Multiple Inheritance

The “Deadly Diamond”

- **Java™** only supports *single inheritance*
- *Multiple inheritance* is okay if the super classes (B and C) do not have a common ancestor (A)
  - This results in the “deadly diamond” inheritance hierarchy
- **All Java™ classes extend class Object** either directly or indirectly
  - *If Java™ allowed multiple inheritance, then all Java™ programs would exhibit the deadly diamond architecture*
Java™ and Multiple Inheritance

The problem with the “deadly diamond”

- Class D object
  - Contains a class B subobject
  - Contains a class C subobject
- Class B object
  - Contains a class A subobject
- Class C object
  - Contains a class A subobject
- Therefore, a class D object contains 2 class A subobjects
- Solving this problems (ala C++)
  - Results in inelegant code
  - Complex and inefficient compilers

Public Interfaces

Interface: another overloaded word

- A user (application programmer) uses, accesses, or interfaces an object through its public interface
- An object’s public interface includes its public
  - Methods
  - Data (often constants)
- The constructor, push, pop, and size are the Stack interface
**interface**

A partial replacement for multiple inheritance

- Permits a class to “reflect the behavior of [multiple] parents” even when the one “extends” has been used

- **An interface** defines a public interface or signature
  
  - Specifies method header or *signature* only
    - Method name
    - Return value type
    - Argument list
  
  - Method body is *not* defined

- Example
  ```java
  public interface ActionListener
  {
      public void actionPerformed(ActionEvent event);
  }
  ```

**Implementing Interfaces**

Using interfaces

- **An interface** is a contract
  
  - Compiler verifies that the implementing class overrides all *interface* methods (it is a compile time error if it doesn’t)

- **An interface** is a data type
  
  - Variables point to objects instantiated from implementing classes

- Example
  ```java
  public class Bar implements ActionListener
  {    ActionListener foo = new Bar();
      public void actionPerformed(ActionEvent event)
      {
          ...
      }
  }
  ```
interface vs Abstract Class

Comparing similar constructs

- **Similarities**
  - Specify `abstract` methods, which must be overridden elsewhere
  - Specify constants (data that is `public`, `static` and `final`)
  - Can be used as a generic type specifier that can reference any object instantiated from a class that implements that interface, which is useful in `upcasting`
  - Can participate in `polymorphism`
  - Can be the right hand operand of `instanceof`
  - Cannot instantiate either an abstract class or an interface

- **Differences**
  - Interfaces do not specify concrete methods
  - Interfaces do not specify instance variables
  - Interfaces do not contain anything that would form a subobject

Interface Summary

Key concepts

- **Interface**
  - Methods are abstract
    - The `abstract` keyword may be used but is superfluous (i.e., not required)
    - They do not have bodies
  - Data are `public`, `static`, `final`
    - The keywords may be used but are superfluous (i.e., not required)
    - They are constant and must be initialized

- **public interface name and file name must agree**
  - Non-public interfaces should also follow this naming convention
  - `public` interfaces can be implemented outside of the package
  - `friendly` interfaces can only be implemented within the package

- **A class can implement multiple interfaces**
  - State `implements` once
  - Specify the interfaces as a comma separated list of interface names
### Interface Example

**Interface syntax**

```java
public interface MyInterface
{
    public void mymethod(); // no method body
}
```

```java
public class IFexample implements MyInterface
{
    public void mymethod() // needed to compile
    {
        System.out.println("IFexample method");
    }
}
```

```java
public libraryService( )
{
    MyInterface IFobject = new IFexample(); // type specifier
    IFobject.mymethod(); // guaranteed
}
```

### Interface Example

“Library” or “server” code (see Sortable.java and SelectSort.java)

```java
public interface Sortable
{
    public int compare(Sortable otherObject);
}
```

```java
public static void sort(Sortable[] list)
{
    for (int bottom = list.length - 1; bottom >= 1; bottom--)
    {
        Sortable currentMax = list[bottom];
        int currentMaxIndex = bottom;
        for (int i = bottom - 1; i >= 0; i--)
        {
            if (currentMax.compare(list[i]) < 0)
            {
                currentMax = list[i];
                currentMaxIndex = i;
            }
        }
        if (currentMaxIndex != bottom)
        {
            list[currentMaxIndex] = list[bottom];
            list[bottom] = currentMax;
        }
    }
}
```
class Widget implements Sortable
{
    int partNumber;
    public Widget(int pn)
    {
        partNumber = pn;
    }
    public int compare(Sortable otherObject)
    {
        if (partNumber < (Widget)otherObject.partNumber)
            return -1;
        else if (partNumber == (Widget)otherObject.partNumber)
            return 0;
        else
            return 1;
    }
}

Chapter 6 Slide 11 of 28

Interface Example Continued

“Application” or “client” code (see Main.java)

// Fills an array with Widgets. Each Widget has a part number, which
// is generated with a pseudo random number generator. The array of
// Widgets is sorted by part number with the selection sort algorithm.

public class Main
{
    public static void main(String[] args)
    {
        Widget[] list = new Widget[20];
        for (int i = 0; i < 20; i++)
            list[i] = new Widget((int)(Math.random() * 100));

        SelectSort.sort(list);
        for (int i = 0; i < 20; i++)
            System.out.println(list[i].getPartNumber());
    }
}

Chapter 6 Slide 12 of 28

Interface Example Continued

“Application” code continued (see Main.java)
An Aside: class Arrays

New with 1.2: java.util.Arrays

- static void sort(type[ ] a)
  - type can be any built-in type
    - The sorting algorithm is a tuned quicksort
    - This algorithm offers n*\log(n) performance

- static void sort(type[ ] a, int fromIndex, int toIndex)
  - fromIndex - the index of the first element (inclusive) to be sorted
  - toIndex - the index of the last element (exclusive) to be sorted
  - type can be object
    - The sorting algorithm is a modified mergesort
    - This sort is guaranteed to be stable: equal elements will not be reordered as a result of the sort
    - guaranteed n*\log(n) performance
    - All elements in the array must implement the Comparable interface

- static int binarySearch(type[ ] a, type key)
  - The array must be sorted into ascending order according to the natural ordering of its elements
  - If the array contains multiple elements with the specified value, there is no guarantee which one will be found
  - Returns the index if key is found, otherwise -(insertion point) - 1
    - Returns a value greater than or equal to 0 if key is found
    - Returns a value less than 0 if not found
  - type may be any built-in type
    - must implement Comparable interface

- interface Comparable { public int compareTo(Object o); }
Callback Methods

Another use for interfaces

- Replacement for function pointers
- Think of Timed & Timer as library code; Clock is written later

```java
interface Timed {
    public void tick();
}

class Clock implements Timed {
    Timer t;
    public Clock() {
        t = new Timer(this);
    }
    public void tick() {
        /* update time display */
    }
}

class Timer extends Thread {
    Timed client;
    public Timer(Timed t) {
        client = t;
    }
    public void run() {
        while (true) {
            sleep(1000); // pause 1 sec
            client.tick();
        }
    }
}
```

Callback Illustrated

An association relationship

```
Timer t = new Timer(this);
t.start();
void tick() {
    
}
```

Clock

```
Timer(Timed t) {
    client = t;
}
public void run() {
    client.tick();
}
```

Timer
Extending Interfaces

Interface inheritance

- Interfaces cannot extend classes
- One interface can extend another interface
- Any class that implements an interface which extends another interface, must define the methods in both interfaces

```java
interface Swappable extends Sortable {
    void swap(Sortable x, Sortable y);
}
```

Copying Objects

The assignment operator (see Driver.java, line 11)

- Copies the value (address) stored in T to noCopy
- T and noCopy point to the same object

```java
Triangle T = new Triangle(0,0, 0,100, 200,200);
Triangle noCopy = T;
if (noCopy == T)
    System.out.println("noCopy == T");
else
    System.out.println("noCopy != T");
```
clone() and Cloneable

Copying objects

- clone() is defined in the Object class
  - protected Object clone()
  - clone() performs a bitwise copy of an object
  - clone() must usually be overridden

- interface Cloneable is defined in java.lang
  - It is a tagging interface
    - It does not define any methods or constants
    - Its purpose is to support instanceof and upcasting (i.e., be a type specifier)

- Implementing Cloneable indicates that it is legal to clone an object
  - Throws a CloneNotSupportedException otherwise
  - Overridden clone may call inherited version: super.clone()
  - Overridden clone may clone individual instance objects (examples follow on the next slides)

Cloning Simple Objects

Bitwise copy

- clone performs a bitwise copy
- Person has simple attributes
  - clone copies the values
  - each object has its own, private copies of the attributes
  - Similar to the C++ copy constructor

```java
public class Person implements Cloneable {
    int height;
    float salary;
}
```

```java
Person p = new Person(71, 50000);
Person copy = (Person)p.clone();
```
Cloning Objects: Shallow Copy

- clone performs a bitwise copy
- Triangle has three Points: v0, v1, and v2
  - clone copies the values (addresses) stored in v0, v1, and v2
  - T and copy have the same three Points

```
Triangle T = new Triangle(0,0, 0,100, 200,200);
Triangle copy = (Triangle)T.clone();
```

Cloning Objects: Deep Copy

- clone copies the object
- Each object attribute must also be cloned
- Must override clone()

```
public Object clone() throws CloneNotSupportedException {
    Triangle copy = (Triangle)super.clone();
    copy.v0 = (Point)v0.clone();
    copy.v1 = (Point)v1.clone();
    copy.v2 = (Point)v2.clone();
    return copy;
}
```
**Cloning Arrays**

See CloneArray.java

- All arrays implement Cloneable
- Elements must implement Cloneable for deep copy

Shallow copy: \[ \text{Point[]} \text{copy} = (\text{Point}[]) \text{points.clone}(); \]
Deepp copy add: \[
\text{for (int } i = 0; i < 10; i++)
\text{copy[i] = (Point)points[i].clone();}
\]

**Inner Classes**

Embedded scope (added at Java™ 1.1)

- An inner class is defined *inside* of another class or method
- An inner class has
  - Full access to the implementation of the object that created it, including *private* features
  - An implicit association (*this* reference) to the object that created it
- Anonymous inner classes are useful for defining callbacks
- Inner classes can be hidden from other classes in the same package (avoiding name exhaustion or conflicts)
- Inner classes may be used to deal with events (implement adapter classes)
- Inner classes may be private (other classes always have either package-"friendly"–or public visibility)
Why Use Inner Classes?

Prevents duplicate data and huge constructor parameter lists

- Inner classes are not essential
  - Make instance variables public
  - Copy data to second object
- Outer class has many, dynamic variables
  - Inner class needed for inheritance
  - Inner class needs outer class variables
  - Outer class variables change frequently

```
class Foo extends Bar {
    private int x;
    private int y;
    private int z;
    private A a = new A();
}
```

```
class A extends B {
    public method M( ) {
        uses x, y, & z;
    }
}
```

Inner Class Example

Used in event handling

```
public class OuterClass {
    private String id = "OuterClass";
    // private OuterClass instance variable

    private class InnerClass {
        String name = "InnerClass";
        // InnerClass instance variable

        public void demo( ) {
            System.out.println(id); // access OuterClass instance variable
            System.out.println(name); // access InnerClass instance variable
        }
    } // InnerClass

    static public void main(String args[ ])
    { OuterClass OC = new OuterClass( ); }

    public OuterClass( ) {
        InnerClass IC = new InnerClass( );
        IC.demo( );
    } // OuterClass
```
Inner Classes Defined In Methods

Inner classes defined inside of methods have access to all of the data and methods of the enclosing class through the this reference

Inner classes defined inside of methods have access to all of the final variables and parameters of the method

The final keyword is allowed with local variables and parameters (added at 1.1 with inner classes)

It is possible that an object instantiated from the inner class could survive the method call

- The inner class object is given copies of the local variables and parameters
- To insure that the copies are up to date (i.e., that the method has not changed the values), the inner class object can only reference final variables

Anonymous Inner Classes

Used for event handling

- Is not given a name (which is why it is anonymous)
- Class can only be instantiated once
- Defined within a method of the enclosing class
- May access
  - Class variables and methods from the enclosing class
  - final data and parameters of the enclosing method
- Usage should not be more complicated than the example below (from chap.07/awt/event3.java)

```java
addWindowListener ( new WindowAdapter( )
{   public void windowClosing(WindowEvent e)
{ System.exit(0); }
} );
```