cs3101-003 Java: lecture #6

- news:
  - homework #5 due today
  - little quiz today
  - it's the last class!
  - please return any textbooks you borrowed from me
- today's topics:
  - interfaces
  - recursion
  - data structures
  - threads

interfaces (1).

- an *interface* is a group of *abstract methods* that are defined by all classes that implement the interface
- an *abstract method* is one that does not have an implementation, i.e., there is no body of code for the method
- *polymorphism* means "having many forms"
  - lets us use different implementations of a single interface
  - *binding* happens when a particular implementation is locked to an interface
  - this can happen at compile time or at run time
  - an example of a run-time or *dynamic binding* is:
    `((Philosopher)current).pontificate();`
    from the example to follow

interfaces (2).

```
public interface Speaker {

  public void speak();

  public void announce( String str );

} // end of Speaker interface
```

interfaces (3).

```
public class Philosopher implements Speaker {
  private String philosophy;

  public Philosopher ( String thoughts ) {
    philosophy = thoughts;
  } // end of Philosopher constructor

  public void speak () {
    System.out.println( philosophy );
  } // end of Philosopher method speak

  public void announce ( String announcement ) {
    System.out.println( announcement );
  } // end of Philosopher method announce

  public void pontificate() {
    for ( int i=0; i<5; i++ )
      System.out.println( philosophy );
  } // end of Philosopher method pontificate

} // end of Philosopher class
```

interfaces (4).

```
public class Dog implements Speaker {

  public void speak() {
    System.out.println( "woof" );
  } // end of Dog method speak

  public void announce( String arf ) {
    System.out.println( "woof: " + arf );
  } // end of Dog method announce

} // end of class Dog
```

interfaces (5).

```
public class Talking {

  public static void main( String[] args ) {

    Speaker current;

    current = new Dog();
    current.speak();

    current = new Philosopher( "I think, therefore I am." );
    current.speak();

    ((Philosopher)current).pontificate();

  } // end of main()

} // end of Talking class
```

recursion.

- recursion is defining something in terms of itself
- there are many examples in nature
- and in mathematics
- and in computer graphics, e.g., the Koch snowflake (from our gui example in the gallery)

power function.

- *power* is defined recursively: $x^y = \begin{cases} \text{if } y == 0, & x^y = 1 \\ \text{if } y == 1, & x^y = x \\ \text{otherwise}, & x^y = x * x^{y-1} \end{cases}$

here it is in a Java method.

```
public int power ( int x, int y ) {
  if ( y == 0 ) {
    return( 1 );
  }
  else if ( y == 1 ) {
    return( x );
  }
  else {
    return( x * power( x, y-1 ));
  }
} // end of power() method
```

- Notice that `power()` calls itself!
- You can do this with any method *except main()*
- BUT beware of infinite loops!!!
- You have to know when and how to stop the recursion — what is the *stopping* condition

---

let's walk through `power(2,4)`.

	call	x	y	return value
1	power(2,4)	2	4	2 * power(2,3)
2	power(2,3)	2	3	2 * power(2,2)
3	power(2,2)	2	2	2 * power(2,1)
4	power(2,1)	2	1	2

- the first is the *original call*
- followed by three *recursive calls*

---

stacks.

- the computer uses a data structure called a *stack* to keep track of what is going on
- think of a *stack* like a stack of plates
- you can only take off the top one
- you can only add more plates to the top
- this corresponds to the two basic *stack operations*:
  - *push* — putting something onto the stack
  - *pop* — taking something off of the stack
- when each recursive call is made, power() is pushed onto the stack
- when each return is made, the corresponding power() is popped off of the stack

---

another example: factorial.

- *factorial* is defined recursively:

$$N! = \begin{cases} \text{if } N == 1, & N! = 1 \\ \text{otherwise}, & N! = N * (N-1)! \end{cases}$$

- (for $N > 0$)

here it is in a Java method.

-
  ```
  public int factorial ( int N ) {
    if ( N == 1 ) {
      return( 1 );
    }
    else {
      return( N * factorial( N-1 ));
    }
  } // end of factorial() method
  ```

---

recursive iteration.

```
public class ex6a {

  Random r = new Random();
  Coin[] pocket = new Coin[5];

  public static void main( String[] args ) {
    ex6a ex = new ex6a();
    for ( int i=0; i<5; i++ ) {
      ex.pocket[i] = new Coin();
    }
    ex.printPocket( 0 );
  } // end of main() method

  public void printPocket( int index ) {
    if ( index < pocket.length ) {
      System.out.print( pocket[index].getValue() + " " );
      printPocket( index+1 );
    }
    else {
      System.out.println();
    }
  } // end of printPocket() method

} // end of ex6a class
```

---

normal iteration.

- where normal iteration looks like this:

  ```
  public void printPocket() {
    for ( int index=0; index<pocket.length; index++ ) {
      System.out.print( pocket[index].getValue() + " " );
    }
    System.out.println();
  } // end of printPocket() method
  ```

---

back to recursive iteration.

- in the recursive version, each call is like one iteration inside the for loop in the iterative version

	call	index	output	next call
1	printPocket(0)	0	pocket[0].getValue()	printPocket(1)
2	printPocket(1)	1	pocket[1].getValue()	printPocket(2)
3	printPocket(2)	2	pocket[2].getValue()	printPocket(3)
4	printPocket(3)	3	pocket[3].getValue()	printPocket(4)
5	printPocket(4)	4	pocket[4].getValue()	printPocket(5)
6	printPocket(5)	5	newline	— —

more on recursion.

- With recursion, each time the method is invoked, one step is taken towards the resolution of the task the method is meant to complete.
- Before each step is executed, the state of the task being completed is somewhere in the middle of being completed.
- After each step, the state of the task is one step closer to completion.
- In the example above, each time $printPocket(i)$ is called, the array is printed from the i-th element to the end of the array.
- In the $power(x, y)$ example, each time the method is called, power is computed for each x^y, in terms of the previous x^{y-1}.
- In the $factorial(N)$ example, each time the method is called, factorial is computed for each N, in terms of the previous $N - 1$.
- One classic example is "Towers of Hanoi". In each turn or iteration, one disk is moved from one tower to another. At each point (i.e., at the start of each recursive call), the state of the towers is in the middle of completion, until the final solution is reached.

---

search.

- Often, when you have data stored in an array, you need to locate an element within that array.
- This is called searching.
- Typically, you search for a *key* value (simply the value you are looking for) and return its *index* (the location of the value in the array)
- As with sorting, there are many searching algorithms.
  - linear search
    - standard linear search, on sorted or unsorted data
  - binary search
    - iterative binary search, on sorted data only
    - recursive binary search, on sorted data only

---

linear search.

```
public int linearSearch( int key ) {
  for ( int i=0; i<pocket.length; i++ ) {
    if ( key == pocket[i].getValue() ) {
      return( i );
    }
  }
  return( -1 );
} // end of linearSearch() method
```

---

binary search (1).

- Binary search is much more efficient than linear search, ON A SORTED ARRAY. (It CANNOT be used on an unsorted array!)
- It takes the strategy of continually dividing the search space into two halves, hence the name *binary*. Say you are searching something very large, like the phone book. If you are looking for one name (e.g., "Gilligan"), it is extremely slow and inefficient to start with the A's and look at each name one at a time, stopping only when you find "Gilligan". But this is what linear search does. Binary search acts much like you'd act if you were looking up "Gilligan" in the phone book.
  - You'd open the book somewhere in the middle, then determine if "Gilligan" appears before or after the page you have opened to.
  - If "Gilligan" appears after the page you've selected, then you'd open the book to a later page.
  - If "Gilligan" appears before the page you've selected, then you'd open the book to an earlier page.
  - You'd repeat this process until you found the entry you are looking for.

binary search (2).

```
public int binarySearch( int key ) {
  int lo = 0, hi = pocket.length-1, mid;
  while ( lo <= hi ) {
    mid = ( lo + hi ) / 2;
    if ( key == pocket[mid].getValue() ) {
      return( mid );
    }
    else if ( key < pocket[mid].getValue() ) {
      hi = mid - 1;
    }
    else {
      lo = mid + 1;
    }
  } // end while
  return( -1 );
} // end of binarySearch() method
```

recursive binary search (1).

```
public int recursiveBinarySearch( int key, int lo, int hi ) {
  if ( lo <= hi ) {
    int mid = ( lo + hi ) / 2;
    if ( key == pocket[mid].getValue() ) {
      return( mid );
    }
    else if ( key < pocket[mid].getValue() ) {
      return( recursiveBinarySearch( key, lo, mid-1 ));
    }
    else {
      return( recursiveBinarySearch( key, mid+1, hi ));
    }
  }
  else {
    return( -1 );
  }
} // end of recursiveBinarySearch() method
```

- invoke with:
  `int i = recursiveBinarySearch( key,0,pocket.length );`

data structures.

- a *data structure* is essentially an *abstract data type*
- it maps a virtual model of data to a real data type
- like an array or a Vector or a class
- there are several classic data structures in computer science
- we'll look at a few:
  - linked list
  - doubly linked list
  - queue
  - stack
- each has rules about how to *add* and *remove* elements
- examples:
  - a queue is also called FIFO — first in, first out
  - a stack is also called LIFO — last in, first out

implementation vs interface.

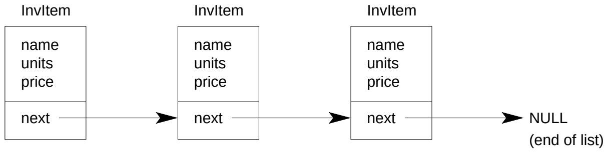
- when talking about data structures, there is a distinction between *implementation* and *interface*
- implementation — refers to the actual underlying implementation; like *linked list* or *doubly linked list*
- interface — refers to an abstract view of the data structure, overlying the implementation; like *queue* or *stack*
- for example, a *queue* can be implemented using a *linked list*
- the interfaces govern how items can be added and removed from the data structure

---

linked list.

- a *linked list* chains instances of a class together using a field called "next", which points to the next instance of the class in the chain
- yes, this looks like another type of array or Vector

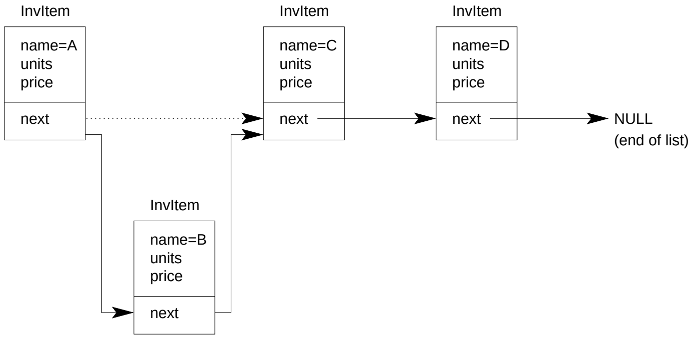
```
public class InvItem {
  private String name;
  private int units;
  private float price;
  InvItem next; // points to the next InvItem in the chain
}
```



---

adding an item to a linked list (1).

- to add an item to a linked list, you simply instantiate one instance of the InvItem and then set the "next" fields in the new item and the item in the linked list after which the new item is being inserted:



---

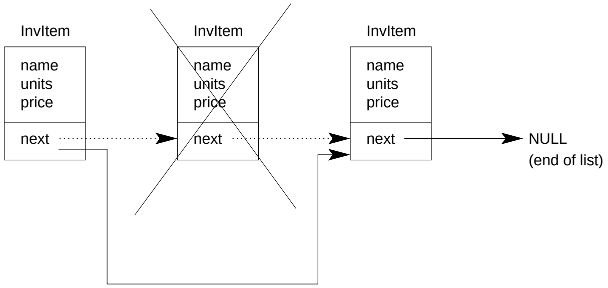
adding an item to a linked list (2).

- here's a code fragment:

```
// top-level declaration
InvItem linkedList;
.
.
.
// somewhere inside a method...
InvItem newItem = new InvItem( name,units,price );
InvItem listItem; // pointer to list item after which
                  // newItem will be inserted
newItem.next = listItem.next;
listItem.next = newItem;
.
.
.
```

removing an item from a linked list.

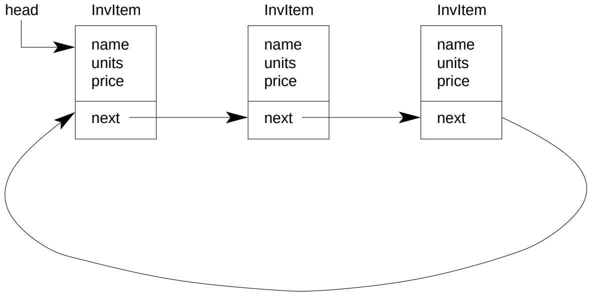
- to remove an item to a linked list, you simply move the “next” field pointers around



---

circular linked list.

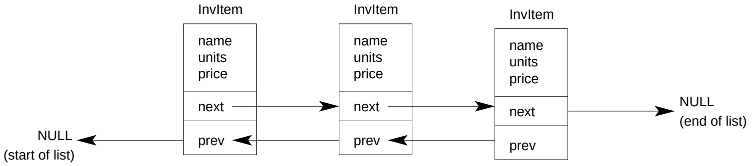
- sometimes linked lists are *circular*, where the “next” field from the last item points back to the first item
- in this case, there is typically an external “pointer” called “head” which points to the first item on the list



---

doubly linked list (1).

- a *doubly linked list* chains instances of a class together using two fields called “next” (which points to the next instance of the class in the chain) and “prev” (which points to the previous instance of the class in the chain).



---

doubly linked list (2).

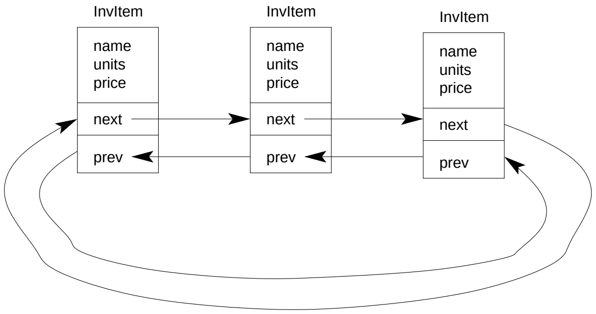
- for example:

```
public class InvItem {
  private String name;
  private int units;
  private float price;
  InvItem next; // points to the next InvItem in the chain
  InvItem prev; // points to the previous InvItem in the chain
}
```

- when adding and removing items to a doubly linked list, you need to set the “next” and “prev” fields, just as like with the (singly) linked list

circular doubly linked list.

- doubly linked lists can also be circular:



---

queue.

- a *queue* is like a check-out line at a store
- you can only add items (people) to the end of the line
- you can only remove items (people) from the front of the line
- hence, a queue is also called a FIFO (first in, first out)
- following are the typical names for queue routines:
  - **enqueue**: for adding items to a queue
  - **dequeue**: for removing items from a queue

---

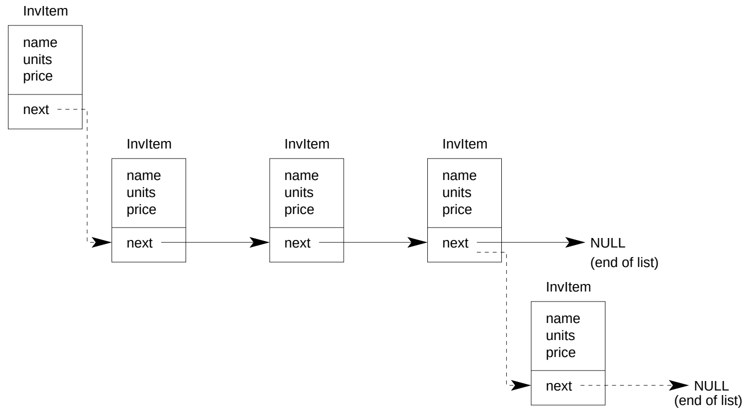
stack.

- a *stack* is like a stack of plates
- you can only add items (plates) to the top of the stack
- you can only remove items (plates) from the top of the stack
- hence, a stack is also called a LIFO (last in, first out)
- following are the typical names for stack routines:
  - **push**: for adding items to a stack
  - **pop**: for removing items from a stack

---

stack vs queue: adding items.

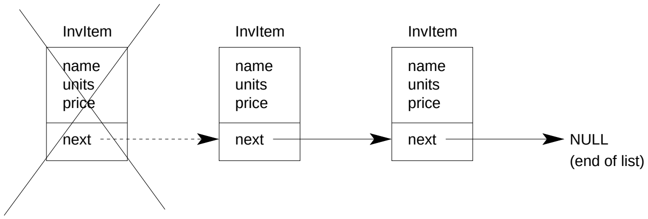
new stack item gets added to the front (top) of the list



new queue item gets added to the end of the list

stack vs queue: removing items.

stack items get removed from the front (top) of the list



queue items get removed from the front of the list

---

threads (1).

- a *thread* is a unit of sequential execution
- it is smaller than a program (application or applet)
- a Java program can have multiple threads, which allow several things to be going on at the same time
- you have already been using *implicit threads* when programming GUIs
  - one thread handled painting updates to your GUI
  - another thread handled events
  - another thread handled your applet's main thread of execution

---

threads (2).

- you can create your own threads using `java.lang.Thread`
- the `Thread` class *implements* a `Runnable` interface
- the main methods to know about are:
  - `start()`
    causes the thread to begin execution by calling the thread's run method
  - `run()`
    runs the body of the thread (like the main method in an application)
  - verb+sleep()+
    suspends the run method for a specified period of time

---

threads — example.

```
public class SimpleThread extends Thread {

  private int id;
  private int delay;

  SimpleThread( int id, int delay ) {
    this.id = id;
    this.delay = delay;
  } // end of SimpleThread constructor

  public void run() {
    System.out.println( "thread " + id + " started" );
    System.out.flush();
    for ( int i=0; i<10; i++ ) {
      try {
        sleep( delay );
      }
      catch( InterruptedException e ) {
        System.out.println( "sleep interrupted: " + e );
      }
      System.out.println( "thread " + id + ": i = " + i );
    }
    System.out.println( "thread " + id + " finished" );
  } // end of run()

} // end of class SimpleThread
```

threads — example, cont.

```
public class TwoThreads {

  public static void main( String[] args ) {
    SimpleThread t1 = new SimpleThread( 1, 1000 );
    SimpleThread t2 = new SimpleThread( 2, 1200 );
    t1.start();
    t2.start();
  } // end of main()

} // end of class TwoThreads
```

---

that's it!

- have a good Spring :-)