Dynamic Binding

COMP360

“Sometimes one creates a dynamic impression by saying something, and sometimes one creates as significant an impression by remaining silent.”

Dali Lama

Three Pillars of OO Programming

- Inheritance
- Encapsulation & Abstraction
- Polymorphism

Creating an Object

When you create an object with the keyword new

- Java loads the class (if not already loaded)
- Java loads all parent classes
- Space is allocated in the heap
- The constructors are called in the proper order
- The constructor returns a reference to the object

Removing the Default Constructor

- If you create a constructor with parameters, then the default no parameter constructor does not work
- You must create a default no parameter constructor if you want one with other constructors

Missing Default Constructor

```
public class Aardvark {
    public Aardvark(int ant) { … }
}
```

- in another class

```
        // This does not work
Aardvark termite = new Aardvark();
```

Calling One Constructor from Another

- One constructor can call another constructor of the same class using the method `this( … )`
- When used as a method name, `this` is the constructor for this class
- Call the this constructor is often used to provide default values when a parameter is not given

Constructors calling Constructors

```
public class Worm{
    public Worm() {
        this("default");
    }
    public Worm(String dirt) {
        System.out.print( dirt );
    }
}
Worm crawly = new Worm("fish");      // displays fish
Worm inch   = new Worm();            // displays default
```

Constructors are not Inherited

- Unlike all other methods, constructors are not inherited by subclasses
- If a class does not have a constructor, the constructor of the super classes will be called in order

What is displayed?

```
class Fish {
   public Fish() {
      this("Fish ");
      System.out.print("trout ");
   }
   public Fish(String minnow) {
      System.out.print(minnow);
   }
}
class Bass extends Fish {
   public Bass() {
      System.out.print("Bass ");
   }
}
public class ConOrder {
   public static void main(String[] x) {
      Bass swim = new Bass();
   }
}
```

A. Bass
B trout Bass
C. trout Fish Bass
D. Fish trout Bass
E. Bass Fish trout

Using the Keyword `super`

- The keyword `super` refers to the superclass of the class in which `super` appears
- This keyword can be used in two ways:
  - To call a superclass constructor
  - To call a superclass method

Superclass Constructor is super

- You must use the keyword super to call the superclass constructor
- Invoking a superclass constructor's name in a subclass causes a syntax error
- Java requires that the statement that uses the keyword super appear first in the constructor

Implicit Calls to Parent Constructors

- If a constructor does not call the super class constructor, Java will automatically call it

```
public class Frog extends Amphibian {
    public Frog() {
      System.out.println("frog");
    }
}
```

- Is the same as

```
public class Frog extends Amphibian {
    public Frog() {
      super( );
      System.out.println("frog");
    }
}
```

Implicit Constructor Call

```
public class ImplicitCon {
    public static void main(String[] x) {
        Child kid = new Child();
    }
}
class Parent {
    public Parent() {
        System.out.println("Parent");
    }
}
class Child extends Parent {
    public Child() {
        System.out.println("Child");
    }
}
```

Displays

```
Parent
child
```

Implicit Constructor Call

```
public class ImplicitCon {
    public static void main(String[] x) {
        Child kid = new Child();
    }
}
class Parent {
    public Parent() {
        System.out.println("Parent");
    }
}
class Child extends Parent {

      // no Child constructor

}
```

Displays

```
Parent
```

Avoiding Super

- The default super() constructor is implicitly called at the beginning of any constructor
- If you call super with a parameter, the default no parameter version will not be called
- If you call the this constructor, with or without parameters, the default super() will not be called
- The default super() might be called in the other constructor called by this(…)

Will this work?

```
class Mammal {
  public Mammal(String name) {
    System.out.println("Mammal"+name);
  }
}
public class Sheep extends Mammal {
   int whatever = 0;
}
```

A. It should work fine
B. Sheep needs a no argument constructor
C. Mammal needs a no argument constructor
D. Both Sheep and Mammal need a no argument constructor

Default Constructor Needed

```
class Mammal {
  public Mammal() {
    System.out.println("Default Mammal");
  }
  public Mammal(String name) {
    System.out.println("Mammal"+name);
  }
}
public class Sheep extends Mammal {
   int irrelevant = 0;
}
```

- Since Sheep has no constructor, the default constructor calls a default constructor in the super class, Mammal

Default Constructor Needed

```
class Mammal {
  public Mammal(String name) {
    System.out.println("Mammal"+name);
  }
}
public class Sheep extends Mammal {
  int whatever = 0;
  public Sheep(String iname) {
    super(iname);
    System.out.println("Sheep");
  }
}
```

- Sheep calls the super constructor with a String parameter matching the Mammal constructor

Destructors

- Destructor functions are the inverse of constructor functions. They are called when an object is destroyed or dealloTreeed
- In C++ a destructor method is always named the same as the class name preceded by a tilde, ~

```
MyClass :: ~MyClass() { … }
```

- They are useful is releasing resources that might not otherwise be released
- Java calls the destructor method finalize

```
public void finalize() { … }
```

Finalize Example

```
public class FinalDemo {
    private int myID;
    public static void main(String[] unused) {
        FinalDemo thing;
        for (int i = 0; i < 1000; i++) {
            thing = new FinalDemo(i);
        }
        System.out.println("Objects created");
        System.gc();     // force garbage collection
        System.out.println("All done");
    }
    public FinalDemo(int num) {
        myID = num;
    }
    public void finalize() {
        System.out.println("finalizing "+myID);
    }
}
```

Dynamic Binding

- Previously we discussed the linker program which binds program names to addresses
- There are times with OO programs that you cannot statically bind a name to an address
- In some cases the program must determine at run time what method to call

Overriding Methods in the Superclass

- A subclass inherits methods from a superclass
- Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass
- This is referred to as *method overriding*

Private Restriction

- A method can be overridden only if it is accessible
- Thus a private method cannot be overridden, because it is not accessible outside its own class
- If a method defined in a subclass is private in its superclass, the two methods are completely unrelated

Override

- A method is overridden when a child class creates a method of the same name and the same number and type of parameters
- The method of the child class will be used

Overload

- A method is overloaded when a child class creates a method of the same name but different type or number of parameters
- When a program calls the method, Java will select the method with the correct parameter type *(polymorphism)*

Override

```
class Plant {
    public void branch( double goat ) {
        System.out.println("Plant "+goat);
    }
}
class Tree extends Plant {
    public void branch( double cow ) {
        System.out.println("Tree "+cow);
    }
}

public class OverRide {
    public static void main(String[] x) {
        Tree pine = new Tree();
        pine.branch( 2 );
        pine.branch( 7.0 );
    }
}
```

Overload

```
class Plant {
    public void branch( double goat ) {
        System.out.println("Plant "+goat);
    }
}
class Tree extends Plant {
    public void branch( int cow ) {
        System.out.println("Tree "+cow);
    }
}

public class OverLoad {
    public static void main(String[] x) {
        Tree pine = new Tree();
        pine.branch( 2 );
        pine.branch( 7.0 );
    }
}
```

What is displayed?

```
class Plant {
    public void branch( double goat ) {
        System.out.print("Plant "+goat);
    }
}
class Tree extends Plant {
    public void branch( double cow ) {
        System.out.print("Tree "+cow);
    }
}

public class OverRide {
    public static void main(String[] x) {
        Tree pine = new Tree();
        pine.branch( 2 );
        pine.branch( 7.0 );
    }
}
```

A. Plant 5.0 Tree 3

B. Tree 3 Plant 5.0

C. Tree 3.0 Tree 5.0

D. Plant 3.0 Tree 5.0

What is displayed?

```
class Plant {
    public void branch( double goat ) {
        System.out.print("Plant "+goat);
    }
}
class Tree extends Plant {
    public void branch( int cow ) {
        System.out.print("Tree "+cow);
    }
}

public class OverLoad {
    public static void main(String[] x) {
        Tree pine = new Tree();
        pine.branch( 2 );
        pine.branch( 7.0 );
    }
}
```

A. Plant 5.0 Tree 3.0

B. Tree 3 Plant 5.0

C. Tree 3 Tree 5.0

D. Plant 3.0 Tree 5.0

@override and @overload

- The @override and @overload statements are similar to comments
- They specify that the following method is overridden or overloaded
- If you put @override or @overload before a method that is not overridden or overloaded, you will get an error

Polymorphism

- Subtype polymorphism allows a function to be written to take an object of a certain class `Parent`, but also work correctly if passed an object that belongs to a class `Child` that is a subclass of `Parent`
- Ad Hoc polymorphism allows functions to be overloaded so that the same function can take parameters of different types and perform differently

Subtype Polymorphism Example

```
public class Animal {
   String talk() {
      return "Grunt";
   }
}
class Tree extends Animal {
   String talk() {
      return "Meow!";
   }
}
class branch extends Animal {
   String talk() {
      return "Woof!";
   }
```

```
public class myProg {
   static void letsHear(Animal goat) {
      System.out.println( goat.talk() );
   }
   public static void main( … ) {
      Tree pine = new Tree();
      branch puppy = new branch();
      letsHear( pine );
      letsHear( puppy );
   }
}
```

Vtables

- In OO languages, methods have a virtual method table or vtable
- The vtable contains links to the methods that will be used by objects of a given class
- The vtable is created by the compiler based on the inheritance tree of the class

Dynamic Method Binding

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

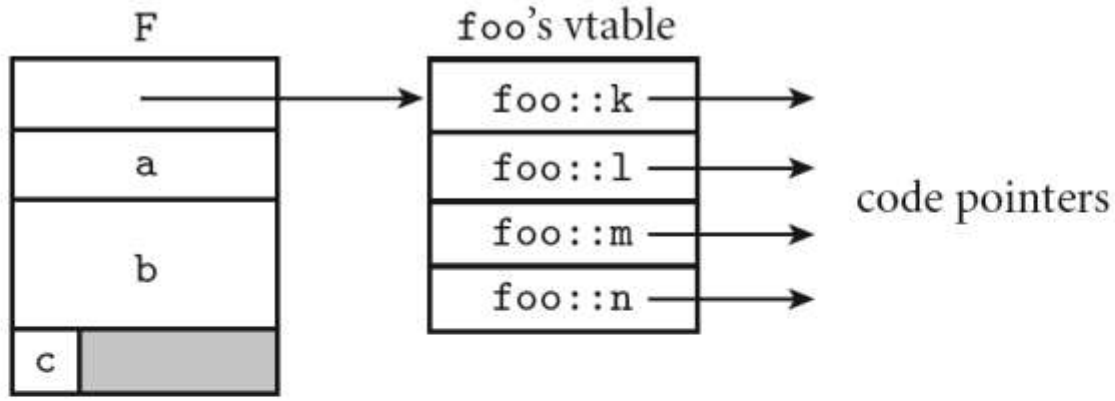

Figure 10.3 **Implementation of virtual methods.** The representation of object `F` begins with the address of the vtable for class `foo`. (All objects of this class will point to the same vtable.) The vtable itself consists of an array of addresses, one for the code of each virtual method of the class. The remainder of `F` consists of the representations of its fields.

from Programming Language Pragmatics by Michael Scott

ELSEVIER

Dynamic Method Binding

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

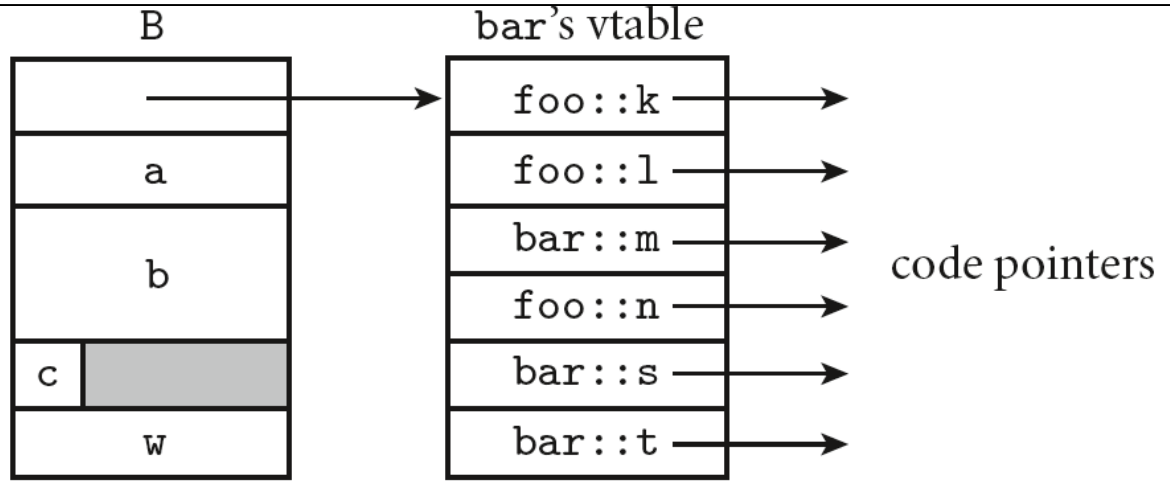

Figure 10.4 **Implementation of single inheritance.** As in Figure 10.3, the representation of object `B` begins with the address of its class's vtable. The first four entries in the table represent the same members as they do for `foo`, except that one—`m`—has been overridden and now contains the address of the code for a different subroutine. Additional fields of `bar` follow the ones inherited from `foo` in the representation of `B`; additional virtual methods follow the ones inherited from `foo` in the vtable of class `bar`.

from Programming Language Pragmatics by Michael Scott

ELSEVIER

Class Reflection

- Note that if you can query the type of an object, then you need to be able to get from the object to run-time type info
- The standard implementation technique is to put a pointer to the type info at the beginning of the vtable

ELSEVIER

Three Pillars of OO Programming

- Inheritance
- Encapsulation & Abstraction
- Polymorphism