



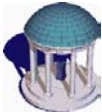
The University of North Carolina at Chapel Hill

COMP 144 Programming Language Concepts
Spring 2003

Object-Oriented Languages: Scope, Lifetimes, Garbage Collection

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



Encapsulation

- **Data abstraction** allow programmers to hide data representation details behind a (comparatively) simple set of operations (an *interface*)
- What the benefits of data abstraction?
 - Reduces *conceptual load*
 - » Programmers need to know less about the rest of the program
 - Provides *fault containment*
 - » Bugs are located in independent components
 - Provides a significant degree of *independence* of program components
 - » Separate the roles of different programmer

**Software
Engineering
Goals**

3

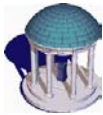


Encapsulation Classes, Objects and Methods

- The unit of encapsulation in an O-O PL is a **class**
 - An abstract data type
 - » The set of values is the set of *objects* (or *instances*)
- Objects can have a
 - Set of *instance attributes* (*has-a relationship*)
 - Set of *instance methods*
- Classes can have a
 - Set of *class attributes*
 - Set of *class methods*
- The entire set of methods of an object is known as the *message protocol* or the *message interface* of the object

**Method calls are
known as messages**

4



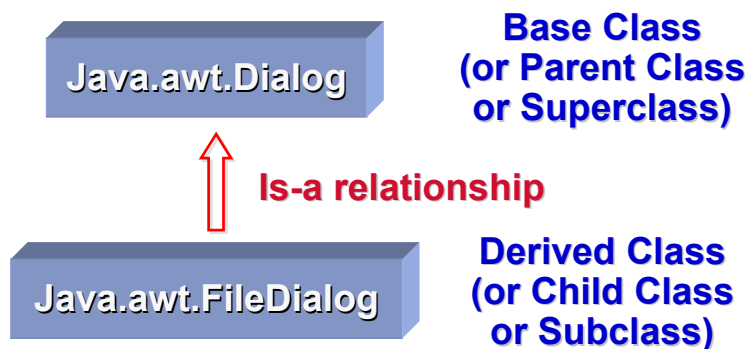
Inheritance

- 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

5



Inheritance Notation



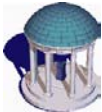
6



Polymorphism

- 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
- The binding of messages to method definition 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

7



Encapsulation Modules and Classes

- The basic unit of OO, the class, is a *unit of scope*
 - This idea originated in module-based languages in the mid-70s
 - » E.g. Clu, Modula, Euclid
- Rules of scope enforce data hiding
 - Names have to be *exported* in order to be accessible by other modules
 - What kind of data hiding mechanisms do we have in Java?
 - » <http://java.sun.com/docs/books/tutorial/java/javaOO/accesscontrol.html>
 - And in Python?
 - » <http://www.python.org/doc/current/tut/node11.html#SECTION00116000000000000000>

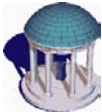
8



Reading Assignment

- Scott
 - Read Ch. 10 intro
 - Read Sect. 10.1
 - » Study the list and queue examples
 - Read Sect. 10.2
 - » Go through the documents linked in slide 8

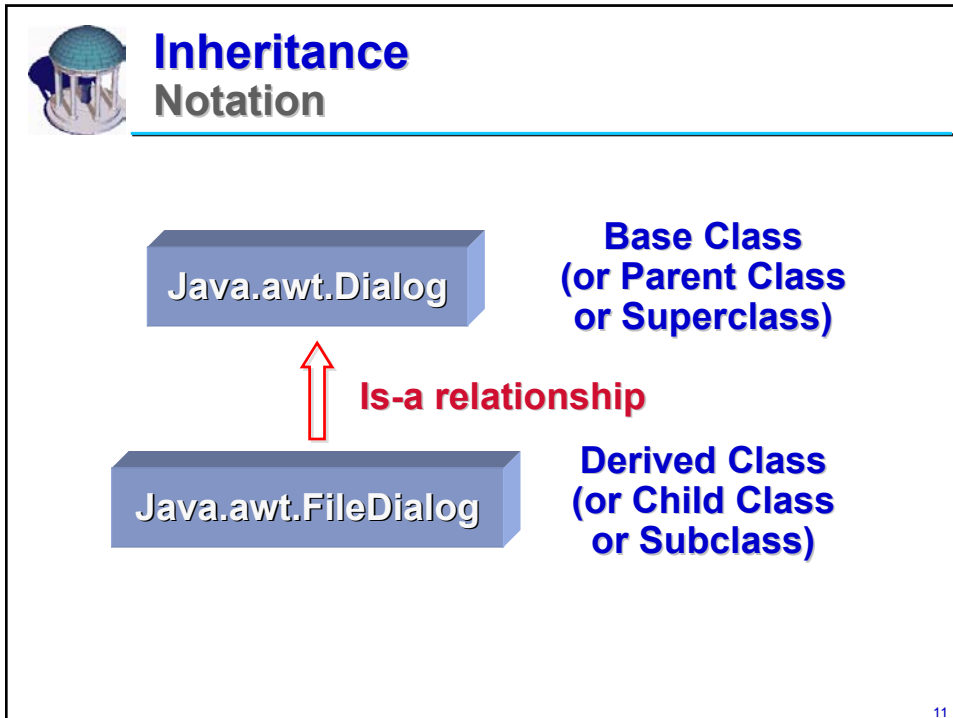
9



Object Lifetime: Constructors

- **Constructors** are methods used to *initialize* the content of an object
 - They do not allocate space
- Most languages allow multiple constructors
 - They are distinguished using different names or different parameters (type and/or number)
 - Java and C++ overload the constructor name, so the appropriate method is selected using the number and the type of the arguments
 - » `Rectangle r;`
 - » Invokes the parameterless constructor
 - Smalltalk and Eiffel support different constructor names

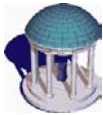
10



Java Example

- Dialog constructors
 - http://java.sun.com/j2se/1.4/docs/api/java/awt/Dialog.html#constructor_summary
- FileDialog constructors
 - http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#23302

12



Constructors in Eiffel

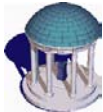
```
class COMPLEX
  creation
    new_cartesian, new_polar
  feature {ANY}
    x, y : REAL;
    new_cartesian (x_val, y_val : REAL) is
    do
      x := x_val; y := y_val;
    end;
    new_polar (rho, theta : REAL) is
    do
      x := ro * cos (theta);
      y := ro * sin (theta)
    end;
    -- other public methods
  feature {NONE}
    -- private methods
end -- class COMPLEX
...
a, b : COMPLEX;
...
!!b.new_cartesian (0, 1);
!!a.new_polar (pi/2, 1);
```

} **Explicit Constructor Declaration**

**Complex
Numbers
Class**

} **Constructors (!! equals new)**

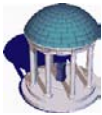
13



References and Values

- Some OO languages use the reference model
 - More elegant
 - Extra level of indirection on every access
 - *E.g.* Java, Simula, Smalltalk
- Other languages use the value model
 - More efficient
 - More difficult to control initialization
 - » *E.g.* uninitialized objects, mutual references
 - *E.g.* C++, Ada 95

14



Constructors in C++

```
foo b;           // calls foo::foo ()
...
foo b (10, 'x'); // calls foo::foo (int, char)

foo a;           // calls foo::foo ()
bar b;           // calls bar::bar ()
...
foo c (a);       // calls foo::foo (foo&)
foo d (b);       // calls foo::foo (bar&)

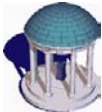
foo c = a;       // calls foo::foo (foo&)
foo d = b;       // calls foo::foo (bar&)

foo a, c, d;     // calls foo::foo () three times
bar b;           // calls bar::bar ()
...
c = a;           // calls foo::operator= (foo&)
d = b;           // calls foo::operator= (bar&)
```

**Initialization
(not assignment)**

**Copy Constructor
operator=**

15



Execution Order

- How is an object of class B derived from class A initialized?
 - In C++ and Java, the constructor of A is invoked before the constructor of B
 - Why?
 - » So the B constructor never sees uninitialized attributes
 - What are the arguments of the A constructor?
 - » In C++, they are explicitly defined
- ```
B::B (B_params) : A (A_args) { ... }
```
- ```
list_node() : prev(this), next(this),  
             head_node(this), val(0) { }
```

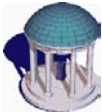
16



Java Example

- See Java Language Specification
 - http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html#41652
 - http://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#23302
- Alternate constructor
- Superclass constructor
 - Unqualified superclass constructor
 - Qualified superclass constructor invocations
 - » Inner classes

17



Object Lifetime: Destructors

- **Destructors** are methods used to *finalize* the content of an object
 - They do not deallocate space
 - Language implementations that support garbage collection greatly reduce the need for destructors
 - » Most C++ compiler do not support GC

18

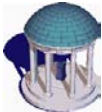


C++ Example

- In general, C++ destructors are used for manual storage reclamation

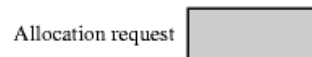
```
class name_list_node : public gp_list_node {
    char *name;           // pointer to the data in a node
public:
    name_list_node () {
        name = 0;        // empty string
    }
    name_list_node (char *n) {
        name = new char[strlen(n)];
        strcpy (name, n); // copy argument into member
    }
    name_list_node () { Destructor
        if (name != 0) {
            delete name; // reclaim space
        }
    }
};
```

19



Heap-based Allocation

- The **heap** is a region of storage in which subblock can be allocated and deallocated
 - This not the *heap data structure*

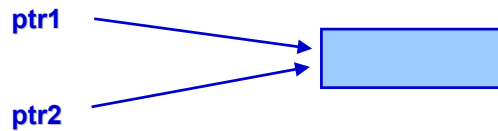


20

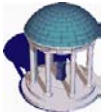


Garbage Collection

- Explicit reclamation of heap objects is problematic
 - The programmer may forget to deallocate some objects
 - » Causing *memory leaks*
 - » In the previous example, the programmer may forget to include the `delete` statement
 - References to deallocated objects may not be reset
 - » Creating *dangling references*



21



Garbage Collection

- Explicit reclamation of heap objects is problematic
 - The programmer may forget to deallocate some objects
 - » Causing *memory leaks*
 - » In the previous example, the programmer may forget to include the `delete` statement
 - References to deallocated objects may not be reset
 - » Creating *dangling references*



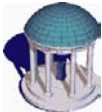
22



Garbage Collection

- Automatic reclamation of the head space used by object that are no longer useful
 - Developed for functional languages
 - » It is essential in this programming paradigm. Why?
 - Getting more and more popular in imperative languages
 - » Java, C#, Python
- It is generally slower than manual reclamation, but it eliminates a very frequent programming error
 - Language without GC usually have memory profiling tools
 - » E.g. <http://www.mozilla.org/performance/tools.html>,
<http://www.pds-site.com/VMGear/profiler/bigger/Objectallocationview.htm>

23



Garbage Collection Techniques

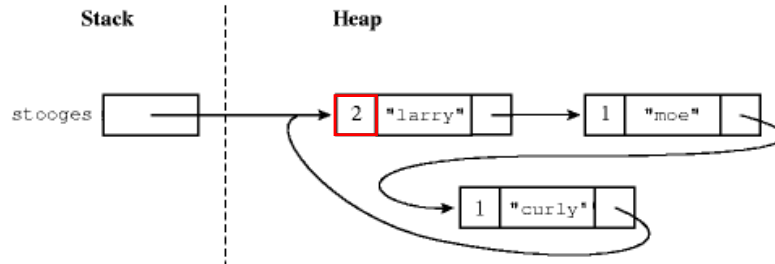
- When is an object no longer useful?
- There are several garbage collection techniques that answer this question in a different manner
 - Reference counting
 - Mark-and-sweep collection

24



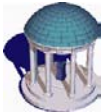
Reference Counting

- Each object has an associated reference counter



- The run-time system
 - keeps reference counters up to date, and
 - recursively deallocates objects when the counter is zero

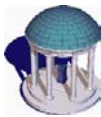
25



Reference Counting Problems

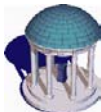
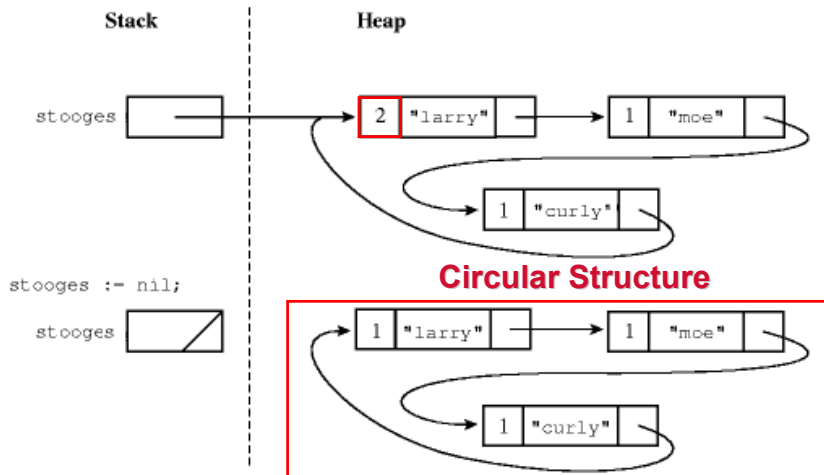
- Extra overhead of storing and updating reference counts
- Strong typing required
 - Impossible in a language like C
 - It cannot be used for variant records
- It does not work with circular data structures
 - This is a problem with this definition of *useful* object as an object with one or more references

26



Reference Counting Circular Data Structures

- Each object has an associated reference counter



Mark-and-Sweep Collection

- A better definition of *useless* object is one that cannot be reached by following a chain of valid pointers starting from *outside* the heap
- Mark-and-Sweep GC applies this definition
- Algorithm:
 - Mark every block in the heap as useless
 - Starting with all pointers outside the heap, recursively explore all linked data structures
 - Add every block that remain marked to the *free* list
- Run whenever the free space is low

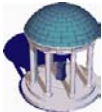
28



Mark-and-Sweep Collection Problems

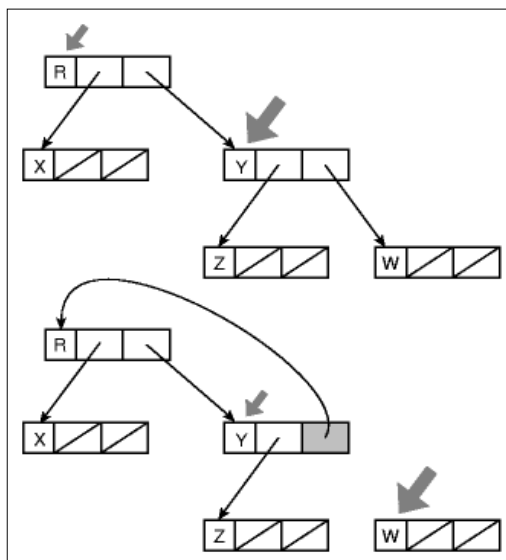
- Block must begin with an indication of its size
 - Type descriptor that indicate their size makes this requirement unnecessary
- A stack of depth proportional to the longest reference chain is required
 - But we are already running are out of space!
 - Pointer reversal embeds the stack in the sequence of references in the heap
 - » The GC reverses each pointer it traverses

29

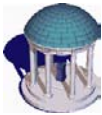


Mark-and-Sweep Collection Pointer Reversal

R is outside the heap

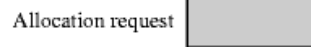


30



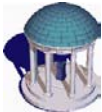
Store-and-Copy

- Use to reduce external fragmentation



- S-C divides the available space in half, and allocates everything in that half until it is full
- When that happens, copy each *useful* block to the other half, clean up the remaining block, and switch the roles of each half

31



Reading Assignment

- Scott
 - Read Sect. 10.3
 - Read Sect. 7.7.2 (dangling references)
 - Read Sect. 7.7.3 (garbage collection)
 - Garbage collection in Java JDK 1.2
 - » <http://developer.java.sun.com/developer/technicalArticles/ALT/RcfObj/>

32