

Multiple Inheritance

- **What is an Interface?**
- **Syntax**
- **Implementing Interfaces**
- **Inheritance**
- **Exceptions**
- **Complexity**

- **Multiple inheritance - inheriting from more than one class.**
 - * **Java doesn't do this.**

What is an Interface?

- **Interface**: Structure that is similar to a class, but has no data or method bodies associated with it.
 - * Collection of abstract methods and static final data.
- Similar to abstract classes but contain no method code.
- Interfaces are supported by classes that are said to “implement” the interfaces.
- Interfaces are Java’s substitute for multiple inheritance.
 - * A single Java class can implement several Java interfaces.
 - * That class doesn’t inherit any code from the interfaces so not true inheritance.
- Interfaces are used to outline desired behavior while leaving the specific implementation of the behavior to the classes.

Syntax

- **The general syntax for an interface declaration is very similar to class definition:**

public interface InterfaceName *extends* InterfaceList

- **Italicized words are optional.**
- **Public interfaces may be used outside of current package.**
 - * **Lack of public keyword limits use of interface to current package.**
- **Convention says to capitalize first letters of interface name.**
 - * **Same convention as for classes.**
- **All methods and variables are implicitly public.**

- **Attributes are treated like constants in an interface.**
 - * **Attributes are always final and static.**
 - * **Attributes must be initialized.**

```
public interface Widget {  
  
    // Constant to hold manufacturer's name  
    static final String MANUFACTURER = "ACME";  
  
    public String getName();  
    public String getDescription();  
}
```

- **An interface method consists of a declaration.**

Syntax

- **Since interfaces will typically be implemented by more than one class, it is important to carefully consider return types and parameter lists.**
 - * **Try to maximize reuse.**
- **Method declarations must end with a semicolon.**
- **All methods in interfaces are public by default.**
 - * **It is illegal to use any other standard modifier.**
- **All attributes are public, final, and static by default whether explicitly stated or not.**
 - * **Good practice to use the modifiers to remind yourself (and others) of this fact.**

Implementing Interfaces

- **Classes that implement an interface must override all methods declared in that interface.**
 - * **Failure to do so results in an abstract class.**
- **Since all interface methods are public, all overriding class methods must also be public.**
- **Implementing class may only apply native and abstract modifiers to interface methods.**
- **Be careful when implementing interface methods.**
 - * **Incorrect number or type of parameters will overload, not override interface method.**

```
public class Gadget implements Widget{
```

```
public Gadget(String nm, String desc)
{ name = new String(nm);
  description = new String(desc); }
```

```
public String getName()
{ return name; }
```

```
public String getDescription()
{ return description; }
```

```
public void setName(String nm)
{ name = new String(nm); }
```

```
public void setDescription(String desc)
{ description = new String(desc); }
```

```
// ***** HIDDEN IMPLEMENTATION *****
```

```
protected String name;
protected String description;
}
public class Setup
```

```
{
public static void main(String[] arg)
{ // Create a new gadget
  Gadget myGadget = new Gadget("FunKey", "A funkey new key");

  // Call the displayInfo method to display the information.
  displayInfo(myGadget);
} // End the main function.

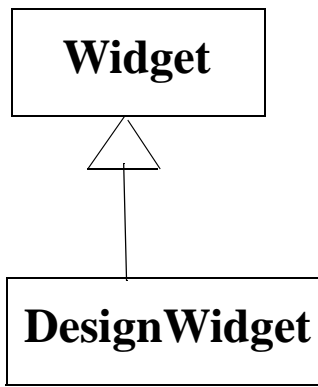
// Method that uses the Widget interface as a parameter and
// allows the gadget information to be printed.
public static void displayInfo(Widget w)
{
  System.out.println("The manufacturer is " + w.MANUFACTURER);
  System.out.println("The name of the widget is " + w.getName());
}
}
```

The manufacturer is ACME
The name of the widget is FunKey

Inheritance

- **Interfaces may extend other interfaces.**
- **Child interfaces still may not define method bodies of parent or own methods.**
- **Any class that implements the child interface must define the body of all methods of both the parent and child interface.**
- **Interfaces CANNOT extend classes.**

*** Why not?**



```
public interface DesignWidget extends Widget {  
  
    // DesignWidgets have the name and description inherited from  
    // Widget as well as methods below.  
    public int getHeight();  
    public void setHeight(int h);  
}
```

```
public class GadgetDesign implements DesignWidget{
```

```
    public GadgetDesign(String nm, String desc)  
    { name = new String(nm);  
      description = new String(desc);  
    }
```

```
// Methods to carry out methods defined on Widget  
    public String getName()  
    { return name; }
```

```
public String getDescription()  
{ return description; }
```

```
public void setName(String nm)  
{ name = new String(nm); }
```

```
public void setDescription(String desc)  
{ description = new String(desc); }
```

```
// Methods to carry out methods defined on DesignWidget
```

```
public int getHeight()  
{ return height; }
```

```
public void setHeight(int h)  
{ height = h; }
```

```
// ***** HIDDEN IMPLEMENTATION *****
```

```
protected String name;  
protected String description;  
protected int height;  
}
```

```
public class Setup
{
    public static void main(String[] arg)
    {
        // Create a new design gadget and set its height.
        GadgetDesign myDesign = new GadgetDesign("FunKey", "A funkey new key");
        myDesign.setHeight(4);

        // Call the displayInfo method to display the information.
        displayInfo(myDesign);
    } // End the main function.

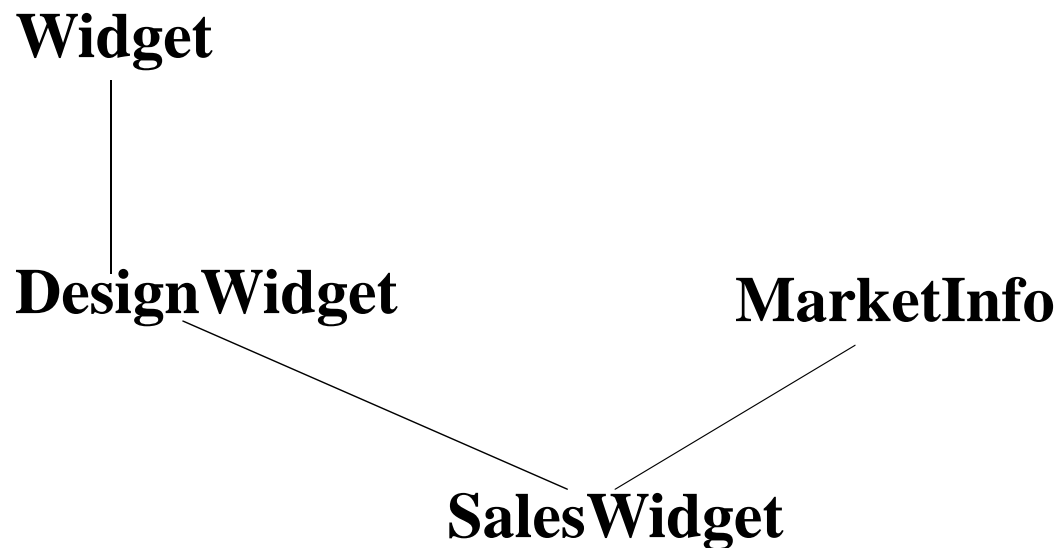
    // Method that uses the Widget interface as a parameter and
    // allows the gadget information to be printed.
    public static void displayInfo(DesignWidget d)
    {
        System.out.println("The manufacturer is " + d.MANUFACTURER);
        System.out.println("The name of the widget is " + d.getName());
        System.out.println("The gadget's height is " + d.getHeight());
    }
}
```

The manufacturer is ACME

The name of the widget is FunKey

The gadget's height is 4

- **Interfaces also allow us to simulate multiple inheritance:**



```
public interface Widget {
```

```
static final String MANUFACTURER = "ACME";  
public String getName();  
public String getDescription();  
}
```

```
public interface DesignWidget extends Widget {  
    public int getHeight();  
    public void setHeight(int h);  
}
```

```
public interface MarketInfo {  
    public double unitCost();  
    public double profitMargin();  
}
```

```
public interface SalesWidget extends DesignWidget, MarketInfo {  
    public float maxDailyProduction();  
}
```

```
public class SalesGadget implements SalesWidget{
```

```
public SalesGadget(String nm, String desc)
{ name = new String(nm);
  description = new String(desc);
}
```

```
// Methods to carry out methods defined on Widget
```

```
public String getName()
{ return name; }
```

```
public String getDescription()
{ return description; }
```

```
// Methods to carry out methods defined on DesignWidget
```

```
public int getHeight()
{ return height; }
```

```
public void setHeight(int h)
{ height = h; }
```

```
// Methods to carry out methods defined on MarketInfo
```

```
public double unitCost()  
{ return cost; }
```

```
public double profitMargin()  
{ return retail - cost; }
```

```
// Methods to carry out methods defined on SalesWidget
```

```
public float maxDailyProduction()  
{ return production; }
```

```
// Methods in addition to those defined on the interfaces.
```

```
public void setName(String nm)  
{ name = new String(nm); }
```

```
public void setDescription(String desc)  
{ description = new String(desc); }
```

```
public void setCost(double amt)  
{ cost = amt; }
```

```
public void setRetail(double amt)
```

```
{ retail = amt; }
```

```
public void setProduction(float prod)  
{ production = prod; }
```

```
// ***** HIDDEN IMPLEMENTATION *****
```

```
protected String name;  
protected String description;  
protected int height;  
protected double cost;  
protected double retail;  
protected float production;  
}
```

```
public class Setup
```

```
{  
public static void main(String[] arg)  
{  
    // Create a new design gadget and set its height.  
    SalesGadget mySales = new SalesGadget("FunKey", "A funkey new key");  
    mySales.setHeight(4);  
    // Note that these methods are not defined in the interface.  
    mySales.setCost(100.00);  
    mySales.setRetail(200.00);  
    mySales.setProduction((float)1000.00);  
  
    // Call the displayInfo method to display the information.  
    displayInfo(mySales);  
  
} // End the main function.
```

// Method that uses the Widget interface as a parameter and

```
// allows the gadget information to be printed.
public static void displayInfo(SalesWidget d)
{
    System.out.println("The manufacturer is " + d.MANUFACTURER);
    System.out.println("The name of the widget is " + d.getName());
    System.out.println("The gadget's height is " + d.getHeight());
    System.out.println("The unit cost is " + d.unitCost());
    System.out.println("The profit margin is " + d.profitMargin());
    System.out.println("The maximum daily production is " +
        d.maxDailyProduction());
}
}
```

The manufacturer is ACME
The name of the widget is FunKey
The gadget's height is 4
The unit cost is 100.0
The profit margin is 100.0
The maximum daily production is 1000.0

Inheritance

- **Note that inheritance was not used in the class structure, just in the interfaces.**
- **We could have used some class inheritance if we had wished.**
 - * **Class inheritance is frequently used in conjunction with interface inheritance.**
- **Note that use of the class is not restricted to methods defined on the interface.**
 - * **Use of object is based on type of variable used to reference it.**
 - * **SalesWidget variable can only use interface methods.**
 - * **SalesGadget variable can use all methods defined on SalesGadget.**

- **One of the most important features of an interface is that it can be used as a data type.**
 - * e.g., parameters and return types
- **An interface variable can be used just as you would any class.**

Exceptions

- **Interfaces may be defined to throw exceptions.**

```
interface Example {  
    public int getPrice(int id)  
        throws java.lang.RuntimeException;  
}
```

- **Rules for overriding methods that throw exceptions:**
 1. **New exception list may only contain exceptions listed in the original exception list or subclasses of the originally listed exceptions.**
 2. **Original exception list is automatically assigned to the new method so new exception list doesn't need to contain any exceptions.**
- **In general, interface's exception list (not class') determines**

exceptions that can and cannot be thrown.

*** Implementing classes cannot add exceptions to interface exception list.**

*** Methods defined on a class only (not on interface) can have any exceptions on exception list.**

Complexity of Algorithms

- **What resources (time, space, I/O) does the algorithm require?**
- **Criteria used when selecting between algorithms that perform the same basic task include:**
 - * **Speed of execution**
 - * **Complexity of data structures used**
 - * **Memory requirements**
- **How do each of these grow as the data set the algorithm works on gets larger?**
- **Suppose the size of the problem can be represented by the number N**
- **Typically N would be the number of data elements to be processed.**

- **Find a formula that expresses, say, the speed of the algorithm in terms of N .**
- **This complexity is often called the *order* of the algorithm and denoted using “big oh” notation $O(N)$.**
- **Example 1: Linear Search**
 - * **Search an array A of N elements for a particular element x**
 - * **For the moment assume that it is known that x is in the array.**

```
int i;
```

```
//empty
```

```
for ( i=0; x != item[i]; i++);
```

- * **On average, how many comparisons must be made?**
- * **What are the best and worst case scenarios?**

Complexity of Algorithms

- **Example 2: Binary Search**
- **Search a sorted array A of N elements for a particular number x .**
- **Natural language version of the algorithm:**
 - 1. Find the midpoint of the array. Then what?**

Constants don't matter (much)

- If one algorithm has order $O(1000n)$, another order $O(n*n)$ which would you use on large sets of data?
- In general, for sufficiently large numbers, the differences among $\log(n)$, n , $n*n$, $\exp(n)$ is so much greater than the difference, say, between n and $c*n$, where c is a constant, that we normally discount constant multiples in measuring complexity. Thus

$$O(n) = O(1000n) = O(n/1000) = \textit{linear complexity}$$

Only the Dominant Term Matters

- What is the difference between an algorithm with complexity $O(n^2)$ and an algorithm of complexity $O(n^2 + 3n + 1)$?
- For large n , $n^2 + 3n + 1 < n^2 + 3n^2 + n^2 < 5n^2$
- Since constants don't matter we consider both algorithms to be of order $O(n^2)$
- Note that from calculus or discrete mathematics we have, for positive integer constant c ,

$$\log_2 n \ll n^c \ll 2^n$$

Complexity of Algorithms

O(n) linear	O(log n) log	O(n log n) nlogn	O(n²) quadratic	O(n³) cubic	O(2ⁿ) exponential
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
31	5	160	1024	32768	4294967296

Array complexity

- **What is the time complexity of *inserting* an element into index 4 of an array?**
 - * **Note that this overwrites the current element at index 4.**
 $a[4] = 'a';$
- **What is the time complexity of *accessing* the element at index 3 of an array? `System.out.println(a[3]);`**
- **Suppose an array, *roster*, of names is to be maintained in sorted order. What is the average time complexity for inserting a new element in the appropriate location without losing any data?**