

A Second Look At Java

Subtype Polymorphism

Person x;

- Does this declare **x** to be a reference to an object of the **Person** class?
- Not exactly—the *type* **Person** may include references to objects of other classes
- Java has subtype polymorphism

Outline

- 15.2 Implementing interfaces
- 15.3 Extending classes
- 15.4 Extending and implementing
- 15.5 Multiple inheritance
- 15.6 Generics

Interfaces

- A method *prototype* just gives the method name and type—no method body
- An interface in Java is a collection of method prototypes

```
public interface Drawable {  
    void show(int xPos, int yPos);  
    void hide();  
}
```

Implementing Interfaces

- A class can declare that it implements a particular interface
- Then it must provide **public** method definitions that match those in the interface

Examples

```
public class Icon implements Drawable {  
    public void show(int x, int y) {  
        ... method body ...  
    }  
    public void hide() {  
        ... method body ...  
    }  
    ...more methods and fields...  
}
```

```
public class Square implements Drawable, Scalable {  
    ... all required methods of all interfaces implemented ...  
}
```

Why Use Interfaces?

- An interface can be implemented by many classes:

```
public class Window implements Drawable ...  
public class MousePointer implements Drawable ...  
public class Oval implements Drawable ...
```

- Interface name can be used as a reference type:

```
Drawable d;  
d = new Icon("i1.gif");  
d.show(0,0);  
d = new Oval(20,30);  
d.show(0,0);
```

Polymorphism With Interfaces

```
static void flashoff(Drawable d, int k) {  
    for (int i = 0; i < k; i++) {  
        d.show(0,0);  
        d.hide();  
    }  
}
```

- Class of object referred to by **d** is not known at compile time
- It is some class that **implements Drawable**, so it has **show** and **hide** methods that can be called

A More Complete Example

- A **Worklist** interface for a collection of **String** objects
- Can be added to, removed from, and tested for emptiness

```

public interface Worklist {
    /**
     * Add one String to the worklist.
     * @param item the String to add
     */
    void add(String item) ;

    /**
     * Test whether there are more elements in the
     * worklist: that is, test whether more elements
     * have been added than have been removed.
     * @return true iff there are more elements
     */
    boolean hasMore() ;
}

```

```
/**
 * Remove one String from the worklist and return
 * it.  There must be at least one element in the
 * worklist.
 * @return the String item removed
 */
String remove();
}
```

Interface Documentation

- Comments are especially important in an interface, since there is no code to help the reader understand what each method is supposed to do
- **Worklist** interface does not specify ordering: could be a stack, a queue, or something else
- We will do an implementation as a stack, implemented using linked lists

```

/**
 * A Node is an object that holds a String and a link
 * to the next Node. It can be used to build linked
 * lists of Strings.
 */
public class Node {
    private String data; // Each node has a String...
    private Node link;   // and a link to the next Node

    /**
     * Node constructor.
     * @param theData the String to store in this Node
     * @param theLink a link to the next Node
     */
    public Node(String theData, Node theLink) {
        data = theData;
        link = theLink;
    }
}

```

```

/**
 * Accessor for the String data stored in this Node.
 * @return our String item
 */
public String getData() {
    return data;
}

/**
 * Accessor for the link to the next Node.
 * @return the next Node
 */
public Node getLink() {
    return link;
}
}

```

```

/**
 * A Stack is an object that holds a collection of
 * Strings.
 */
public class Stack implements Worklist {
    private Node top = null; // top Node in the stack

    /**
     * Push a String on top of this stack.
     * @param data the String to add
     */
    public void add(String data) {
        top = new Node(data, top);
    }
}

```

```

/**
 * Test whether this stack has more elements.
 * @return true if this stack is not empty
 */
public boolean hasMore() {
    return (top!=null);
}

/**
 * Pop the top String from this stack and return it.
 * This should be called only if the stack is
 * not empty.
 * @return the popped String
 */
public String remove() {
    Node n = top;
    top = n.getLink();
    return n.getData();
}
}

```


A Test

```
Worklist w;  
w = new Stack();  
w.add("the plow.");  
w.add("forgives ");  
w.add("The cut worm ");  
System.out.print(w.remove());  
System.out.print(w.remove());  
System.out.println(w.remove());
```

- Output: The cut worm forgives the plow.
- Other implementations of **Worklist** are possible: **Queue**, **PriorityQueue**, etc.

Outline

- 15.2 Implementing interfaces
- **15.3 Extending classes**
- 15.4 Extending and implementing
- 15.5 Multiple inheritance
- 15.6 Generics

More Polymorphism

- Another, more complex source of polymorphism
- One class can be derived from another, using the keyword **extends**
- For example: a class **PeekableStack** that is just like **Stack**, but also has a method **peek** to examine the top element without removing it

```

/**
 * A PeekableStack is an object that does everything a
 * Stack can do, and can also peek at the top element
 * of the stack without popping it off.
 */
public class PeekableStack extends Stack {

    /**
     * Examine the top element on the stack, without
     * popping it off. This should be called only if
     * the stack is not empty.
     * @return the top String from the stack
     */
    public String peek() {
        String s = remove();
        add(s);
        return s;
    }
}

```

Inheritance

- Because **PeekableStack** extends **Stack**, it inherits all its methods and fields
- (Nothing like this happens with interfaces—when a class implements an interface, all it gets is an obligation)
- In addition to inheritance, you also get polymorphism

```
Stack s1 = new PeekableStack();  
PeekableStack s2 = new PeekableStack();  
s1.add("drive");  
s2.add("cart");  
System.out.println(s2.peak());
```

Note that **s1.peak()** is not legal here, even though **s1** is a reference to a **PeekableStack**. It is the static type of the reference, not the object's class, that determines the operations Java will permit.

Question

- Our **peek** was inefficient:

```
public String peek() {  
    String s = remove();  
    add(s);  
    return s;  
}
```

- Why not just do this?

```
public String peek() {  
    return top.getData();  
}
```

Answer

- The **top** field of **Stack** is **private**
- **PeekableStack** cannot access it
- For more efficient **peek**, **Stack** must make **top** visible in classes that extend it
- **protected** instead of **private**
- A common design challenge for object-oriented languages: designing for reuse by inheritance

Inheritance Chains

- A derived class can have more classes derived from it
- All classes but one are derived from some class
- If you do not give an **extends** clause, Java supplies one: **extends Object**
- **Object** is the ultimate base class in Java

The Class **Object**

- All classes are derived, directly or indirectly, from the predefined class **Object** (except **Object** itself)
- All classes inherit methods from **Object**:
 - **toString**, for converting to a **String**
 - **equals**, for comparing with other objects
 - **hashCode**, for computing an **int** hash code
 - etc.

Overriding Inherited Definitions

- Sometimes you want to redefine an inherited method
- No special construct for this: a new method definition automatically overrides an inherited definition of the same name and type

Overriding Example

```
System.out.print(new Stack());
```

- The inherited **toString** just combines the class name and hash code (in hexadecimal)
- So the code above prints something like:
Stack@b3d
- A custom **toString** method in **Stack** can override this with a nicer string:

```
public String toString() {  
    return "Stack with top at " + top;  
}
```

Inheritance Hierarchies

- Inheritance forms a hierarchy, a tree rooted at **Object**
- Sometimes inheritance is one useful class extending another
- In other cases, it is a way of factoring out common code from different classes into a shared base class

```

public class Label {
    private int x,y;
    private int width;
    private int height;
    private String text;
    public void move
        (int newX, int newY)
    {
        x = newX;
        y = newY;
    }
    public String getText()
    {
        return text;
    }
}

```

```

public class Icon {
    private int x,y;
    private int width;
    private int height;
    private Gif image;
    public void move
        (int newX, int newY)
    {
        x = newX;
        y = newY;
    }
    public Gif getImage()
    {
        return image;
    }
}

```

Two classes with a lot in common—but neither is a simple extension of the other.

```

public class Graphic {
    protected int x,y;
    protected int width,height;
    public void move(int newX, int newY) {
        x = newX;
        y = newY;
    }
}

```

```

public class Label
    extends Graphic {
    private String text;
    public String getText()
    {
        return text;
    }
}

```

```

public class Icon
    extends Graphic {
    private Gif image;
    public Gif getImage()
    {
        return image;
    }
}

```

Common code and data have been factored out into a common base class.

A Design Problem

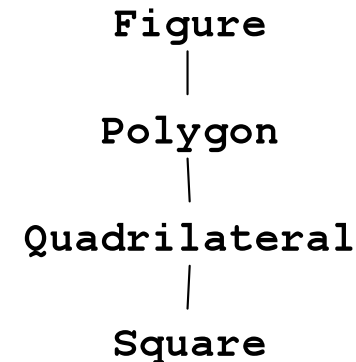
- When you write the same statements repeatedly, you think: that should be a method
- When you write the same methods repeatedly, you think: that should be a common base class
- The real trick is to see the need for a shared base class early in the design, before writing a lot of code that needs to be reorganized

Subtypes and Inheritance

- A derived class is a subtype
- From Chapter Six:

A subtype is a subset of the values, but it can support a superset of the operations.

- When designing class hierarchies, think about inheritance of functionality
- Not all intuitively reasonable hierarchies work well for inheriting functionality



Outline

- 15.2 Implementing interfaces
- 15.3 Extending classes
- 15.4 Extending and implementing
- 15.5 Multiple inheritance
- 15.6 Generics

Extending And Implementing

- Classes can use **extends** and **implements** together
- For every class, the Java language system keeps track of several properties, including:

A: the interfaces it implements
B: the methods it is obliged to define
C: the methods that are defined for it
D: the fields that are defined for it

Simple Cases For A Class

- A method definition affects C only
- A field definition affects D only
- An **implements** part affects A and B
 - All the interfaces are added to A
 - All the methods in them are added to B

A: the interfaces it implements

B: the methods it is obliged to define

C: the methods that are defined for it

D: the fields that are defined for it

Tricky Case For A Class

- An **extends** part affects all four:
 - All interfaces of the base class are added to A
 - All methods the base class is obliged to define are added to B
 - All methods of the base class are added to C
 - All fields of the base class are added to D

A: the interfaces it implements

B: the methods it is obliged to define

C: the methods that are defined for it

D: the fields that are defined for it

Previous Example

```
public class Stack implements Worklist {...}  
public class PeekableStack extends Stack {...}
```

■ **PeekableStack** has:

- A: **Worklist** interface, inherited
- B: obligations for **add**, **hasMore**, and **remove**, inherited
- C: methods **add**, **hasMore**, and **remove**, inherited, plus its own method **peek**
- D: field **top**, inherited

A Peek At **abstract**

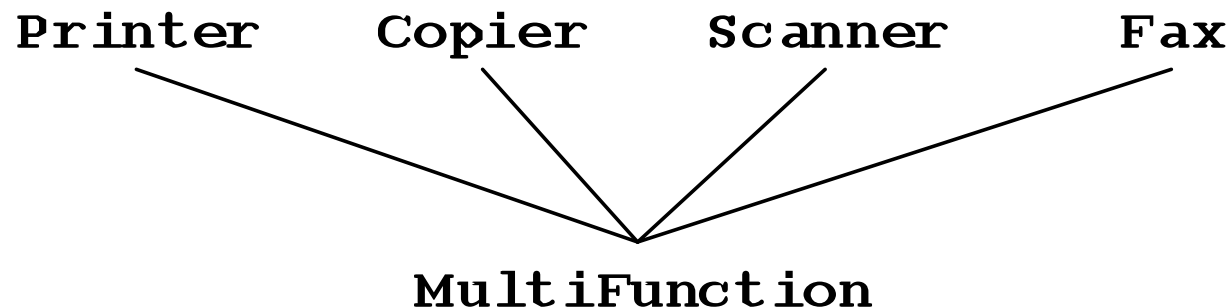
- Note that C is a superset of B: the class has definitions of all required methods
- Java ordinarily requires this
- Classes can get out of this by being declared **abstract**
- An **abstract** class is used only as a base class; no objects of that class are created
- We will not be using **abstract** classes

Outline

- 15.2 Implementing interfaces
- 15.3 Extending classes
- 15.4 Extending and implementing
- **15.5 Multiple inheritance**
- 15.6 Generics

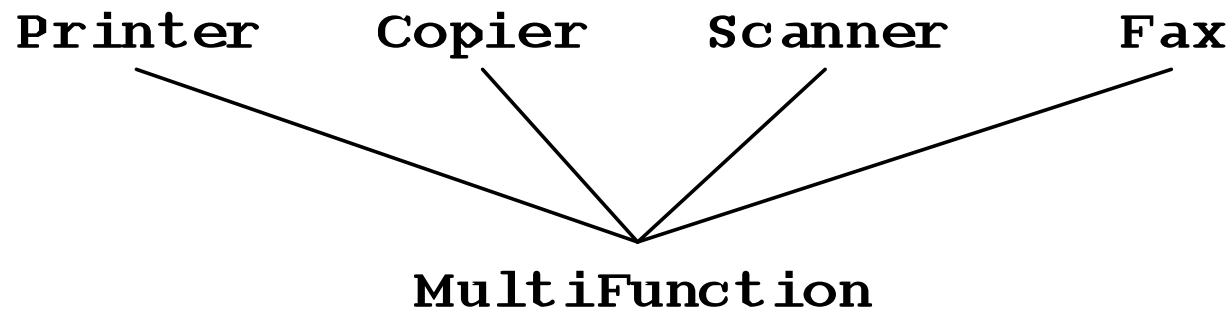
Multiple Inheritance

- In some languages (such as C++) a class can have more than one base class
- Seems simple at first: just inherit fields and methods from all the base classes
- For example: a multifunction printer



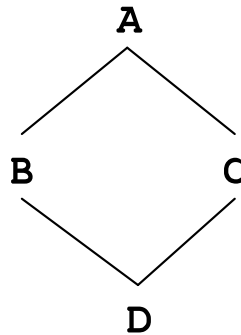
Collision Problem

- The different base classes are unrelated, and may not have been designed to be combined
- **Scanner** and **Fax** might both have a method named **transmit**
- When **MultiFunction.transmit** is called, what should happen?



Diamond Problem

- A class may inherit from the same base class through more than one path



- If **A** defines a field **x**, then **B** has one and so does **C**
- Does **D** get two of them?

Solvable, But...

- A language that supports multiple inheritance must have mechanisms for handling these problems
- Not all that tricky
- The question is, is the additional power worth the additional language complexity?
- Java's designers did not think so

Living Without Multiple Inheritance

- One benefit of multiple inheritance is that a class can have several unrelated types (like **Copier** and **Fax**)
- This can be done in Java by using interfaces: a class can implement any number of interfaces
- Another benefit is inheriting implementation from multiple base classes
- This is harder to accomplish with Java

Forwarding

```
public class MultiFunction {
    private Printer myPrinter;
    private Copier myCopier;
    private Scanner myScanner;
    private Fax myFax;

    public void copy() {
        myCopier.copy();
    }
    public void transmitScanned() {
        myScanner.transmit();
    }
    public void sendFax() {
        myFax.transmit();
    }
    ...
}
```

Outline

- 15.1 Implementing interfaces
- 15.2 Extending classes
- 15.3 Extending and implementing
- 15.4 Multiple inheritance
- **15.5 Generics**

An Early Weakness in Java

- Previous **Stack** example: a stack of strings
- Can't be reused for stacks of other types
- In ML we used type variables for this:

```
datatype 'a node =  
    NULL |  
    CELL of 'a * 'a node;
```

- Ada and C++ have something similar, but Java originally did not

Living Without Generics

- Until the 2004 additions to Java, programmers had to work around this
- For example, we could have made a stack whose element type is **Object**
- The type **Object** includes all references, so this would allow any objects to be placed in the stack

```

public class ObjectNode {
    private Object data;
    private ObjectNode link;
    public ObjectNode(Object theData,
        ObjectNode theLink) {
        data = theData;
        link = theLink;
    }
    public Object getData() {
        return data;
    }
    public ObjectNode getLink() {
        return link;
    }
}

```

Similarly, we could define **ObjectStack**
(and an **ObjectWorklist** interface) using
Object in place of **String**

Weaknesses

- No compile-time type checking on the element types

```
ObjectStack s1 = new ObjectStack();  
s1.add("hello");  
s1.add(s1);
```

- Usually, that kind of code is an error, and programmers want the compiler to help identify it

Weaknesses

- To recover the type of the stacked object, we will have to use an explicit *type cast*:

```
ObjectStack s1 = new ObjectStack();  
s1.add("hello");  
String s = (String) s1.remove();
```

- This is a pain to write, and also inefficient
- Java checks at runtime that the type cast is legal—the object really is a **String**

Weaknesses

- Primitive types must first be stored in an object before being stacked:

```
ObjectStack s2 = new ObjectStack();  
s2.add(new Integer(1));  
int i = ((Integer) s2.remove()).intValue();
```

- Again, laborious and inefficient
- **Integer** is a predefined wrapper class
- There is one for every primitive type

True Generics

- In 2004, Java was extended
- It now has parameterized polymorphic classes, interfaces, methods, and constructors
- You can tell them by the distinctive notation using angle brackets after the type name

```
interface Worklist<T> {  
    void add(T item);  
    boolean hasMore();  
    T remove();  
}
```

formal *type parameter* defines
a type variable **T** inside this
interface

uses of the type variable **T**

```
Worklist<String> w;  
...  
w.add("Hello");  
String s = w.remove();
```

actual *type parameter* when
we use the type; now **add** takes
a **String**, and **remove** returns
a **String**

```
public class Node<T> {  
    private T data;  
    private Node<T> link;  
    public Node(T theData, Node<T> theLink) {  
        data = theData;  
        link = theLink;  
    }  
    public T getData() {  
        return data;  
    }  
    public Node<T> getLink() {  
        return link;  
    }  
}
```



```

public class Stack<T> implements Worklist<T> {
    private Node<T> top = null;
    public void add(T data) {
        top = new Node<T>(data, top);
    }
    public boolean hasMore() {
        return (top!=null);
    }
    public T remove() {
        Node<T> n = top;
        top = n.getLink();
        return n.getData();
    }
}

```

Using Generic Classes

```
Stack<String> s1 = new Stack<String>();  
Stack<Integer> s2 = new Stack<Integer>();  
s1.add("hello");  
String s = s1.remove();  
s2.add(1);  
int i = s2.remove();
```

- Notice the coercions: **int** to **Integer** (“boxing”) and **Integer** to **int** (“unboxing”)
- These also were added in 2004