



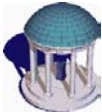
The University of North Carolina at Chapel Hill

COMP 144 Programming Language Concepts
Spring 2003

Dynamic Method Binding, Inheritance

Stotts, Hernandez-Campos

1



Fundamental Concepts in OOP

- **Encapsulation**
 - Data Abstraction
 - Information hiding
 - The notion of class and object
- **Inheritance**
 - Code reusability
 - Is-a vs. has-a relationships
- **Polymorphism**
 - Dynamic method binding

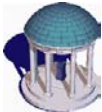
2



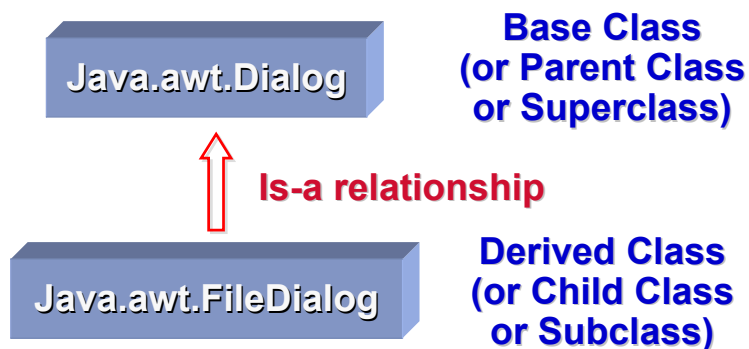
Inheritance

- Encapsulation improves code reusability
 - Abstract Data Types
 - Modules
 - Classes
- However, it is generally the case that the code a programmer wants to reuse is close but not exactly what the programmer needs
- **Inheritance** provides a mechanism to extend or refine units of encapsulation
 - By adding or *overriding* methods
 - By adding attributes

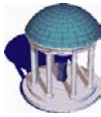
3



Inheritance Notation

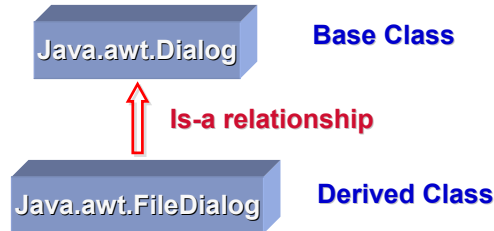


4



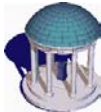
Inheritance

Subtype



- The derived class has all the members (*i.e.* attributes and methods) of the base class
 - Any object of the derived class can be used in any context that expect an object of the base class
 - `fp = new FileDialog()` is **both** an object of class Dialog and an object of class File Dialog

5



Method Binding

```
class person { ...  
class student : public person { ...  
class professor : public person { ...
```

Classes Student and Professor derive from class Person

```
student s;  
professor p;  
  
person *x = &s;  
person *y = &p;
```

```
void person::print_mailing_label () { ...  
...
```

Method `print_mailing_label` is polymorphic

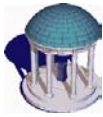
```
s.print_mailing_label (); // student::print_mailing_label (s)  
p.print_mailing_label (); // professor::print_mailing_label (p)
```

```
x->print_mailing_label (); // ??  
y->print_mailing_label (); // ??
```

Results depend on the binding: static or dynamic

`Print_mailing_label` redefined for student and professor classes

6

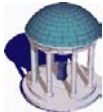


Method Binding

Static and Dynamic

- In **static method binding**, method selection depends on the type of the variable `x` and `y`
 - Method `print_mailing_label()` of class `person` is executed in both cases
 - Resolved at compile time
- In **dynamic method binding**, method selection depends on the class of the objects `s` and `p`
 - Method `print_mailing_label()` of class `student` is executed in the first case, while the corresponding methods for class `professor` is executed in the second case
 - Resolved at run time

7



Polymorphism and Dynamic Binding

- The is-a relationship supports the development of *generic operations* that can be applied to objects of a class and all its subclasses
 - This feature is known as *polymorphism*
 - E.g. `paint()` method is polymorphic (accepts multiple types)
- The binding of messages to method definitions is instance-dependent, and it is known as **dynamic binding**
 - It has to be resolved at run-time
 - Dynamic binding requires the `virtual` keyword in C++
 - Static binding requires the `final` keyword in Java

8



Polymorphism example

```

//: Shapes.java
package c11;
import java.util.*;
interface Shape {
    void draw();
}
class Circle implements Shape {
    public void draw() {
        System.out.println("Circle.draw()");
    }
}
class Square implements Shape {
    public void draw() {
        System.out.println("Square.draw()");
    }
}

```

Upcasting (Type Safe in Java)

```

class Triangle implements Shape {
    public void draw() {
        System.out.println("Triangle.draw()");
    }
}
public class Shapes {
    public static void main(String[] args) {
        Vector s = new Vector();
        s.addElement(new Circle());
        s.addElement(new Square());
        s.addElement(new Triangle());
        Enumeration e = s.elements();
        while (e.hasMoreElements())
            ((Shape)e.nextElement()).draw();
    }
}

```

Poly-morphism

What if you want to know the exact type at run-time?



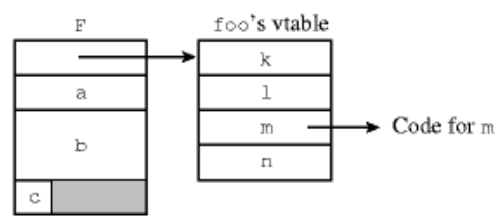
Dynamic Binding Implementation

- A common implementation is based on a *virtual method table* (vtable)
 - Each object keeps a pointer to the vtable that corresponds to its class

```

class foo {
    int a;
    double b;
    char c;
public:
    virtual void k { ...
    virtual int l { ...
    virtual void m {};
    virtual double n( ...
    ...
} F;

```





Dynamic Binding Implementation

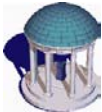
- Given an object of class foo, and pointer f to this object, the code that is used to invoke the appropriate method would be

to call $f \rightarrow m()$:

```

r1 := f    this (self)
r2 := *r1  -- vtable address
r2 := *(r2 + (3-1) * 4)  -- assuming 4 = sizeof(address)
call *r2  (polymorphic) method invocation
    
```

11

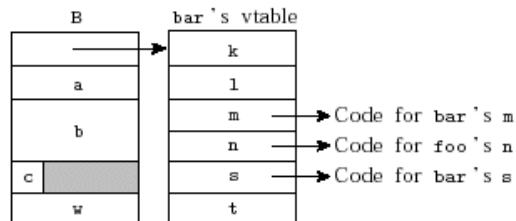


Dynamic Binding Implementation Simple Inheritance

- Derived classes extend the vtable of their base class
 - Entries of overridden methods contain the address of the new methods

```

class bar : public foo {
    int w;
public:
    void m (); //override
    virtual double s ( ...
    virtual char *t ( ...
    ...
} B;
    
```



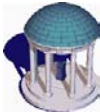
12



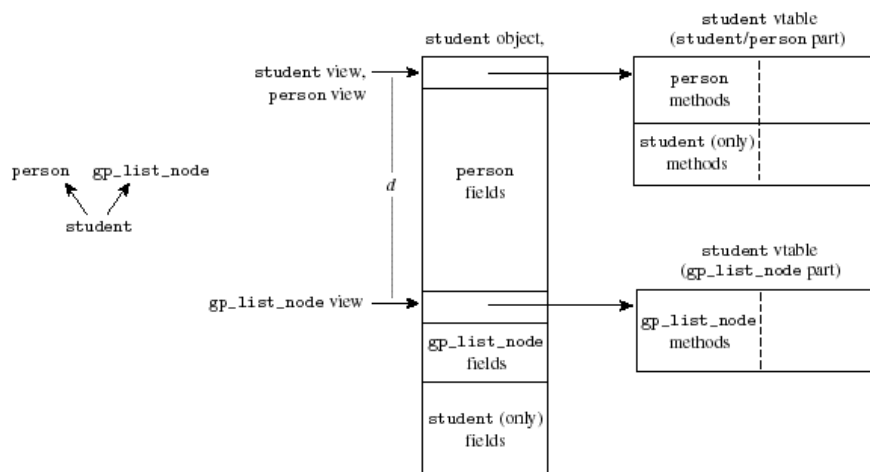
Dynamic Binding Implementation Multiple Inheritance

- A class may derive from more than one base class
 - This is known as multiple inheritance
- Multiple inheritance is also implemented using vtables
 - Two cases
 - » Non-repeated multiple inheritance
 - » Repeated multiple inheritance

13



Dynamic Method Binding Non-Repeated Multiple Inheritance



14



Dynamic Method Binding

Non-Repeated Multiple Inheritance

- The view of this must be corrected, so it points to the correct part of the objects
 - An offset d is used to locate the appropriate vtable pointer
 - » d is known at compile time

to call `my_student.debug_print`:

```
this (self)
r1 := my_student
r1 := r1 + d
r2 := *r1
r3 := *(r2 + (3-1) × 8)
r2 := *(r2 + (3-1) × 8 + 4)
r1 := r1 + r2
call *r3
```

-- student view of object
-- gp_list_node view of object
-- address of appropriate vtable
-- method address
-- this correction
-- this

15



Dynamic Method Binding

Repeated Multiple Inheritance

- Multiple inheritance introduces a semantic problem: method name collisions
 - Ambiguous method names
 - Some languages support inherited method renaming (e.g. Eiffel)
 - Other languages, like C++, require a reimplementaion that solves the ambiguity
 - Java *solves* the problem by not supporting multiple inheritance
 - » A class may inherit multiple interfaces, but, in the absence of implementations, the collision is irrelevant

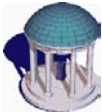
16



Reading Assignment

- Scott
 - Read Sect. 10.4
 - Read Sect. 10.5 intro and 10.5.1

17



RTTI and Introspection

- *Run-time type identification* make it possible to determine the type of an object
 - E.g. given a pointer to a base class, determine the derived class of the pointed object
 - The type (class) must be known at compile time
- *Introspection* makes general class information available at run-time
 - The type (class) does not have to be known at compile time
 - This is very useful in component architectures and visual programming
 - E.g. list the attributes of an object

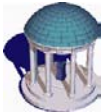
18



RTTI and Introspection

- RTTI and introspection are powerful programming language features
 - They enables some powerful design techniques
 - We will discuss them in the context of Java
- This discussion will follow Chapter 11 in *Thinking in Java* by Bruce Eckel
 - <http://www.codeguru.com/java/tij/tij0119.shtml>
 - By the way, this is an excellent book freely available on-line

19



The need for RTTI Polymorphism Example

```
///  
package c11;  
import java.util.*;  
interface Shape {  
    void draw();  
}  
class Circle implements Shape {  
    public void draw() {  
        System.out.println("Circle.draw()");  
    }  
}  
class Square implements Shape {  
    public void draw() {  
        System.out.println("Square.draw()");  
    }  
}
```

Upcasting (Type Safe in Java)

```
class Triangle implements Shape {  
    public void draw() {  
        System.out.println("Triangle.draw()");  
    }  
}  
public class Shapes {  
    public static void main(String[] args) {  
        Vector s = new Vector();  
        s.addElement(new Circle());  
        s.addElement(new Square());  
        s.addElement(new Triangle());  
        Enumeration e = s.elements();  
        while (e.hasMoreElements())  
            ((Shape)e.nextElement()).draw();  
    }  
}
```

**Poly-
morphism**

What if you want to know the exact type at run-time?

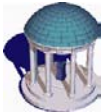
20



The Class Object

- Type information is available at run-time in Java
- There is a *Class object* for each class in the program
 - It stores class information
- Class objects are loaded in memory the first time they are needed
 - A Java program is not completely loaded before it begins!
- The class **Class** provides a number of useful methods for RTTI
 - <http://java.sun.com/j2se/1.3/docs/api/java/lang/Class.html>

21



Example

```
class Candy {
    static {
        System.out.println("Loading Candy");
    }
}
class Gum {
    static {
        System.out.println("Loading Gum");
    }
}
class Cookie {
    static {
        System.out.println("Loading
Cookie");
    }
}
```

Executed at Load Time

```
public class SweetShop {
    public static void main(String[] args) {
        System.out.println("inside main");
        new Candy();
        System.out.println("After creating
Candy");
        try {
            Class.forName("Gum");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        System.out.println("After
Class.forName(\"Gum\")");
        new Cookie();
        System.out.println("After creating
Cookie");
    }
}
```

Returns a reference to class Gum

22

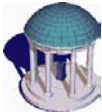


Example

- Output
 - JVM-1

inside main
Loading Candy
After creating Candy
Loading Gum
After Class.forName("Gum")
Loading Cookie
After creating Cookie

23



Example

- Output
 - JVM-2

Loading Candy
Loading Cookie
inside main
After creating Candy
Loading Gum
After Class.forName("Gum")
After creating Cookie

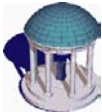
24



The Class Object

- Class literals also provide a reference to the Class object
 - E.g. `Gum.class`
- Each object of a primitive wrapper class has a standard field called `TYPE` that also provides a reference to the Class object
 - <http://java.sun.com/j2se/1.3/docs/api/java/lang/Boolean.html>

25



RTTI

- The type of a object can be determined using the **`instanceof`** keyword
 - See `PetCount.java`
 - It can be rewritten using Class literal, see `PetCount2.java`
 - Notice that an object of a derived class is an instance of the its base classes (*i.e.* any predecessor in the inheritance hierarchy)
- RTTI is very useful when reusing classes without extending them
- **`Class.isInstance()`** also implements the **`instanceof`** functionality

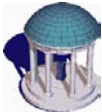
26



Introspection

- *Introspection* makes general class information available at run-time
 - The type (class) does not have to be known at compile time
 - E.g. list the attributes of an object
- This is very useful in
 - Rapid Application Development (RAD)
 - » Visual approach to GUI development
 - » Requires information about component at run-time
 - Remote Method Invocation (RMI)
 - » Distributed objects

27



Reflection

- Java supports introspection through its reflection library
 - <http://java.sun.com/j2se/1.3/docs/api/java/lang/reflect/package-summary.html>
 - See classes Field (attributes), Method and Constructor
- Examples:
 - ShowMethods.java

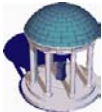
28



Python

- The Inspect module provides introspections mechanism
 - <http://www.python.org/doc/current/lib/module-inspect.html>
 - See:
 - » `getmembers(object[, predicate])`
 - » `getsource(object)`
 - » `getclasstree(classes[, unique])`
 - » `getmro(cls)`

29



Java Beans

- Tutorial
 - <http://java.sun.com/docs/books/tutorial/javabeans/index.html>
- The JavaBeans API makes it possible to write **component software** in the Java programming language.
- **Components** are self-contained, reusable software units that can be **visually composed** into composite components, applets, applications, and servlets using visual application builder tools.
- JavaBean components are known as *Beans*.

30

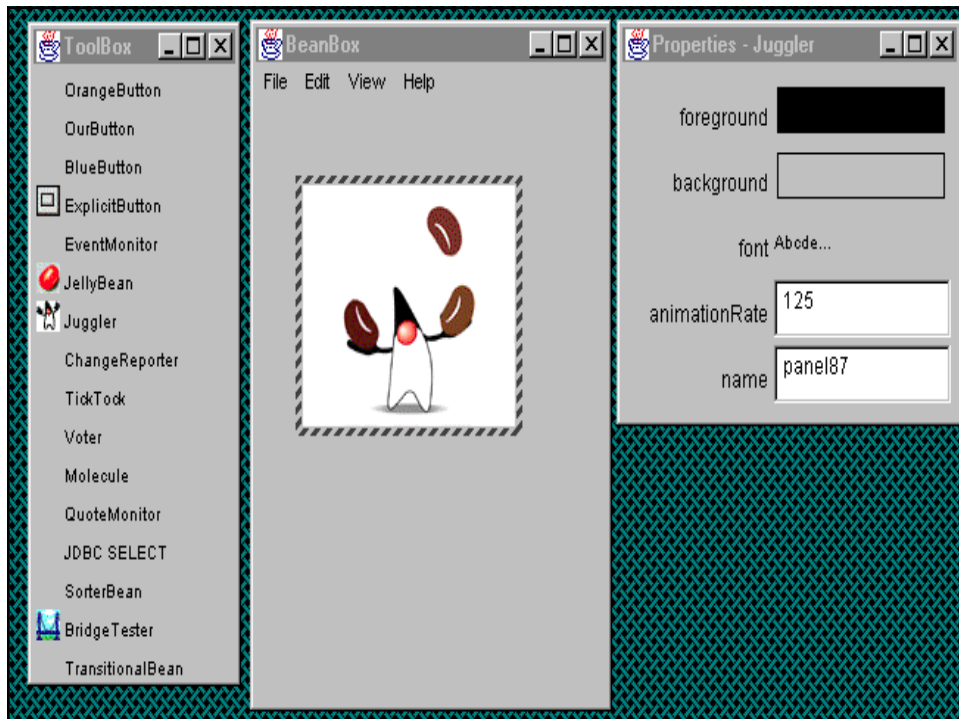


Demonstration

- BeanBox application

The BeanBox is a simple tool you can use to test your Beans, and to learn how to visually manipulate their properties and events. The BeanBox is not a builder tool. You'll use the BeanBox to learn about Beans.

31





Reading Assignment

- Bruce Eckel *Thinking in Java*
 - Chapter 11, RTTI
 - » <http://www.codeguru.com/java/tij/tij0119.shtml>
- Java Beans
 - Tutorial
 - » <http://java.sun.com/docs/books/tutorial/javabeans/index.html>
 - Play with the BeanBox