(Integer.valueOf(3))`
- ***unboxed:*** `list.get(i) --------> list.get(i).intValue();`
- *wraning:* Automatic boxing and unboxing even works with arithmetic expressions
  - *such as:* apply the increment operator to a wrapper reference

```
Integer n = 3;
n++;
```

- *wraning:* just one point different ------- (the **==** operator)

  > 我们知道，== 比较的是 object reference (location)
  >
  > 因此：这样一般是失败的
  >
  > Integer a = 1000;
  >
  > Integer b = 1000;
  >
  > if(a == b) ...

- *wraning:* call the `equalss` method when **comparing wrapper objects**

> boxing and unboxing 是编译器提供的，而不是由虚拟机提供
>
> 换句话是，编译器插入了必要的字节码，虚拟机执行额外的字节码

Some subtleties

- `null`

```
Integer n = null;
System.out.println(2 * n); // Throw NullPointerException
```

- mix `Integer` and `Double` in a conditional expression

```
Integer n = 1;
Double x = 2.0;
System.out.println(true ? n : x); // prints:  1.0
```

> 1、unboxing n
>
> 2、promoted to `double`
>
> 3、boxing into `Double`

parseInt method

- *define:* convert a string to an integer
- *use:* `int x =  Integer.parseInt(s);`

CAUTION

> 是否可以解决在method中修改外部primitive variable的问题呢?
>
> 答案是不行的:
>
> public static void triple(Integer x) // won't work

> {
>
> // 修改x
>
> }
>
> 理由：Wrapper is immutable

- *use* `holder` *types:* defined in the `org.omg.CORBA` package : `IntHolder,BooleanHolder...`

```
public static void triple(IntHolder x)
{
    x.value = 3 * x.value;
}
```

5.5 Methods with a Variable Number of Parameters

- *define:* provide methods that can be called with a variable number of parameters

  > called " varargs" method

- *such as:* `System.out.printf("%d",n);`
  - *printf method define:*

```
public class PrintStream
{
    public PrintStream printf(String fmt,Object... args)
    {
        return format(fmt,args);
    }
}
```

    - *wraning:* `...`
      - *define:* is a part of the Java code,denotes that the method can receive an arbitrary number of objects

        > 接受任意数量的object
        >
        > 如果提供primitive实参，则进行boxing，变成object
        >
        > 扫描fmt，对于低i个占位符，寻找args[i]

      - Object... the same to Object[]

        > paratemer : Object...
        >
        > argument :Object[]

  - *such as:*

```
System.out.println("%d %s", new Object[] { new Integer(n),"widgets});
```

The meaning of ...

- *define:*
  - parameter: double...
  - argument :double[]

    > parameter :Type...
    >
    > argument :Type[]

- *such as:*

```
public static double max(double... values)
{
    double largest = Double.NEGATIVE_INFINITY;
    for( double v : values)
        if(v > largest)
            largest = v;
    return v;
}
// such as
// argument : new double[] {1.0,2,3,6.5};
```

5.6 Enumeration Classes

> All enumerated types are subclass of the class `Enum`.
>
> `java.lang.Enum<E>`

- such as:

```
public enum Size {SMALL,MEDIUM,LARGE,EXTRA_LARGE};
```

  - *wraning:* The type defined by this declaration is actually a class

    > 你可以添加字段和方法和构造函数
    >
    > 因为 只有SMALL,MEDIUM,LARGE,EXTRA_LARGE四个Size的**实例**，不会去生成新的实例，因此你不需要使用equals，而是简单的使用==就能完成比较了

  - *example:*

```
public enum Size
{
    // four instances,constructor: Size(String)
    SMALL("S"),
    MEDIUM("M"),
    LARGE("L"),
    EXTRA_LARGE("XL");

    private String abbreviation;

    private Size(String abbreviation)
    {
```

```
        this.abbreviation = abbreviation;
    }
    public String getAbbreviation()
    {
        return abbreviation;
    }
}
```

- use
  - override **toString** method : return the name of the enumerated constant

    > enumerated constant : SMALL,MEDIUM,...
    >
    > 因此，`Size.SMALL.toString()` returns `"SMALL"`

  - **valueOf** *method:* the converse of `toString` method

    > `Size s = Enum.valueOf( Size.class, "SMALL");`
    >
    > 引用Size的enumerated constant 为SMALL的实例

  - **values** *method:* returns an array of all values of the enumeration

    > `Size[] values = Size.values();`
    >
    > elements: `Size.SAMLL,Size.MEDIUM,...`

  - **ordinal** *method*: yiedlds the position of an enumerated constant

    > 返回 enumerated constant 在enum中定义的位置，例如
    >
    > `Size.MEDIUM.oradinal()` return 1

- example

```
package enums;
import java.util.*;
public class EnumTest
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter a size:(SMALL,MEDIUM,LARGE,EXTRA_LARGE)");
        String input = in.next().toUpperCase();
        Size size = Enum.valueOf(Size.class,input);
        System.out.println("size = " + size);
        System.out.println("abbreviation = " + size.getAbbreviation());
        if(size == Size.EXTRA_LARGE)
            System.out.println("Good job !");
    }
}
enum Size
{
    SMALL("S"),MEDIUM("M"),LARGE("L"),EXTRA_LARGE("XL");
    private String abbreviation;

    private Size(String abbreviation)
```

```
    {
        this.abbreviation = abbreviation;
    }
    public String getAbbreviation()
    {
        return abbreviation;
    }
}
```

5.7 Reflection

Reflective

- *define:* A program that can analyze the capabilities of classes
- use:
  - Analyze the capabilities of classes at runtime (运行时分析类的功能)
  - Inspect objects at runtime　(运行时检查对象)

    > for example : write a single `toString` method that works for all classes
  - Implement generic array manipulation code (实现泛型数组操作代码)
  - `Method` objects that work just like function pointer in language such as C++ (Method)

5.7.1 The Class Class

runtime type identification

- *wraning:* Java runtime system always maintains it on all objects

  > Java运行时系统维护在object上所谓的运行时类型识别

Class class

- *define:* holds the infomation
- use :

  > getClass 、forName、T.class

  - *such as:*

    ```
    Employee e;
    ...
    Class cl = e.getClass();
    ```

    - ***getClass*** *method of any obejcts:* returns an instance of Class type (define in Object)
  - ***getName*** *method of Class:* returns the name of the class

    - *such as:* `System.out.println(e.getClass().getName() + " " + e.getName());`
    - ***forName*** *method of Class:* get the Class object
        - *such as:*

```
String className = "java.util.Random";
Class cl = Class.forName(className);
```

    - **T.class** :
        - such as:

```
Class cl1 = Random.class;
Class cl2 = int.class;
Calss cl3 = Double[].class;
```

- *wraning:* Class object really decrible a type,which may or may not be a class

> Class对象只是描述了一个类型信息，关于被描述的类型是否是class并不关心

- *wraning:* The Class obejct actually a generic class

> `Employee.class` is of type `Class<Employee>`
>
> 但是一般可以使用 Class 来代替，不用type parameter（Chapter 8）

CAUTION

- `Double[].class.getName()` returns `"[Ljava.lang.Double;"`
- `int[].class.getName()` returns `"[I"`

> 这边输出的奇怪的字符串是历史遗留问题造成的

- use the == operator to compare class objects

> 虚拟机为每个类型管理了一个**唯一的Class 对象实例**，因此可以用 ==
>
> 例如: `if(e.getClass() == Employee.class) ...`

- ***newInstance*** *method:* create a new instance of the same class type
    - *such as:* `e.getClass().newInstance();`

> 创建了和e同类型的对象，然而构造函数是调用的 no-argument constructor
>
> 如果没有该构造函数，则抛出异常

    - such as: combination of `forName` and `newInstance`

```
String s = "java.util.Random";
Object m = Class.forName(s).newInstance();
```

> 如果想要使新实例调用有参数的构造函数，使用Constructor class 中的newInstance方法

> 类比：Class相当于C++中的type_info，getClass相当于C++中的typeid，然而Java的更加强大，能实例化。

5.7.2 A Primer On Catching Exceptions

> 详细的异常处理在 Chapter 7 中介绍

throw an exception

- *when:* an error occurs at runtime

  > 提供一个捕获异常和处理异常的程序使得程序更加灵活
  >
  > 如果没有处理异常的话，程序终止并向控制台输出消息
  >
  > 从而给出异常的类型

two kinds of exceptions

- *define:* `unchecked exceptions` and `checked exception`
- *checked exception:* compiler checks that you provide a handler

  > 编译器会检查你是否提供了处理程序
  >
  > Class.forName是checked exception

- checked exception : compiler will not checks

  > 有些异常编译器是不会检查是否有处理程序的，因此我们需要避免这类异常的出现

use

- *simplest handler implementation:*

```
try
{
    statements that might throw exception
}catch (Exception e)
{
    handler action
}
```

- example

```
try
{
    String name =  ... ;// get class name
    Class cl = Class.forName(name); // might throw exception
    //...
}catch (Exception e)
{
    e.printStackTrace(); // 堆栈追踪 : the method of Trowable class
}
```

> `Throwable` class is the superclass of the `Exception` class

5.7.3 Using Reflection to Analyze the Capabilities of Classes

- *wraning:* we can examine the structure of a class

> 利用反射来分析类的功能是反射机制中最重要的部分，允许我们对类的结构进行检查

Three classes

- *define :* **Field** , **Method**, **Constructor** in the java.lang.reflecti package **describle the fields, methods and constructor of a class**
- ***getName*** *method :* return the name of the item

> 三个类都有这个方法

- **Field** class has ***getType*** *method:* return an object that type is `Class`

> 该方法获取字段的类型描述对象引用，Method和Constructor，两者都有获取参数类型的方法，Method还能获取方法的返回类型

- ***getModifiers*** *method:* returns an integer with various bits turned on and off

> 三个类都有该方法，能够返回 修饰符的状态，比如public或者static有没有等状态
>
> Modifier class能够用于分析该方法的返回值
>
> 例如：`isPublic, isPrivate, isFinal,...`
>
> 你还可以使用`Modifier.toString`打印 修饰符

Class class

- ***getFields*** *method:* return **arrays** of the **public fields**
- ***getMethods*** *method :* return **arrays** of the **public methods**
- ***getConstructors*** *method :* return **arrays** of the **public constructor**

> also includes public member of superclass

- ***getDeclaredFields***、***getDeclaredMethos***、***getDeclaredConstructor*** *methods:* return arrays consisting of all fields,methods,and constructor that are declared in the class

> include private, package, and protected members,but not members of superclass

5.7.4 Using Reflection to Analyze Objects at Runtime

- ***get*** *method in Field class:* return value of field
  - *such as:*

```
Employee harry = new Employee("Hello World",1,1,1,1);
Class cl = harry.getClass();
Field f = getDeclaredField("name");
Object v = f.get(harry); // the value of the name field of the harry object,the String object "Hello World"
```

> 实际上，这段代码是有问题的，一旦 name field is private，那么就会抛出一个 `IllegalAccessException` 异常，get只能获取accessible fields，如果非要访问，只能重写访问权限

  - *wraning:* 如果field 不是 object，而是primitive类型的字段，那么get获取会自动进行boxing，或者如果你想获取例如double字段值，则将get替换为getDouble

- ***setAccessible*** *method:* override access control

> 可以用于 Field，Method，Constructor对象

  - *such as:* `f.setAccessible(true); // now OK to call f.get(harry)`
  - *wraning:* The `setAccessible` method is a method of the `AccessibleObject` class,the `common superclass` (常见超类) of the `Field ,Method,Constructor` classes.

> 这个功能主要用于调试器（debuger）、持久存储器（persistent storage）等等类似机制。
>
> We use it for generic `toString` method later in this seciton

- ***set*** *method* : The call `f.set(obj,value)` sets the field represented by `f` of the object `obj` to the new `value`

5.7.5 Using Reflection to Write Generic Array Code

- *create arrays dynamiclly:*

```
Empoloyee[] a = new Employee[100];
//...
// if array is full
a = Arrays.copyOf(a,2*a.length);
```

> 怎么写出一个通用的方法呢

  - first attempt

```
public static Object[] badCopyOf(Object[] a,int newLength) // not useful
{
    Object[] newArray = new Object[newLength];
    System.arraycopy(a,0,newArray,0,Math.min(a.length,newLength));
    return newArray;
}
```

> Employee[] cast into Object[] : OK
>
> Object[] cast into a Employee[] : ClassCastException

- static ***newInstance*** method of the `Array` class: constructs a new array

```
public static Object goodCopyOf(Object a,int newLength)
{
    // first step
    Class cl = a.getClass();
    if(!cl.isArray()) return null;
    // second step
    Class compent = cl.getCompentType();
    int length = Array.getLength(a);
    // thord  step
    Object newArray = Array.newInstance(compentType,newLnegth);

    System.arraycopy(a,0,newArray,0,Math.min(length,newLength));
    rerurn newArray;
}

// call
int[] a = {1,2,3,4,5}
a = (int[]) goodCopyOf(a,10);
```

- *methods:*
  - **isArray** method of Class class: returns if is arrays type
  - **getComponentType** method of Class class: returns arrays type
  - **newInstance** method of Array class : returns new array
  - **arraycopy** method of System class: copy

> int[] 可以 和Object 转换
>
> 然而 不能和 Object[] 进行转换 (Object[] is an array of obejcts)

5.7.6 Invoking Arbitary Methods

> 调用任意方法，可以使用interface，然而反射机制使得我们可以进行任意调用

invoke method

- *define:* let you call the method that is wrapped in the current `Method` object
- *signature:* `Object invoke(Object obj,Object... args)`
  - *obj :* implicit paramter

> 对于static方法，该参数设置为null

  - *args :* explicit paramter

- *example:*

```
double s = (Double)m2.invoke(harry);
```

> m2 : represents the `getSalary` method of the `Employ` class

- obtain a `Method` object:
  - *getMethod signature:* `Method getMethod(String name,Class... parameterTypes)`
  - example

```
Method m1 = Employee.class.getMethod("getName");
Method m2 = Employee.class.getMethod("raiseSalary",double.class);
```

> 使用interface 和 lambda会做得更好 （Chapter 6）

5.8 Desgin Hints for Inheritance

useful when using inheritance

1. Place common operations and fields in the superclass
2. Don't use protected fields

> 所有在同一个包中的类可以访问到保护字段，不管是不是子类
>
> 但是保护方法是很有用的，留给子类去重新定义（可以被子类调用）

3. Use inheritance to model the "is-a" relationship
4. Don't use Inheritance unless all inherited methods make sense
5. Don't change the expected behavior when you override a method

> 当你重写方法时不要更改预期的行为

6. Use ploymorphism,not type information
7. Don't overuse reflection