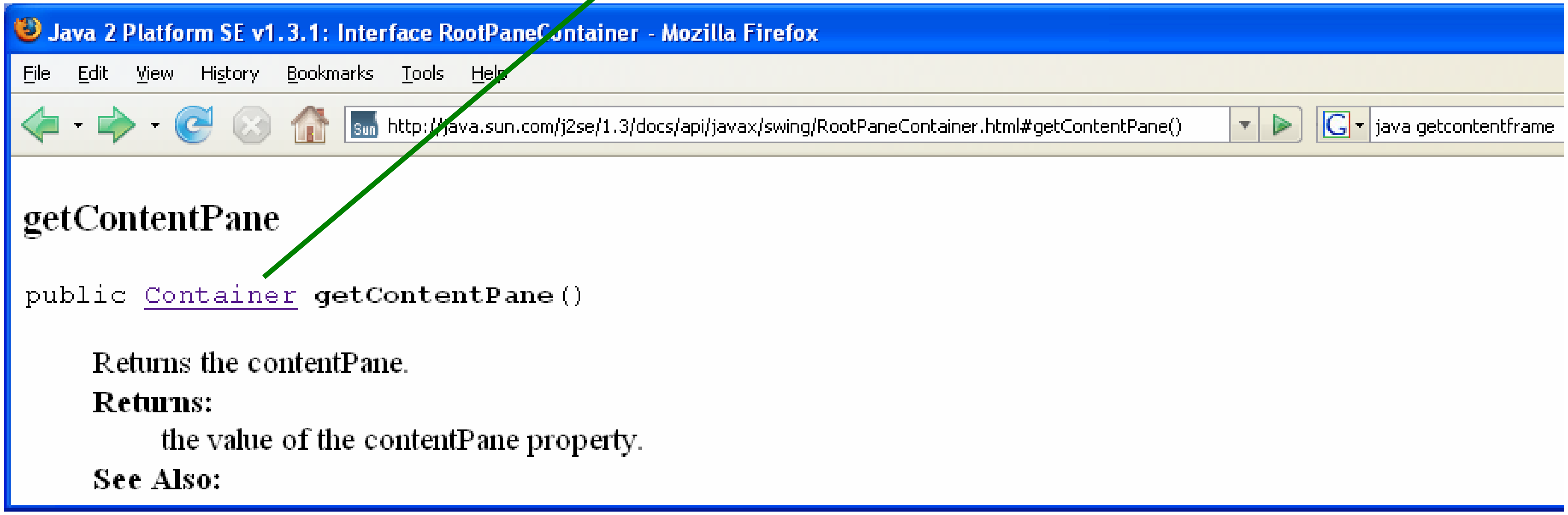
ECE 462
Object-Oriented Programming using C++ and Java

Lecture 5

Yung-Hsiang Lu
yunglu@purdue.edu

Review of Lab 2

- Every derived class of JComponent supports **paintComponent**. In general, this function should be overridden in the derived class (as we did in lab 2)
- getContentPane() $\Rightarrow$ return a container

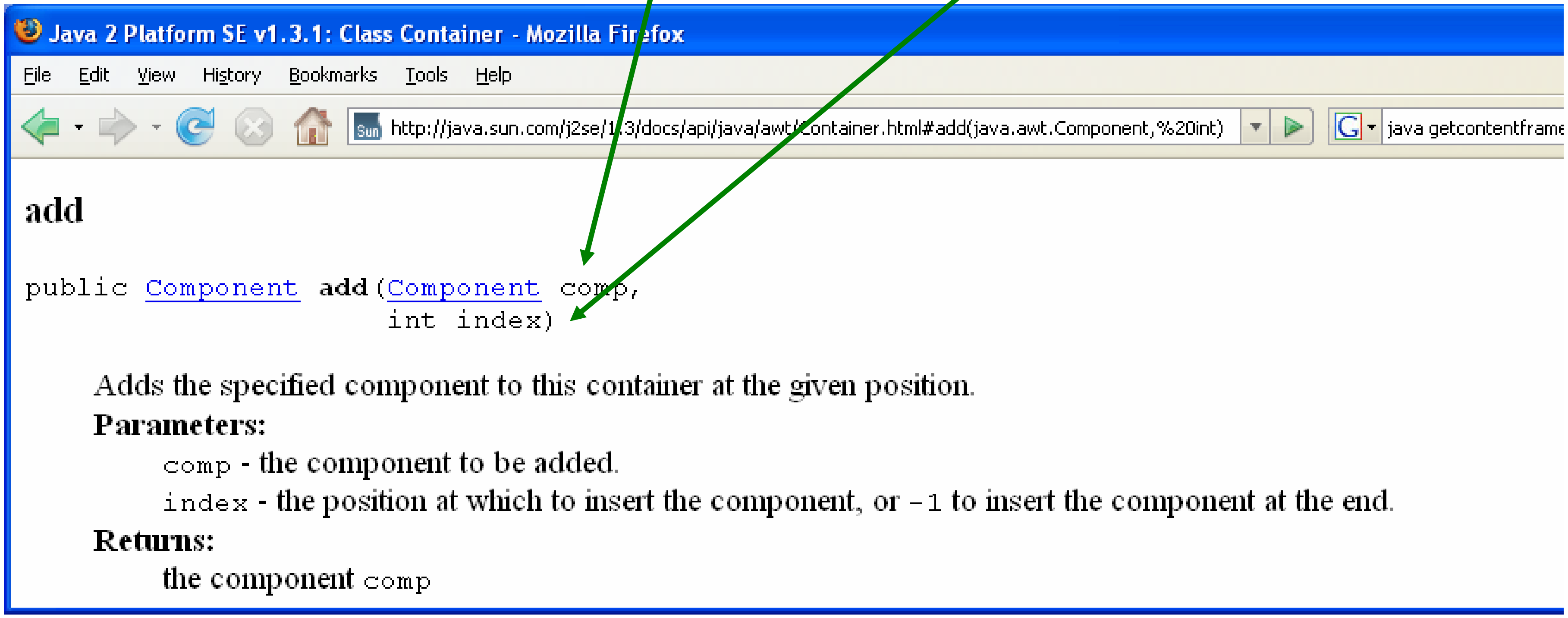

Java 2 Platform SE v1.3.1: Interface RootPaneContainer - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://java.sun.com/j2se/1.3/docs/api/javax/swing/RootPaneContainer.html#getContentPane()
java getcontentframe

getContentPane

```
public Container getContentPane()
```

Returns the contentPane.
Returns:
the value of the contentPane property.
See Also:

- getContentPane().add(jDisplay, BorderLayout.SOUTH);


Java 2 Platform SE v1.3.1: Class Container - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://java.sun.com/j2se/1.3/docs/api/java/awt/Container.html#add(java.awt.Component,%20int)
java getcontentframe

add

```
public Component add(Component comp,
                     int index)
```

Adds the specified component to this container at the given position.

Parameters:

`comp` - the component to be added.

`index` - the position at which to insert the component, or `-1` to insert the component at the end.

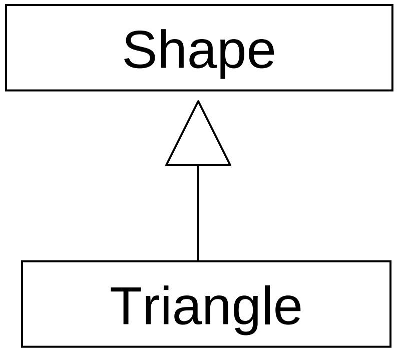
Returns:

the component `comp`

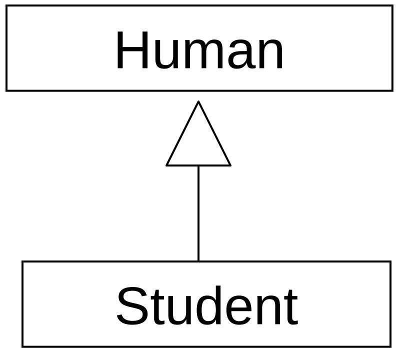
- The function requires a Component object as the first parameter. jDisplay is an object from a class that is derived from Component; hence, jDisplay is a valid parameter

⇒ if an object of class X is expected as a parameter of a function, an object from a class derived from X is also a valid parameter.

```
func1(Shape sobj) { ... }
Shape obj1 = new Shape();
Triangle obj2 = new Triangle();
func1(obj1);      // valid
func1(obj2);      // valid
```



```
func2(Triangle tobj) { ... }
Shape obj1 = new Shape();
Triangle obj2 = new Triangle();
func2(obj1);      // invalid
func2(obj2);      // valid
```

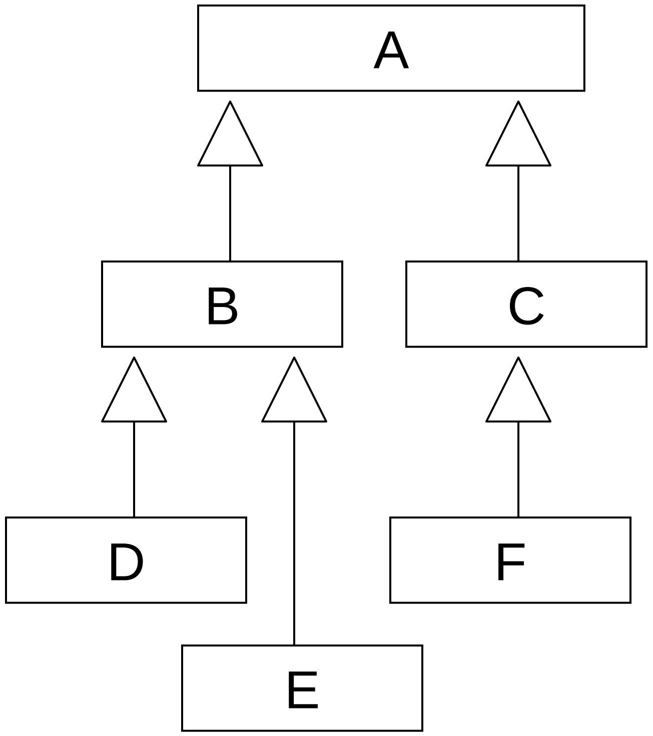


- A Student object is also a Human object. If a Human object is expected, it is valid to provide a Student object.
- Not all Human objects are Student objects. If a Student object is expected, it is **invalid** to provide a Human object.

- Whenever a component object needs to be repainted, the **paintComponent(Graphics gfx)** function is called.
- If it overridden in a derived class (for example, ShapeDisplay), the one in the derived class is called.
- This is an example of polymorphism.
  - Functions of the same name, the same return type, and the same parameter list.
  - Functions are implemented in derived classes.
- In a window, there may be many objects of different classes: buttons, menus, textfields, panels ... These objects are added to the window (e.g. in a list).
- When the window needs to be redrawn (e.g. restore from minimized), the window “asks” (i.e. send a message) to each object and the object decides how to redraw itself.

```
class Container {
    public void add (A aobj);
}
B bobj = new B ...
C cobj = new C ...
D dobj = new D ...
E eobj = new E ...
F fobj = new F ...
cont.add(bobj);
cont.add(cobj);
cont.add(dobj);
cont.add(eobj);
cont.add(fobj);
for each object x in the container object cont {
    x.func(...);     // will call func of the respective class
}
```

```
class A {
   public void func(...)
}
```

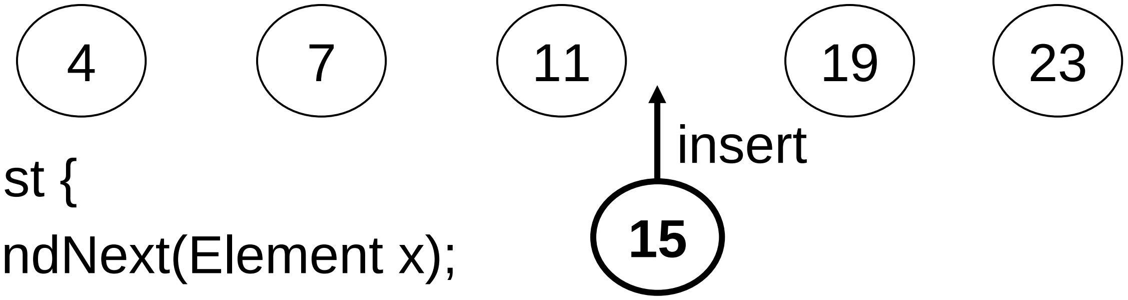


Inheritance is a Contract

- A derived class has **all** the properties of the base class, including attributes and methods.
- A virtual function overrides the **implementation** of the method but the derived class still **provides** the method (respond to the message)
- If one class does not have **all** the properties of another class, the first should **not** be a derived class.
- If you are not sure, do **not** create a derived class.

Class SortedList

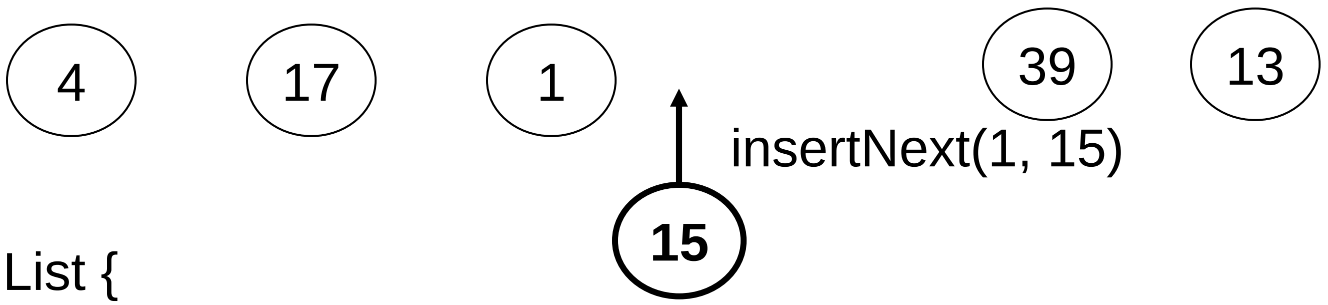
A sorted list is a class in which elements are sorted.



```
class SortedList {
        Element findNext(Element x);
            // return the smallest element that is larger than x
        void insert(Element x);
            // insert x so that x is between the largest element that
            // is smaller than x and the smallest element that is larger x
        void remove(Element x); // remove x
}
```

Class List

A list is a class in which elements can be inserted at any location.



```
class List {
      Element findNext(Element x); // return the element after x
      void insertNext(Element x, Element y); // insert y after x
      void insertBefore(Element x, Element y); // insert y before x
      void insertHead(Element x); // insert x as the first element
      void insertTail(Element x); // insert x as the last element
}
```

Self Test

What is the relationship between SortedList and List?

A. SortedList: base class; List: derived class, “is a”
B. List: base class; SortedList: derived class, “is a”
C. no relationship
D. List should have ("has a") SortedList as an attribute
E. SortedList should have ("has a") List as an attribute

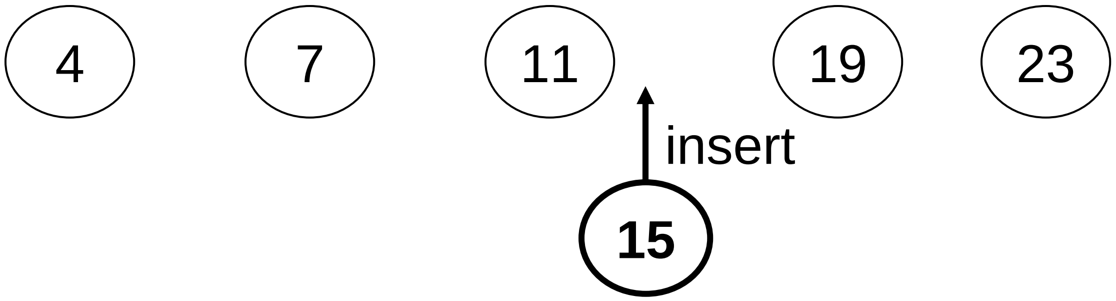
Self Test

What is the relationship between SortedList and List?

A. SortedList: base class; List: derived class, “is a”
B. List: base class; SortedList: derived class, “is a”
C. no relationship
D. List should have ("has a") SortedList as an attribute
E. SortedList should have ("has a") List as an attribute

SortedList uses List?

```
class SortedList {
    List * mylist;
    void insert(Element x) {
      Element * aheadx = mylist -> findHead();
      while (x > aheadx) { aheadx = mylist -> findNext (aheadx); }
      mylist -> insertBefore(aheadx, x);
    }
}
```



Now, what is your answer?

“Has a” as Encapsulation

- Does SortedList have a List? i.e. SortedList uses List to maintain the order of elements by inserting at the right location? Maybe. **Maybe not**.
- SortedList does not have to use List. It may use
  - array
  - binary search tree
  - priority queue
  - list
  - …   // can be changed without breaking your code
- You **don’t have to know** what is inside SortedList. You only need to know its interface $\Rightarrow$ **encapsulation**

- Encapsulation and inheritance are the foundation of code reuse.
- Encapsulation: You can use classes without knowing how they are implemented. In fact, the implementation may change without breaking your code.
- Inheritance: You can use the attributes and methods already declared, defined, or implemented in the class hierarchy.
- Do not allow the visibility of attributes, methods, or to create new classes, if you have doubt. Encapsulation is more likely to keep your code working.

General Principle of Inheritance

- identify the objects' behavior
- extract common behavior and create the base class
- provide specialized behavior in the derived class
- base class: more general, fewer attributes

⇒ Since the interface of SortedList and the interface of List do not have “**superset**” relationship, they should **not** form a class hierarchy.

- Do not create too many classes. If you can express the properties of an object by **attributes**, do not create new classes. For example, Human, Male, Female, Boy, Girl, Man, Woman ... They can be expressed by two attributes “gender” and “age”

Abstract Class

- Sometimes, it makes no sense to create an object of a class (called "**instantiation**"). It makes sense to create objects of that class' derived classes. For example, no object for Shape.
- Why to create a class without instantiation?
  – to provide a common interface for derived classes, for example, getArea
  – to provide common attributes, for example, line style and thickness

Lab 3: Creating Multi-File C++ using Managed Project in eclipse