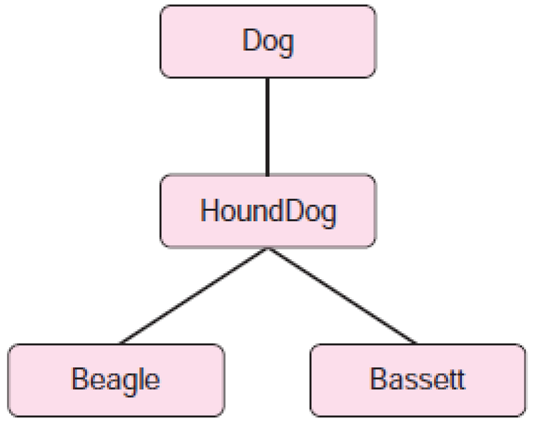
Polymorphism

Polymorphism

- By exploiting *similarity* among classes, inheritance makes it possible to build new classes from existing classes.
- In contrast to inheritance, polymorphism underscores the *differences* of class behavior in an inheritance hierarchy.
- The word *polymorphism* , derived from the Greek words *polus* and *morphe* , means “many shapes” or “many forms.”

Two Simple Forms Of Polymorphism

- **Ad-hoc Polymorphism—Method Overloading**
  - Method/Constructor Overloading, a form of polymorphism, is also known as ad-hoc polymorphism.
- **Upcasting**
  - A second form of polymorphism comes in the guise of upcasting. Recall that upcasting in an inheritance hierarchy allows an object of a derived type to be considered an object of a base type.
    - 1. Dog elvis;
    - 2. elvis = new HoundDog();
    - 3. elvis = new Beagle();
    - 4. elvis = new Bassett();

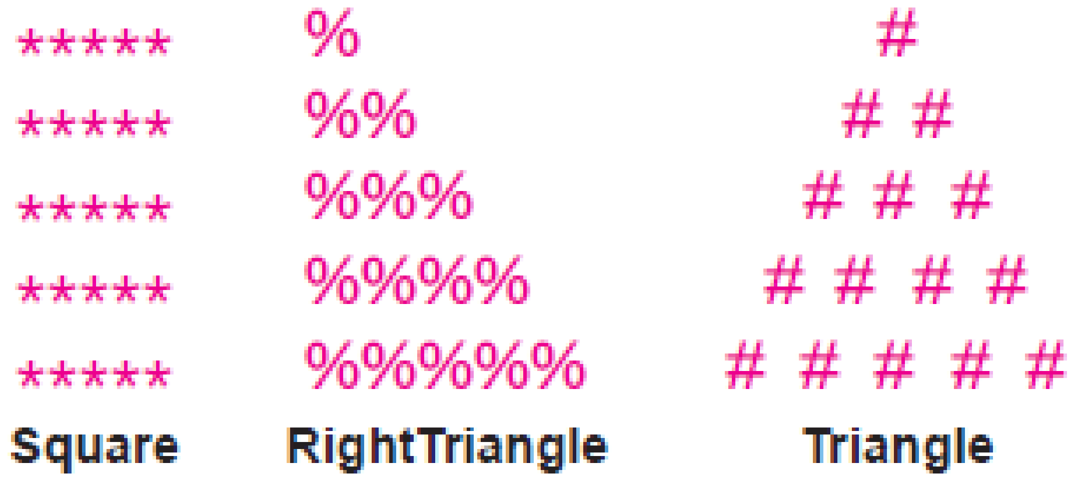


Continue..

- Because a HoundDog *is-a* Dog , a HoundDog reference can be upcast to Dog (line 2). Similarly, a Beagle reference and a Bassett reference can also be considered Dog references (lines 3 and 4). The reference elvis is *polymorphic* , that is, elvis has “many forms” and elvis can refer to a Dog object, a HoundDog object, a Beagle object, or a Bassett object.

Dynamic (Late) Binding

- A third form of polymorphism, *dynamic or late binding,* accentuates the *behavioral differences* among objects of different classes in a hierarchy.



```
*****    %              #
*****    %%            # #
*****    %%%          # # #
*****    %%%%        # # # #
*****    %%%%%      # # # # #
Square  RightTriangle  Triangle
```

Example Explanation

- When TestDraw is compiled and translated into bytecode, the Java compiler *cannot* determine which draw (…) method is applicable. The compiler knows that shape refers to a kind of Shape , but it does not know which kind. The appropriate draw(...) method is not discernible until the program runs and the user chooses one of three shapes. Consequently, the compiled version of the program, that is, the bytecode that executes on the Java Virtual Machine, does not specify which draw(...) method is appropriate. The choice of the correct draw(...) method is postponed until the program executes; that is, the choice is postponed until *runtime* .
- “Polymorphism via *dynamic or late binding* refers to choosing the appropriate method not at compile time, but at runtime.”

Continue..

- Dynamic binding is a convenience. If Java did not automatically support late binding, we could achieve the same effect explicitly, if less elegantly, using a sequence of if-else statements , instanceof ’s, and downcasts:

How Dynamic Binding Works?

- Shape is the apparent or declared type of shape.
- The *real type* or *actual type* of a reference variable is the type of the object that is created by the new operation.
- Let's arbitrarily assume that the user, TestDraw , chooses to draw a right triangle. In this case, the real type of shape is RightTriangle (line 15). When the draw(...) method is invoked by shape (see line 22), Java begins searching for a fully implemented draw(...) method. The search begins in the RightTriangle class (the real type of shape). If the RightTriangle class has implemented a draw(...) method then the search ends, and that method is called. If not, then Java searches the parent of RightTriangle . Searching continues all the way up the hierarchy until an implemented draw(...) method is found (or until the Object class is reached).

Continue..

- How does the compiler handle shape.draw() , which at compile time is ambiguous? It checks the apparent type of shape and works with that. Since Shape declares a draw(...) method, anything below Shape in the hierarchy also has a draw(...) method. Even though the Shape class does not *implement* a draw(...) method, Shape does *declare* a draw method.
- Consequently the statement shape.draw(1,1) causes no confusion to the compiler. The compiler happily accepts the statement and, during runtime, the appropriate version of draw(...) is selected. Were draw(...) not declared in Shape , a compile time error would be issued:
  - C:\JavaPrograms\TestDraw.java:19: cannot find symbol
  - symbol : method draw(int,int)
  - location: class Shape
  - shape.draw(1,1);
  - ^

Exceptions to Late Binding

- Late binding is the rule, but there are exceptions. Late binding allows the programmer to avoid a tedious sequence of if statements. However, there are situations where late binding does not make sense.
- Unlike the draw(…) method of Example 13.2, a final , private , or static method cannot be overridden in a derived class and has only one form. Consequently, a call to a final, private , or static method presents no ambiguity to the compiler. Because such a method has but one version, a method call can be associated with the correct method implementation at compile time, that is, before the program executes. There is no need to wait until runtime to connect the call to the appropriate version of the method.
- Java uses late binding for all method invocations except final, private and static methods.

Interfaces and Polymorphism

- Using an interface can tie classes together into a nice package with the power of polymorphism added to the bundle.
- An interface can be used to achieve polymorphism.
- As abstract class the polymorphism can be achieved with interfaces similarly.

EarylBinging & Late Binding

- In EarlyBinding the targeted method is found at the time of compilation and in late binding ,the targeted method is found at the run time.most scripts langugaes uses the concept of late   binding and the compiled languages uses the concept of early binding