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

Interface, Lambda Expression, and Inner Classes

In this chapter

- **6.1 Interfaces** (*接口*)
- **6.2 Examples of Interfaces** (*接口实例*)
- **6.3 Lambda Expression** (*Lambda表达式*)
- **6.4 Inner Classes** (*内部类*)
- **6.5 Proxies** (*代理*)

> Inteface : 指出了 类"能做什么"，没有指出"怎么去做"
>
> 一个类可以 implement 多个接口
>
> lambda expression： 简化代码块表示，优雅地使用回调或者变量行为
>
> inner class： 在其他类中定义，它们的方法可以访问周围类的字段，当你设计协助类的集合时，内部类是有用的
>
> proxie： 对于构造系统级工具非常有用

6.1 Interfaces

6.1.1 The Interface Concept

interface

- define : a set of requirements for the classes

  > 提供给类的一组需求

- example

```
public interface Comparable
{
    int compareTo(Obejct other);
}
```

  > 例如 Array.sort sort an arrray of objects,但是前提是 objects 所属的类必须implement 哪个 Comparable 接口

- *wraning:* 在 Java SE 5.0中，Comparable 接口被增强为泛型

```
public interface Comparable<T>
{
    int compareTo(T other);
}
```

  > 旧版本需要强类型转换

- *wraning:* All methods of an interface are automatically `public`

  > 因此接口中的方法在声明时不需要 使用 public 关键字

- *wraning:* `x.compareTo(y);` 结果和0对比
- *wraning:* interface never have instance fields

  > SE 8之前，你无法在interface中实现方法，只能声明
  >
  > 之后，你可以定义一些特定的方法（static or dufault method）
  >
  > 但是定义的方法不能引用实例字段，因为interface是没有实例字段的

- Supply instance fields and method that operate on them(接口中的方法)

  > 为接口方法提供实现和实例字段是 实现接口的类的任务
  >
  > interface 可以看成没有实例字段的抽象类，但是还是有一些区别的（之后提到 ）

Class implement an interface

- *two steps:*

  1. 声明要实现接口的类

     ```
     class Employee implements Comparable
     ```

  2. 提供接口中所有方法的定义

     > 假设比较薪水

     ```
     public int compareTo(Object otherObject)
     {
         Employee other = (Employee)otherObject;
         return Double.compare(salary,other.salary);
     }
     ```

  > type parameter

  ```
  class Employee implements Comparable<Employee>
  {
      public int compareTo(Employee other)
      {
          return Double.compare(salary,other.salary);
      }
      ...
  }
  ```

  > 为什么需要Comparable接口呢？Java是强类型的，编译器需要检查类型是否具备比较方法
  >
  > 如果知道比较对象是Comparable object，那么就能够确定
  >
  > sort内部的实现比较：
  >
  > **if (((Comparable)a[i]).compareTo(a[j]) > 0)**
  >
  > **{**
  >
  > // ...
  >
  > **}**

- *wraning:* 接口在继承中的一些注意事项

  > superclass实现了 Comparable 接口，那么subclass如果想要自己的Comparable接口实现呢？重写？
  >
  > subclass不能进行下面的措施：

```
public Manager extends  Employee
{
    public int compareTo(Employee other)
    {
        Manager otherManager = (Manager)other; // 当比较对象是employee和manager问题就大了，因此你可以使用 getClass来 抛出ClassCastException异常来报告这种错误
        ...
    }
}
```

6.1.2 Properties of Interfaces

- Interface are not class

  > `x = new Comparable(...); // ERROR`
  >
  > `Comparable x = new Employee(...); // OK`
  >
  > 指向实例了接口的类

- check wether implements or not

  > `if (anObject instanceof Comparable){...}`
  >
  > 检查anObject是否实现了接口

- *wraning:* 接口是可以被继承出新接口的，我们可以为接口定义常量，这些常量将会被实现了接口的类自动继承。

- wraning:class can implement multiple interfaces

  > 类可以实现多个接口
  >
  > 例如：Cloneable和Comparable
  >
  > **class Employee implements Cloneable, Comparable**

6.1.3 Interface and Abstact Classes

> 为什么不使用抽象类而要引入接口的概念呢，因为可以继承多个接口，而继承只有单继承

6.1.4 Static Methods

> 从SE 8开始，允许我们在接口中添加静态方法

- example

```
public interface Path
{
    public static Path get(String first, String... more)
    {
        return FileSystems.getDefault().getPath(first,more);
    }
}
```

6.1.5 Default Methods

- *default keyword:* You can supply a default implementation of any interface method

  > 提供默认方法，但是每个实现接口的类都会覆盖，那么用处在哪呢?

  - example

```
public interface MouseListener
{
    default void mouseClicked(MouseEvent event){}
    default void mousePressed(MouseEvent event){}
    default void mouseRelease(MouseEvent event){}
    default void mouseEntered(MouseEvent event){}
    default void mouseExited(MouseEvent event){}
}
```

  > 用处在于你不被全部都实现，有时我们只是想要实现一部分!

- an default method can call other methods

```
public interface Collection
{
    int size();
    default boolean isEmpty()
    {
        return size() == 0;
    }
    ...
}
```

  > 默认的方法可以调用其他方法！上面的这个例子：将size和isEmpty联合起来，因为isEmpty依赖于size，所以实现是固定的，因此可以在接口类中就进行默认实现，类就没有必要去覆盖了，默认方法还有其他一些优点，接口演化....

6.1.6 Resolving Default Method Conflicts

> 解决默认方法冲突

- 如果多个接口和超类都有同样的方法，那么会发生什么呢?

  > 1、如果superclass提供了方法1，而接口也有默认方法1，那么接口的默认方法1被覆盖
  >
  > supperclass最大
  >
  > 2、如果superclass实现的接口中已经有方法1了，而实现的其他接口中也有方法1，那么你必须重写新接口中的方法（不管是不是默认方法都会冲突）

6.2 Examples of Interfaces

6.2.1 Interfaces and Callbacks

callback pattern

- *define:* sepcify the action that should occurt whenerver a particular event happens

  > 指定了当特定事件发生时应该执行的操作

  - **Timer** class

    > javax.swing package中
    >
    > 当你构造一个构造器时，设置时间间隔并告诉他在时间间隔结束后应该做什么

    - 怎么告诉计时器去做什么呢?

      > 面向过程的语言可能使用回调函数来传递给计时器使用，Java是面向对象的语言，采用面向对象方法：pass an object of some class
      >
      > 计时器对象会自动调用该对象实现的接口，比传递函数灵活，因为对象拥有额外的信息。
      >
      > 好了，现在我们需要知道定时器提供的接口是什么?
      >
      > ActionListener
      >
      > 计时器对象使用该接口中 方法：`void actionPerformed(ActionEvent event);`

    - example

```
/**
*   每10秒发出一条消息然后 beep
*/
class TimerPrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("At the tone,the time is " + new Date());
        Toolkit.getDefaultToolkit().beep();
    }
}

// 传递给计时器去使用
ActionListener listener = new TimePrinter();
Timer t = new Timer(1000,listener);

// 开始记时
t.start();
```

6.2.2 The Comparator Interface

- compare strings by length

  > 根据字符串长度来比较，我们不能实现compareTo in two ways
  >
  > Arrays.sort 还有其他形式，即参数是comparator接口对象

```
public interface Comparator<T>
```

```
{
    int compare(T first, T end);
}

class LengthComparator implements Comparator<String>
{
    public int compare(String first,String second)
    {
        return first.length() - second.length();
    }
}

Comparator<String> comp = new LenthComparator();
if(comp.compare(words[i],words[j]) > 0)...


//
String[] friends = {"Peter","paul","Mary"};
Arrays.sort(friends, new LengthComparator());
```

> 我们注意到实现接口的类的对象有的情况下是匿名的（lambda使得简化）
>
> 在于是用实现接口的对象来调用方法，还是被比较对象自己调用接口实现方法

6.2.3 Object Cloning

```
Employee original = new Employee(...);
// 1
Employee copy = original;
copy.raiseSalary(10); // original changed
// 2
Employee copy orignal.clone();
copy.raiseSalary(10); // original unchanged
```

- **clone** method : the default cloning operation is "shallow"

  > clone实现的拷贝默认是浅层拷贝，就跟参数传参拷贝一样，这意味着如果字段是引用的话也照样只拷贝引用，这样可能会出现问题。并不是真正意义上的拷贝。

  - **deep copy**

    > 因此有时我们需要进行深拷贝，重新定义clone

- *wraning:* 有时我们避免使用clone，而是实现Cloneable接口

  > 注意！clone是protected method，但是我们重写接口要public

  - Clonable is `maker interfaces` (标记接口)

    > 标记接口没有方法，唯一的目的就是允许在类型中使用：
    >
    > `if (obj intanceof Cloneable) ...`
    >
    > 最好不要在我们的程序中使用标记接口

- *wraning:* 即使默认的clone是足够的，你也需要实现Cloneable接口，让clone变成public，调用super.clone

```
class Employee implements Clonable
{
    ...
    public Employee clone() throws CloneNotSupportedException  // public
    {
        // call Object.clone
        Employee cloed = (Employee)super.clone();
        // clone mutable fields
        cloned.hireDay = (Date)hireDay.clone();
        return cloned;
    }

}
```

> 我们是描述throws CloneNotSupportedException好呢，还是这样写好？=>

```
public Employee clone()
{
    try
    {
        Employee cloned = ...
        ...
    }
    catch(CloneNotSupportedException e)
    {
        return null;
    }
}
```

- *wraning:* 所有的数组都有非protected clone

```
int[] arr = {1,2,3};
int[] copy = arr.clone();
```

6.3 Lambda Expressions

6.3.1 Why Lambdas?

Lambda expression

- *define:* a block of code that you can pass arround so it can be executed later,once or multiple times

```
class Worker implements ActionLitener
{
    public void actionPerformed(ActionEvent event)
    {
        // do some work
    }
}
```

> 当你想要重复执行此代码时，你构造一个Worker实例，然后将该实例交给计时器对象。

- *wraning:* 在没有lambda之前，提供代码块在Java中是不容易的，需要显式由对象来接管

6.3.2 The Syntax of Lambda Expressions

- 从排序实例开始思考

```
// 比较字符串
first.length() - second.length()
```

> Java是强类型语言，first和second是什么类型?

```
(String first,String second)
    ->first.length() - second.length()
// 于是，这样，一个简单的lambda表达式就有了
// 注意传参的规范 (the specification of any variables that must be passed to the code)
```

> **An expression with parameter variable has been called a lambda expression**

- one form of lambda expression：**parameters，->, expression** (形参列表结合表达式)

  > 如果表达式多则写入 {} 中

```
(String first,String second)->
{
    if(first.length() < second.length()) return -1;
    else if(first.length() > second.lenth()) return 1;
    else return 0;
}
```

> 进一步，如果lambda没有形参列表，则仍然需要提供：***()***

```
() ->
{
    for(int i = 100;i >= 0; i--)
        System.out.println(i);
}
```

> 再然后，如果lambda表达式的参数类型可以被推断出来，那么就可以省略

```
Comparator<String> comp
    = (first,second) -> first.length() - second.length();
// 由于赋值给 Comparator<String> 因此 可以推断两个都是String （后面会介绍）
```

> 特殊点，如果方法带有唯一一个可推断的形参则可以忽略括号

```
ActionListener listener
 = event -> System.out.println("The time is " + new Date());
```

> 返回类型一直都是自动推倒的，因此不需要指定返回类型

```
(String first,String second) -> first.lenth() - second.lenth()
```

> 注意返回值需要每个情况都返回

```
(int x) -> {if (x >= 0) return 1;} // error
```

6.3.3 Functional Interfaces

> Java实现了很多封装了代码块的接口，lambda和这些接口兼容
>
> 例如：ActionListener or Comparator

- *wraning:* 只要接口对象提供了单一的抽象方法，就可以使用lambda

  > 这样的接口称之为 : functional interface

  - wraning: 为什么函数接口必须有单一抽象方法？接口中所有方法都抽象不行吗？

```
Arrays.sort( words,
    (fist,second) -> first.length() - second.length() );
```

> Arrays.sort第二个形参接受一个Comparator<**String**>对象，该对象引用实现了Comparator<**String**>的类对象,函数调用compare函数执行lambda表达式，比传统的内部类有效多了
>
> 注意：这些对象和类完全依赖于实现

- conversion example:

```
Timer t = new Timer(1000, event ->
{
    System.out.println("At the tone,time is " + new Date());
    Toolkit.getDefaultToolKit().beep();
});
```

> Lambda表达式就是拥有单一方法的接口的方法的实现（方法体，我们将lambda看成函数）
>
> ” Lambda表达式转换到函数接口 “
>
> Object不是函数接口，你甚至不能将表达式赋值给Object

- generic functional interfaces : define in `java.util.function`

```
BiFunction<String,String,Integer> comp
= (frist,second) -> first.length() - second.length();
// 但是这样并不会对sort有帮助
```

> Arrays.sort的第二个形参并不是要BiFunction，而是要Comparator
>
> 因此我们需要确定lambda使用的目的然后提供特定的功能接口

- 还有一个常用的接口 : Predicate

```
public interface Predicate<T>
{
    boolean test(T t);
    //.. default and static methods
}
```

> Predicate 内的test用于判定成立条件
>
> 我们可以使用lambda来实现

```
list.removeIf(
    e->e = null;
)
// 形参: Predicate<...> // type argument is the element type
```

6.3.4 Method References

> 有时，我们已经有方法想传递给其他代码

```
Timer t = new Timer(1000,System.out::println);
//System.out::println等价于 x->System.out::println(x)
```

- *wraning:* `System.out::println` is a method reference
- example : sort strings regardless of letter case

```
Arrays.sort(strings,String::compareToIgnoreCase);
```

Three principal cases

- obejct : : instanceMethod

> 等价相同参数的lambda

- Class : : staticMethod

> 等价相同参数的lambda

- Class : : instanceMethod

> 等价第一个参数为对象本身，其余参数为方法参数的lambda
>
> 例如：`String::compareToIgnoreCase`等价于
>
> `(x,y)-> x.compareToIgnoreCase(y);`

- *wraning:* you can capture **this** 、**super**

```
super::instanceMethod
this::equals
```

  - example

```
class Greeter
{
    public void greet()
    {
        System.out.println("Hello World");
    }
}
class TimedGreeter extends Greeter
{
    public void greet()
    {
        Timer r = new Timer(1000,super::greet);
        t.start();
    }
}
```

6.3.5 Constructor References

> 构造函数的引用和方法的引用是一样的，区别在于构造函数的名字：new

- example : `Person::new` is a reference to a Person constructor

> 至于哪个构造函数取决于上下文，方法的引用重载则取决于接口中方法的参数

- You can form constructor references with array types : such as `int[] :: new`

> `int[] :: new` : constructor reference with one paramter：length
>
> 同价lambda : x -> new int[x];

  - example

```
Person[] people = stream.toArray(Person[] :: new);
```

> stream.toArray() : return an Object array
>
> `Object[] people = stream.toArray();`
>
> stream.toArray(paramter)
>
> `Person[] people = stream.toArray(Person[] :: new);`

6.3.6 Variable Scope

- 在lambda中获取封闭方法或者类中的变量

```
public static void repeatMessage(String text,int delay)
{
    ActionListener listener = event ->
    {
      System.out.println(text);   // text variable
      //...
    };
    new Timer(delay,listener).start();
}
```

> 返回后text变量是如何保存不变的?

> 被lambda所捕获，我们知道，lambda可以转换为单一方法的对象，从而将free variable拷贝到对象的实例变量中去

  - wraning: you can only reference variable whose value doesn't change

```
public static void countDown(int start,int delay)
{
    ActionListener listener = event ->
    {
        start--; // 不能改变捕获的变量 (Can't mutate captured variable)
        System.out.println(start);
    };
}
```

  - wraning: 不能引用正在改变的变量

```
public static void repeat(String text,int count)
{
    for(int i = 0;i <= count;++i)
    {
        System.out.println( i + ":" + text);
        // 不能引用正在改变的变量i
    }
    new Timer(1000,listrener).start();
}
```

- *wraning:* 能够被捕获的变量是 effectively final

> 必须是有效的最终变量 由于text始终指向同一个String object，因此可以被捕获

- *wraning:* 名字作用域相同

```
Path first = Paths.get(...);
Comparator<String> comp =
    (first,second)->first.length() - second.length();
// error:first重定义
```

- wraning: use **this** in a lambda expression

```
public class Application
{
    public void init()
    {
        ActionListener listener = event ->
        {
          System.out.println(this.toString());  // 调用Appication实例的toString
          //...
        };
    }
}
```

6.3.7 Processing Lambda Expressions

> 让我们编写可以处理lambda表达式的方法!

- deferred execution

  > 使用lambda的要点是延迟执行，需要延迟执行的代码包装在lambda中

- 为什么要延迟执行代码?
  - 单独线程中执行的代码
  - 多次运行代码
  - 在算法中正确地运行代码（例如排序的比较操作）
  - 必要时运行代码
- example : 多次执行

`repeat(10, ()-> System.out.println("Hello World"));`

> 10：count
>
> println：action
>
> 为了使得repeat第二个参数接受一个lambda，我们需要一个functional interface
>
> 这边我们可以使用 Runable :

```
public static void repeat(int n,Runable action)
{
    for(int i = 0; i < n;++i)
        action.run(); // lambda expression在这边开始执行
}
//我们甚至可以自己写个functional interface
public interface IntConsumer
{
    void accept(int value);
}
public static void repeat(int n,IntConsumer action)
{
    for(int i = 0;i < n;++i)
        action.accept(i);
}
repeat(10,i->System.out.println("Countdown:" + (9-i)));
```

Functional Interface	Parameter Types	Return Type	Acstract Method Name	Description	Other Methods
`Runable`	none	void	run	Runs a action without aruguments or return value	-
`Supplier<T>`	none	T	get	Supplies a value of type T	-
`Consumer<T>`	T	void	accpet	Consumes a value of type T	andThen
`BiConsumer<T>`	T, U	void	accept	Consumes values of types T and U	andThen
`Function<T,R>`	T	R	apply	A function wit argument of type T	compose, andthen, identity
`BiFunction<T,U,R>`	T,U	R	apply	A function with arguments of types T and U	andThen
`BinaryOperator<T>`	T,T	T	apply	A binary operator on the type T	andThen, maxBy, minBy
`UnaryOperator<T>`	T	T	apply	A unary operator on the type T	compose, andThen, identity
`Predicate<T>`	T	boolean	test	A boolean-valued function	and, or, nagate, isEqual
`BiPredicate<T>`	T,U	boolean	test	A boolean-valued function with two arguments	and, or, negate

> **Functional Interface for Primitive Types**

Functional Interface	Parameter Types	Return Type	Abstract Method name		
`BooleanSupplier`	none	boolean	`getAsBoolean`		
P `Supplier`	none	p	`getAs` P		
P `Consumer`	p	void	`accept`		
`Obj` P `Consumer<T>`	T,p	void	`accept`		
P `Functional<T>`	p	T	`apply`		
P `To` Q `Function`	p	q	`apply`		
`To` P `Function<T>`	T	P	`applyAs` Q		
`To` P `Bifunctional<T,U>`	T,U	p	`applyAs` P		
P `UnaryOperator`	p	p	`applyAs` P		
P `BinaryOperator`	p,p	p	`applyAs` P		
P `Predicate`	p	boolean	`test`		

> **p,q** is **int,log,double;**
>
> **P,Q** is **Int,Long,Double;**

- wraning: 如果你定义了functional interface，可以使用**@FunctionalInterface**来注释标记

6.3.8 More About Comparators

- **has static methods for creating comparators:**

  > 这些方法大多数都是返回lambda表达式（也就是functional interface，这边也就是Comparators）

  - `Array.sort.(people,Comparator.comparing(Person::getName));`
    - **comparing method（键提取器 ket extractor）:** 接受第一个参数作为比较对象"key "（键）构造根据"键提取器"的延迟比较接口实现

    > 第二个参数需要一个Comparator接口去调用compare方法
    >
    > 而comparing就是去创建了这么一个Comparator
    >
    > comparing第一个参数为Function接口，于是，Person::getName的实现变成了Function接口中抽象的实现，返回一个lambda表达式传递给sort，表达式被接口引用延迟执行比较操作，而比较操作的对象就是getName获取的字符串
    >
    > 伪代码：`return (c1,c2)-> Function.apply(c1).compareTo(Function.apply(c2));`

  - `Arrays.sort(people,Comparator.comparing(Persion::getLastName).thenComparing(Persion::getFirstName));`
    - 如果last name相同，比较first name

- `Arrays.sort(people,Comparator.comparing(Persion::getName,(s,t)->Integer.compare(s.length(),t.length())));`
  - comparing method 变体：第二个参数用于处理 “键“（第一个参数）
- `Array.sort(people,Comparator.comparingInt(p->p.getName().length());`
  - **comparingInt method** : 构造比较int长度的延迟接口实现
- `Arrays.sort(people,Comparator.comparing(Person::getMiddleName,Comparator.nullFirst(Comparator.nuturalOrder()));`
  - 当getMiddleName返回null怎么办？ 需要第二个参数构造默认比较顺序
  - **nullFirst**需要comparator，nuturalOrder()构造任何类的正常顺序，reverOrder则是相反顺序
  - **naturalOrder().reversed()** 和 **reverseOrder()** 一样

> Functional interface更加深刻的认识：包装方法，延迟调用，方法参数需要和接口方法参数匹配
>
> 常用`Function<T,R>`接口：接受参数类型T，返回类型R

6.4 Inner Classes

- *define :* a class is defined inside another class
- 内部类特点
  - 内部类中的方法可以access定义它们的作用域中的数据，包括其它私有数据
  - 内部类可以隐藏在同一个包中的其他类中
  - 当你不想写大量代码进行回调时，anonymous inner class是非常方便的

    > 不过，现在我们已经有Lambda了

6.4.1 Use of an Inner Class to Access Object State

> An inner class method gets to access both its own data and those of outer object creating it

- wraning: an object of an inner class always gets an implicit reference to the object that created it
  - the outer class reference is set in the constructor

    > 编译器修改内部类构造函数，为其添加一个外部类引用参数

  - outer is not a Java keyword.

    > 这边没有关键字使用该隐式参数

- example

```
class TalkingClock
{
    private int interval;
    ...
    public void start()
    {
        ActionListener listener = new TimePrinter(); // 这边构造函数默认传入this

        Timer t = new Timer(interval,listener);
```

```
        t.start();
    }
    public class TimerPrinter implements ActionListener // 内部类定义
    {
        public void actionPerformed(ActionEvent event)
        {
            ...
        }
    }
}
```

6.4.2 Special Synax Rules for Inner Classes

- *expression :* `OuterClass.this`

```
public void actionPerformed(ActionEvent event)
{
    ...
    if(TalkingClock.this.beep) Toolkit.getDefaultToolkit().beep();
}
```

- explicitly constructor: `outerObject.new InnerClass(constructor parameter)`

```
ActionListener listener = this.new TimerPrinter();
```

> 通过这个显示语法，你可以让OutClass的其他对象传递给内部类

```
TalkingClock jb = new TalkingClock(1000,true);
TalkingClock.TimerPrinter listrener = jb.new TimerPrinter();
```

  - 外部 refer to inner class : `OutClass.InnerClass`

- wraning:
  - `static field` in inner class : must be `final`
  - `static method` in inner class : cannot (没有限制，但是静态方法只能访问该静态字段和方法，复杂度超过好处，所以一般不使用)

6.4.3 Are Inner Classes Useful? Actually Necessary? Secure?

- 内部类是编译器的现象，虚拟机并不知情，内部类被转换为特别名字的类

> 例如：TimePrinter是TalkingClock内部类，被转换为:`TalkingClock$TimePrinter.class`

  - `javap -private ClassName` 来查看
    - `javap - private innerClass.TalkingClock\$TimePrinter`

```
public class TalkingClock$TimePrinter
{
    public TalkingClock$TimePrinter(TalkingClock);
    public void actionPerformed(java.awt.event.ActionEvent);
    final TalkingClok this$0; // 编译器生成的对外部类对象的引用
}
```

```
class TalkingClock
{
    private int interval;
    private boolean beep;
    public TalkingClock(int,boolean);
    static boolean access$0(TalkingClock);//编译器生成的一个访问方法，返回beep
    public void start();
}
```

> **if (beep)就等价于if(TalkingClock.access$0(outer))**
>
> 这都是编译器干的

6.4.4 Local Inner Classes

```
public void start()
{
    class TimePrinter implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            System.out.println("At the tone,the time is" + new Date());
            if(beep)...
        }
    }
    ActionListener listener = new TimePrinter();
    Timer t = new Timer(interval,listener);
    t.start();
}
```

> Local class 不能声明访问说明符（public、private、protected），因为只能在该方法内被访问

6.4.5 Accessing Variables form Outer Methods

- local class 不仅可以访问 field of their outer class,还可以访问局部effectively final变量

> 外部传里面需要是不变的，object variable一般是不变的，primitive variable一般可变，我们可以使用这样的策略来补救：`int[] count = new int[1];` 大小为1的数组
>
> 改变count[0]

6.4.6 Anonymous Inner Classes

- *define :* 当你只是需要该类的一个对象，你不需要给类名字

```
public void start(...)
{
    ActionLisener listener = new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            ...
        }
    }
}
```

> 这边的语法是很神秘的: 创建了一个实现了ActionLiestener接口的类对象

- ***syntax :***

```
new SuperType(constrcutor parameters)
    {
        inner class methods and data
    }
```

  - two SuperType : interface or class

    > 如果是接口，则匿名内部类 是 接口的实现类
    >
    > 如果是类，则匿名内部类是 superclass的subclass

- *wraning:* An anonymous inner class cannot have constructors

> 因为构造函数名字需要和类名相同，但是匿名了，所以也就不能去写构造函数
>
> 构造功能则转变为`SuperClass (constructor parameters)`
>
> **当SuperType是interface时**，constrcutor parameters为空的，即不能有参数

```
new InterfaceType()
{
    methods and datas
}
```

```
从一般内部类------>局部内部类----（只需要一个对象）---->匿名内部类---（只需要一个方法）-->lambda表达式

public void start (int interval,boolean beep)
{
    Timer t = new Timer(interval,event->
                        {
                           System.out.println("At the tone,time:" + new Date());
                            if(beep)Toolkit.getDefaultToolkit().beep();
                        });
    t.start();
}
```

匿名类应用

```
ArrayList<String> firends = new ArrayList<>();
friends,add("Harry");
friends.add("Tony");
invite(friends);

invite(new ArrayList<String>()
       {// 匿名类块
           { // 初始化块
               add("harry");
               add("Tony");
           }
       })
```

```
class C
{
    public static void method()
    {
        new Object(){}.getClass().getEnclosingClass();
        // new Object(){} 获取当前匿名类
        // getClass() 获取当前匿名类的Class类型
        // getEnclosingClass() 获取包裹着当前匿名类的外部类的类类型
        // 也就是 "Class C"
    }
}
```

> Inner Class 通过this.new构造 实现对outerclass的访问 不可直接编写构造函数（编译器会自己来搞）
>
> Anonymous Inner Class 丢掉类名 对象唯一
>
> - 引用局部变量 参数 对象字段都会创建final实例字段存储
> - 引用静态字段则无影响
>
> Static Inner Class 抑制 this.new 构造，可进行构造函数编写

6.4.7 Static Inner Classes

> 有时，我们使用Inner Class只是想将其隐藏在另外一个类当中
>
> 不需要使用内部类来引用外部类对象
>
> 我们可以使用static inner class

- example

```
/*
问题：从一个数组中找出最大值和最小值
1、我们可能会先想到：先找最大，再找最小
2、考虑到效率问题，我们觉得遍历一次比较两次找出最大和最小
3、由此，我们需要一个方法来执行遍历查找并返回结果
4、结果是两个值，由此我们可能会去定义一个Pair class来管理这两个值
5、考虑到其他人也会编写存储其他值(比如String值)的Pair，我们将Pair类定义在需要的地方
6、Pair只存储结果，并不引用外部类对象字段值，于是采用static inner class
```

```
*/

class ArrayAlg
{
    public static class Pair
    {
        ...
    }
}
```

- *wraning:* **only inner classes can be declared static**

> 因此构造内部类对象不需要 对象引用
>
> 意味着我们可以在static方法中构造内部类对象(static method可是没有隐式对象引用参数的)

```
public static Pair minmax(double[] d)
{
    ...
    return new Pair(min,max);
}
```

> 当inner class不需要access outer class时，我们将使用static inner class
>
> 有些程序员使用术语：nested class （嵌套类） 来描述 static inner class

> interface中的inner class默认是public static inner class
>
> static inner 可以拥有static methods 和 static fields，不同于inner class
>
> inner class不能定义static fields、static method

6.5 Proxies

- *define :* Create new classes that implement a given set of interface at runtime

> 运行时创建接口的实现类

6.5.1 When to Use Proxies

- construct an object of a class that implements one or more interfaces

> 这些接口的确切性质你在编译时可能不知道

- proxy class implements the interface that you specify

> proxy class 可以实现你指定的接口

- method
  - All methods requirtd by the interfaces

- All methods defined int the Object class (toString, equal, an so on)

> 但是你不能在运行时定义新的代码，而是要提供一个 invocation handler

- invocation handler : an object of any class that implements the **InvocationHandler** interface
  - **invoke** method of the interface :
    - `Object invoke(Object proxy,Method method,Object[]args);`

      > 每当在proxy object上面调用方法时，invoke 方法就被调用
      >
      > method with args

6.5.2 Creating Proxy Objects

- **newProxyInstance** of **Proxy** class
  - parameter
    - A class loader : 后续(Chapter 9)介绍,使用 null 默认加载器
    - An array of Class objects: for each interface to be implemented
    - An invocation handler : 调用处理
- purpose of proxy
  - Routing method calls to remote servers
  - Associating user interface events with actions in a running program
  - Tracing method calls for debugging purpose
- example

```
package proxy;

import java.lang.reflect.*;
import java.util.*;

public class ProxyTest
{
    public static void main(String... args)
    {
        Integer value = 10;
        InvocationHandler handler = new TraceHandler(value);// 处理程序
        Obejct proxy = Proxy.newProxyInstance(null,new Class[]{Comparable.class},
handler);// 代理Comparable接口实现
        // .. 对 value进行比较操作，由于比较接口被代理，因而转而调用handler实现的invoke
    }
}

class TraceHandler implements InvocationHandler
{
    private Object target;

    public TraceHandler(Object t)
    {
        target = t;
    }
    public Object invoke(Object proxy,Method m,Object... args)
```

```
    throws Trowable
    {
        // Todo : ...代理的处理程序
        return m.invoke();
    }
}
```

> 这边的应用是 处理Integer中的Comparator接口
>
> 当Comparator中的方法被调用时，被代理，调用处理程序 invoke

6.5.3 Properties of Proxy Classes

- only one field : invocation handler
- wraning : 所有proxy class override the `toString、equals、hashCode` methods of the `Object` class, 它们都会 call invoke method on the invocation handler

  > clone 和 getClass没有被重定义

- **getProxyClass** method : `Class proxyClass = Proxy.getProxyClass(null,interfaces);`