Lecture 32
Prepare to play Poker --- Assigning values to Poker hands

- Computing the value of a Poker hand
  - The value of a Poker hand depends on the type of Poker hand

- We can give the following general method that determines the value of a Poker hand

```java
/* -------------------------------
  valueHand(): return value of a hand
  ------------------------------- *
*/

public static int valueHand( Card[] h )
{
    if ( isFlush(h) && isStraight(h) )
        return valueStraightFlush(h);
    else if ( is4s(h) )
        return valueFourOfAKind(h);
    else if ( isFullHouse(h) )
        return valueFullHouse(h);
    else if ( isFlush(h) )
        return valueFlash(h);
    else if ( isStraight(h) )
        return valueStraight(h);
    else if ( is3s(h) )
        return valueThreeOfAKind(h);
    else if ( is2s(h) )
        return valueTwoPairs(h);
    else if ( is1s(h) )
        return valueOnePair(h);
    else
        return valueHighCard(h)  // Lowest Poker hand;
}
```

- Assigning values to Poker hands: another encoding scheme

  - In order to compare Poker hands, we want to assign values to Poker hands such that:

    \[
    \text{Value( a better Poker hand )} > \text{Value( a worse Poker hand )}
    \]
- *Inter* and *intra* Poker hand comparisons

  - 2 kinds of comparisons:
    - *Inter Poker hands*: comparison between *different kinds* of Poker hands
    - *Intra Poker hands*: comparison between *same kind* of Poker hands

- *Inter* Poker hand comparisons:

  Royal Flush
  > Straight Flush
  > 4 of a Kind
  > Full House
  > Flush
  > Straight
  > 3 of a Kind
  > 2 pairs
  > 1 pair
  > High Card

- *Intra* Poker hand comparisons:

  Royal Flush: all equal value
  Straight Flush: highest rank card wins
  4 of a Kind: highest rank quads wins
  Full House: highest rank set (3) cards wins
  Flush: highest rank card wins
  Straight: highest rank card wins
  3 of a Kind: highest rank set (3) cards wins
  2 pairs: highest pair wins
  if tie, lowest pair wins
  if also tie, highest unmatched card wins
  1 pair: highest pair wins
  if tie, highest unmatched card wins
  if also tie, second highest unmatched card wins
  if also tie, lowest unmatched card wins
  High Card: highest rank card wins
  if tie, second highest rank card wins
  and so on...
- Value encoding method that takes care of both *inter* and *intra* Poker hand comparision

  - **Encoding method** for the *value* of a *Poker hand*:

    ![Diagram](image)

    - **Within** the *same type (kind)* of *Poker hand*:
      - A *higher ranked* hand is assigned a *higher value*
      - We make sure the *assigned value* does *not exceed* the *prescribe range*
Example:

Flush range

500000

7 high flush

23457

8 high flush

23467

Ace high flush

23567

600000

Explanation:

- Suppose you have 3 flush hands:
  - Spade Flush with cards 2, 3, 4, 5, 7
  - Heart Flush with cards 2, 3, 4, 6, 7
  - Club Flush with cards 2, 3, 5, 6, 7

- The different flush hand must receive different values in order to tell them apart !!!
- **Inter Poker hand encoding**
  - The *inter Poker hand* encoding is used to separate the *different kinds* of *Poker hand*.
  - We can use the following *function* to do this job:

  ![Table](image)

- **Note:**
  - *We must* make sure that:

  ![IntraValue](image)

  *Otherwise, we cannot satisfy* the *value encoding method* given in:

  ![Diagram](image)
Computer playing Poker

- Playing the Poker game between 2 players
  - Writing Poker game is now very easy using the methods contained in the Poker class
  - Here is a program that simulate a 2 player Poker game:

```
Shuffle the deck of card;
Deal 5 cards to player1
Deal 5 cards to player2

if ( Poker.valueHand(player1) > Poker.valueHand(player2) )
  System.out.println("Player 1 wins");
else if ( Poker.valueHand(player1) < Poker.valueHand(player2) )
  System.out.println("Player 2 wins");
else
  System.out.println("Player hands are a tie");
```
Program in Java:

```java
public class PlayPoker {
  public static void main(String[] args) {
    DeckOfCards a;

    Card[] player1 = new Card[5];
    Card[] player2 = new Card[5];

    int i;

    a = new DeckOfCards();
    System.out.println(a);

    System.out.println("Shuffle cards....");
    a.shuffle(1000);                    // Shuffle up...
    System.out.println(a);

    // Deal!
    for (i = 0; i < 5; i++) {
      player1[i] = a.deal();
      player2[i] = a.deal();
    }

    System.out.print("Player 1's hand: ");
    for (i = 0; i < 5; i++) {
      System.out.print(player1[i] + " ");
    }
    System.out.println(" - value = " + Poker.valueHand(player1));

    System.out.println("Player 2's hand: ");
    for (i = 0; i < 5; i++) {
      System.out.print(player2[i] + " ");
    }
    System.out.println(" - value = " + Poker.valueHand(player2));
    System.out.println();
    System.out.println();

    if ( Poker.valueHand(player1) > Poker.valueHand(player2) )
      System.out.println("Player 1 wins");
    else if ( Poker.valueHand(player1) < Poker.valueHand(player2) )
      System.out.println("Player 2 wins");
    else
      System.out.println("Player hands are a tie");
    System.out.println();
  }
}```
The **String class**

- **String** is a *user-defined class* that is created by the programmers who implemented the Java library.
- Here is the **beginning** of the description of the **String class** in the Java documentation:

```
java.lang

Class String

java.lang.Object
  - java.lang.String

All Implemented Interfaces:
  - Serializable, CharSequence, Comparable<String>

public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.
```
Defining **String** variables:

- Using the **object definition syntax** (See: click here)

```java
String s;          // s is an identifier
s = new String( "Hello World " );  // creates a String object and
        // calls a constructor method
```

- The **designer** of the Java language wanted to **treat Strings special** and they made the Java compiler recognize a **short hand notation** for string creation:

```java
String s;          // s is an identifier
s = "Hello World "; // Short hand for: new String("Hello World");
```

```
public static void main(String[] args)
{
    ...
    String s;
    s = "Hello";
    ...
}
```

---

**Recall that:**

- `s = "Hello";` is **short hand for:**
- `s = new String("Hello");`

---

**RAM memory**

```
main
```

```
S
```

---

5600

```
H
```

```
E
```

```
I
```

---

5600
- **length()**: returns the **length** of the string

**Examples:**

```java
String s = "Abc", t = "Hello";
s.length()   returns 3
t.length()   returns 5
```

- **charAt(n)**: returns the **character** at position **n** in the string

**Examples:**

```java
String s = "Abc", t = "Hello";
s.charAt(1)  returns 'b'
t.charAt(1)  returns 'e'
```

- **substring(i,j)**: returns the **substring** from position **i** **upto (not including)** position **j** of the string

**Examples:**

```java
String s = "Abc", t = "Hello";
s.substring(0,2)  returns "Ab"
t.substring(1,4)  returns "ell"
```
**Concatenating strings:**

- Using an *instance method* (defined inside the *String class*):

```java
String s = "Abc", t = "Hello";
s.concat("xyz") returns "Abcxyz"
t.concat(s) returns "HelloAbc"
```

Note: We would normally store the return value in a *String variable*

```java
String r;
r = t.concat(s); // r = "HelloAbc"
```

- The *designers of the Java language* wanted to *treat Strings special* and they made the *Java compiler* recognize a *short hand notation* for *string concatenation*:

```java
String s = "Abc", t = "Hello";
s + "xyz" // Synonym for: s.concat("xyz")
t + s // Synonym for: t.concat("s")
```

Note: We would normally store the return value in a *String variable*

```java
String r;
r = t + s // Same as: r = t.concat(s);
```
Operations on Strings: *instance methods*

- **Fact:**

  - *String* is a *class* in Java

  In other words, someone wrote:

  ```java
  public class String
  {
    instance variables...
    
    instance methods such as:
    
    public char charAt()
    {
      ...
    }
    
    public int length()
    {
      ...
    }
    
    public String substring(int a, int b)
    {
      ...
    }
  }
  
  All operations on Strings are implemented as *instance methods*

- The **general format** to use an *operation on a String object*:

  ```java
  String s1;
  s1.StringOperationMethod( parameters )
  ```
Comparison operations (instance methods) on Strings

- String:
  - A String consists of a sequence of characters

- Representing characters by numbers:
  - The characters are represented (encoded) using the Unicode encoding method.
  - The first 128 codes in the Unicode table are used to represent English characters.
  - The first 128 codes in the Unicode encoding scheme is in fact the ASCII code.
  - The ASCII code table:

```
<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
</tr>
<tr>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
</tr>
<tr>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
</tr>
<tr>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
</tr>
<tr>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
</tr>
<tr>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>80</td>
</tr>
<tr>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
</tr>
<tr>
<td>90</td>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
<td>96</td>
<td>97</td>
<td>98</td>
</tr>
<tr>
<td>99</td>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
</tr>
<tr>
<td>108</td>
<td>109</td>
<td>110</td>
<td>111</td>
<td>112</td>
<td>113</td>
<td>114</td>
<td>115</td>
<td>116</td>
</tr>
<tr>
<td>117</td>
<td>118</td>
<td>119</td>
<td>120</td>
<td>121</td>
<td>122</td>
<td>123</td>
<td>124</td>
<td>125</td>
</tr>
<tr>
<td>126</td>
<td>127</td>
<td>128</td>
<td>129</td>
<td>130</td>
<td>131</td>
<td>132</td>
<td>133</td>
<td>134</td>
</tr>
</tbody>
</table>
```

x == y
test whether x and y are pointing to the same String object

RAM memory

x 5600

main

y 5600

Hello

5600

RAM memory

x 5600

main

y 7800

Hello

7800

x != y

5600
public class StringEq1
{
    public static void main( String[] args )
    {
        String a, b, c;

        a = "Hello";
        a = a + " World"; // a = "Hello World"

        b = a; // b points to the same string as a
        System.out.println("a = " + a);
        System.out.println("b = " + b);

        if ( a == b )
            System.out.println("a == b");
        else
            System.out.println("a != b");

        /*-----------------------------------------------*/

        c = "Hello World"; // c contains the same string as a
        // But: c does not point to the same string
        System.out.println("a = " + a);
        System.out.println("c = " + c);

        if ( a == c )
            System.out.println("a == c");
        else
            System.out.println("a != c");
    }
}
• **Comparing if strings are equal based on their content**

  - The *instance method* used to *compare equality of two strings* `s1` and `s2` based on their *content*:

    ```java
    s1.equals(s2)
    returns:
    true if string s1 and s2 contains the same characters
    false otherwise
    ```

  - **Example:**

    ```java
    public class StringEq1
    {
        public static void main( String[] args )
        {
            String a, b, c;
            a = "Hello";
            a = a + " World";  // a = "Hello World"
            b = a;             // b points to the same string as a
            System.out.println("a = " + a);
            System.out.println("b = " + b);
            if ( a.equals(b) )
                System.out.println("a EQUALS b");
            else
                System.out.println("a NOT EQUALS b");
            /* ----------------------------------------- */
            c = "Hello World";  // c contains the same string as a
            // But: c does not point to the same string
            System.out.println("a = " + a);
            System.out.println("c = " + c);
            if ( a.equals(c) )
                System.out.println("a EQUALS c");
            else
                System.out.println("a NOT EQUALS c");
        }
    }
    ```
• Comparing if strings are equal without considering upper/lower case difference
  
  The `instance method` used to compare `equality of two strings` $s_1$ and $s_2$ without considering upper/lower case difference:

  ```java
  $s_1$.equalsIgnoreCase($s_2$)
  
  returns:
  
  true if string $s_1$ and $s_2$ contains the same characters when ignoring case difference
  false otherwise
  ```

Examples:

```
"abc".equalsIgnoreCase("ABC") returns true
```

Java program that illustrates this fact:

```java
public class StringEq3
{
    public static void main(String[] args)
    {
        System.out.println("\"abc\".equals("\"ABC\" ) - "+"abc".equals("ABC" ) );
        System.out.println("\"abc\".equalsIgnoreCase("\"ABC\" ) - "
        +"abc".equalsIgnoreCase("ABC" ) );
    }
}
```

Output:

```
"abc".equals("ABC") = false
"abc".equalsIgnoreCase("ABC") = true
```
• Ranking 2 strings: `compareTo()

○ The *instance method* used to *rank of two strings* `s1` and `s2` in *alpha-numerical order*:

```
s1.compareTo(s2)
```

returns:

```
0  if s1.equals(s2)
< 0  if s1 lexicographically precedes s2
> 0  if s1 lexicographically succeeds s2
```

○ The *lexicographical ordering* of the characters are *defined* by the *ordering* of the characters in the *ASCII table*:

```
0  NUL  1  SOH  2  STX  3  ETX  4  EOT  5  ENQ  6  ACK  7  BEL
8  BS  9  HT  10  NL  11  VT  12  NP  13  CR  14  SO  15  SI
16  DLE  17  DC1  18  DC2  19  DC3  20  DC4  21  NAK  22  SYN  23  ETB
24  CAN  25  EM  26  SUB  27  ESC  28  FS  29  GS  30  RS  31  US
32  SP  33 !  34 "  35 #  36 $  37 %  38 &  39 '
40 (  41 )  42 *  43 +  44 ,  45 -  46 .  47 /
48 0  49 1  50 2  51 3  52 4  53 5  54 6  55 7
56 8  57 9  58 :  59 ;  60 <  61 =  62 >  63 ?
64 @  65 A  66 B  67 C  68 D  69 E  70 F  71 G
72 H  73 I  74 J  75 K  76 L  77 M  78 N  79 O
80 P  81 Q  82 R  83 S  84 T  85 U  86 V  87 W
88 X  89 Y  90 Z  91 [  92 \  93 ]  94 ^  95 _
96 `  97 a  98 b  99 c  100 d  101 e  102 f  103 g
104 h  105 i  106 j  107 k  108 l  109 m  110 n  111 o
112 p  113 q  114 r  115 s  116 t  117 u  118 v  119 w
120 x  121 y  122 z  123 {  124 |  125 }  126 ~  127 DEL
```

In other words, a *partial lexicographical ordering* is:

```
'A' < 'B' < 'C' < ... < 'Z' < '[' < '\' < ... < 'a' < 'b' < ... < 'z' ...
```

○ Example:

```
"abc".compareTo("abc")  returns 0
"abc".compareTo("xyz")  returns < 0
"abc".compareTo("ABC")  returns > 0
```
• Ranking 2 strings without considering case: `compareToIgnoreCase()`
  
  ◦ The `instance method` used to `rank` of two strings `s1` and `s2` without considering upper/lower case in `alpha-numerical order`:

    ```java
    s1.compareToIgnoreCase( s2 )
    ```
    
    returns:
    
    0    if `s1.equals(s2)` ignoring upper/lower case
    < 0  if `s1 lexicographically precedes s2` ignoring upper/lower case
    > 0  if `s1 lexicographically succeeds s2` ignoring upper/lower case

  
  ◦ Example:

    ```java
    "abc".compareToIgnoreCase( "abc" ) // returns 0
    "abc".compareToIgnoreCase( "xyz" ) // returns < 0
    "abc".compareToIgnoreCase( "ABC" ) // returns > 0
    ```

  ◦ Java program:

    ```java
    public class StringCompI {
        public static void main( String[] args ) {
            System.out.println( "abc".compareToIgnoreCase( "abc" ) == 
                + "abc".compareToIgnoreCase( "abc" ) );
            System.out.println( "abc".compareToIgnoreCase( "xyz" ) == 
                + "abc".compareToIgnoreCase( "xyz" ) );
            System.out.println( "abc".compareToIgnoreCase( "ABC" ) == 
                + "abc".compareToIgnoreCase( "ABC" ) );
        }
    }
    ```
Finding substrings (instance methods) in Strings

- Finding the first occurrence of a substring
  - The instance method used to find (locate) the first occurrence of a substring sub in a string s is:

```java
s.indexOf(sub)
```

returns:

the (integer) index (position) of the first occurrence of the substring sub in string s

-1 if substring sub is not found in string s

Examples:

```
0123456789...
"Hello World, Bye World".indexOf("World")    returns: 6

0123456789012345678901
"Hello World, Bye World".indexOf("By") = 13

0123456789012345678901
"Hello World, Bye World".indexOf("by") = -1
```
- Finding the *last* occurrence of a *substring*

  - The *instance method* used to *find* (*locate*) the *last occurrence* of a *substring* `sub` in a *string* `s` is:

```java
s.lastIndexOf( sub )
```

returns:

- the (integer) index (position) of the last occurrence of the substring `sub` in string `s`
- `-1` if substring `sub` is not found in string `s`

Examples:

```
0123456789012345678901
"Hello World, Bye World".lastIndexOf( "World" ) returns: 17
```
public class IndexOf1
{
    public static void main( String[] args )
    {
        String s;
        int i;

        // 0123456789012345678901
        s = "Hello World, Bye World";
        i = s.indexOf( "World" );
        System.out.println( "0123456789012345678901" );
        System.out.println( "+ s + \\
                        + ".indexOf( \"World\" ) = " + i );
        i = s.indexOf( "By" );
        System.out.println( "0123456789012345678901" );
        System.out.println( "+ s + \\
                        + ".indexOf( \"By\" ) = " + i );
        i = s.indexOf( "by" );
        System.out.println( "0123456789012345678901" );
        System.out.println( "+ s + \\
                        + ".indexOf( \"by\" ) = " + i );
        /* ---------------------------------------- */
        i = s.lastIndexOf( "World" );
        System.out.println( "0123456789012345678901" );
        System.out.println( "+ s + \\
                        + ".lastIndexOf( \"World\" ) - " + i );
    }
}
Some **transformation operations** (instance methods) on Strings

- Transform a string into *lower case letters*
  - The *instance method* used to **transform all upper case letters** in a string $s$ into *lower case letters* is:

    ```java
    s.toLowerCase()
    ```

    **returns:**
    
    a new string where all upper case letters in $s$ are replaced by lower case letters

    **Note:**
    
    the original string $s$ is unchanged

**Examples:**

```
"AbC".toLowerCase() returns "abc"
"A&C".toLowerCase() returns "a&C"
```
• Transform a string into upper case letters
  
  ◦ The *instance method* used to *transform all lower case letters* in a *string s* into *upper case letters* is:

  ```
  s.toUpperCase()
  ```

  **returns:**

  A new string where all lower case letters in *s* are replaced by upper case letters

  **Note:**

  The original string *s* is unchanged

---

**Examples:**

```
"AbC".toLowerCase() returns "ABC"
"Ab%".toLowerCase() returns "AB%"
```
• Transform a string by *replacing a substring once*

  The *instance method* used to *replace* the *first occurrence of a substring old* inside a string *s* by the *substring new* is:

  ```java
  s.replaceFirst( old, new )
  ``

  **returns:**

  a new string where the first occurrence of "old" in *s* is replaced by "new"

  **Note:**

  the original string *s* is unchanged

---

**Examples:**

```
"Hello World, Bye World".replaceFirst( "World", "Class" )
```

returns "Hello Class, Bye World"
- Transform a string by replacing all occurrences of a substring
  - The instance method used to replace all occurrences of a substring old inside a string s by the substring new is:

    ```java
    s.replaceAll(old, new)
    ```

    returns:

    a new string where all occurrence of "old" in s is replaced by "new"

    Note:

    the original string s is unchanged

Examples:

```java
"Hello World, Bye World".replaceAll("World", "Class")
returns "Hello Class, Bye Class"
```
• A shared property of all instance methods in the String class --- that make strings in Java immutable

  ○ Property of all instance methods in the String class:

    • The instance methods in the String class do not change the implicit parameter

    Examples:

    ```java
    s = "AbC";
    s.toLowerCase() returns: "abc"
    Afterwards: s = "AbC"
    
    toLowerCase() does not change the implicit parameter s
    (s is still equal to "AbC")
    
    s = "Hello World, Bye World";
    s.replaceFirst( "World", "Class" ) returns: "Hello Class, Bye World"
    
    Afterwards: s = "Hello World, Bye World"
    
    replaceFirst( ) does not change the implicit parameter s
    (s is still equal to "Hello World, Bye World")
    ```
Converting *String* to *array of char* and vice versa

```java
char[] A;
A = new char[3];
A[0] = 'a';
A[1] = 'b';
```

```
String A;
A = "abH";
```
- **String and array of char in Java**
  - *String* and *array of char* are treated *differently* in Java

- **Example of differences:**
  - **In Java:**
    - a *String* is a *single object*
    - an *array of char* are a *serie (multiple) of variables*
  - **In Java:**
    - a *String* is *immutable* (because the *instance methods* in the *String* class will *not update the characters* inside a *String*
    - an *array of char* is *not protected* by a *private access modifier* (so we can *update* the characters stored in an *array of char*)
• **Converting a string into an array of char**

  - *Because Java treats strings and arrays of char differently*, we need a *conversion mechanism* to tell the Java compiler to *convert* one into another.

    *(No conversion mechanism is necessary in C/C++ to make strings into arrays of char and vice versa).*

  ◦ *(Instance) method in Java that *converts* a string into an array of char:

```
String s;
s = "....";
s.toCharArray() returns an array of char
    containing all characters in string s
```

Example:

```
String s;
s = "abc";
char[] A;
```
The other way: Converting an array of char into a String

To "convert" an array of char into a string object, we use a constructor method in the string class:

Example:

```java
char[] A = new char[3];
String s;
s = new String(A); // Make a string using the characters in array A
```

Example: Java program

```java
public class ArrayChar1 {
    public static void main( String[] args ) {
        String s = "abc";
        char[] A;
        /* -----------------------------------------------
         * Convert string s into an array of char
         * ----------------------------------------------- */
        A = s.toCharArray();
        System.out.println(s);
        for ( int i = 0; i < A.length; i++ )
            System.out.println("A[" + i + "] = " + A[i]);
        System.out.println("n");

        /* -----------------------------------------------
         * Convert an array of char A into string
         * ----------------------------------------------- */
        A = new char[4];
        s = new String(A);
        for ( int i = 0; i < A.length; i++ )
            System.out.println("A[" + i + "] = " + A[i]);
        System.out.println(s);
    }
}
```