



## Number, Array and String

### Session 5

---



Numbers

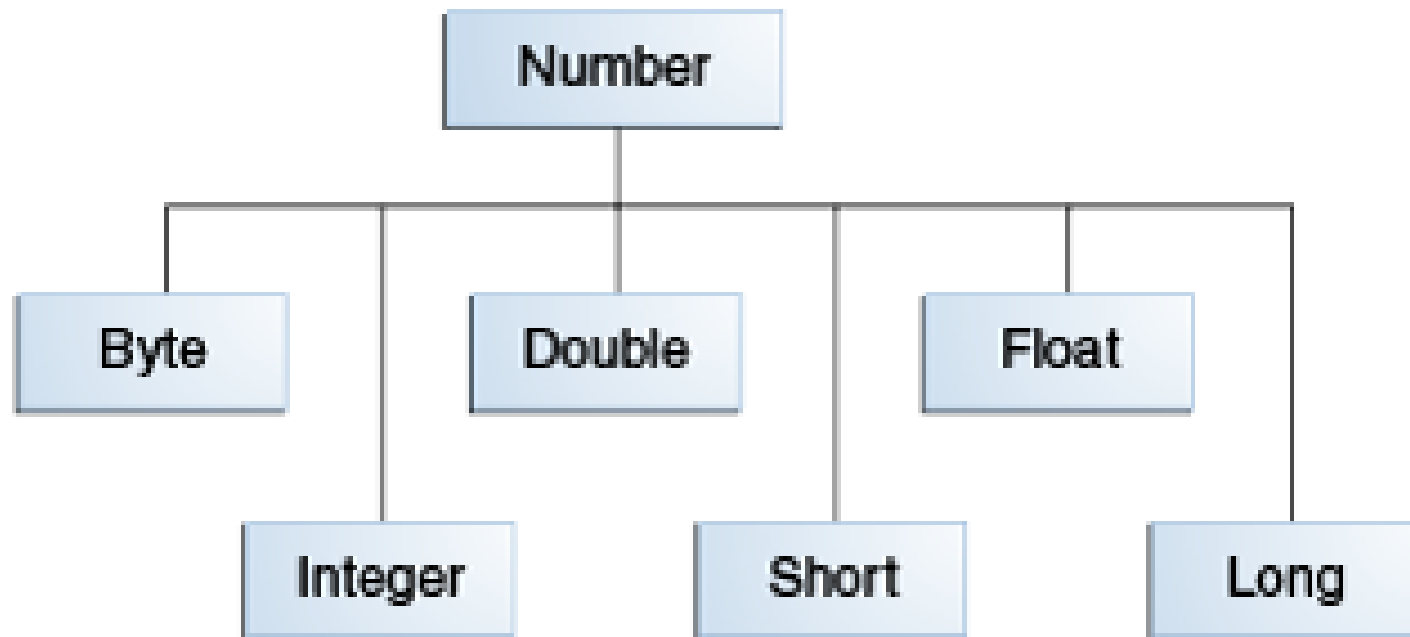
Characters

Strings

Autoboxing and Unboxing?

Java platform provides *wrapper* classes for each of the primitive data types.

All of the numeric wrapper classes are subclasses of the abstract class Number.



## Number object:

- As an argument of a method that expects an object.
- To use constants defined by the class, such as `MIN_VALUE` and `MAX_VALUE`.
- To use class methods for converting values to and from other primitive types.

- ***printf***, ***format*** and ***DecimalFormat*** class are used to formatting numeric.
- The ***printf*** and ***format*** Methods
- Syntax for these methods

*public PrintStream format(String format, Object... args)*

- Example:

```
int i = 461012;
System.out.format("The value of i is: %d%n", i);
```

# Formatting Numeric Print Output

```
public class DecimalFormatDemo {
    static public void customFormat(String pattern, double value ) {
        DecimalFormat myFormatter = new DecimalFormat(pattern);
        String output = myFormatter.format(value);
        System.out.println(value + " " + pattern + " " + output);
    }
    static public void main(String[] args) {
        customFormat("###,###.###", 123456.789);
        customFormat("###.##", 123456.789);
        customFormat("000000.000", 123.78);
        customFormat("$###,###.###", 12345.67);
    }
}
```

- The `Math` class in the `java.lang` package provides methods and constants for doing more advanced mathematical computation, including:
  - Constants and Basic Methods: **`Math.E`**, **`Math.PI`**,...
  - Basic static methods: **`ceil(double d)`**, **`floor(double d)`**, **`abs(int i)`**,...
  - Exponential and Logarithmic Methods: **`exp(double d)`**, **`sqrt(double d)`**, **`pow(double base, double exponent)`**
  - Trigonometric Methods: **`cos(double d)`**, **`sin(double d)`**
  - Random Numbers: The **`random()`** method returns a pseudo-randomly selected number between 0.0 and 1.0.

- Character class also offers a number of useful class (i.e., static) methods for manipulating characters.

*Character ch = new Character('a');*

- Some methods in this class
  - boolean **isLetter**(char ch)
  - boolean **isDigit**(char ch)
  - boolean **isUpperCase**(char ch)
  - char **toUpperCase**(char ch) ...
- A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler.



- String literals such as "Hello" in Java are implemented as instances of the `String` class.
- Strings are constant and immutable, that is, their values cannot be changed once they are created.
- String buffers allow creation of mutable strings.
- A simple `String` instance can be created by enclosing a string literal inside double quotes as shown in the following code snippet:

```
String name = "John";  
String s = new String("Hello World!!!");
```

# Introduction to String

- Java also provides special support for concatenation of strings using the plus (+) operator and for converting data of other types to strings as depicted in the following code snippet:

```
...
String str = "Hello"; String str1 = "World";
// The two strings can be concatenated by using the operator '+'
System.out.println(str + str1);
// This will print 'HelloWorld' on the screen
```

The *java.lang.String* class is a `final` class, that is, no class can extend it.

The *java.lang.String* class differs from other classes, in that one can use `+=` and `+` operators with *String* objects for concatenation.

# Introduction to String

`length(String str)`

- The `length()` method is used to find the length of a string. For example,
- `String str = "Hello";`
- `System.out.println(str.length());` // output: 5

`charAt(int index)`

- The `charAt()` method is used to retrieve the character value at a specific index.
- The index ranges from zero to `length() - 1`.
- The index of the first character starts at zero. For example,
- `System.out.println(str.charAt(2));` // output: 'l'

`concat(String str)`

- The `concat()` method is used to concatenate a string specified as argument to the end of another string.
- If the length of the string is zero, the original `String` object is returned, otherwise a new `String` object is returned.
- `System.out.println(str.concat("World"));`  
// output: 'HelloWorld'

## `compareTo(String str)`

- The `compareTo()` method is used to compare two `String` objects.
- The comparison returns an integer value as the result.
- The comparison is based on the Unicode value of each character in the strings.
- The result will return a negative value, if the argument string is alphabetically greater than the original string.
- The result will return a positive value, if argument string is alphabetically lesser than the original string and the result will return a value of zero, if both the strings are equal.
- For example,  

```
System.out.println(str.compareTo("World"));
// output: -15
```
- The output is **15** because, the second string **"World"** begins with **'W'** which is alphabetically greater than the first character **'H'** of the original string, **str**.
- The difference between the position of **'H'** and **'W'** is **15**.
- Since **'H'** is smaller than **'W'**, the result will be **-15**.

## `indexOf(String str)`

- The `indexOf()` method returns the index of the first occurrence of the specified character or string within a string.
- If the character or string is not found, the method returns **-1**. For example,  

```
System.out.println(str.indexOf("e")); // output: 1
```

# Introduction to String

## lastIndexOf(String str)

- The `lastIndexOf()` method returns the index of the last occurrence of a specified character or string from within a string.
- The specified character or string is searched backwards that is the search begins from the last character. For example,
- `System.out.println(str.lastIndexOf("l")); // output: 3`

## replace(char old, char new)

- The `replace()` method is used to replace all the occurrences of a specified character in the current string with a given new character.
- If the specified character does not exist, the reference of original string is returned. For example,
- `System.out.println(str.replace('e','a'));`  
`// output: 'Hallo'`

## substring(int beginIndex, int endIndex)

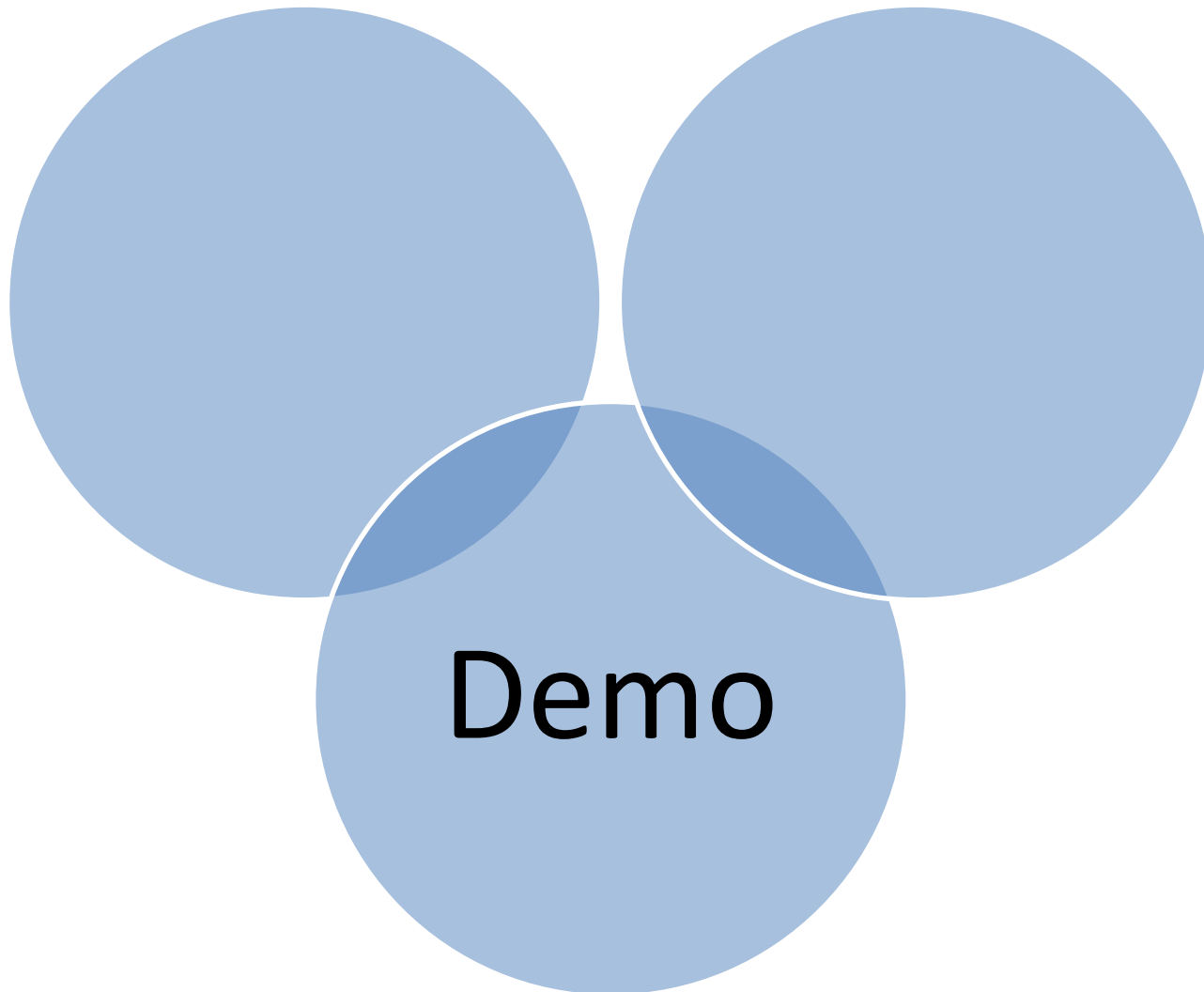
- The `substring()` method is used to retrieve a part of a string, that is, substring from the given string.
- One can specify the start index and the end index for the substring.
- If end index is not specified, all characters from the start index to the end of the string will be returned. For example,
- `System.out.println(str.substring(2,5)); // output: 'llo'`

## toString()

- The `toString()` method is used to return a `String` object.
- It is used to convert values of other data types into strings. For example,
  - `Integer length = 5;`
  - `System.out.println(length.toString()); // output: 5`
- Notice that the output is still 5. However, now it is represented as a string instead of an integer.

## trim()

- The `trim()` method returns a new string by trimming the leading and trailing whitespace from the current string. For example,
  - `String str1 = " Hello ";`
  - `System.out.println(str1.trim()); // output: 'Hello'`
- The `trim()` method will return **'Hello'** after removing the spaces.



# Working with StringBuilder Class



StringBuilder objects are similar to String objects, except that they are mutable and flexible.

Internally, the system treats these objects as a variable-length array containing a sequence of characters.

The length and content of the sequence of characters can be changed through methods available in the StringBuilder class.

For concatenating a large number of strings, using a StringBuilder object is more efficient.

The StringBuilder class also provides a `length()` method that returns the length of the character sequence in the class.



# Working with StringBuilder Class



Unlike strings a `StringBuilder` object also has a property `capacity` that specifies the number of character spaces that have been allocated.

The capacity is returned by the `capacity()` method and is always greater than or equal to the length.

The capacity will automatically expand to accommodate the new strings when added to the string builder.

`StringBuilder` object allows insertion of characters and strings as well as appending characters and strings at the end.

# Working with StringBuilder Class

`StringBuilder()`

- Default constructor that provides space for 16 characters.

`StringBuilder(int capacity)`

- Constructs an object without any characters in it.
- However, it reserves space for the number of characters specified in the argument, capacity.

`StringBuilder  
(String str)`

- Constructs an object that is initialized with the contents of the specified string, str.

# Methods of StringBuilder Class

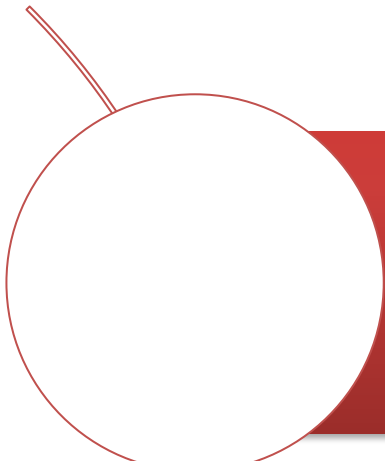
`append()`

`insert()`

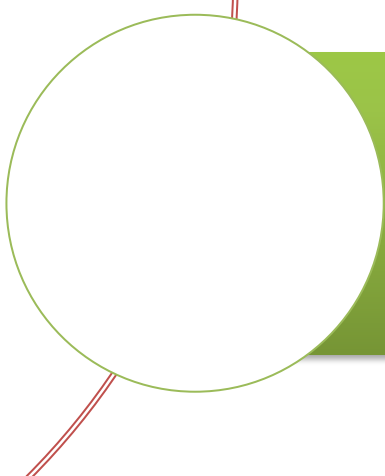
`delete()`

`reverse()`

# Working with StringBuffer Class



**StringBuffer** is a peer class of **String** that provides much of the functionality of strings.



**StringBuffer** may have characters and substrings inserted in the middle or appended to the end.

# StringBuffer Constructors

## StringBuffer( ):

- It reserves room for 16 characters without reallocation.
- `StringBuffer s=new StringBuffer();`

## StringBuffer( int size)

- It accepts an integer argument that explicitly sets the size of the buffer.
- `StringBuffer s=new StringBuffer(20);`

## StringBuffer(String str):

- It accepts a **String** argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.
- `StringBuffer s=new StringBuffer("GeeksforGeeks");`

# Methods of StringBuffer Class

---

`length()`

`capacity()`

`insert()`

`append()`

`reverse()`

# StringBuilder vs StringBuffer

---

No.	StringBuffer	StringBuilder
1.	Thread safe and synchronized	non-synchronized
2.	slower	faster

# String Arrays

Sometimes there is a need to store a collection of strings.

`String` arrays can be created in Java in the same manner as arrays of primitive data types.

For example, `String[] empNames = new String[10];`

This statement will allocate memory to store references of 10 strings.

However, no memory is allocated to store the characters that make up the individual strings.

Loops can be used to initialize as well as display the values of a `String` array.



- A regex (a regular expression) is a series of patterns used to determine the form of strings. If a string matches a pattern, the string is called match.
- Ex: **[0-9]{3,7}**: This regular expression matches strings of 3 to 7 numeric characters.
  - **[0-9]**: representing 1 digits
  - **{3,7}**: represents the number of occurrences (at least 3, at most 7)

```

1  package democ6;
2
3  import java.util.Scanner;
4  public class Regex {
5      public static void main(String[] args)
6      {
7          Scanner ip = new Scanner(System.in);
8          System.out.print("Phone number: ");
9          String mobile = ip.nextLine();
10         String pattern = "0[0-9]{9,10}";
11         if(mobile.matches(pattern))
12             System.out.println("That's phone number!");
13         else
14             System.out.println("That isn't phone number!");
15     }
16 }

```

Regex

Does the mobile test  
match the pattern?

**s.matches(regex)**

```

1  package democ6;
2
3  import java.util.Scanner;
4  public class Regex {
5      public static void main(String[] args)
6      {
7          Scanner ip = new Scanner(System.in);
8          System.out.print("Phone number: ");
9          String mobile = ip.nextLine();
10         String pattern = "0[0-9]{9,10}";
11         if(mobile.matches(pattern))
12             System.out.println("That's phone number!");
13         else
14             System.out.println("That isn't phone number!");
15     }
16 }

```

Regex

Does the mobile test  
match the pattern?

**s.matches(regex)**

## Regular Expression

Regular Expression	Description
.	Matches any character
^regex	Finds regex that must match at the beginning of the line.
[abc]	Set definition, can match the letter a or b or c.
[abc][vz]	Set definition, can match a or b or c followed by either v or z.
[^abc]	When a caret appears as the first character inside square brackets, it negates the pattern. This pattern matches any character except a or b or c.
[a-d1-7]	Ranges: matches a letter between a and d and figures from 1 to 7, but not d1.
X Z	Finds X or Z.
XZ	Finds X directly followed by Z.
\$	Checks if a line end follows.

Number of Occurrences	
{M,N}	At least M, at most N times
{N}	n times
?	0-1
*	0-N
+	1-N
	1

← **[0-9]{3, 7}** ↑

# Example RegEx

```

1  import java.util.Scanner;
2  public class PhoneNumber {
3      public static void main(String[] args)
4      {
5          Scanner ip = new Scanner(System.in);
6
7          System.out.print("Email: ");
8          String email = ip.nextLine();
9
10         System.out.print("Phone number of Cantho: ");
11         String phone = ip.nextLine();
12
13         String reEmail = "\\w+@\\w+\\.\\w+";
14         if(!email.matches(reEmail))
15             System.out.println("Not email!!");
16         String rePhone = "0292\\d{7}";
17         if(!phone.matches(rePhone))
18             System.out.println("Not phone number of Cantho");
19     }
20 }
  
```

# Wrapper Classes



Java provides a set of classes known as wrapper classes for each of its primitive data type that 'wraps' the primitive type into an object of that class.

In other words, the wrapper classes allow accessing primitive data types as objects.

The wrapper classes for the primitive data types are: Byte, Character, Integer, Long, Short, Float, Double, and Boolean.

The wrapper classes are part of the `java.lang` package.

Primitive type	Wrapper class
byte	Byte
char	Character
float	Float
double	Double
int	Integer
long	Long
short	Short
boolean	Boolean

# What is the need for wrapper classes?



The use of primitive types as objects can simplify tasks at times.

For example, most of the collections store objects and not primitive data types.

Many of the activities reserved for objects will not be available to primitive data types.

Also, many utility methods are provided by the wrapper classes that help to manipulate data.

Wrapper classes convert primitive data types to objects, so that they can be stored in any type of collection and also passed as parameters to methods.

Wrapper classes can convert numeric strings to numeric values.



## `valueOf()`

- The `valueOf()` method is available with all the wrapper classes to convert a type into another type.
- The `valueOf()` method of the `Character` class accepts only `char` as an argument.
- The `valueOf()` method of any other wrapper class accepts either the corresponding primitive type or `String` as an argument.

## `typeValue()`

- The `typeValue()` method can also be used to return the value of an object as its primitive type.

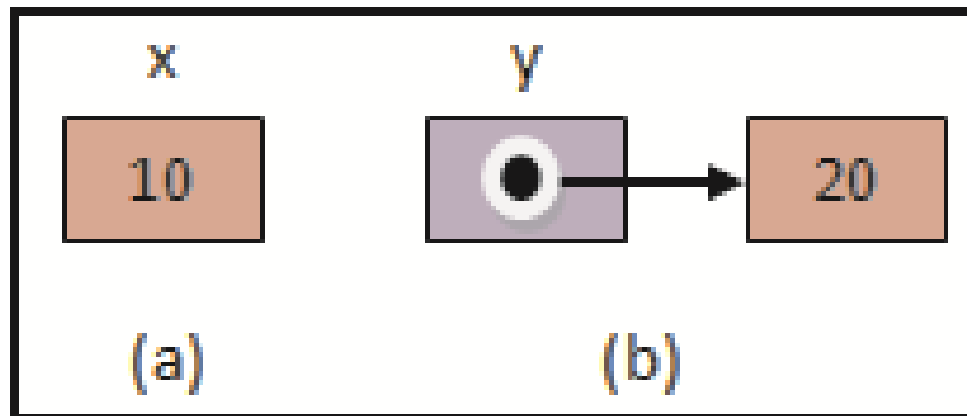
- The difference between creation of a primitive type and a wrapper type is as follows:

**Primitive type**

```
• int x = 10;
```

**Wrapper type**

```
• Integer y = new Integer(20);
```





**FPT UNIVERSITY**



## Autoboxing

- The automatic conversion of primitive data types such as int, float, and so on to their corresponding object types such as Integer, Float, and so on during assignments and invocation of methods and constructors is known as autoboxing.
- For example,
  - `ArrayList<Integer> intList = new ArrayList<Integer>();`
  - `intList.add(10); // autoboxing`
  - `Integer y = 20; // autoboxing`

## Unboxing

- The automatic conversion of object types to primitive data types is known as unboxing.
- For example,
  - `int z = y; // unboxing`

```
public class AutoUnbox {

    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        Character chBox = 'A'; // Autoboxing a character
        char chUnbox = chBox; // Unboxing a character

        // Print the values
        System.out.println("Character after autoboxing is:" + chBox) ;
        System.out.println("Character after unboxing is:" + chUnbox);
    }
}
```

# ĐẠI HỌC FPT CẦN THƠ

