Day 2
Object Oriented Programming: Encapsulation

Webster defines encapsulation as being “enclosed by a capsule.”
Real world examples of encapsulation surround us:

A cabinet hides (encapsulates) the “guts” of a television, concealing from TV viewers the internal apparatus of the TV. Moreover, the television manufacturer provides users with an interface -- the buttons on the TV or perhaps a remote control unit. To operate the TV and watch *The Simpsons* or *Masterpiece Theatre*, viewers utilize this interface. The inner circuitry of a TV is of no concern to most TV viewers.

A computer is another example of real world encapsulation. The chips, boards, and wires of a computer are never exposed to a user. Like the TV viewer, a computer user operates a computer via an interface -- a keyboard, screen, and pointing device.

Cameras, vending machines, slot machines, DVD players, lamps, cars, video games, clocks, and even hourglasses are all physical examples of encapsulation.

Each of the devices enumerated above supplies the user with an interface — switches, buttons, remote controls, whatever -- for operation. Technical details are tucked away and hidden from users. Each encapsulated item functions perfectly well as a “black box.” Certainly, Joe User need not understand how his Radio Shack gadget is constructed in order to operate it correctly. A user-friendly interface and perhaps an instruction manual will suffice.

Encapsulation has a similar (though somewhat expanded) meaning when applied to software development and object oriented programming:

> The ability to provide users with a well-defined interface to a set of functions in a way which hides their internal workings. In object-oriented programming, the technique of keeping together data structures and the methods (procedures) which act on them.

– The Online Dictionary of Computing

Java provides encapsulation, as defined above, via classes and objects. As we have already seen, classes bundle data and methods into a single unit. *Classes encapsulate.*

**Example:**

Consider the Square class that is defined below. Square contains a few data fields, three constructors and several methods. Most of the code should be fairly easy to understand.

```java
public class Square {
    // notice that the private (public) keyword must be repeated for each new declaration
    private int dimension;
    private char character; // character to be used for drawing

    //constructors
    public Square() //default constructor
    {
        dimension = 0;
        character = '\n';
    }
}
```
public Square(int x, char ch)  //two argument constructor
{
    dimension = x;
    character = ch;
}

//accessor methods -- getter methods
public int getDimension()
{
    return dimension;
}

public char getCharacter()
{
    return character;
}

//Mutator Methods -- setter methods
public void  setDimension(int x)
{
    dimension = x;
}

public void setCharacter(char ch)
{
    character = ch;
}

public int area() // calculates the area of a square
{
    return dimension*dimension;
}

public int perimeter() // calculates the perimeter of a square
{
    return 4*dimension;
}

public void  draw(int x, int y)  // draws a square at position (x,y)
{
    // moves down y lines, indents x spaces and
    // draws a Square with the designated character

    // move the cursor down y lines
    for ( int i = 1; i <= y; i++)
    {
        System.out.println();
    }

    for (int len = 1; len<= dimension; len++)
    {
        // indent x spaces
        for (int i = 1; i <= x; i++)
        {
            System.out.print(" ");
        }

        //draw one "line" of the Square using character
        for (int j = 1; j <= dimension; j++)
        {
            System.out.print(character);
        }
    }
}
Java and Object Oriented Programming  
Day 2 - Encapsulation

```java
System.out.println();
}

public static void main(String args[]) {
    Square s;
    s= new Square(6,"*"); // dimension is 6 X 6
    s.draw(8,15);
}
```

Although this class is very similar in style to its C++ counterpart, you might notice a few differences.

- Java differs from C++ in the placement of the keywords `public` and `private`. (There are no public and private sections in Java as in C++.)
- Java does not allow separate interface and implementation sections, as does C++.
- Class Square has a `main` method. No C++ class has a main method. In C++, main is independent and outside of any class. (The main method in Square, however, is for testing purposes only.)
- Class Square must be saved in a file called `Square.java`. C++ places no restrictions on the name of a file.

The Square class provides a simple example of encapsulation – data and methods, attributes and functionality, are combined into a single unit, a single class. Furthermore, all data are accessed not directly but through the class methods, i.e. via an interface.

The user of Square – the client -- need not know how the class is implemented – only how to use the class. Variable names are of no concern to the client. If the client wishes to know the length of a Square object, the interface provides an accessor method. If the client wants to change the size of a Square object, the client simply uses the mutator method available through the interface. That the length of the sides of a square is held in a variable called `dimension` is irrelevant to the client. Further, if the implementation of the class is modified, programs utilizing the Square class will not be affected provided the interface remains unchanged. Like a TV or a camera, the inner workings of the Square class are encapsulated and hidden from the client. Public methods provide the interface just as the remote control unit provides the interface for a TV viewer.

Another term that is often associated with encapsulation is information hiding. Many authors regard encapsulation and information hiding as synonyms. However OO purists might define encapsulation as the language feature allowing the bundling of data and methods into one unit and information hiding as the design principle that restricts clients from the inner workings of a class. With this distinction, programs can have encapsulation without information hiding. Regardless of how you define your terms, classes should be designed with a well-defined interface in which implementation decisions and details are hidden from the client.
Square Objects and References

You will notice that the Square class has a main() method. main() is included for testing as well as for illustrative purposes. Every class need not contain a main() method.

Take a look at the main() method. Notice that a Square object is created or *instantiated* in two steps:
1. Square r;
2. r = new Square(6, "*");

Variable r is *NOT* a Square object but a *reference* to a Square object. This is in contrast to C++. The declaration in Line 1 does not cause a Square object to be constructed. No constructor is called. The reference r can hold the *address* of a Square object only once a Square object is created. Initially r has the value *null*.

Line 2 constructs, builds, instantiates, or creates a Square object and also assigns the address of this newly minted Square to reference r. Notice the use of the *new* operator as well as a call to the (two-argument) constructor.

Java utilizes *references* but not pointers. The distinction is subtle. Like a pointer, a Java reference holds an address, but a reference may not be manipulated like a C++ pointer. In C++, pointer arithmetic is a way of life. For example, if p is a C++ pointer, then p++ is a perfectly legal manipulation -- not so in Java.

It is common practice to speak of a reference variable as an object. Often, one says that “r is a Square object” or “r is a Square.” The fact is, however, that r is a *reference* to a Square object, but r is not a Square object itself. This distinction between a reference and an object is an important one.

Using the Square Class

Once we have designed the Square class, it is a snap to use it in a program. The following program allows a person to continually invoke the methods of the Square class, possibly to draw any number of squares on the screen.

It is certainly tempting to place the program application into the main method of the Square class. The program would certainly work and do the job. Nonetheless, in the object-oriented paradigm, *objects perform tasks by sending messages to other objects*. Drawing a Square on the screen, means sending a message to a Square object. (“Square, draw thyself!”). An object of the following class, aptly named Picasso, communicates with a Square object, *square*, by sending messages to square -- messages like “draw yourself,” “get your area,” “change your width” etc.
public class Picasso {
    private Square square; // needs a Square object with which to communicate

    public Picasso() {
        // default constructor
        // creates a Square object from the user’s specifications
        System.out.print("Size: ");
        int s = MyInput.readInt();
        System.out.print("Drawing character: ");
        char c = MyInput.readChar();
        square = new Square(s, c);  // here is a Square object.
    }

    private int menu() {
        // a helper function to present choices to the user
        int choice;

        do // loop until user enters a number in the range 1-6
        {
            System.out.println("1: set side");
            System.out.println("2: set drawing character");
            System.out.println("3: draw");
            System.out.println("4: get area");
            System.out.println("5: get perimeter");
            System.out.println("6: exit");

            System.out.print("Choice: ");
            choice = MyInput.readInt();
        } while (choice < 1 || choice > 6);
        return choice;
    }

    public void drawPlus() {
        // Based on the user’s choice, this method
        // sends messages to the Square object, square. The messages
        // "ask" square to draw itself, get its area, perimeter etc.
        int choice;
        do {
            choice = menu();
        }
switch (choice)
{
    case 1: System.out.println("Enter side:");
            // send square a message
            square.setDimension(MyInput.readInt());
            break;

    case 2: System.out.println("Enter character:");
            square.setCharacter(MyInput.readChar());
            break;

    case 3: System.out.println("Enter x and y:");
            square.draw(MyInput.readInt(), MyInput.readInt());
            break;

    case 4: System.out.println("Area is \(\) \+ square.area());
            break;

    case 5: System.out.println("Perimeter is \(\) \+ square.perimeter() \);
            break;

    case 6: System.out.println("Bye");
}
} //while (choice != 6);

public static void main(String args[])
{
    Picasso artist = new Picasso();  //a Picasso object
    artist.drawPlus();  // triggers the action
}
} //end Picasso.  This class should be saved in Picasso.java.

In addition to main(), the Picasso class has two public methods:

- a constructor which creates a Square object (square) and
- a drawPlus() method which sends messages to the Square object.

These messages tell square to draw itself, give its area, its perimeter etc. Picasso also has a private “helper” method – menu(). (Rather than writing this separate helper method, the code for menu() could have been incorporated into the drawPlus() method or a menu object might have been created from a Menu class).

In short, two objects, artist (a Picasso object) and square (a Square object), communicate with each other to accomplish a task.

The main method of our application is simple and uncomplicated.
- The main method creates an instance of the Picasso class (artist) and
- starts the action by calling the drawPlus() method that sends a message to artist to start drawing (artist.drawPlus()).

That’s all. There is no other functionality in main().
In contrast to C++ programs, the main function of our application does nothing more than create an instance of itself and trigger the action. In his text, *Java and Object Orientation* (Springer, 2002), John Hunt writes:

The main method should not be used to define the application program. This tends to happen when people move from C or C++ to Java, since in C the main function is exactly where the main functionality is placed. It is, therefore, unfortunate that the name `main` is used for this method. The main method should only ever do a few things:

- Create an instance of the class in which it is defined. It should never create an instance of another class.
- Send the newly created instance a message so that it initializes itself.
- Send the newly created instance a message that triggers the application’s behavior.

Nonetheless, this design philosophy is not universally accepted. Many programmers do place quite a bit of functionality into the main method. In general, however, we should keep main as uncomplicated as possible. Of course, there are always exceptions.

**Designing Classes and Objects**

Once a class has been specified, coding the class is usually not too difficult. Specifying and designing the classes is usually a bit trickier. Object design is both an art and a science. Mastery comes with practice.

In the next example, we develop a simple design for a video poker game.

**Example:**

Casinos are not lacking in video poker machines like the one pictured below.
To initiate a video poker session, a player deposits an arbitrary number of coins into the video poker machine. This amount is the bankroll.

To play the game:
- The player makes a bet (one to five coins but not more than the bankroll).
- A hand of five cards is dealt from a deck of 52 cards to the player. The deck is reshuffled for each game.
- The player decides which cards he/she wishes to hold.
- New cards are dealt for those cards that the player wishes to discard.
- The hand is scored.
- If the hand is a winner the winning amount is added to the bankroll otherwise the bet is deducted from the bankroll.

The player can quit and cash out at any time. The player can continue to play as long as the bankroll is not depleted. The player can add to the bankroll before each game.

Our current goal is to design an object based model for a video poker game. For our purposes, a model involves designing classes that represent the objects of video poker, and identifying the relationships among these classes.

There is no single correct choice for determining the classes. Moreover, you will probably change and/or refine your classes several times during the design process. Program design is not linear; it is iterative.

Object-oriented design is a topic that fills volumes and is far too complicated for an in-depth discussion here. Nonetheless, we can develop a somewhat simple design process using the following steps:

1. Determine the classes.
2. Determine the responsibilities of the classes.
3. Determine the interactions among the classes.

Designing the Classes for Video Poker

We will assume that:
- Once a bet is made, it cannot be retracted and play automatically begins.
- All user input is valid. For example, we take for granted that a bet is at most five coins and that no bet exceeds the current bankroll. Of course, any useful system would need to check all user input.

1. Determine the Classes

Classes describe objects and objects are things. A common methodology for determining which objects might be appropriate involves marking the nouns – the things -- in the problem specification. Although every noun does not necessarily give rise to a class, examining the nouns is a good starting point. Below is the problem specification replicated with the nouns highlighted.

To initiate a video poker session, a player deposits an arbitrary number of coins into the video poker machine. This amount is the bankroll. To play the game:

- The player makes a bet (one to five coins but not more than the bankroll).
- A hand of five cards is dealt from a deck of 52 cards to the player.
- The deck is reshuffled for each game.
Java and Object Oriented Programming
Day 2 - Encapsulation

- The player decides which cards he/she wishes to hold.
- New cards are dealt for those cards that the player wishes to discard.
- The hand is scored.
- If the hand is a winner the winning amount is added to the bankroll otherwise the bet is deducted from the bankroll.

The player can quit and cash out at any time. The player can continue to play as long as the bankroll is not depleted. The player can add to the bankroll before any individual game.

We list the nouns - our class candidates:
video poker session
player
coins
video poker machine
amount
bankroll
game
bet
hand
card
deck

Obviously, some words are redundant. For example, amount and bankroll refer to the same thing. Video poker session, game and video poker machine for the most part represent the game. Also, a coin probably does not warrant a class of itself. So for now let’s settle on six classes:
poker game
player
bankroll
bet
hand
card

2. Determine Responsibilities of Each Class.

What does a class do? What is each class’ responsibility? What are the actions, the behaviors of each class?

As the nouns help indicate classes, the verbs of the problem specification can aid in finding class responsibilities. Of course, just as every noun did not necessarily give rise to a class, every action may not manifest itself as a verb, and every verb does not designate a class action or responsibility. A dose of good common sense is helpful too. Look for any kind of action in the problem description:

To initiate a video poker session, a player deposits an arbitrary number of coins into the video poker machine. This amount is the bankroll. To play the game:

- The player makes a bet (one to five coins but not more than the bankroll).
- A hand of five cards is dealt from a deck of 52 cards to the player. The deck is reshuffled for each game.
- The player decides which cards he/she wishes to hold.
- New cards are dealt for those cards that the player wishes to discard.
- The hand is scored.
- If the hand is a winner the winning amount is added to the bankroll otherwise the bet is deducted from the bankroll.
The player can **quit** and **cash out** at any time. The player can **continue** to play as long as the bankroll is not **depleted**. The player can **add** to the bankroll before any individual game.

Use the highlighted words to determine the actions and responsibilities of each class.

Let’s start with a player. What can a player do? We make a list based on the actions noted above. A player can:

- Initiate a game.
- Deposit coins (add to the bankroll).
- Play the game.
- Make a bet.
- Decide which cards to hold/discard.
- Cash out (quit).
- Decide to play again.

Here are some possible actions for the other classes.

**Poker game:**
- Accept coins for bankroll.
- Accept a bet.
- Deal a hand.
- Update a hand.
- Score a hand.
- Update the bankroll.

**Bet:**
- Give its amount.
- Set an amount.

**Deck:**
- Shuffle itself.
- Deal a card.

**Card:**
- Give its suit.
- Give its value.

**Hand:**
- Give its score.
- Update itself.

**Bankroll:**
- Update itself.
- Give its value.
Iterative Refinement

Perhaps some refinement is in order:

A few player actions might be collapsed into one action. For example, making a bet and initiating a game are really the same thing (assuming the bet cannot be retracted). Once a bet is made the game begins. Also, deciding whether or not to play again is really implied by either making a bet or not. With a few modifications, the responsibilities of a player are now:

- Deposit coins.
- Make a bet.
- Decide which cards to hold/discard.
- Quit.

The Poker Game might need a bit of refinement.

Since there is no graphical interface, the Poker Game (the machine) should probably display a menu. Also, the actions:

- Accept a bet.
- Deal a hand.
- Update a hand.
- Score a hand.

constitute a single, ordered collection of messages sent to other objects and together constitute the steps for playing a game. Thus we might combine these actions into a single action: play_game

The other classes, being somewhat simpler, probably need no refinement. One last refinement: Should the player end the game or the game end itself? Let's decide to let the game end itself.

Thus the classes and actions are:

<table>
<thead>
<tr>
<th>Player</th>
<th>Game</th>
<th>Bet</th>
<th>Deck</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deposit coins</td>
<td>Display menu</td>
<td>Get its amount</td>
<td>Shuffle</td>
</tr>
<tr>
<td>Make a bet</td>
<td>Accept coins for bankroll</td>
<td>Set its amount</td>
<td></td>
</tr>
<tr>
<td>Decide which cards to hold/discard</td>
<td>Play game</td>
<td>End</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Update the bankroll</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Card</th>
<th>Hand</th>
<th>Bankroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>Give its suit</td>
<td>Give its score</td>
<td>Set its value</td>
</tr>
<tr>
<td>Give its value</td>
<td>Make new hand</td>
<td>Give its value</td>
</tr>
<tr>
<td></td>
<td>Update a hand</td>
<td></td>
</tr>
</tbody>
</table>
3. Determine the interactions among the classes

Classes interact with other classes via messages. Below we list the messages that one class might send another throughout the game. These messages tell us how the classes interact.

- Poker Game asks Player for an initial amount and Player returns some amount.
- Poker Game asks Player for a Bet and Player returns the bet.
- Poker Game updates the Bet.
- Poker Game asks Hand for a new hand and Hand returns a hand.
- Hand asks Deck for Cards and Deck returns five Cards.
- Poker Game asks Player for hold cards and Player returns a list of cards to be held.
- Poker Game asks Hand for an update and Hand returns an updated version of a hand.
- Poker Game asks Hand for its score and Hand returns a score.
- Poker Game updates Bankroll.
- Poker Game displays menu.

Finally, from the class interactions, it is obvious that some classes must collaborate with others to get the job done. Sometimes one class needs another class. A Bet object can exist without any other object. However, a Hand cannot function without a Deck, so a Hand has a Deck to get its job done. Similarly a Deck must collaborate with a Card. Moreover, PokerGame collaborates with Player, Bet, Bankroll and Hand. Should a Player interact directly with the Bankroll (making the Bankroll a collaborator) or simply return an amount to the Game that has sole control of the bankroll? There is really no correct answer to this question and others like it. Some designs are obviously better than others, but sometimes it is up to the style of the programmer.

The Video Poker Class After Some Refinement

Taking into account all three steps, the following list of classes gives one design for a video poker game. This is not necessarily a final design. The process is iterative. Changes will occur. Also, no algorithm for scoring a hand has even been discussed. What we do have is a first model of a program, a model that is not cast in cement.
Encapsulation is the very foundation of object-oriented programming. In the next chapter we study the other two underlying themes: inheritance and polymorphism. However, for the remainder of this section, we discuss a few various and sundry topics that are essential when designing classes. We will also take another look at static methods and variables, along with Java’s String class, StringBuffer class, and the Java garbage collector.

Garbage Collection

Consider the following code segment which instantiates three Square objects:

1.  
2.  
3.  

A Square reference, r, is declared in line 1. Subsequently, r is assigned three different Square objects:

After line 1:

r

side 4
character ~
After line 2:

Notice that three Square objects have been created but two of them are no longer referenced. Is there a memory leak?

In Java, the memory for unreferenced objects is automatically reclaimed. This clean-up process is called garbage collection. Garbage collection is not a great name, because it is more like recycling.

What causes an object to become garbage?

1. An object’s reference may go out of scope. For example, a reference may be declared locally in a method. When the method terminates, the reference variable no longer exists. The referenced object is reclaimed.

2. A reference that is set to object x may no longer refer to x. Again, the garbage collector reclaims the object.

In contrast, C++ provides programmers with the delete operator as well as the responsibility of memory management. Java relieves the programmer of that responsibility.

Automatic garbage collection does not mean that all programs are free of memory leaks. An obsolete reference (a reference to an object no longer used in a program) will cause a memory leak. The garbage collector cannot determine that a referenced object is no longer used in the program, i.e., the reference is obsolete. If the garbage collector recognizes that an object is referenced, then the object’s memory is not reclaimed – whether or not the object is of any further use. To avoid such obsolete reference memory leaks, simply assign a value of null to an obsolete reference. The garbage collector will do the rest.
Consider the two code fragments below:

```java
Square r = new Square(5, '*');
r.draw(2, 4);
Triangle t = new Triangle(6, '*');
t.draw(4, 4);
Circle c = new Circle(4, '*');
c.draw(2, 2);
// code that does not use the objects created above
```

```java
Square r = new Square(5, '*');
r.draw(2, 4);
r = null;
Triangle t = new Triangle(6, '*');
t.draw(4, 4);
t = null;
Circle c = new Circle(4, '*');
c.draw(2, 2);
c = null;
// more code but without above objects
```

Although the Square, Triangle and Circle objects become obsolete, references to these objects remain. References r, c, and t continue to hold the addresses of these obsolete objects. The garbage collector will not reclaim the memory for these three objects.

### Static Data and Methods

We have described classes as templates or blueprints for creating objects. In fact, classes are a bit more than that.

Although a class is not an object, a class can have **both** class variables and class methods. Class variables and class methods can exist whether or not any object is created. Class variables and class methods are indicated with the keyword `static`.

If a class contains a static variable then:

- All objects/instances of the class **share** that variable.
- There is only one version of the variable defined for the whole class.
- The variable belongs to the class.
- The variable exists regardless of whether or not any objects have been created.
- The variable may be accessed using either the class name or an object name, if an object has been created.

Static or class variables are often used for constants. The Math class contains two class constants: Math.PI and Math.E (approximate value: 2.71828).

A class/static method, as we have already seen, is a method that:

- exists as a member of a class,
- may be invoked using either the class name or the name of an object,
- may not access instance variables. (Since class methods may be invoked regardless of whether or not any object have been created, object/instance variables cannot be accessed by static methods.)
- is often used when it is important to know how many class instances exist or to restrict the number of instances of a class.
The Java System library contains many class/static methods. The methods of Math are all static as are the (final) variables. Of course, allowing static methods and variables is contrary to the principles of object-oriented programming since Java is providing a mechanism for what amounts to global variables and methods!

Example:

The following simple class contains two class/static variables:

- The variable count keeps track of the number of Circle objects that have been created. One version of count exists for the entire class. All objects access this same variable.

- The static variable pi is a constant. Notice the keyword final. Also note that pi is declared public so that any other class may access it as Circle.pi.

The class method, getCount(), returns the number of Circle objects that have been created. Notice that getCount() accesses count – a class/static variable. Remember: a static method cannot access an instance variable, since instance variables may not even exist.

The example also illustrates use of the keyword this. Sometimes the parameter has the same name as one of the instance variables. To differentiate between the parameter and the private variable, use the keyword this. As in C++, this is a reference to the invoking object.

```java
public class Circle {

    static private int count = 0; //class variable
    public final static double pi = 3.14159; //class variable
    private double radius;

    //constructors
    public Circle() { //default constructor
        radius = 0;
        count++;    
    }
    public Circle(double radius) { //one argument constructor
        this.radius = radius; //this.radius designates variable from the class
        count++; 
    }

    public double area() {
        return pi * radius * radius;
    }

    public static int getCount() { // this is a CLASS method
        return count;
    }

}
```
Java and Object Oriented Programming  
Day 2 - Encapsulation

The following small class uses both the static data and methods of Circle.

```java
public class Test {
    public static void main(String args[]) {
        System.out.println("" + Circle.pi + " + Circle.getCount());
    }
}
```

The output from this class is:
```
3.14159 0
```

**Inner Classes**

An *inner class* is a class that is defined within the scope of another class. That is, an inner class is a nested class. Java specifies that if class *Inner* is nested inside class *Outer*:

- Inner has access to the data and methods of Outer – Inner can “see out.”
- Outer does not have access to members of Inner – Outer cannot “see in.”
- Inner is visible to Outer but not outside of Outer

Thus, for example, the following code will compile:

```java
public class Outer {
    private int a;
    private int b;
    private class Inner {
        private int c;
        private int d;
        public Inner() {
            a = 5; // Inner had access to a and b of Outer
            b = 6;
            c = 3;
            d = 4;
        }
    }
    public Outer() {
        Inner x = new Inner();
    }
}
```

However, this next fragment contains an **error**.

```java
public class Outer {
    private int a;
    private int b;
    private class Inner {
        private int c;
        private int d;
    }
```
Java and Object Oriented Programming
Day 2 - Encapsulation

public Outer()
{
    a = 5;
    b = 6;
    c = 3;  //Error: Outer does not have access to c and d.
    d = 4;
}

Inner classes are a convenience, not a necessity. Later, when we study event-driven programming, you will see the real expediency of inner classes.

The next example illustrates inner classes in the construction of a stack.

**Example:**

Recall that a stack is a list of items such that data is inserted and deleted from one end only, the top of the stack. A stack is often called a last-in-first-out list (LIFO). Stack operations include:

- push(x) – add item x to the top of the stack
- pop() – remove and return the top item from the stack
- empty() – returns true if the stack contains no items
- peek() – returns the top item from the stack but leaves the stack unchanged.

A common method of implementing a stack utilizes a linked structure:

As the diagram indicates, each node in the linked structure contains two fields: a field that holds stack data (data) and a reference to the next node (next). Each of the stack implementations below uses the class Node. The first uses an inner class and the second does not. Notice that the second version of Stack uses accessor and mutator methods to access the fields in a Node.

<table>
<thead>
<tr>
<th>Using an inner class:</th>
<th>Using an external node class</th>
</tr>
</thead>
</table>
| public class Stack    // a stack of char
{                     |
 private class Node    |
 {                     |
     private char data;
     private Node next;
     //constructors
     public Node()       |
     //default constructor
     {                   |
         data = " ";    |
         next = null;    |
     }                   |
     public Node(char x) |
     {                   |
         // one argument constructor
         data = x;
         next = null;     |
     }                   |
}                     |
| public class Node     |
 {                     |
     private char data;
     private Node next;
     //constructors
     public Node()       |
     //default constructor
     {                   |
         data = " ";    |
         next = null;    |
     }                   |
     public Node(char x) |
     {                   |
         // one argument constructor
         data = x;
         next = null;     |
     }                   |
private Node top; // top is a reference

public Stack() // default constructor
{
    top = null;
}

public boolean empty()
{
    return top == null;
}

public void push(char x)
{
    Node p = new Node(x);
    p.next = top;
    top = p;
}

public char pop()
{
    if (empty())
    {
        System.out.println(
            "Stack underflow");
        System.exit(0);
        return 0;
    }
    else
    {
        char x = top.data;
        top = top.next;
        return x;
    }
}

public char peek()
{
    if (empty())
    {
        System.out.println(
            "Stack underflow");
        System.exit(0);
        return 0;
    }
    else
    {
        char x = top.data;
        return x;
    }
}

public char getData()
{
    return data;
}

public void setData(char x)
{
    data = x;
}

public void setLink(Node p)
{
    next = p;
}

public Node getLink()
{
    return next;
}

private Node top; // top is a reference

public class Stack1 // a stack of characters
{
    private Node top;

    public Stack1() // default constructor
    {
        top = null;
    }

    public boolean empty()
    {
        return top == null;
    }

    public void push(char x)
    {
        Node p = new Node(x);
        p.setLink(top);
        top = p;
    }

    public char pop()
    {
        if (empty())
        {
            System.out.println(
                "Stack underflow");
            System.exit(0);
            return 0;
        }
        else
        {
            char x = top.data;
            top = top.next;
            return x;
        }
    }

    public char peek()
    {
        if (empty())
        {
            System.out.println(
                "Stack underflow");
            System.exit(0);
            return 0;
        }
        else
        {
            char x = top.data;
            return x;
        }
    }

    public char getData()
    {
        return data;
    }

    public void setData(char x)
    {
        data = x;
    }

    public void setLink(Node p)
    {
        next = p;
    }

    public Node getLink()
    {
        return next;
    }
}
The String class

In C, Strings are special character arrays. C++ supports C’s view of strings but, in the spirit of object oriented programming, also supplies a string class. Like C++ strings, Java strings are genuine objects. However, unlike C++, Java String objects are immutable. Once a String object is created it cannot be changed, although a String reference may be changed. This is a simple but subtle point that is not as restrictive as it might seem at first.

Java provides the following methods for handling strings.

**Constructors**

```java
String s = new String();  //Empty string
String s = new String("Popeye");
String s = "Olive Oyl";
```

**public char charAt(int index)** returns the character located at index.

```java
String s = "Dopey";
s.charAt(3) returns ‘e’.
```

**public int compareTo(String s)**

```java
String s = "Grumpy";
String t = "Happy";
s.compareTo(t) returns a negative number.
s.compareTo(s) returns 0.
t.compareTo(s) returns a positive number.
```
public int compareToIgnoreCase(String s)
    String s = “ABC”;
    String t = “abc”;
    s.compareTo(t) returns a negative number.
    s.compareToIgnoreCase(t) returns 0.

public String concat(String s)
    String s = “Olive”;
    s.concat(“Oyl”) returns the string “Olive Oyl”.

public boolean endsWith(String suffix)
    String s = “Dopey”;
    s.endsWith(“ey”) returns true.

boolean equals(String s)
    String s = “ABC”;
    String t = “abc”;
    s.equals(t) returns false

public boolean equalsIgnoreCase(String s)
    String s = “ABC”;
    String t = “abc”;
    s.equalsIgnoreCase(t) returns true.

public int length()
    String s = “Sponge Bob”;
    s.length() returns 10.

public String replace(char oldChar, char newChar)
    String s = “Slinky”;
    s.replace(‘l’, ‘t’) returns “Stinky”

public boolean startsWith(String prefix)
    String s = “Mississippi”;
    s.startsWith(“Miss”) returns true
    s.startsWith(“miss”) returns false

public String substring(int index)
    String s = “Ethel Mertz”;
    s.substring(4) returns “l Mertz” (returns rest of string from position 4)

public String substring(int start, int end)
    String s = “Ethel Mertz”;
    s.substring(4,8) returns “l Me” (end is the position following the last character to extract).

public char[] toCharArray()
    String s = “Doc”;
    s.toCharArray() returns the array {‘D’, ‘o’, ‘c’}.

public String toLowerCase()
    String s = “Doc”;
    s.toLowerCase() returns “doc”.
public String toString()
    String s = “Doc”;
    s.toString() returns “Doc” – the same string. However, we will be using toString() to get a
    string representation of other objects.

public String toUpperCase()
    String s = “Doc”;
    s.toUpperCase() returns “DOC”.

public String trim()
    returns the string with all leading and trailing white space removed.
    String s = “    Dopey    “;
    s.trim() returns “Dopey”.

Static (Class) Methods

public String valueOf(char [] data)
    Suppose x is the character array containing ‘D’ ‘o’ ‘c’, then
    String.valueOf(x) returns the String “Doc”.

public String valueOf(double x)
    String.valueOf(3.14159) returns the String “3.14159”.

public String valueOf(int x)
    String.valueOf(314) returns the String “314”.

public String valueOf(char c)
    String.valueOf(‘a’) returns the String “a”

Concatenation

The + and += operators are overloaded to allow string concatenation.

Example:
    String s = “Happy”;
    String t = “Grumpy”
    String w = s+“ “+t; // w is “Happy Grumpy”

    String s = “Sleep”
    s += “y”;
    System.out.println(s); // the output is “Sleepy”

String Objects Immutable

Wait a minute... Aren’t String objects immutable?

Here is what happens: s += “y” is the same as s = s + “y”;

A new string object is created which is the concatenation of String s and “y” (i.e., “Sleep” and “y”).
The address of this new string is assigned to reference s. The string “Sleep” is no longer
referenced by s (and may very well be garbage collected).
When using strings, you should also be wary of using the == operator for comparison:

Consider the following code fragment:

```java
String s = "ABC";
String t = "ABC";

System.out.println("s is " +s);
System.out.println("t is " + t);
System.out.println("s == t "+ (s==t));

s = "AB";
s+='C';

System.out.println("s is " +s);
System.out.println("t is " + t);
System.out.println("s == t "+ (s==t));
System.out.println("s.equalTo(t) " + s.equalTo(t));
```

The output:

```
s is  ABC
t is ABC
s == t true

s is  ABC
t is ABC
s == t false
s.equals(t) true
```

Remember that s and t are references. The == operator compares addresses. Initially, s and t both hold the address of string literal “ABC”. However, after a bit of manipulation, s no longer holds that same address even though the characters of string s and string t are the same. Consequently, s==t returns false. In contrast, s==t returns true when using the equals() method, because equals() compares the values of the strings and not the references.

***************************************************
StringBuffer Class -- when a string must be changed
***************************************************

A StringBuffer object has a capacity that automatically expands as needed. As we have seen, a String object is immutable, i.e., once created, a String object cannot change. On the other hand, you may add or delete characters from a StringBuffer object. For those programs which are heavy in string modifications, StringBuffer may improve performance. Otherwise the String class is preferable.

**Constructors**

```java
StringBuffer s = new StringBuffer();       // empty string with initial capacity 16 characters
StringBuffer s  = new StringBuffer(50);  // empty string with capacity 50
StringBuffer s = new StringBuffer ("Hello"); // initializes s to “Hello”
```
Methods:
The following methods append a String (or a string representation of another type) to a StringBuffer object and return a reference to the modified StringBuffer. Unlike the corresponding String operation, a new StringBuffer is not created.

public StringBuffer append(char c)
public StringBuffer append(char[] c)
public StringBuffer append(int i)
public StringBuffer append(double x)
public StringBuffer append(String s)

    StringBuffer s = new StringBuffer("Sleep");
    StringBuffer t;
    t = s.append("y");
    System.out.println(s);
    System.out.println(t);

    output: Sleepy
    Sleepy

public int capacity( )
    StringBuffer s = new StringBuffer();
    int x = s.capacity();  // x has the value of 16, by default

public char charAt(int index)

public StringBuffer delete(int start , int end)
removes all characters from start position up to the character before the end position. All following characters are shifted, shortening the string buffer. A reference to the shortened string buffer is returned.

    StringBuffer s = new StringBuffer("Hello");
    StringBuffer t;
    t = s.delete(0,3);
    System.out.println(s);
    System.out.println(t);

    output: lo
    lo

public StringBuffer deleteCharAt(int index)

    StringBuffer s = new StringBuffer("Hello");
    StringBuffer t;
    t = s.deleteCharAt(1);
    System.out.println(s);
    System.out.println(t);

    Output  Hllo
    Hllo
public StringBuffer insert(int index, char[] s)
public StringBuffer insert(int index, anyPrimitiveType s)
public StringBuffer insert(int index, String s)
    StringBuffer s = new StringBuffer("ABC");
    StringBuffer t;
    t = s.insert(1,"XYZ");
    System.out.println(s);
    System.out.println(t);

    Output: AXYZBC
           AXYZBC

public int length()

public StringBuffer replace(int start, int end, String s)
deletes the substring of characters starting at start up to the character before end and replaces it with String s.

    StringBuffer s = new StringBuffer("Grumpy");
    StringBuffer t;
    t = s.replace(0,2,"L");
    System.out.println(s);
    System.out.println(t);

    Output: Lumpy
           Lumpy

public StringBuffer reverse()
    StringBuffer s = new StringBuffer("Grumpy");
    StringBuffer t;
    t = s.reverse();
    System.out.println(s);
    System.out.println(t);

    Output: ypmurG
           ypmurG
public void setCharAt(int index, char ch)
StringBuffer s = new StringBuffer("Grumpy");
s.setCharAt(2,'i');
System.out.println(s); // output: Grimpy

public void setLength(int len)
truncates or pads the contents of the String buffer.
StringBuffer s = new StringBuffer("Grumpy");
s.setLength(25);
System.out.println(s+"xxx");

Output: Grumpy xxx // 19 spaces between Grumpy and xxx

public String substring(int index)
public String substring(int start, int end)
Note: substring returns a String

String w;
StringBuffer s = new StringBuffer("Grumpy");
w = s.substring(3);
System.out.println(w); // Output is mpy

public String toString()
returns the String stored in the StringBuffer.

StringBuffer sb = new StringBuffer("Sleazy");
String s = sb.toString();

Some Notes on Strings and String Buffers

• The concatenation operator "+" cannot be applied to a StringBuffer. Instead, use append.

• A program that repeatedly uses the + operator for String concatenation may experience some deterioration in performance. A slowdown may occur because whenever two Strings are concatenated, the following three actions occur in sequence:
  1. A temporary StringBuffer object is created.
  2. The two String objects are copied to the temporary StringBuffer.
  3. The new concatenated String is created.

• For programs that have heavy use of concatenation, it may be better to first use the StringBuffer method, append, and then convert the StringBuffer object to a String object using the toString() method.

• There is a method
  boolean equals(StringBuffer s)
that tests the references and not the contents of a StringBuffer. This is in contrast to the equals method of the String class, which compares the contents.

• The StringBuffer class adds no new functionality to Java. However, for programs that build Strings from user input or from a file, using StringBuffer can be more efficient and improve performance.
The BigInt Class - An Example Using Strings

In any Java application the maximum value of a long int is 9223372036854775807 – a mere 19
digits. The following class, BigInt, is capable of storing and adding integers of arbitrary length.
Since Java’s long integers contain at most nineteen digits, the BigInt class stores an integer as a
string of characters (e.g. “9876543212345678987654321”).

The methods of BigInt include:

- Two constructors
- An addition method **BigInt add(BigInt x)**
- Accessor and mutator methods

The following partial implementation of BigInt includes the constructors, as well as accessor (get)
and mutator (set) methods.

```java
public class BigInt {
    private String number; // holds a large integer as a string

    public BigInt() // default is 0
    {
        number = new String("0");
    }

    public BigInt(String number)
    {
        this.number = number;
    }

    public String getBigInt() // accessor
    {
        return number;
    }

    public void setBigInt(String number) // mutator
    {
        this.number = number;
    }
}
```

Adding Two BigInt objects.

The **add** method of BigInt uses a character stack. In OO terminology, a **BigInt object sends
messages (push, pop) to a stack object**.

To illustrate the addition algorithm, consider the sum of 9876 and 34:

**Step 1: Pad operand if necessary**

If one of the two operands has fewer digits that the other (as is the case with 9876 and 34), add
leading zeroes to the shorter operand so that both numbers consist of the same number of digits:
“9876” and “0034.” Recall that big numbers are implemented with strings.
Step 2: Initialize Stacks
Push the digits (leftmost first) of the first operand onto stack, s1.
Push the digits of the second operand onto stack, s2.
Initialize a third stack, s3, to empty.
Set (int) carry = 0:

Once the stacks have been initialized:

Step3: Loop until stacks are empty
Pop the top digit from each stack and add these two digits along with carry: 6 + 4 + 0 = 10.
Push the units digit of the resulting sum onto s3 and store the ten's digit in carry. Since the sum was 10, we push 0 onto s3, and set carry = 1

Continue the process until s1 and s2 are empty:

7 + 3 + 1 = 11
push 1 onto s3
carry = 1

8 + 0 + 1 = 9
push 9 onto s3
carry = 0
$$9 + 0 + 0 = 9$$
push 9 onto s3

$$\text{carry } = 0$$

Both stacks s1 and s2 are empty. Stop.

**Step 4: Retrieve the sum**
If carry == 1, push `'1` onto s3
The sum, 9910, is stored on the stack, s3.

The implementation is fairly simple:

```java
public BigInt add(BigInt x) // a member of the BigInt class
{
    Stack s1 = new Stack(); // for first operand -- Stack of char
    Stack s2 = new Stack(); // for second operand
    Stack s3 = new Stack(); // for sum
    String answer = new String("");
    BigInt temp;
    char ch1, ch2, ch3; // digits as characters
    int n1, n2, sum; //digits as numbers
    int carry = 0;

    // So that both stacks are the same size, push leading
    // zeroes on the stack which holds the smaller BigInt object.
    // At most one of the following two loops will execute.
    for (int i = 1; i <= (number.length()- x.number.length()); i++)
        s2.push('0');
    for (int i = 1; i <= (x.number.length()- number.length()); i++)
        s1.push('0');

    //push all digits for first operand on s1
    for (int i = 0; i < number.length(); i++)
        s1.push(number.charAt(i));

    //push all digits for the second operand on s2
    for (int i = 0; i < x.number.length(); i++)
        s2.push(x.number.charAt(i));

    // add digit by digit, keeping track of the carry digit
    while (!s1.empty())
    {
        ch1 = s1.pop(); // characters are on the stack
        ch2 = s2.pop(); //
        carry = ch1 + ch2 - '0';
        if (carry == 1)
            answer = '1' + answer;
        else
            answer = (char) (carry + '0') + answer;
    }
    return new BigInt(answer);
}
```

Using the BigInt Class

The following program allows a user to interactively add a list of arbitrarily long integers. A Sum object adds and stores the sum of \( n \) large integers. The value of \( n \) is supplied by the user.

```java
public class Sum {
    BigInt sum;
    public Sum() // constructor, initializes sum to 0
    {
        sum = new BigInt();// a call to default BigInt constructor
    }

    public void add(int count) // adds n BigInt objects
    {
        String num;// integers come from the user in the form of a String
        // read n integers and add each to sum
        for (int i = 1; i <= count; i++)
        {
            System.out.print(i +": ");
            num = MyInput.readString();
            sum = sum.add(new BigInt(num));
        }
    }

    public String getSum() // returns sum as a string
    {
        return sum.getBigInt(); //send a message to a BigInt object
    }
}
```
public static void main(String args[]) {
    Sum sum = new Sum();
    // trigger the action
    int n;
    System.out.print("How big is the list? ");
    n = MyInput.readInt();
    sum.add(n);

    // output the result
    System.out.println("Sum is "+ sum.getSum());
}

Output:
How big is the list? 4
1: 9999999999999999999999999999999999999
2: 9999999999999999999999999999999999999
3: 9999999999999999999999999999999999999
4: 1
Sum is 29999999999999999999999999999999999998

The DecimalFormat Class

A DecimalFormat object allows you to format floating point numbers using a pattern string. The DecimalFormat class is contained in the package java.text.

Constructor:
public DecimalFormat(String pattern)

Methods:

String format(double x) returns a String version of x which formatted according to the specifications of the pattern string.

void applyPattern(String pattern) supplies a new pattern string to a DecimalFormat object.

One form of a pattern string includes a decimal point and any number of "#" and "0" characters. A "0" in position i, indicates that a digit is required in position i, even if the digit is a leading 0. A "#" in position i indicates that the character in position i may be a digit or a blank. The character will be a digit as long as it is not a leading or trailing zero in which case the character in position i will be a blank.

<table>
<thead>
<tr>
<th>Number</th>
<th>pattern string</th>
<th>formatted number</th>
</tr>
</thead>
<tbody>
<tr>
<td>123.123456</td>
<td>0.###</td>
<td>123.123</td>
</tr>
<tr>
<td></td>
<td>0.</td>
<td>123.</td>
</tr>
<tr>
<td>8.125</td>
<td>00.##</td>
<td>08.13 (rounds)</td>
</tr>
<tr>
<td></td>
<td>##.0000</td>
<td>8.1250</td>
</tr>
<tr>
<td>.123</td>
<td>###,##</td>
<td>0.1 (Yes one zero will appear before the decimal)</td>
</tr>
<tr>
<td></td>
<td>0000.########</td>
<td>0000.123</td>
</tr>
</tbody>
</table>
Example
The following program demonstrates the DecimalFormat class with several pattern strings.

```java
import java.text.*;
public class Circle1
{
    static private int count = 0;   //class variable
    private double radius;

    //constructors
    public Circle1()
    {
        radius = 0;
        count++;
    }
    public Circle1 (double radius)
    {
        this.radius = radius;
        count++;
    }

    public double area()
    {
        return Math.PI*radius*radius;  // static constant from the Math class
    }

    public static int getCount()  // this is a CLASS method;
    {
        return count;
    }

    public static void main(String args[])
    {
        Circle1 c1 = new Circle1(23.5456789);
        Circle1 c2 = new Circle1(.1234567);

        DecimalFormat formatter = new DecimalFormat(".###");
        System.out.println("Area of c1:"+formatter.format(c1.area()));

        String pattern = "0.##";
        formatter.applyPattern(pattern);
        System.out.println("Area of c2:"+formatter.format(c2.area()));
        formatter.applyPattern("##");
        System.out.println("Area of c2:"+formatter.format(c2.area()));
        System.out.println("Number of circles: "+ getCount());
    }
}
```

Output:
Area of c1: 1741.694
Area of c2: 0.05
Area of c2:  .05
Number of circles: 2
The program works as follows:

1. `DecimalFormat formatter = new DecimalFormat(".###");`
   A DecimalFormat object (formatter) is created and initialized with the pattern "###"
   The pattern "###" indicates that there should be at most three decimal places.
   The symbol "#" is a placeholder for a digit or a blank.

2. `formatter.format(c1.area())`
   A message is passed to the formatter object. The message is `format(c1.area())` i.e.
   format the area according to the current pattern. Output is 1741.694 – three decimal places.

3. `String pattern = "0.##";`
   `formatter.applyPattern(pattern);`
   A new pattern string ("0.##") is given to the formatter object. The "0" indicates that a digit
   must precede the decimal point. The whole pattern indicates that there must be at least
   one digit preceding the decimal and at most two after the decimal. Output is 0.05

4. `formatter.applyPattern("##");`
   A new pattern is supplied to the formatter. The pattern specifies that there must be at
   most two decimal places. Nothing is specified to the left of the decimal. Output is .05

**The Random Class**

For most applications the random number generator, `random()`, contained in `java.lang.Math` is sufficient. This method returns a random number $x$ such that $0 < x < 1$.

Java provides a more complex class for generating pseudo-random numbers. The package `java.util.Random` includes methods that generate pseudo-random integers as well as doubles. Unlike the `random()` method of the Math class, however, the methods of the Random class are not static and a Random object must be instantiated.

**Constructor:**

```
public Random()
```

**Methods:**

- `int nextInt(int n)` returns a pseudo-random integer in the range 0 to n-1 inclusive.
- `int nextDouble()` returns a pseudo-random double in the interval $[0, 1)$.
- `double nextGauss()` returns a pseudo-random number from a normal (bell shaped) distribution with mean 0 and standard deviation 1 (Gaussian distribution).
Example:
The following Class Dice uses the Random class to simulate rolling dice. The main method is merely a test function that calculates the number of time snake eyes appears in 1000 rolls of the dice.

```java
import java.util.*;

public class Dice {
    Random random;
    public Dice() {
        random = new Random(); // get a random object
    }
    int roll() {
        int die1 = random.nextInt(6) + 1; // integer from 1 to 6,
        int die2 = random.nextInt(6) + 1;
        return die1 + die2;
    }

    public static void main(String args[]) {
        // a test method
        Dice dice = new Dice();
        int snakeEyes = 0;
        for(int i = 1; i <= 1000; i++)
            if(dice.roll() == 2)
                snakeEyes++;
        System.out.println("Number of Snake eyes in 1000 rolls: " + snakeEyes);
    }
}
```

Output from three runs:
Number of Snake eyes in 1000 rolls: 31
Number of Snake eyes in 1000 rolls: 35
Number of Snake eyes in 1000 rolls: 14  // yes it can happen!!!

Next stop: inheritance and polymorphism.