Java_Notes2

Lec17 Slide3

Interface

//1.Declare a Interface class

```
public interface Dancer {
/* All methods in an interface are public by default, so you don't need to mark them as
public. They're also abstract, so you don't have to provide any method bodies. Doing that
is the responsibility of any class that implements the interface. */
/*any class that implements an interface must provide bodies for the methods of the interface.*/

        int dance(); //these are a method that all subclasses will use
        String dancerName();
}
```

//2.Declare subclasses using implements keyword.

```
/* these classes can use completely different methods and fields inside */
public class Ninja implements Dancer {...public Pirate(String n, String s) {...}, public void
drink(String rumName) {...}             }
public class Zombie implements Dancer {...public Zombie(String name),public void
shuffle() {...},  …                     }
public class Pirate implements Dancer {...}
```

//3. These subclasses can now act as "Dancer." write class that takes advantage of this

```
public class DanceBattle {...public void fight() {...
    // A danceDuel() expects two parameters of type Dancer. Because
    // of polymorphism, Ninja, Zombie and Pirate objects can all do
    // double-duty as Dancers.
    danceDuel(jubei, gurg); danceDuel(silver, gurg); danceDuel(jubei, silver);..}...

    // Simulate a duel between two dancers.
    public void danceDuel(Dancer a, Dancer b) { … } …                    }
```

//4. Keep in mind

• The name of the interface (Dancer in our example) is a type, just like a class name.
• You can't instantiate an object of this type. It has no constructor.
• But you can have Dancer variables, which will store an object of any class as long as it implements Dancer.
• if you make a polymorphic assignment are the methods of "Dancer" - the common interface

Example

```
Dancer dancer = gurg; // Polymorphic
Dancer otherDancer = redbeard; // assignment

dancer.dance(); // VALID.
print(otherDancer.danceName());    // VALID.
dancer.attack(); // NOT VALID!
otherDancer.sail(); // NOT VALID!
```

Superclass/Subclass Assignments

```
super: ClassA          ClassA varA1 = varA;          //good
   ↑                          ClassA varA1 = varB; //good
Sub:    ClassB                ClassB varB1 = varB; //good
                       ClassB varB = (classB) varA ;        //must be casted ("Downcasting")
```

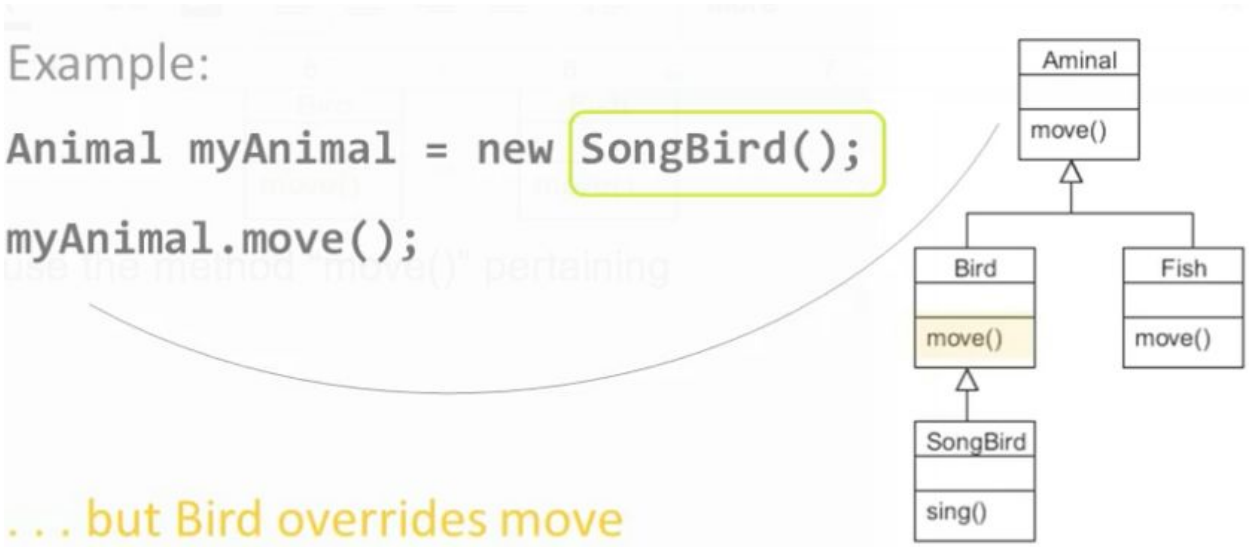
Examples

1. Animal a1 = new Bird();
   Bird myBird = (bird) a1; //works a1 is an instance of bird
2. Animal a2 = new Fish();
   Bird myBird = (Bird) a2; //fails a2 is NOT an isntance of Bird
3. Animal a3 = getsomeAnimal();
   Bird myBird = (Bird) a3; //might or might not work

```
//if we wanted to insure we get the correct object do…
        if( a3 instanceof Bird) Bird myBird = a3;
```

//dynamic binding

a. In this case since myAnimal is a "Bird()" type it will use the method "move()" pertaining to bird not animal.



b. The question is which move method. During run time its going to be established that the variable myAnimal is referencing an instance of type songBird but SongBird does not override move. so we look one level up. Bird overrides move so the move method of bird is going to be called.
http://www.youtube.com/watch?v=58Yhyg8Iw7A

slides

•For polymorphic method calls, the method is bound at runtime.

•During execution, the JVM examines the object and figures out its true type.

•Then it directs the method call to the right class.

•Thus, binding is dynamic – it happens at runtime.

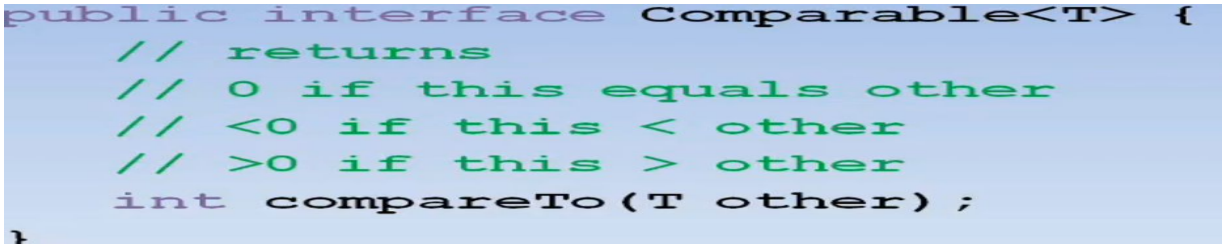
Static Binding

Normally, when you put a function call in your code, the compiler links it to the right function in the right class. The previous is “static binding”, however, for cases with interfaces real time information is not available at compile time.

• How? By looking at object type and matching the name to a method in that class.

Comparable

a. comparable is an interface that is built into java. found in java.util



```
public interface Comparable<T> {
    // returns
    // 0 if this equals other
    // <0 if this < other
    // >0 if this > other
    int compareTo(T other);
}
```

I//uses Arrays.sort(words)

```
//Use comparable to compare and sort object using Arrays.sort()
public class Person implements Comparable<Person> {
        String firstName;
        String lastName;
        int age;
        public Person(String first, String last, int a) { //constructor
                firstName = first; lastName = last; age = a; }
        public String toString() {
                return String.format("%s %s (%d years old)",
                                firstName, lastName, age);}
        // This sorts:
        // In decreasing order of age.
        // If ages are the same, ties are broken by reverse
        // lexicographical order of last names.
        // If last names are the same, ties are broken by first name.
        public int compareTo(Person other) {
                // If ages are different, we can just return the difference.
                // Otherwise, break ties by last name.
                if (age != other.age) return other.age - age;

                // If last names are the same, break ties by first name.
                if (other.lastName.compareTo(lastName) != 0)
                        return other.lastName.compareTo(lastName);

                return firstName.compareTo(other.firstName);
        }

}
```

```
import java.util.Arrays;

public class Sorting {

        public static void main(String[] args) {
                String[] words = "hello world what is going on today".split(" ");
                System.out.println("Before: " + Arrays.toString(words));
                Arrays.sort(words);
                System.out.println("After: " + Arrays.toString(words));
                // A bunch of Person objects. Try changing up the ages, last names
                // and so on.
                Person a = new Person("Jackie", "Chan", (int)1e9); Person
                b = new Person("Bruce", "Willis", (int)1e9);
                Person c = new Person("Arnold", "Schwarzenegger", (int)1e9);
                Person d = new Person("Chuck", "Norris", (int)(1e9 + 1));
                Person e = new Person("Jeff", "Dean", (int)(1e9 + 10));
                Person f = new Person("Chuck", "Dean", (int)(1e9 + 10));

                Person[] expendables = new Person[] {c, e, f, a, d, b};

                // Since Person implements Comparable, you can call Arrays.sort on
                // it.
                System.out.println("Before: " + Arrays.toString(expendables));
                Arrays.sort(expendables);
                System.out.println("After: " + Arrays.toString(expendables));
        }

}
```

//output
Before: [hello, world, what, is, going, on, today]
After: [going, hello, is, on, today, what, world]

Before: [Arnold Schwarzenegger (1000000000 years old), Jeff Dean (1000000010 years old), Chuck Dean (1000000010 years old), Jackie Chan (1000000000 years old), Chuck Norris (1000000001 years old), Bruce Willis (1000000000 years old)]
After: [Chuck Dean (1000000010 years old), Jeff Dean (1000000010 years old), Chuck Norris (1000000001 years old), Bruce Willis (1000000000 years old), Arnold Schwarzenegger (1000000000 years old), Jackie Chan (1000000000 years old)]

http://docs.oracle.com/javase/tutorial/collections/interfaces/order.html

• When comparing objects a and b, we call a.compareTo(b);
• This should return an int that is:
• Negative, if a < b in our preferred ordering • Positive, if a > b in our preferred ordering • 0, if a = b in our preferred ordering.
• Not symmetric!
a.compareTo(b) will be the *negative* of b.compareTo(a).
In general, you will always match the types:

```
class Blah implements Comparable<Blah> { ...
// The comparison function demanded by Comparable
public int compareTo(Blah other) { ...
} }
```

• You can imagine that a.compareTo(b) just returns a – b, as if the two were numbers.
• Thinking this way has two advantages:
• You can always remember which way the signs go.
• You can actually use subtraction to simplify your compareTo() code.

Under the hood?
• What's really happening in Arrays.sort()?
• When it compares arr[i] and arr[j], it does this:

```
Comparable<Person> a = arr[i]; // Polymorphic
 Comparable<Person> b = arr[j]; // Assignment
int c = a.compareTo(b);  ...
```

Lec19 Inheritance

- to inherit from a class use the **extends** keyword
- private members are inaccessible to the derived class but the private members are still part of the derived class so some special stuff happens.
- the inner core of a derived class is sometimes referred to as a super object and can be referenced using the **super** keyword. Its like "this" but i guess you can think of it like a super this.

example from code

inheritance constructors

```
public class Cat {

    public String name;
    private int age;
    private Color color;

    public Cat(String name, int age, Color color) {
        this.name = name;
        this.age = age;
        this.color = color;
    }
```

```
public class GrumpyCat extends Cat {

    // GrumpyCat usually has something to say,
    // unlike normal Cats.
    private String statement;

    // The constructor needs to set all fields, including the ones
    // inherited from Cat!
    public GrumpyCat(String name, int age, Color color, String statement) {
        // The first three fields are inherited from Cat. However,
        // we cannot reference this.age and this.color, because they
        // are private members of Cat. (this.name is fine because it is public)
        // However, we can use a constructor of Cat to set them - after all
        // initializing the state of Cat is what a Cat constructor is for.
        // We do this with super(), analogous to the way we used this() in Cat.java
        super(name, age, color);

        // After the superconstructor call initializes the
        // superobject, we can initialize the GrumpyCat-specific
        // fields in the normal way. Note that the superconstructor
        // call must always be the first line of code in a constructor.
        // The exception is if we're writing a default constructor, in
        // which case the super() call can be left out. It still happens,
        // but you don't need to do it by hand.
        this.statement = statement;
    }
```

• Remember: Always call the superconstructor first!
• There is one exception to this structure:
• If the parent class has a *default constructor* (i.e., no parameters) then you don't need to explicitly call the superconstructor.
• The default superconstructor will get called *implicitly* in such cases.
• Since no parameters are needed, calling it is simple.

• Why doesn't this happen in the non-default case? Because if parameters are needed, there's no way for the compiler to know what values to use.
• Either way, a superconstructor always gets called. No escaping that.
In Summary
• There is an is-a relationship:
• *Subclass_Object is-a Superclass_Object*
• Doesn't work the other way!

override

• The inherited *methods* can be *overridden* by simply providing an alternative method body in the subclass.
• Note that this overriding *hides* the original method.
• It can still be accessed within the derived class using super.
• Suppose you wanted to override the method speak() in LOLCat.
• If you accidentally typed seapk(), the compiler will not know that this was a typo.
• It'll just think you wanted a method called seapk().
• However, if you use @Override, it will search the parent class
and complain that it has no method by that name.

the object class
• For generality, Java makes sure that *every* class has a superclass.
• Even when a class doesn't explicitly extend anything.
• This universal base class is named Object, and itself doesn't
have any superclass.
• To add to the confusion, you can make an object of type
Object...
• Anything that doesn't extend a class explicitly is invisibly
extending Object.
• Object has a default (i.e., no parameters) constructor, which is invisibly called to satisfy the superconstructor call
• Object has a few methods inherited by every class.
• The methods toString() and equals() are among them.
• The default toString() output that looks like "Cat@5d0385c1" is the output of Object's toString().
• These methods can be overridden, of course. Every time you add a toString() method to a class, you're actually *overriding* the version inherited from Object.

Object Reference

```
Object o1 = new Object();
Object o2 = new Object();
o1 == o2
```

`o1 == o2` is pretty much equivalent to comparing two pointers in C/C++, yes. But there are a two main differences between references in Java and pointers in C/C++ that are quite important:

- Java references can't do pointer arithmetic: you can't "add 3" to a reference, you can *only* let it point to another (known) object
- Java references are stongly typed: you can't "reinterpret" what lies on the other end of a reference *unless*you reinterpret it as a type that *that object actually is*.

Also a short note about the word "reference": C++ has references that act quite differently from both pointers in C and references in Java (but I don't know enough about C++ to tell you the specifics).

polymorphism
• Because inheritance creates an is-a relationship, *every* class has an is-a relationship to Object.

• No matter what class you write, it can always be stored in an Object reference.
• Object x = “I’m a String!”;
• Object y = new Cat();
• Object z = new Scanner(System.in);
• Remember that is-a is a *transitive* relation, so if: • X is-a Y, and Y is-a Z
• Then X is-a Z

protected
protected methods are methods that are public to its’ subclasses but private to anyone else

Final, redux
• We’ve previously seen the final modifier used on variables, to indicate they are constants.
• It is possible to mark a *method* as final.
• This means that it cannot be overridden in any subclasses! • Similarly, a *class* can be marked as final too.
• This means that it cannot be extended to make any subclasses. • For example, String is final.

Abstract classes
• By adding the modifier abstract to the class declaration, we can make a class that behaves a little like an interface.
• It can have fields and normal methods.
• At the same time, it can define *abstract methods*, that have no bodies.
• Use the abstract modifier to indicate these methods.
• Interface methods are abstract, but we don’t bother with the keyword there – since they can’t be anything else.
• Bodies must be provided for them by any subclass. • Unless the subclass is abstract too, of course.

• We cannot instantiate abstract classes – *even if they have constructors!*
• Since they lack some method bodies, they are incomplete, and their objects cannot be built.
• Like interfaces, they provide a *framework* for building other classes.
• Unlike interfaces, they can provide some state and behavior.
• In the form of their fields and non-abstract methods, which are inherited by their subclasses.

```
//how to !instantiate abstract classes.
        public static void main(String[] args) {
                // This line won't compile, since you can't instantiate
                // an abstract class. Uncomment it to see the error.
//              Shape s = new Shape(3, 4);

                // We can't instantiate a Shape object, but we can
                // store descendants of Shape in a Shape object.
                // In this case, Circle. This is standard polymorphic
                // assignment - we saw the same thing in the Dancer
                // example with interfaces.
                Shape c = new Circle(10, 10, 5);
```

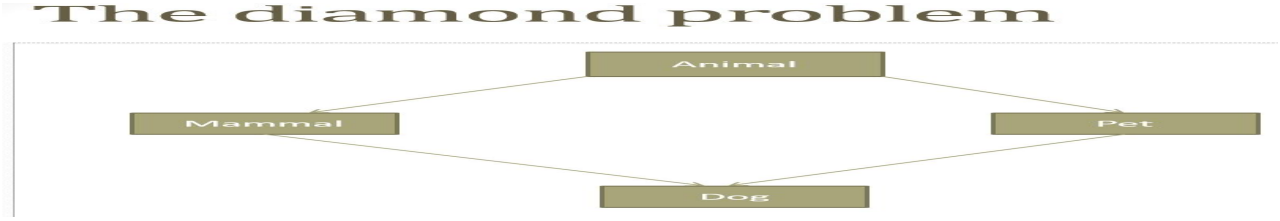
lecture21

Java Packages
You can have two classes with the same name, as long as they're in different packages

- java doesn't allow for multiple inheritance

errors
• Compile error:
• Syntax/grammar mistakes, program won't compile
• Easy to fix, just isolate the problem and take care of it.
• Runtime error:
• Program compiles and runs, but crashes
For handling runtime errors, Java provides the try-catch statement, which works with a specific class hierarchy.
• The Exception class, and its descendants
• Unavoidable in many cases.
• But can be *foreseen*, and therefore handled.
• Example: Expect a user to enter an invalid value, and check it.
• Program will crash if given empty file as input, handle that case.
• Logical error:
• Program compiles and runs, doesn't crash
• But does something we didn't want.
• Your program is doing exactly what you told it to.
• So tell it what you actually *meant* to say.

multiple inheritance



•Say Animal implements eat(), and Mammal and Pet both override it.

•Which version does Dog inherit?

•Things like this get troublesome, so Java doesn't allow multiple inheritance.

Exception Handling
• When an error happens, the method where it happened will *throw* an Exception object.
• This object needs to be *caught* and dealt with.
• Because this is an error case, normal program flow will stop when the exception is thrown.
• If the offending function doesn't contain an *exception handler*, the exception object will be passed to the parent function – the function that called it.
• And if that doesn't, then to the function that called it, and so on all the way up the call stack.
• This will happen until:
• It reaches a function with a matching exception handler and gets taken care of.

• It reaches main(), finds no exception handler, and gets thrown to the user, crashing the program.
• In general, when a handler catches an exception, it does something to deal with.
• Try something different, retry the same old thing, tell the user something, etc.
• Sometimes, the handler may just rethrow the exception further up the call stack, so that someone higher up can handle it. (Passing the buck, basically.)
• Sometimes, it'll catch the exception but throw *another* exception it built out of this one, again leaving the final resolution to somebody further up the call stack.

```
try {
   ...
}
catch(SomeException e) {
   ...
}
catch(SomeOtherException e) {
   ...
}
catch(YetAnotherException e) {
   ...
}
```

Different possible exceptions. Each one has its own class.

Each class must be descended from Exception.

```
finally {
  // Code that is ALWAYS executed
}
```

Finally

• This is a place to put code that will *always* be run.
• This happens even if:
• The program crashes somewhere in the try-block.
• A return statement inside the try-block gets used.
• The whole thing is in a loop and break/continue gets used. • And so on...
• This is a good place to clean up after yourself.
• Release resources, close files and network connections, all the housekeeping stuff you would normally do at the end.

lec 23
Exception Handling
the is-a relationship from inheritance means that we can exploit polymorphism.

•All descendants of Exception can polymorphically act like Exceptions, so this is a sort of catch-all statement.

•E.g., if some code throws a FileNotFoundException, a catch that handles IOExceptions will catch it.

•The same catch will be triggered by a SocketException too.

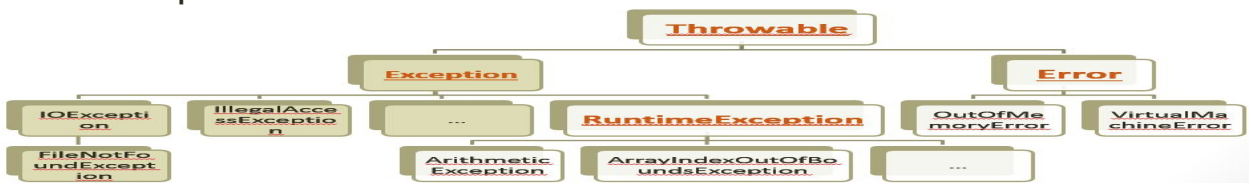
•In fact, if we want to catch *any* exception at all, we can just use a catch(Exception …) clause.

•If you decide to catch an error, then a catch(Error …) will catch any type of Error.

•And a catch(Throwable …) will catch absolutely anything that could be thrown, Exception or Error.

Partial Hierarchy

Checked Exceptions



- **Not all types of runtime errors are useful to catch.**
- **No reasonable way to recover from OutOfMemoryError, and some exceptions (like ArithmeticException from dividing by zero).**
- **This leads to the distinction between *checked* and *unchecked* exceptions.**
- **The compiler will force you to handle checked exceptions in some way (or you can't compile), but unchecked exceptions do not need to be handled.**
- **They can be, but the compiler doesn't care either way.**

Throws / Throw

- **This indicates that the function is giving up its responsibility to handle the exceptions it throws.**
- **So now the compiler won't force you to use try-catch.**
- **The responsibility of exception handling now goes to the calling function. -which can either do a try-catch, or pass the buck again, using throws.**
- **At some point, you have to throw an exception to indicate the error. This is done using the throw keyword.**

```java
public class App {

    public static void main(String[] args) {

        Test test = new Test();

        //if i were to do a test.run(); it would complain this needs a try-catch
        //because there was a throws in Exeption.java
        try{
            test.run();
        }
        catch(IOException e){
            //IOException was the exeption we decide to use
            //we found this in java documentation for exeptions
            System.out.println(e.getMessage() ); //this prints the
                                                 //error from exep...java
        }

    }

}
```

```java
public class Test {


    //•This "TRHOWS" indicates that the function is giving up its
    //responsibility to handle the exceptions it throws.
    //•So now the compiler won't force you to use try-catch.
    //•The responsibility of exception handling now goes to the calling function.
    //-which can either do a try-catch, or pass the buck again, using throws.
    public void run() throws IOException{

        //Some kind of return value form some complex sprocess!
        // 0 = success
        // anything else = error coe
        int code = 0; //will output: Running sucessfully
        //int code = 1;    //will output: My message in Test class

        if(code != 0){
            //Something's wrong!
            throw new IOException("My message in Test class");
        }

        System.out.println("Running successfully");

    }
}
```

lect 24

Creating our own exception

//if you do NOT pass it any argument it will return “Division by Zero!”
//if you DO, however, passit it an argument it will return the argument you passed in.

```
public class DivisionByZeroException extends Exception {
    public DivisionByZeroException() {
        super("Division by Zero!");
    }

    public DivisionByZeroException(String message) {
        super(message);
    }
}
```

```
    public static void main(String[] args) {
        try {
            Scanner keyboard = new Scanner(System.in);

            System.out.println("Enter numerator:");
            int numerator = keyboard.nextInt();
            System.out.println("Enter denominator:");
            int denominator = keyboard.nextInt();

            if (denominator == 0) {
                throw new DivisionByZeroException();
            }

            double quotient = numerator / (double) denominator;
            System.out
                    .println(numerator + "/" + denominator + " = " + quotient);
        } catch (DivisionByZeroException e) {
            System.out.println(e.getMessage());
            secondChance();
        }

        System.out.println("End of program.");
    }

    public static void secondChance() {
        Scanner keyboard = new Scanner(System.in);

        System.out.println("Try again:");
        System.out.println("Enter numerator:");
        int numerator = keyboard.nextInt();
        System.out.println("Enter denominator:");
        System.out.println("Be sure the denominator is not zero.");
        int denominator = keyboard.nextInt();

        if (denominator == 0) {
            System.out.println("I cannot do division by zero.");
            System.out.println("Aborting program.");
            System.exit(0);
        }

        double quotient = ((double) numerator) / denominator;
        System.out.println(numerator + "/" + denominator + " = " + quotient);
    }
```

ArrayList **http://www.newthinktank.com/2012/01/java-video-tutorial-11/**

```java
import java.util.ArrayList; // The ArrayList library
import java.util.Iterator; // The Iterator Library
import java.util.Arrays; // The Arrays Library

public class LessonEleven {

    public static void main(String[] args)
    {

        // You can create an ArrayList variable
        ArrayList arrayListOne;

        // Then create an ArrayList object
        // You don't have to declare the ArrayList size like you
        // do with arrays (Default Size of 10)
        arrayListOne = new ArrayList();

        // You can create the ArrayList on one line

        ArrayList arrayListTwo = new ArrayList();

        // You can also define the type of elements the ArrayList
        // will hold
        ArrayList<String> names = new ArrayList<String>();

        // This is how you add elements to an ArrayList
        names.add("John Smith");
        names.add("Mohamed Alami");
        names.add("Oliver Miller");

        // You can also add an element in a specific position
        names.add(2, "Jack Ryan");

        // You retrieve values in an ArrayList with get
        // arrayListName.size() returns the size of the ArrayList
        for( int i = 0; i < names.size(); i++)
        {
            System.out.println(names.get(i));
        }

        // You can replace a value using the set method
        names.set(0, "John Adams");

        // You can remove an item with remove
        names.remove(3);

        // You can also remove the first and second item with
        // the removeRange method
        // names.removeRange(0, 1);

        // When you print out the ArrayList itself the toString
        // method is called
        System.out.println(names);

        // You can also use the enhanced for with an ArrayList
        for(String i : names)
        {
            System.out.println(i);
        }
        System.out.println(); // Creates a newline

        // Before the enhanced for you had to use an iterator
        // to print out values in an ArrayList

        // Creates an iterator object with methods that allow
        // you to iterate through the values in the ArrayList
        Iterator indivItems = names.iterator();

        // When hasNext is called it returns true or false
        // depending on whether there are more items in the list

        while(indivItems.hasNext())
        {
            // next retrieves the next item in the ArrayList
            System.out.println(indivItems.next());

        }

        // I create an ArrayList without stating the type of values
        // it contains (Default is Object)
        ArrayList nameCopy = new ArrayList();
        ArrayList nameBackup = new ArrayList();

        // addAll adds everything in one ArrayList to another
        nameCopy.addAll(names);
        System.out.println(nameCopy);

        String paulYoung = "Paul Young";

        // You can add variable values to an ArrayList
        names.add(paulYoung);
```

```
094
095             // You can add variable values to an ArrayList
096             names.add(paulYoung);
097
098             // contains returns a boolean value based off of whether
099             // the ArrayList contains the specified object
100
101             if(names.contains(paulYoung))
102             {
103                 System.out.println("Paul is here");
104             }
105
106             // containsAll checks if everything in one ArrayList is in
107             // another ArrayList
108             if(names.containsAll(nameCopy))
109             {
110                 System.out.println("Everything in nameCopy is in names");
111             }
112
113             // Clear deletes everything in the ArrayList
114             names.clear();
115
116             // isEmpty returns a boolean value based on if the ArrayList
117             // is empty
118             if (names.isEmpty())
119             {
120
121                 System.out.println("The ArrayList is empty");
122
123             }
124
125             Object[] moreNames = new Object[4];
126
127             // toArray converts the ArrayList into an array of objects
128             moreNames = nameCopy.toArray();
129
130             // toString converts items in the array into a String
131             System.out.println(Arrays.toString(moreNames));
132
133
134         }
135
136 }
```

LinkedList
http://www.newthinktank.com/2013/03/linked-list-in-java/

Link.java

```
01 public class Link {
02
03     // Set to public so getters & setters aren't needed
04
05     public String bookName;
06     public int millionsSold;
07
08     // Reference to next link made in the LinkList
09     // Holds the reference to the Link that was created before it
10     // Set to null until it is connected to other links
11
12     public Link next;
13
14     public Link(String bookName, int millionsSold){
15
16         this.bookName = bookName;
17         this.millionsSold = millionsSold;
18
19     }
20
21     public void display(){
22
23         System.out.println(bookName + ": " + millionsSold + ",000,000
   Sold");
24
25     }
26
27     public String toString(){
28
29         return bookName;
30
31     }
32
33     public static void main(String[] args) {
34
35         LinkList theLinkedList = new LinkList();
36
37         // Insert Link and add a reference to the book Link added just prior
38         // to the field next
39
40         theLinkedList.insertFirstLink("Don Quixote", 500);
41         theLinkedList.insertFirstLink("A Tale of Two Cities", 200);
42         theLinkedList.insertFirstLink("The Lord of the Rings", 150);
43         theLinkedList.insertFirstLink("Harry Potter and the Sorcerer's
   Stone", 107);
44
45         theLinkedList.display();
47         System.out.println("Value of first in LinkedList " +
   theLinkedList.firstLink + "\n");
48
49         // Removes the last Link entered
50
51         theLinkedList.removeFirst();
52
53         theLinkedList.display();
54
55         System.out.println(theLinkedList.find("The Lord of the
   Rings").bookName + " Was Found");
56
57         theLinkedList.removeLink("A Tale of Two Cities");
58
59         System.out.println("\nA Tale of Two Cities Removed\n");
60
61         theLinkedList.display();
62
63     }
64
65 }
```

inkList.java

```
001 class LinkList{
002
003     // Reference to first Link in list
004     // The last Link added to the LinkedList
005
006     public Link firstLink;
007
008     LinkList(){
009
010         // Here to show the first Link always starts as null
011
012         firstLink = null;
013
014     }
015
016     // Returns true if LinkList is empty
017
018     public boolean isEmpty(){
019
020         return(firstLink == null);
021
022     }
023
024     public void insertFirstLink(String bookName, int millionsSold){
025
026         Link newLink = new Link(bookName, millionsSold);
027
028         // Connects the firstLink field to the new Link
```

```
029
030             newLink.next = firstLink;
031
032             firstLink = newLink;
033
034         }
035
036         public Link removeFirst(){
037
038             Link linkReference = firstLink;
039
040             if(!isEmpty()){
041
042                 // Removes the Link from the List
043
044                 firstLink = firstLink.next;
045
046             } else {
047
048                 System.out.println("Empty LinkedList");
049
050             }
051
052             return linkReference;
053
054         }
055
056         public void display(){
057
058             Link theLink = firstLink;
059
060             // Start at the reference stored in firstLink and
061             // keep getting the references stored in next for
062             // every Link until next returns null
063
064             while(theLink != null){
065
066                 theLink.display();
067
068                 System.out.println("Next Link: " + theLink.next);
069
070                 theLink = theLink.next;
071
072                 System.out.println();
073
074             }
075
076         }
077
078         public Link find(String bookName){
079
                                                    stLink;
rketing/for-signs/
078         public Link find(String bookName){
079
080             Link theLink = firstLink;
081
082             if(!isEmpty()){
083
084                 while(theLink.bookName != bookName){
085
086                     // Checks if at the end of the LinkedList
087
088                     if(theLink.next == null){
089
090                         // Got to the end of the Links in LinkedList
091                         // without finding a match
092
093                         return null;
094
095                     } else {
096
097                         // Found a matching Link in the LinkedList
098
099                         theLink = theLink.next;
100
101                     }
102
103                 }
104
105             } else {
106
107                 System.out.println("Empty LinkedList");
108
109             }
110
111             return theLink;
112
113         }
114
115         public Link removeLink(String bookName){
116
117             Link currentLink = firstLink;
118             Link previousLink = firstLink;
119
120             // Keep searching as long as a match isn't made
121
122             while(currentLink.bookName != bookName){
123
124                 // Check if at the last Link in the LinkedList
125
126                 if(currentLink.next == null){
127
128                     // bookName not found so leave the method
```

```
129
130                     return null;
131
132                 } else {
133
134                     // We checked here so let's look in the
135                     // next Link on the list
136
137                     previousLink = currentLink;
138
139                     currentLink = currentLink.next;
140
141                 }
142
143             }
144
145             if(currentLink == firstLink){
146
147                 // If you are here that means there was a match
148                 // in the reference stored in firstLink in the
149                 // LinkedList so just assign next to firstLink
150
151                 firstLink = firstLink.next;
152
153             } else {
154
155                 // If you are here there was a match in a Link other
156                 // than the firstLink. Assign the value of next for
157                 // the Link you want to delete to the Link that's
158                 // next previously pointed to the reference to remove
159
160                 System.out.println("FOUND A MATCH");
161                 System.out.println("currentLink: " + currentLink);
162                 System.out.println("firstLink: " + firstLink);
163
164                 previousLink.next = currentLink.next;
165
166             }
167
168             return currentLink;
169
170         }
171
172 }
```

//finish these later: might not need to study for tests
Sets
TreeSet
HashSet: uses HashCode() and equals)
compareTo : used to detect duplicates
Maps
http://www.programcreek.com/2009/02/the-interface-and-class-hierarchy-for-collections/

Colections : ArrayList, TreeSet;

These are basically more complicated arrays, you can store more complicated form of arrays in them.
• The basic methods are all pretty much the same, and quite simple.
• add(), remove(), contains(), size() • get(), put(), etc.

syntax
ArrayList var = new ArrayList();
//or for example if i wanted to use Strings
ArrayList<String> names = new ArrayList<string>();

Remember
for(String i; names) ...iterates all

TreeSet var = new TreeSet(); // sorts everything for you when you add
// if user types a duplicate i believe it will only store one

```
TreeMap tm = new TreeMap();     //sorts according to first name
// Put elements to the map      //duplicates allowed
tm.put("John Doe", new Double(3434.34));
```

Enums

• In the absence of enums, we would do something like this.
• static final int SPRING = 1; • static final int SUMMER= 2;
- The problem with using the previous as a substitution is for example
• Say we have this method: double meanTemperature(int season) { ... }
• Someone using your code can use the following call: • meanTemperature(17);
• This is invalid – 17 doesn't refer to a season.
• ints can take on far too many values, but we only need 4.
• Why not just throw an exception if an invalid value is passed?
• Nothing wrong with that, but it feels like bad design.
• Your attitude should be: Why allow a design to *potentially* cause an error, when I can just make the error *impossible*? • Prevention is better than cure.

The enum solution
public enum Season {
SPRING, SUMMER, FALL, WINTER;

}
• Each value can be accessed as: Season.SPRING, Season.WINTER, etc.

Additional benefits
• Every enum inherits a values() method that allows us to iterate over all valid values.
• Can iterate over all valid seasons with perfectly safe code.
for(Season s: Season.values()) { ... }
• This will not need changing, even if seasons are added, removed or modified.
• Constant ints cannot achieve such elegance.

• We can also add extra information to enums, and even give them methods (and constructors).

• Why? Because the enum syntax is really just a clever disguise for creating a new class that extends a preexisting Enum class.
• So all enums inherit methods like toString() and values()
• For example, we might initialize each season with a date range (say, startMonth and endMonth) indicating when the season begins and ends.
• Adding a toString() method to an enum means we can get it to print nicely formatted information, instead of just the dumb looking shouty text SPRING.

Enum Syntax

```
//declaring an enum & overriding its tostring
public enum Month {
	JANUARY,
	FEBRUARY,
	MARCH,
	//  …   //

	// We'll override toString() so the months come out like
	// "January" rather than "JANUARY".
	@Override
	public String toString() {
		// We'll take the standard toString() inherited from Object
		// and just lower case everything after the first character.
		String normalString = super.toString();
		return normalString.charAt(0) + normalString.substring(1).toLowerCase();
	}
}// closes public enum Month bracket
```

```
//Using values() and weird for loop
	public enum Seasons {
		SPRING, SUMMER, FALL, WINTER;
	}
	// Returns the mean temperature for a given season.
	public static double meanTemperature(Seasons season) {
		switch(season) {
		case FALL:
			return 60;
		//...//
		}
	}
	public static void main(String[] args) {
		System.out.println(meanTemperature(Seasons.WINTER));
		// The values() method is inherited by all enums (you don't have to define it)
		// and returns a list of all the elements in the enum.
		for (Seasons s: Seasons.values()) {
			System.out.println(s + " " + meanTemperature(s));
		}
	}
```

```
//Polymorphic weird constructor thing with enums
	public enum Holiday {
		// This enum has a constructor (defined below), which is the reason for the slightly odd syntax below.
		// Notice how these are just constructor calls, without using the new operator.
		APRIL_FOOLS_DAY(Month.APRIL, 1, "April Fool's Day"),
		HALLOWEEN(Month.OCTOBER, 31, "Halloween"),
		TALK_LIKE_A_PIRATE_DAY(Month.SEPTEMBER, 19, "Talk like a Pirate day");

		// The three instance fields.
		private Month month;
		private int day;
		private String desc;
```

```
		// The constructor.
		Holiday(Month month, int day, String desc) {
			this.month = month;
			this.day = day;
			this.desc = desc;
		}

		@Override
		public String toString() { return desc;	}

		// Since we can add methods to the enum, we'll
		// add two to extract months and days.
		public Month getMonth() { return month; }
		public int getDay() {  return day;		}
	}

public class HolidayDemo {
	public static void main(String[] args) {
		for (Holiday h: Holiday.values()) {
			System.out.println(h + " falls on " + h.getMonth() + " "
					+ h.getDay());
		}
	}
}
```

//using collections with enums nice little trick

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;


public class CardParty {

	public static void main(String[] args) {
		// Make some arbitrary card and print it out.
		Card aceOfSpades = new Card(Suit.SPADES, Rank.ACE);
		System.out.println(aceOfSpades);

		// Use values() to generate all possible cards in the deck.
		List<Card> deck = new ArrayList<Card>();
		for (Suit s: Suit.values())					// Loop over all suits
			for (Rank r: Rank.values()) {		// Loop over all ranks
				// Add them to the list.
				deck.add(new Card(s, r));
			}

		// This handy method randomly permutes any collection.
		Collections.shuffle(deck);
		// Print out the deck.
		for (Card card: deck) {
			System.out.println(card);
		}
	}

}
```

Final details

- Final is on Friday, December 06.
  - 10 AM – 12:50 PM (Not regular class time!)
  - Usual classroom
- It's technically 3 hours long, but the exam itself won't be written to take more than 60-90 minutes.
- Covers pretty much everything, but will touch very lightly on GUIs and threads.
  - Neither topic is especially easy to test in a written exam.
- We'll have 2-3 quizzes up later this week, as a sort of review.