Object Oriented Programming – I

Review-Part 5

MODULE – XI:

Multiple Inheritance

Some object-oriented languages such as C++ allow a class to have more than one parent. For example, class C can extend both classes A and B. This means C will inherit methods from both A and B. What if both A and B have a method m(), and inside class C we call method m()? Whose method is called, A's or B's?

This becomes really tricky if A and B both extend a super class X. Suppose X has method m() and A and B both override m().

Now consider: X x = new C();
x has current type X so x.m() is a legal method call.
But the true type C determines which version of m is called. So which version is it, A's or B's?

In Java, each class can only extend one other class. Thus, there is no ambiguity. We can get something like multiple inheritance by allowing a class to implement more than one interface.

Why Do We Need Multiple Inheritance?
Consider the Shape hierarchy from the last lecture. What if we want to add a RegularPolygon to the hierarchy?

A RegularPolgon is-a Polygon
A Square is-a RegularPolygon
A Rectangle is not a RegularPolygon
A Hexagon is a RegularPolygon

There is no way to correctly place RegularPolygon in the tree hierarchy for the classes and have all the "is-a" relations correct. Instead, let use create RegularPolygon as an interface. Now, we can have classes like Octagon and Hexagon extend Polygon, and have Octagon, Hexagon and Square "implement" RegularPolygon.

Java Interfaces

A Java interface is a non-primitive type like a class, but it cannot contain instance methods, fields, or constructors.

Specifically, an interface can contain:

- static final fields
- static public nested types
- static methods
- abstract public instance methods (method stubs)
- starting in Java 9, private methods will be allowed

The main purpose is to contain public abstract methods.

To create an interface:

```
public interface MyInterface {
        void methodStub1(int x, int y);
        int methodStub2();
}
```
(since all methods stubs in an interface have to be public and abstract, we can drop the "**public abstract**" part)

An interface can extend 0 or more interfaces. (You place "extends ..." just like you do with a class, and for multiple interfaces, you separate them with commas.)

To use an interface in a class: a class can implement 0 or more interfaces.

```
public class MyClass implements MyInterface {
        // here MyClass inherits the abstract methods methodStub1 and methodStub2
        // because a class cannot have abstract methods, we must override these method stubs with
methods with bodies
}
```

Here is another example:

```
public class Square extends Rectangle implements RegularPolygon {
  ....
}
```

If you will implement more than one interface, separate the interface names with commas. Implementing an interface is -exactly- like extending a class. So, a class that implements an interface inherits all the methods (or in this case method stubs) of the interface.

A class (that is not abstract) cannot contain method stubs. So, the class must override each of the method stubs from the interface.

Note how this simplifies the multiple inheritance problem above. A class may inherit a method stub from more than one parent, but it can only inherit an instance method that contains a body from its class parent.

NOTE: Students tend to get mixed up on "extends" vs. "implements". They mean exactly the same thing. Java uses "**implements**" for the specific situation where we are indicating that a class as an interface as a supertype. All other situations where we are indicating a supertype, we use "**extends**".

Interface Example
We created the RegularPolygon interface

public interface RegularPolygon {

The first thing we added is a method to compute the area of a regular polygon. There are two ways we could do that. Since interfaces allow static methods, we did so:

```
public static double areaOfRegularPolygon(RegularPolygon p) {
  ... a formula using p.getNumberAngles() and p.getSideLength()
}
```

We could also make a non-static abstract method, but give it a default method body:

```
public default double getArea() {
   ... a formula using this.getNumberAngles() and this.getSideLength()
}
```

Note one benefit of interfaces: we can create a method that only works on RegularPolygons by specifying that as an input. Since interfaces are a non-primitive type, we can make the current type of an object be a RegularPolygon as long as the true type implements the RegularPolygon interface.

Next, note that we need the instance methods getNumberAngles and getSideLength. How do we make sure that RegularPolygon has them? We add them as abstract methods.

Since all instance methods of an interface MUST be public and abstract, the Java style is to drop those two modifiers (but you can keep them if you want).

```
public interface RegularPolygon {
    int getNumberAngles(); // we can omit public abstract
    double getSideLength(); // we can omit public abstract

    // interfaces can have static methods
    public static double areaOfRegularPolygon(RegularPolygon p) {
      ... a formula to compute the area using getNumberAngles and getSideLength ....
    }

    //interfaces can have default methods
    public default double getArea() {
      ... a formula using this.getNumberAngles() and this.getSideLength()
    }
}
```

Now, anything that is a RegularPolygon will inherit the getNumberAngles and getSideLength method stubs and (if not an abstract class), must override them to make them normal methods.

Notice one trick I used. Polygon already has getNumberAngles as a normal method. So, anything that implements RegularPolygon will already get a normal getNumberAngles method inherited from Polygon and so will not have to override the method stub.

The classes that implement RegularPolygon will still have to override the getSideLength method stub.

```
public class Hexagon extends Polygon implements RegularPolygon {

    private double sideLength;

    public Hexagon(double sideLength) {
      super(6);
      this.sideLength = sideLength;
    }

    public double getSideLength() {  ← this method is required by RegularPolygon so I decided to name my getter/setters appropriately
       return sideLength;
    }

    public void setSideLength(double sideLength) {
       this.sideLength = sideLength;
    }

    public double perimeter() {       ← this method is required by Shape
      return getNumberAngles() * getSideLength();
    }

    public double area() {            ← this method is require by Shape, but we have a static method in the interface for the area
       return RegularPolygon.areaOfRegularPolygon(this);    ← remember that "this" is a variable storing the object this method is acting on
   -OR-
       return this.getArea();
    }
}
```

MODULE – XII:
LinkedLists:

Arrays: A collection of variables of the same type stored in contiguous memory.

Benefits: Very fast access to any arbitrary element.

Downside: Can't change size after created and can't insert/delete without needing to shift values. To change the size, we must create a new array and copy all the data over.

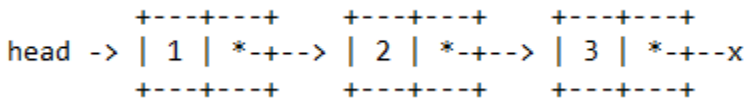
Linked Lists: A data structure that stores values of the same type

Benefits: Can easily increase or decrease size as needed. Can easily insert or delete at any location.

Downside: Fast access to only a few elements.

A linked list is formed of "nodes". Each node contains an element and a pointer to the next node of the list.

The first node of the list is called the "head" of the list.



NOTE: LinkedList is a class of the Java API. However, we are going to implement our ow class so you can understand how linked lists work.

The LinkedList class of the API is identical to our class except that each box also has an array pointing to the box that comes before it. Such a linked list is sometimes called a "double linked list".

What type should the element be? Whatever type we want to store in the list. This seems to imply that we will need to create a different linked list class for each possible type we want to store, but starting in Java 5, we can specify a "**generic type**".

A generic type is a place holder (usually a single capital letter) that indicates that the type will

be specified later.

```
public class LLNode<T> {
```

indicates that there is a generic type associated with LLNode. When we create an instance of the LLNode, we will specify what the type T is.

(A class can have as many generic types as you wish associated with it. For example, if had a class called Box with two generic types, we would declare the class as: public class Box<K,T> { where K and T are the two letters used as placeholders for the two generic types.)

Inside LLNode, we can use T just like any other type:

```
public class LLNode<T> {
        private T element;
        private LLNode<T> next;
        public LLNode(T element, LLNode<T> next) {
                this.element = element;
                this.next = next;
        }
        public T getElement() {
                return element;
        }
```

```
	public LLNode<T> getNext() {
		return next;
	}
	public void setNext(LLNode<T> next) {
		this.next = next;
	}
}
```

The use of LLNode<T> in the setNext and the constructor and the next field forces Java to require that the LLNode that next points to must hold the same type as this node holds. That way we can force every node of the list to hold the same type. This was impossible before generics. Instead, we would need to either specify a separate list class for each type or we would have to store Object and then use lots of **instanceof** expressions and typecasts to enforce that only one type of Object is stored.

When we create an instance of LLNode, we will specify what the type will be. For example, we can store JFrame:

```
LLNode<JFrame> node = new LLNode<JFrame>(new JFrame(), null);
```

or we can store String:

```
LLNode<String> node2 = new LLNode<String>("Hi", null);
```

Types and generics:

The generic type is used by the compiler so it only affects the current type. The compiler makes sure that the types that you specify match.

new LLNode<String>("Hi", null); ← legal! "Hi" is type String and that matches the specified generic.

new LLNode<String>(new JFrame(), null); ← ILLEGAL! JFrame() is not a String. The generic was specified to be String, and so the element's type must match.

LLNode<String> node = new LLNode<Object>("Hi", null); ← ILLEGAL! The type of variable node is LLNode<String> so only LLNode's that have the generic specified as String can be assigned to the variable.

LLNode<Object> node = new LLNode<String>("Hi", null); ← ILLEGAL! The type of variable node is LLNode<Object> so only LLNode's that have the generic specified as Object can be assigned to the variable. It does not matter that String is narrower than Object!

WHY IS THIS NOT ALLOWED?

If this were allowed, we would have a big problem because the following line is legal because all the types match:

```
node.setNext(new LLNode<Object>(new JFrame(), null));
```

node has the generic specified as Object, and so the setNext takes values of type LLNode<Object>.

However, node was storing an LLNode<String>, and now that LLNode<String> has an LLNode<Object> assigned to it's next field. A violation of the types!

Why didn't node know it was storing an LLNode<String>? Because the generics only apply to the current type (the compile-time type). When the code is running, Java does not

know what the current type is, only the true type. The true type does not include the generic. The true type is just LLNode.

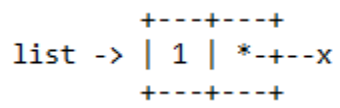
MODULE – XIII:

Continuing with Linked Lists

Recall the LLNode class:

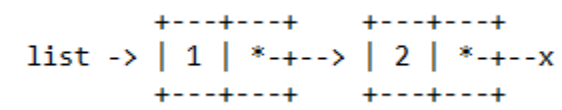
```
public class LLNode<T> {
        private T element;
        private LLNode<T> next;
        public LLNode(T element, LLNode<T> next) {
                this.element = element;
                this.next = next;
        }
        public T getElement() {
                return element;
        }
        public LLNode<T> getNext() {
                return next;
        }
        public void setNext(LLNode<T> next) {
                this.next = next;
        }
}
```

Now, let us create the linked list:



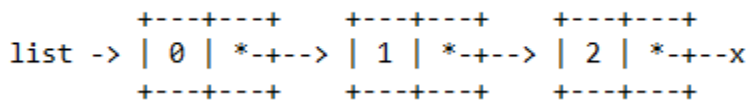
LLNode<Double> list = new LLNode<Double>(1.0, null);

Now, let us add 2 after 1. Creating the node is the same, but we have to change the value of the first node's next pointer to point to the second node:



list.setNext(new LLNode<Double>(2.0, null));

Now, let us add 0 before 1. We need to create a box that points to 1. Where is 1's box currently stored? In list. Then we must change the value of list to point to the new box:



list = new LLNode<Double>(0, list)

A final example: Place 1.5 between 1 and 2 in the above list.

We must be careful to do things in the right order, or we will break the list.

1) Create a new node for 1.5.
2) Move the next pointer for the 1.5 node to point to the 2 node.
3) Move the next pointer for the 1 node to point to the 1.5 node.

If we did step 3 before step 2, we would lose all access to the 2 node. What is the name of the next pointer of 1? list.next.next

To do step 2, we need to change the new's next pointer to list.next.next, i.e. what the 1's box is pointing at.

To do step 3, we need to change list.next.next to point to the new box.

We can do this in one line of carefully organized code!

Step 1) LLNode<Double> newNode = new LLNode<Double>(1.5, null);

Step 2) newNode.setNext(list.getNext().getNext());

Step 3) list.getNext().setNext(newNode);

Or in one line:

list.getNext().setNext(new LLNode<Double>(1.5, list.getNext().getNext());

Abstract Data Types

An abstract data type is a data structure that guarantees certain behaviors to the user, but it keeps its implementation details hidden.

By hiding the implementation details, we are able to change them if we discover a better way to do things without breaking the code that uses the data structure. We also can prevent code that uses the data structure from accidentally breaking the data structure.

Examples:

Strings are an ADT. You are guaranteed certain behavior such as accessing a character from a location and appending strings, but you are not told how they are implemented (though they are probably implemented as an array of chars).

JFrames are an ADT. You are guaranteed certain behavior such as changing its size, displaying it on the screen, but you do not know exactly how the Java Swing developers chose to implement the window.

We will create the LinkedList as an abstract data type. The list is going to be a list of LLNodes, but we will keep the details away from the users

of the LinkedList so that code that uses the LinkedList can not accidentally break the list.

Creating a LinkedList class

A LinkedList will store a list of LLNodes. The only field we need is to store the node that is the first node of the list. We call this the "first" or "head" of the linked list.

There are several ways we can implement the linked list. We decided that if a list is empty, its front should be a null pointer, to indicate that there are no nodes in the list. Another option would be to create a special node that acts as a "caboose" to the linked list. No implementation technique is wrong as long as the list works properly. If we correctly create the LinkedList as an abstract data type, we should be able to switch between implementations an code that uses the LinkedList class will still operate exactly the same.

Note that the LinkedList will need to use a generic to specify the type that will be stored in the list, and we want each LLNode in the list to use the same generic.

```
public class LinkedList<T> {
        private LLNode<T> firstNode;
        public LinkedList() {
                firstNode = null;
        }
```

Now, let use crete a method to add an element to the front of the list.

```
        public void addToFront(T element) {
                firstNode = new LLNode<T>(element, firstNode);
        }
```

If we want to allow polymorphism and extending for this class, we should use getter/setter methods for firstNode, but we don't want to make them public because that would let any code using the LinkedList (rather than any code that "is" a LinkedList) to need to understand how the head of the linked list works. In particular, the getter/setter methods return an LLNode. Having code outside the LinkedList know about LLNodes would violate the "keep implementation details hidden" nature of an abstract data type.

The solution is to make the getter/setter methods "protected". Recall that protected means it can be used in this class or any class that extends this class. (Why would private not work? Hint: getter/setter's are used so to making extending the class easier.)

```
public void addToFront(T element) {
        setFirst(new LLNode<T>(element, getFirst());
}
```

How about testing if a list is empty?

```
public boolean isEmpty() {
        return getFirst() == null;
}
```

How about removing an element at the the front?

```
public T removeFromFront() {
```

We need to move the front from the current node to the next node. But before we do that, we should save the value stored in the front so we can return it at the end.

```
public T removeFromFront() {
        T save = getFirstNode().getElement();
        setFirst(getFirstNode().getNext();
         return save;
}
```

But, what if the list is empty, then the front will be null, and we will get a NullPointerException!

It is not a good idea to throw a NullPointerException because that will not mean anything to the programmers who are using our code.

The problem is that the list is empty. That should not have to deal with a null value that is specific to our implementation of the linked list.

We will still throw an exception, but now we will throw a more meaningfully named exception: NoSuchElementException

****
* Short Aside on Exceptions. We will cover them in more detail later.
* An exception is another way to return from a program. It is "less elegant" than the normal return so we use it for special situations such as errors
* You have already seen some exceptions: NullPointerException, ArrayIndexOutOfBoundsException
* Exception is just a class of Java so it can be created like any other class, but to "return" with an exception value, we "throw" the exception
* There are 2 kinds of exceptions: checked exceptions and unchecked exceptions
* - The difference is that for checked exceptions, we have to be explicit about how our code is handling the exception.
* For example, if our code is going to throw the exception, we must indicate that in the method header.
* **********

The NoSuchElementException is in the java.util class, and it is a checked exception so we must indicate that the method may throw it in the method header

```
public T removeFromFront() throws NoSuchElementException {
        if (isEmpty())
                throw new NoSuchElementException();
        else {
                T save = getFirstNode().getElement();
                setFirst(getFirstNode().getNext());
                return save;
        }
}
```

MODULE – XIV:

Restricting Generic Types and Wildcards: Suppose we have the following method:

```
public static void displayFrames(LinkedList<JFrame> frameList) {
        for (JFrame frame : frameList)
                frame.setVisible(true);
}
```

This will only work on LinkedLists that have the generic type set to JFrame. What if we want to restrict the generic?

```
LinkedList<GeometricFrame> list2 = new LinkedList<GeometricFrame>();

list2.addToFront(new GeometricFrame());

list2.addToFront(new GeometricFrame());
```

However, if we call displayFrames(list2) we get a compile error because you can't convert LinkedList<GeometricFrame> to LinkedList<JFrame>.

Our solution is to restrict the generic type of LinkedList to be anything that is JFrame or narrower.

E extends JFrame : When you declare generic type E, you restrict E to be JFrame or narrower

? extends JFrame : When you use the "don't care" wildcard, you say the generic type must be JFrame or narrower

? super JFrame : When you use the "don't care" wildcard, you say the generic type must be JFrame or wider

You can also extend generics:

E extends T
? extends T
? super T

Here are the two ways to write displayFrames. First, declaring a generic restricted to JFrame or narrower:

```
public static <E extends JFrame> void displayFrames(LinkedList<E> frameList) {
  for (JFrame frame : frameList)
    frame.setVisible(true);
}
```

Or using the "Don't care" wild card:

```
public static void displayFrames(LinkedList<? extends JFrame> frameList) {
  for (JFrame frame : frameList)
    frame.setVisible(true);
}
```

In both cases we declare frame to be type JFrame. We know whatever is stored in frameList is JFrame or narrower.

Iterables and Generic Types

Loops and Abstract Data Types

We can try to think of everything that a program that uses our LinkedList will want to do, but that is impossible.

Instead, other programs will need to write their own loops to run through the linked list.

This creates a problem: to run through the linked list requires having a node pointer, but if our linked list is to be a abstract data type, we do not want to require outside classes to have to deal with the linked list implementation details.

Java provides a pair of interfaces that let us provide a means for other code to loop through our linked list while still hiding the implementation details. These interfaces are Iterable and Iterator.

Iterable: indicates we can loop through the data stored in an instance

Iterator: the instance that performs the routines needed to do the loop

The Iterator interface

The Iterator interface is used to generalize the idea of a loop.

Basically, something that is type Iterator (implements Iterator) can be used as a loop. The Iterator interface has two non-default instance methods (that means we have to override these):

boolean hasNext() → returns true if there are more elements in the list

T next() → returns the next element in the list, and "iterates" so the next time we call next(), we get the next element of the list

(Notice from the API page that the Iterator interface takes a generic, so when we implement it, we need to specify the generic. Here we will specify the generic of Iterator by creating a generic in our class that implements the Iterator.)

Assuming these two methods have been properly overridden, we can now write a loop using just the Iterator interface.

```
Iterator<?> iterator =  ?????                // pretend we have some way to get an iterator

        while (iterator.hasNext()) {

                System.out.println(iterator.next());

}
```

Note that the iterator lets us write loops for an abstract data type.

If we create an Iterator for our LinkedList class, the programmer writing the loop only needs to know how Iterators work and not any details about the LinkedList nodes.

Recall how we write a loop for a LinkedList:

```
        LLNode<?> nodeptr = list.getFront();      // assuming getFront is public!
        while (nodeptr != null) {
                System.out.println(nodeptr.getElement());    // print each element
                nodeptr = nodeptr.getNext();
         }
```

To write the Iterator class, we need to go back to our loop using LLNodes. The key parts of the loop are:

1) initalization: We have to set up the loop. Here the code is "LLNode<T> nodeptr = getFirstNode()"

2) condition: We need to indicate when loop should continue and when it should stop: "nodeptr != null"

3) retrieval: We need to get the next element from the list "nodeptr.getElement()"

4) increment: We need to move to the next element in the list "nodeptr = nodeptr.getNext()"

Now, we need to create a class for our linked list that implements the Iterator interface.

What should the hasNext do? The condition of the loop (nodeptr != null).

What should the next do? Both the retrieveal (return nodeptr.getElement()) and the increment (nodeptr = nodeptr.getNext())

That leaves the initialization step (nodeptr = getFirstNode()). Where should that happen? The constructor!

```
        public class LinkedListIterator<T> implements Iterator<T> {
```

```
    private LLNode<T> nodeptr;

    public LinkedListIterator(LLNode<T> firstNode) {
      nodeptr = firstNode;
    }

    @Override
    public boolean hasNext() {
      return nodeptr != null;
    }

    @Override
    public T next() {
      T element = nodeptr.getElement();
      nodeptr = nodeptr.getNext();
      return element;
    }
  }
}
```

Notice the use of the generic. Iterator takes a generic (we can see that by looking at the API page for Iterator.)

So, we declare a generic in the LinkedListIterator header (the LinkedListIterator<T>), and now that we have a generic T declared, we use it when we implement Iterator (Iterator<T>). That forces the value returned by the next method to have the type of what we are iterating over. (The type that is stored in the linked list.)

(You may notice that we violated the OO-rules of using getter/setter methods. I chose not to do that so that the connection between the Iterator methods and the linked list loop is easy to see. We probably should add in getter/setter methods. Otherwise, creating a class that extends LinkedListIterator will be more challenging to get correct.)

However, the API for Iterator says that we need to throw a NoSuchElementException if next() is called when there are no more elements in the linked list.

We did not have time to add that in lecture, but it is easy to add an if statement to see if the nodeptr is null, and if it is throw the exception.

Now that we created the a class that implements Iterator, how do we connect that class to the LinkedList class? By using the Iterable interface.

The Iterable Interface

Iterable is an interface of the API.

The Iterable type represents an object that contains data that we can loop (or iterate) over.

By having the LinkedList class implement Iterable, we are indicating that we can iterate over the elements of the Linked List.

Every class that is an abstract data type and can store multiple elements should implement the Iterable interface.

Notice (from the API page) that the Iterable interface takes a generic. We will use the same generic that is stored in the LinkedList.

The Iterable interface has 1 (non-default) method:

```
Iterator<T> iterator()
```

This method returns the iterator for the abstract data type. Again, we are going to make sure that the generic used by iterator matches the generic stored in the linked list. And how do we do that? By having the Iterable interface use exactly the same generic.

```
public class LinkedList<T> implements Iterable<T> {
```

Now we have to override the iterator method inherited from Iterable:

```
@Override
public Iterator<T> iterator() {
// we need to return an appropriate object that implements Iterator
}
```

What should the iterator method return? An instance of the LinkedListIterator that we created above!

```
@Override
public Iterator<T> iterator() {
  return new LinkedListIterator<T>(getFirstNode());
}
```

Note that when we override a method, the name and parameter signature must be identical.

The return type, if non-primitive, is allowed to be narrower than the overridden method's return type. So we could also write:

```
@Override
public LinkedListIterator<T> iterator() {
  return new LinkedListIterator<T>(getFront());
}
```

Using the Iterable/Iterator interfaces:

Now, we can write a loop outside of the LinkedList class. (We could not before because the getFirstNode() method is protected.)

Ex:

```
LinkedList<String> list = new LinkedList<String>();
list.addToFront("Cleveland");
```

```
        list.addToFront("Cincinnati");
        list.addToFront("Columbus");


        Iterator<String> it = list.iterator();
        while (it.hasNext())
        System.out.print(it.next() + " ");

      System.out.println();
```

Foreach loops:

Foreach loops are a Java shortcut for Iterable classes and for arrays.

The form of a foreach loop is:

```
for (T i : Iterable<T>)
```

and it reads as "foreach type T in iterable"

For example, if list is a LinkedList<Integer>, we could have:

```
for (Integer i : list)
```

which reads as "foreach Integer in list"

Here is an example:

```
LinkedList<Integer> list = new LinkedList<Integer>();
list.addToFront(1);
list.addToFront(2);
list.addToFront(3);
for (Integer x : list) {
        System.out.print(x + " ");
}
```

Note that the foreach loop is just a shortcut. Java takes the foreach loop on Iterable and coverts it to use the iterator:

```
for (Double element : list)
```

is automatically converted by the compiler to:

```
for (Iterator<Double> it = list.iterator(); it.hasNext(); ) {
  Double element = it.next();
```

This was not covered in lecture, but the foreach loop also works with arrays eventhough array are not Iterable type.

```
double[] a = {1, 2, 3, 4, 5};
for (double x : a) {
  System.out.println(x * x + " ");
}
```

The foreach loop on array is also a shortcut that is automatically converted by the compiler to a normal for loop

```
for (String s : a)
```

is the same as

```
for (int index = 0; index < a.length; index++) {
  String s = a[index];
```

MODULE – XV:
REVIEW: Non-primitive Types: Classes, Abstract Classes, and Interfaces

Classes and abstract classes can form a "class hierarchy" that is a tree.

An abstract class can contain everything a class can contain plus abstract methods. You can't directly instantiate an abstract class.

An interface adds to the hierarchy, but outside of the "class hierarchy tree".

Interfaces can contain

- static methods
- static final fields
- static nested classes
- most importantly, public abstract instance methods

In Java 8, interfaces can have "**default**" method bodies for their abstract methods. When a class inherits the method stub / abstract method, if the class does not already have a method (or abstract method) of that name, the Java compiler will create the overloaded method for you automatically and textcopy the default method body into the code. This is NOT the same as inheriting the method.

Polymorphism Review of key ideas:

- An object is many types at the same time: its true type plus all types that the true type extends or implements.
- An object's true type is what it is when created (new MyClass() creates an object with true type MyClass)
- An object's current type is whatever it is currently being used as via typecasts or Java's type rules

The compiler verifies that you are using the current type properly

- You can only access fields available to the current type
- You can only call a method with a name and parameter signature that exists in the current type. The true type determines what method version is actually run.

Review of creating a hierarchy:

We can use classes, abstract classes, or interfaces.

Classes and abstract classes form the "main" hierarchy that is a tree.

Interfaces are used when we need to model ideas that do not fit into a tree.

1) "**is-a**": If A "is-a" B then A and B are non-primitive types where A extends or implements B. if A is not a B, then we do not relate A and B in a hierarchy.

2) "**has-a**": If A "has-a" B then B is a method in A. likewise, if A does not have a B, then we do not put B into A.

Generally, try to get most of your hierarchy into the class tree uses classes and abstract classes. This way you can take full advantage of instance method inheritance including instance fields. Where things don't fit nicely into a tree, you use interfaces. Using interfaces generally requires a little more coding to get the inheritance to work because interfaces can't have instance fields.

Review or rules for writing good O-O code:

1) Make fields private

2) Create public (or protected) getter/setter methods for the fields (if we want access to them)

3) Anywhere other than the constructor that we use the fields, we do not use the fields themselves, but we use the getter/setter methods.

Additional note: we avoid duplicating fields or storing data in multiple places. we avoid writing duplicate code

Why do we make fields private and have public (or protected) getter/setter methods?

So we can control how an object of a class behaves. We indicate that all direct access should be through the methods, and we can override the methods if we need to change behavior. We lose this control if the fields are not private.

Here is an example of making the fields of Rectangle protected instead of private. We can create a subclass of Square: public class BadSquare extends Square → this says that BadSquare "is-a" Square but since the fields are not private, BadSquare can change the fields directly

```
public void badIdea(double x, double y) {
        super.width = x;
        super.length = y;
}

BadSquare b = new BadSquare(3);
b.badIdea(3, 6);
b.getLength()  => 6
b.getWidth()   => 3
```

So b is not really a Square, we are not able to enforce that the width and length are the same.

This is broken because the fields are not private! Make the fields of Rectangle private and this behavior of BadSquare is impossible. Why must we use the getter/setter methods and not the fields in all methods (except the getter/setter methods themselves and the constructors)? Again, if we do not, we can break the code. Suppose a programmer decides to add more methods to Rectangle:

```
public void doubleWidth() {
        width *= 2;
}
```

Now, what happens to Square:

```
Square s = new Square(5);
s.doubleWidth()
s.getWidth()    => 10
s.getLength()   => 5
s.area()        => 25
```

Square stopped working! The solution is -not- to override doubleWidth inside Square. That would mean Square may have to override all methods of Rectangle. The problem is Rectangle. We need to use the getter/setter methods and not the fields. Then no matter what we do to Rectangle, Square will still work perfectly.

```
public void doubleWidth() {
        setWidth(2 * getWidth());
}
```

Now:

```
        Square s = new Square(5);
        s.area()      => 25
        s.doubleWidth();
        s.area()      => 100
        s.getWidth()  => 10
        s.getLength() => 10
```

This is the power of polymorphism! We don't have to rewrite Square with every change to Rectangle! But we get this power as long as we code correctly.