从而以多种方式实现 SimpleListInterface<T>.

```
public class SimpleAListT<T> implements SimpleListInterface<T>
{  // data and method bodies in here }

public class SimpleLListT<T> implements SimpleListInterface<T>
{  // data and method bodies in here }
```

接入接口，实现操作

此时，变量类型与方法调用是相同的

```
SimpleListInterface<String> List1 = new SimpleAListT<String>();
SimpleListInterface<String> List2 = new SimpleLListT<String>();
List1.add(new String("Hello"));
List2.add(new String("Hello"));
```

Lecture 20

图形界面

- 初代 Java 用 AWT (在各系统上不统一)
- Java 1.2 用 Swing ☆
- JavaFx，可与 Swing 一起使用

Swing
- JFrame — objects, the windows in graphical applications (在其中放置图象与部件)
- JApplets — 不怎用
- JLabels — 简单组件，放在 JFrame 里，可设置字体类型、大小、彩色

与用户先交互，需添加按钮.


```
import java.awt.*;
import javax.swing.*;
public class ex22a
{
   public static void main(String [] args)
   {
      JFrame theWindow = new JFrame("Example 22a");
      theWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JLabel theMessage = new JLabel("Hello Graphics!");
      theMessage.setFont(new Font("Times", Font.ITALIC, 60));
      theMessage.setForeground(Color.RED);
      theMessage.setOpaque(true);
      theMessage.setBackground(Color.BLUE);
      theWindow.add(theMessage);

      //theWindow.setSize(600, 200);
      theWindow.pack();
      theWindow.setVisible(true);
      System.out.println("Done with main!!");
   }
}
```

创建 JFrame 对象
使其关闭时退出程序
创建 JLabel
改变字体颜色，使其不透明
把 JLabel 加进 JFrame
设置 JFrame
使其可见
设置输入一之前，现在程序跑到我们关闭窗口.

JButton — 除了显示文字，还能响应鼠标的反馈

— 在JButton内点击，生成一个ActionEvent对象作为回应，该对象自动传给ActionListener对象

实际上是方法actionPerformed()的接口

Swing：点击，生成ActionEvent，ActionListener内的actionPerformed()方法被执行

个事件驱动编程

Idea：① 某事件被触发

② 触发对象生成一个事件对象

③ 事件对象传递给事件侦听对象

④ 事件侦听对象的方法被执行处理事件

ActionListener Object

public void
actionPerformed()
{
// code
}

[事件处理操作链到正确事件生成器很重要，否则②生成但无响应]

例

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class ex22b
{
    public static void main(String [] args)
    {
        JFrame theWindow = new JFrame("Example 22b");//创建JFrame对象
        theWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//使其关闭时退出程序
        JButton theButton = new JButton("Change Color");//创建...
        theButton.setFont(new Font("Serif", Font.ITALIC + Font.BOLD, 90));
        ActionListener listen = new MyListener();//创建MyListener对象
        theButton.addActionListener(listen);
        theWindow.add(theButton);
        theWindow.pack();
        theWindow.setVisible(true);
    }
}

class MyListener implements ActionListener
{
    static Color [] theColors = {Color.RED, Color.BLUE, Color.CYAN, Color.YELLOW,
                                 Color.ORANGE, Color.MAGENTA, Color.GREEN};
    private int index = 0;//定义index=0
    public void actionPerformed(ActionEvent e)//每次触发事件都执行actionPerformed方法
    {
        JButton theEventer = (JButton) e.getSource();
        theEventer.setForeground(theColors[index]);
        index = (index + 1) % theColors.length;
        theEventer.setBackground(theColors[index]);
    }
}
```

注意！

//将JButton链到Listener

MyListener是一个实现ActionListener的分离的类

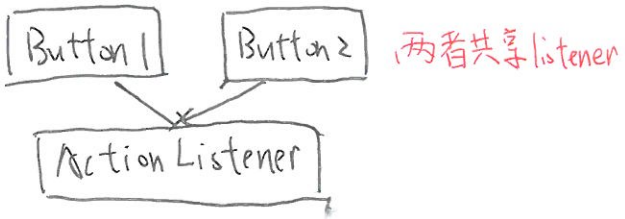
//颜色设置

对象状态每次更新

当需要多个组件时使用layout manager

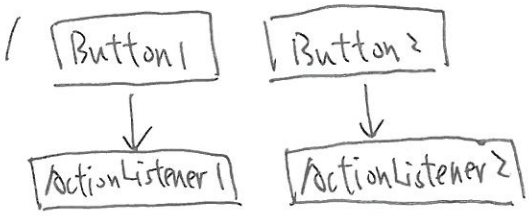
- FlowLayout（从左到右 从上到下）
- GridLayout（将组件放在大小相等的二维网格）

若多个组件生成事件，需要管理listener



两者共享listener

listener一旦改变，一次影响2个object的引用



分离的listener

不一定互相影响，(可以交互导致影响)

多个组件需要彼此交互

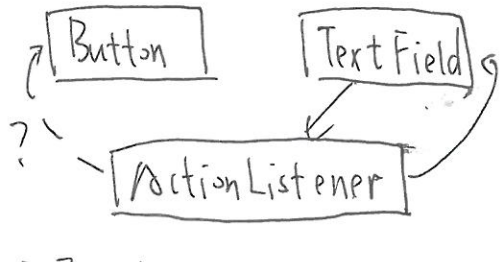
Listener需要访问其它组件 (允许其访问所有涉及组件)

JTextField — 可以输入文本的组件

- 点击Enter后，生成事件，与 JButton相同
- 用它进行输入以改变JLabel或JButton的内容

例，JFrame中有一个JButton和一个JTextField

↘ 输入文本，用于JButton "label"部分



ActionListener可以通过TextField从形参到方法

↘ 始终可以返回生成事件的对象

需要访问Button相应改变文本如何实现？ — 将所有对象封装在一个类中

此时，所有的引用都是instance variables within a single object

— ActionListener自定义也要封装在其中，作为inner class

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class ex22d
{
   private JFrame theWindow;
   private JButton theButton;
   private JTextField theText;
   private MyListener theListener;
   public ex22d()
```

↓

instance variables

初始化函数中的一切

```
   {
     theWindow = new JFrame("Example 22d");
     theWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     theButton = new JButton("Change Color");
     theButton.setFont(new Font("Serif", Font.ITALIC + Font.BOLD, 72));
     theListener = new MyListener();
     theButton.addActionListener(theListener);
     theText = new JTextField();
     theText.setFont(new Font("Serif", Font.BOLD, 72));
     theText.addActionListener(theListener);
     theWindow.setLayout(new GridLayout(2,1));
     theWindow.add(theButton);
     theWindow.add(theText);

     theWindow.pack();
     theWindow.setVisible(true);
   }
```

```
class MyListener implements ActionListener  // inner class
{
      final Color [] theColors = {Color.RED, Color.BLUE, Color.CYAN,
                                  Color.ORANGE, Color.MAGENTA};
      int index = 0;
      public void actionPerformed(ActionEvent e)
      {
          Component theEventer = (Component) e.getSource();
          if (theEventer == theButton)
          {
                theEventer.setForeground(theColors[index]);
                index = (index + 1) % theColors.length;
                theEventer.setBackground(theColors[index]);
          }
          else if (theEventer == theText)
          {
                theButton.setText(theText.getText());  //访问所有对象

                theText.setText("");
                theWindow.pack();
          }
      }
}
public static void main(String [] args)  //主程序组成
{
      new ex22d();
}
}
```

Lecture 21

Exception：程序执行时发生错误、异常、意外事件

在较新的语言中，我们能将异常处理与"main line" code 分离开来。

异常是被抛出然后捕获的对象

异常处理：使用 try-catch blocks

```
try
{
    // code that will normally execute //正常代码
}
catch (ExceptionType1 e)
{   // code to "handle" this exception //处理异常
}
catch (ExceptionType2 e)
{   // code to "handle" this exception //处理异常
}

... // can have many catches

finally
{   // code to "clean up" before leaving try block
    // (optional)
}
```

1° 如果一切顺利（没有异常发生），

执行try中的代码，然后执行（可选）finally代码

（省略 catch blocks）

2° 如果try block出现异常

执行立即跳出try block（try block未完成执行）

在catch block寻找异常处理程序。

2.1 catch block与异常匹配

对其处理并执行 catch block

然后（可选）执行 finally block

2.2 未找到与发生事件匹配的异常处理

异常不会被处理

然后（可选）执行finally block

异常被传播

```
try
{
    // statements before exception are executed
    // EXCEPTION OCCURS
    // statements after exception are skipped
}
catch (ExceptionType1 e)
{   // code to "handle" this exception
}
catch (ExceptionType2 e)
{   // code to "handle" this exception
}
```

只要finally block存在，无论如何它都会执行

（无异，异处，异未处）

- 如果异常被处理

执行在处理 try/catch block 后立即恢复，且不回到抛出点

termination model of exception handling

(与恢复 model 相反，恢复 model 从发生异常处恢复执行)

- 如果异常被传播

动态异常传播

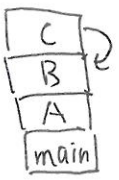
找不到处理程序 / 崩溃，报错

\ GUI 依然执行，但不一致 。

```
public void A()
{
    B();
    C();
}

public void B()
{
    C();
}

public void C()
{
    // EXCEPTION
}

main()
{
    A();
}
```

main calls A; A calls B; B calls C

C 出现异常，传播到了 B



再考虑 A：A 直接调 C

C 出现异常，传播到了 A

以上就是动态异常传播

- 相同方法的相同异常被处理，在不同的地方
- 处理取决于 call chain sequence

检查异常 VS 非检查异常

1° checked exceptions

- 若一个方法有能处理异常，方法必须声明它抛出了异常

Done in throws clause 于方法头

- 若检查异常没有被处理也没有被声明抛出

导致编译错误

- checked exceptions 包含 IOException 和 InterruptedException (及其子类)

2° unchecked exceptions

- 方法有能否处理非检查异常？

如果异常没被处理，不必说 throw (与 checked exceptions 不同)

- 编译器不关注异常是否处理
- 未处理且发生，将传播
- 包含 NumberFormatE~, NullPointerE~, ArrayIndexOutOfBoundsE~ ...

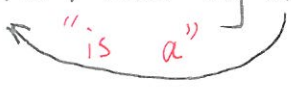
Catching Exceptions 捕获异常

异常处理器按照其在代码中列出的顺序检查 (从上至下)

- 捕获一个异常的父类将捕获子类异常对象("is a")

catch (IOException e) > catch FileNotFoundException

catch (Exception e) > catch any exception



- 异常处理器无法执行,将发生编译错误 → 例 catch(Exception e1){}. 编译错误,IOE在第一行就处理了
  catch(IOException e2){}

以从具体到一般的列顺序列出异常

GUI以多线程运行

如果抛出异常的线程没有被处理,线程将终止 — 主线程,直接程序崩溃
该线程运行GUI中一个事件,其他线程继续,GUI也执行

这不意味正确运行.

Exception classes也能充当继承

Lecture 22

Java方法可以调用其它任何Java公共方法

- main()本身是一个方法,从主方法中可以调用其它方法
- 应当有种方法可以自我调用—RECURSIVE CALL(递归调用)

从数学上开发其思想

阶乘 $N! = N*(N-1)!$ 递归调用的大小(N-1)比原始调用(N)小

对于N!

$$N! = \begin{cases} N*(N-1)! & N>0 \\ N! = 1 & N=0 \end{cases}$$

另例:计算另一个整数的整数幂

$$M^N = \begin{cases} M*M^{N-1} & ,N>0 \text{ 递} \\ M^N = 1 & ,N=0 \text{ 基} \end{cases}$$

first call made is last call to complete

递归算法的三个原则:

1°必须存在某种递归情况,算法能"自我调用"
2°必须存在某种 base case,此时不发生递归调用
3°递归调用最终必须引发 base case
(reduce the problem size)

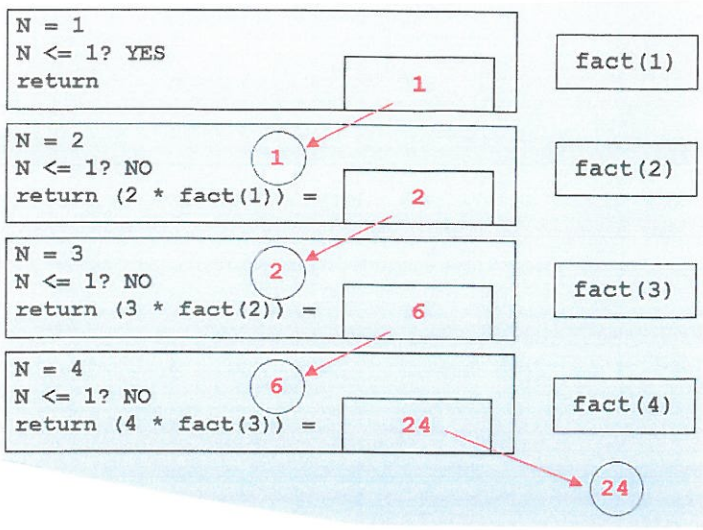
计算机实现递归

```
public static int fact(int N)
{
   if (N <= 1)
               return 1;
   else
               return (N * fact(N-1));
}
```

调用完成后,return才执行

递归实际工作

- 每次调用方法时，都为其分配一个 activation record (AR)
- 每个新的 AR 都放在 run-time stack 的顶部
- 方法终止时，AR 从运行栈的顶部删除
- 因此，第一个方法进栈的 AR 最后一个被删除



备：递归也可以通过 loops 实现

阶乘、幂都能实现，但 iteration 迭代最好

递归比同等的迭代需要更多参数

↘用于测试 base case

递归是接口的一部分会出现问题，因为接口指定方法头

（添加非递归方法解决，一满足头，再调用递归）

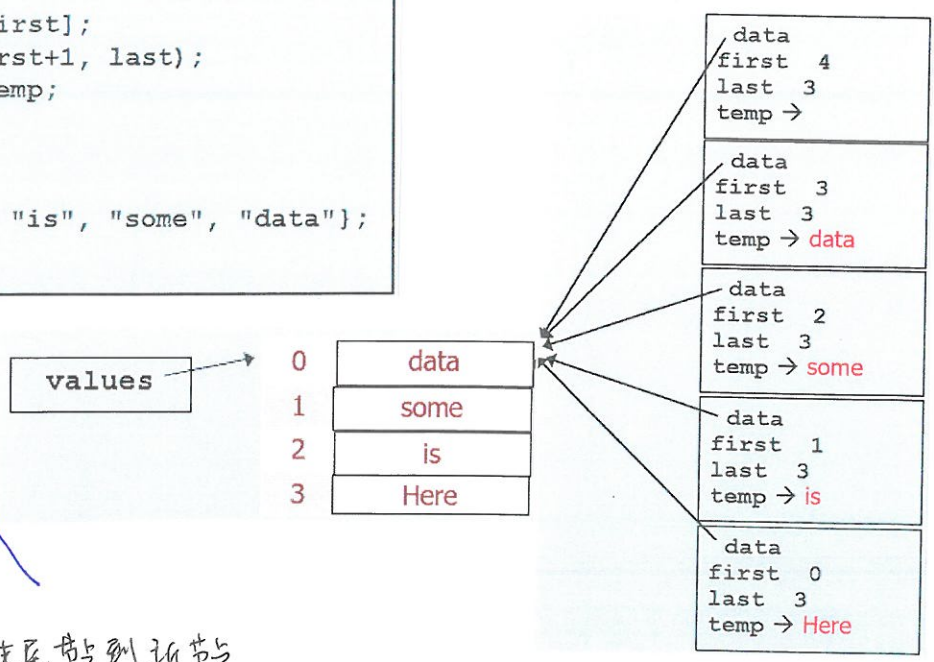
例：反转对象数组的方法

```
public static void reverse(String [] data)
{
    rec_reverse(data, 0, data.length-1);
}

public static void rec_reverse(String [] data,
                               int first, int last)
{
    if (first <= last)
    {
       String temp = data[first];
       rec_reverse(data, first+1, last);
       data[last-first] = temp;
    }
}

String [] values = {"Here", "is", "some", "data"};
reverse(values);
```

→方法头一般这样

但为实现递归，需追踪数组始末

→将额外参数添进递归方法



public boolean add(T val)

从在列表末尾添加一个新值

一建立一个包含该值以及当前末尾节点的新节点

```
public boolean add(T val)
{
        Node curr = front;
        if (curr == null)
        {
                front = new Node(val);
                size++;
                return true;
        }
        else
                return addAtEnd(curr, val);
}
```

// 和同public，以满足接口

// special case，此时不必递归

↓列表是空的，在front加入新节点

// 递归添加到列表末尾，注意对于current Node的引用参数.

```
private boolean addAtEnd(Node curr, T val)
{
        if (curr.next == null)
        {
                curr.next = new Node(val);
                size++;
                return true;
        }
        else
                return addAtEnd(curr.next, val);
}
```

// private method

// Base case - 当前节点为最后节点

// 在后面添加新节点.

// 递归添加到下一个节点