

Inheritance and Polymorphism

Topics

- Introduction to Inheritance
- Polymorphism

2

Introduction to Inheritance (1 of 2)

- In the real world, many objects are a specialized version of more general objects
 - Example: grasshoppers and bees are specialized types of insect
 - In addition to the general insect characteristics, they have unique characteristics:
 - Grasshoppers can jump
 - Bees can sting, make honey, and build hives

3

Introduction to Inheritance (2 of 2)

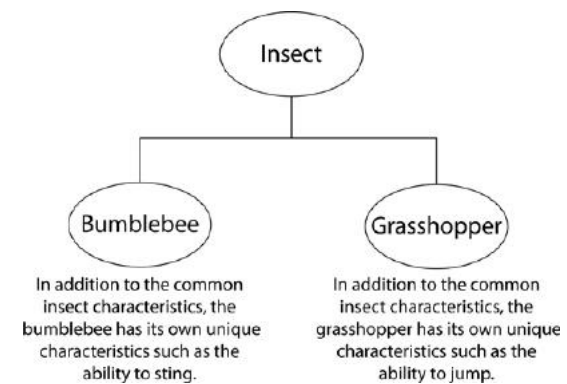


Figure 11-1 Bumblebees and grasshoppers are specialized versions of an insect

4

Inheritance and the “Is a” Relationship

(1 of 4)

- “Is a” relationship: exists when one object is a specialized version of another object
 - Specialized object has all the characteristics of the general object plus unique characteristics
 - Example: Rectangle is a shape
Daisy is a flower

5

Inheritance and the “Is a” Relationship

(2 of 4)

- Inheritance: used to create an “is a” relationship between classes
- Superclass (base class): a general class
- Subclass (derived class): a specialized class
 - An extended version of the superclass
 - Inherits attributes and methods of the superclass
 - New attributes and methods can be added

6

Inheritance and the “Is a” Relationship

(3 of 4)

- For example, need to create classes for cars, pickup trucks, and SUVs
- All are automobiles
 - Have a make, year model, mileage, and price
 - This can be the attributes for the base class
- In addition:
 - Car has a number of doors
 - Pickup truck has a drive type
 - SUV has a passenger capacity

7

Inheritance and the “Is a” Relationship

(4 of 4)

- In a class definition for a subclass:
 - To indicate inheritance, the superclass name is placed in parentheses after subclass name
 - Example: `class Car(Automobile):`
 - The initializer method of a subclass calls the initializer method of the superclass and then initializes the unique data attributes
 - Add method definitions for unique methods

8

Inheritance in UML Diagrams (1 of 2)

- In UML diagram, show inheritance by drawing a line with an open arrowhead from subclass to superclass

9

Inheritance in UML Diagrams (2 of 2)

10

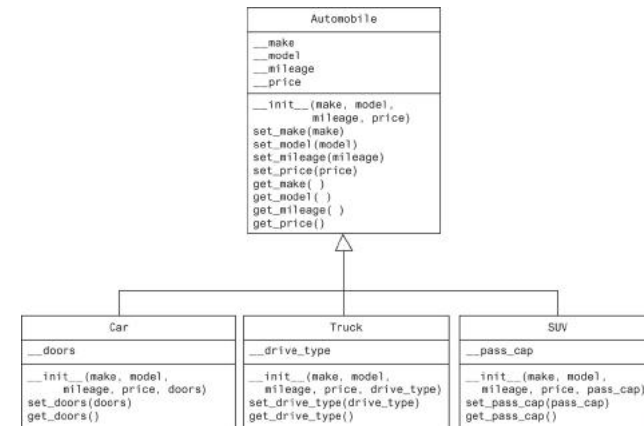


Figure 11-2 UML diagram showing inheritance

10

Polymorphism (1 of 2)

- **Polymorphism**: an object's ability to take different forms
- Essential ingredients of polymorphic behavior:
 - Ability to define a method in a superclass and override it in a subclass
 - Subclass defines method with the same name
 - Ability to call the correct version of overridden method depending on the type of object that called for it

11

Polymorphism (2 of 2)

- In previous inheritance examples showed how to override the `__init__` method
 - Called superclass `__init__` method and then added onto that
- The same can be done for any other method
 - The method can call the superclass equivalent and add to it, or do something completely different

12

The `isinstance` Function

- Polymorphism provides great flexibility when designing programs
- `AttributeError` exception: raised when a method is receives an object which is not an instance of the right class
- `isinstance` function: determines whether object is an instance of a class
 - Format: `isinstance(object, class)`

13

Summary

- This chapter covered:
 - Inheritance, including:
 - “Is a” relationships
 - Subclasses and superclasses
 - Defining subclasses and initializer methods
 - Depicting inheritance in UML diagrams
 - Polymorphism
 - The `isinstance` function

14

Recursion

Topics

- Introduction to Recursion
- Problem Solving with Recursion
- Examples of Recursive Algorithms

2

Introduction to Recursion (1 of 3)

- Recursive function: a function that calls itself
- Recursive function must have a way to control the number of times it repeats
 - Usually involves an `if-else` statement which defines when the function should return a value and when it should call itself
- Depth of recursion: the number of times a function calls itself

3

Introduction to Recursion (2 of 3)

4

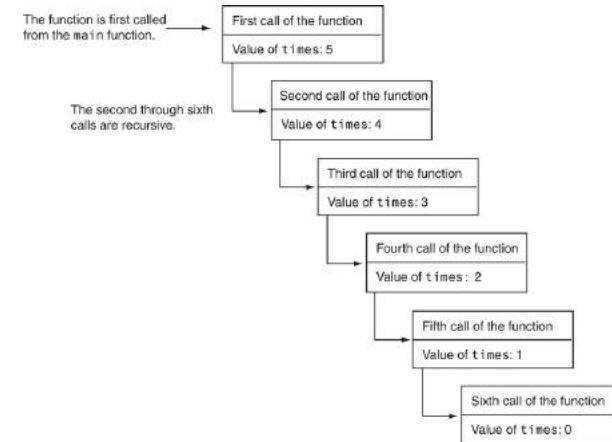


Figure 12-2 Six calls to the message function

4

Introduction to Recursion (3 of 3)

5

```
Recursive function call  def message(times):
                        if times > 0:
                            print('This is a recursive function.')
                            message(times - 1)
```

Control returns here from the recursive call. There are no more statements to execute in this function, so the function returns.

Figure 12-3 Control returns to the point after the recursive function call

5

Problem Solving with Recursion (1 of 2)

- Recursion is a powerful tool for solving repetitive problems
- Recursion is never required to solve a problem
 - Any problem that can be solved recursively can be solved with a loop
 - Recursive algorithms usually less efficient than iterative ones
 - Due to *overhead* of each function call

6

Problem Solving with Recursion (2 of 2)

- Some repetitive problems are more easily solved with recursion
- General outline of recursive function:
 - If the problem can be solved now without recursion, solve and return
 - Known as the *base case*
 - Otherwise, reduce problem to smaller problem of the same structure and call the function again to solve the smaller problem
 - Known as the *recursive case*

7

Using Recursion to Calculate the Factorial of a Number

- In mathematics, the $n!$ notation represents the factorial of a number n
 - For $n = 0$, $n! = 1$
 - For $n > 0$, $n! = 1 \times 2 \times 3 \times \dots \times n$
- The above definition lends itself to recursive programming
 - $n = 0$ is the base case
 - $n > 0$ is the recursive case
 - $\text{factorial}(n) = n \times \text{factorial}(n-1)$

8

Using Recursion (1 of 3)

```
# The factorial function uses recursion to
# calculate the factorial of its argument,
# which is assumed to be nonnegative.
def factorial(num):
    if num == 0:
        return 1
    else:
        return num * factorial(num - 1)
```

9

Using Recursion (2 of 3)

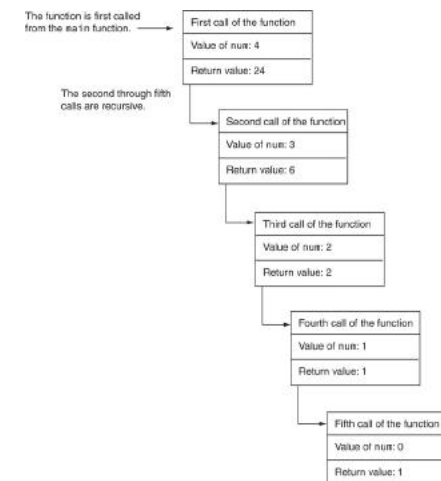


Figure 12-4 The value of num and the return value during each call of the function

10

Using Recursion (3 of 3)

- Since each call to the recursive function reduces the problem:
 - Eventually, it will get to the base case which does not require recursion, and the recursion will stop
- Usually the problem is reduced by making one or more parameters smaller at each function call

11

Direct and Indirect Recursion

- Direct recursion: when a function directly calls itself
 - All the examples shown so far were of direct recursion
- Indirect recursion: when function A calls function B, which in turn calls function A

12

Examples of Recursive Algorithms (1 of 2)

- Summing a range of list elements with recursion
 - Function receives a list containing range of elements to be summed, index of starting item in the range, and index of ending item in the range
 - Base case:
 - `if start index > end index return 0`
 - Recursive case:
 - `return current_number + sum(list, start+1, end)`

13

Examples of Recursive Algorithms (2 of 2)

```
# The range_sum function returns the sum of a
# specified
# range of items in num_list. The start parameter
# specifies the index of the starting item. The end
# parameter specifies the index of the ending item.
def range_sum(num_list, start, end):
    if start > end:
        return 0
    else:
        return num_list[start] +
            range_sum(num_list, start + 1, end)
```

14

The Fibonacci Series

- Fibonacci series: has two base cases
 - if $n = 0$ then $\text{Fib}(n) = 0$
 - if $n = 1$ then $\text{Fib}(n) = 1$
 - if $n > 1$ then $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$
- Corresponding function code:

```
# The fib function returns the nth number
# in the Fibonacci series
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

15

Finding the Greatest Common Divisor

- Calculation of the greatest common divisor (GCD) of two positive integers
 - If x can be evenly divided by y , then
 - $\text{gcd}(x, y) = y$
 - Otherwise, $\text{gcd}(x, y) = \text{gcd}(y, \text{remainder of } x/y)$
- Corresponding function code:

```
# The gcd function returns the greatest common
# divisor of two numbers.
def gcd(x, y):
    if x % y == 0:
        return y
    else:
        return gcd(x, x % y)
```

16

The Towers of Hanoi (1 of 5)

- Mathematical game commonly used to illustrate the power of recursion
 - Uses three pegs and a set of discs in decreasing sizes
 - Goal of the game: move the discs from leftmost peg to rightmost peg
 - Only one disc can be moved at a time
 - A disc cannot be placed on top of a smaller disc
 - All discs must be on a peg except while being moved

17

The Towers of Hanoi (2 of 5)

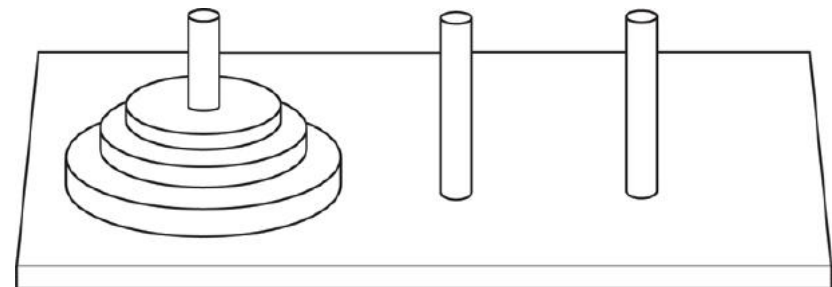


Figure 12-5 The pegs and discs in the Tower of Hanoi game

18

The Towers of Hanoi (3 of 5)

19

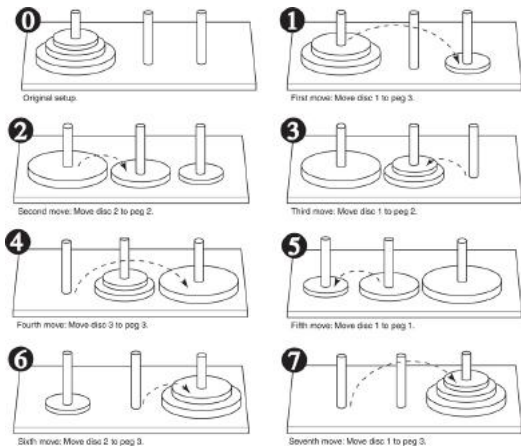


Figure 12-6 Steps for moving three pegs

19

The Towers of Hanoi (4 of 5)

- Problem statement: move n discs from peg 1 to peg 3 using peg 2 as a temporary peg
- Recursive solution:
 - If $n == 1$: Move disc from peg 1 to peg 3
 - Otherwise:
 - Move $n-1$ discs from peg 1 to peg 2, using peg 3
 - Move remaining disc from peg 1 to peg 3
 - Move $n-1$ discs from peg 2 to peg 3, using peg 1

20

The Towers of Hanoi (5 of 5)

```
# The moveDiscs function displays a disc move in
# the powers of Hanoi game.
# The parameters are:
#   num:      The number of discs to move.
#   from_peg: The peg to move from.
#   to_peg:   The peg to move to.
#   temp_peg: The temporary peg.
def move_discs(num, from_peg, to_peg, temp_peg):
    if num > 0:
        move_discs(num - 1, from_peg, temp_peg, to_peg)
        print('Move a disc from peg', from_peg, 'to peg',
              to_peg)
        move_discs(num - 1, temp_peg, to_peg, from_peg)
```

21

Recursion versus Looping

- Reasons not to use recursion:
 - Less efficient: entails function calling overhead that is not necessary with a loop
 - Usually a solution using a loop is more evident than a recursive solution
- Some problems are more easily solved with recursion than with a loop
 - Example: Fibonacci, where the mathematical definition lends itself to recursion

22

Summary

- This chapter covered:
 - Definition of recursion
 - The importance of the base case
 - The recursive case as reducing the problem size
 - Direct and indirect recursion
 - Examples of recursive algorithms
 - Recursion versus looping