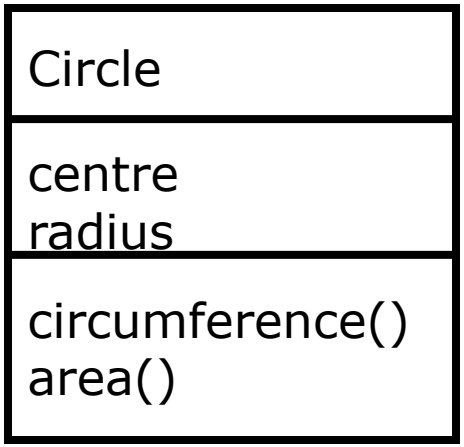
6.Introducing Classes

Sisoft Technologies Pvt Ltd
SRC E7, Shipra Riviera Bazar, Gyan Khand-3, Indirapuram, Ghaziabad
Website: www.sisoft.in Email:info@sisoft.in
Phone: +91-9999-283-283

Classes

- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.



Classes

- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.
- The basic syntax for a class definition:

```
class  ClassName [extends
SuperClassName]
{
        [fields declaration]
        [methods declaration]
}
```

- Bare bone class – no fields, no methods

```
public class Circle {
    // my circle class
}
```

Adding Fields: Class Circle with fields

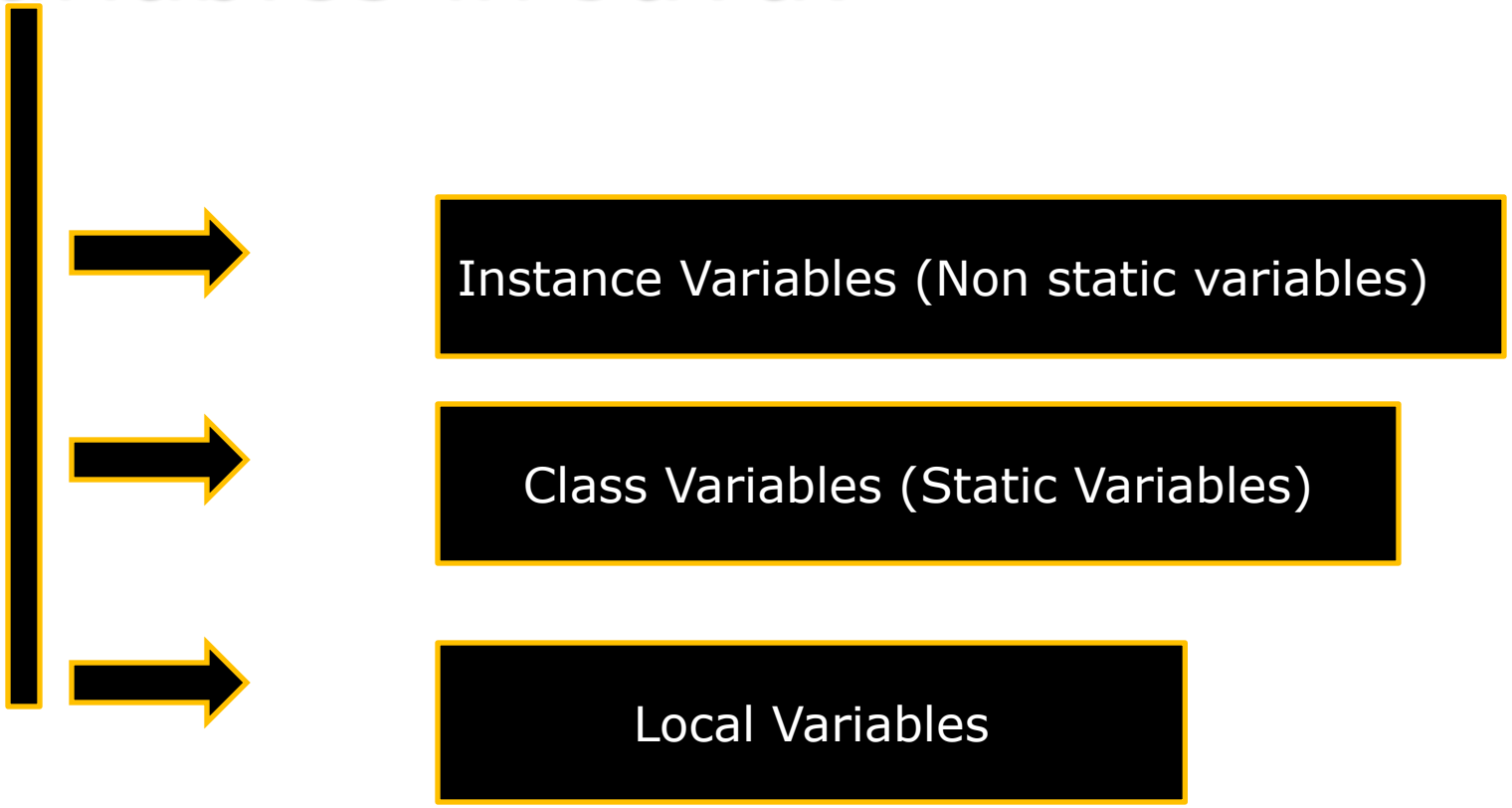
- Add *fields*

```
public class Circle {
        public double x, y;  // centre coordinate
        public double r;     //  radius of the circle


}
```

- The fields (data) are also called the variables.

Variables in Java:



1.Instance variables :

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the key word 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object.s state that must be present through out the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.

2.Class/static variables :

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
- Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.

2.Class/static variables

- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name . *ClassName.VariableName*.
- When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

Example:

```
import java.io.*;

public class Employee{
   // salary  variable is a private static variable
   private static double salary;

   // DEPARTMENT is a constant
   public static final String DEPARTMENT = "Development ";

   public static void main(String args[]){
      salary = 1000;
      System.out.println(DEPARTMENT+"average salary:"+salary);
   }
}
```

This would produce following result:

```
Development average salary:1000
```

Note: If the variables are access from an outside class the constant should be accessed as Employee.DEPARTMENT

3. Local variables

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

Local variables Ex :

```
public class Test
{
    public  void  pupAge()
    {
         int  age = 0;
         age = age + 7;
     System.out.println("Puppy age is : " + age);
    }
public static void main(String  args[])
{
        Test test = new Test(); test.pupAge();
 } }
```

This would produce following result:

Puppy age is: 7

Access and Non Acc Modifiers

- Modifiers are keywords that you add to those definitions to change their meanings
- The Java language has a wide variety of modifiers, including the following:
  - Java Access Modifiers
  - Non Access Modifiers
- To use a modifier, you include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement

Access and Non Acc Modifiers

- *public* class className { // ... }
  private boolean myFlag;
- *static final* double weeks = 9.5;
  protected static final int BOXWIDTH = 42;
- *public static* void main(String[] arguments) {
- // body of method }

Access Control Modifiers:

- Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:
- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

Non Access Modifiers:

Java provides a number of non-access modifiers to achieve many other functionality.

- The *static* modifier for creating class methods and variables
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.
- The *synchronized* and *volatile* modifiers, which are used for threads.

Static

- Static variable
- Static Class
- Static method

static Variables

- Java instance variables are given separate memory for storage. If there is a need for a variable to be common to all the objects of a single java class, then the static modifier should be used in the variable declaration.
- Any java object that belongs to that class can modify its static variables.
- Also, an instance is not a must to modify the static variable and it can be accessed using the java class directly.
- Static variables can be accessed by java instance methods also.
- When the value of a constant is known at compile time it is declared ‘final’ using the ‘static’ keyword.

static Methods

- Similar to static variables, java static methods are also common to classes and not tied to a java instance.
- Good practice in java is that, static methods should be invoked with using the class name though it can be invoked using an object. ClassName.methodName(arguments) or objectName.methodName(arguments)
- General use for java static methods is to access static fields.
- Static methods can be accessed by java instance methods.
- Java static methods cannot access instance variables or instance methods directly.
- Java static methods cannot use the ‘this’ keyword

static Classes

static Classes

- For java classes, only an inner class can be declared using the static modifier.
- For java a static inner class it does not mean that, all their members are static. These are called nested static classes in java.

final Fields

- A field can be declared final
- Both class and instance variables (static and non-static fields) may be declared final.
- Effectively final declares the variable to be constant
- Example:

```
private final int PERFECT_SCORE = 100
```

final Methods

- A **final** **method** cannot be overridden or hidden by subclasses.
- This is used to prevent unexpected behavior from a subclass altering a method that may be crucial to the function or consistency of the class

```
public final void m2() {...}
public static final void m4() {...}
```

final class

- To stop a class from being inherited from (sub-classes) we declare it as a final class e.g.

If Person was declared as final

- **final class Person {**
- …
- // the body of the class
- …
- }

final class

Then the following would not be allowed:

- **class Programmer extends Person {**
- …
- // the body of the class
- …
- }

Modifier – Abstract

- Description
  - Represents generic concept
  - Can not be instantiated
- Abstract class
  - Placeholder in class hierarchy
  - Can be partial description of class
  - Can contain non-abstract methods
  - Required if any method in class is abstract

Adding Methods

- A class with only data fields has no life. Objects created by such a class cannot respond to any messages.
- Methods are declared inside the body of the class but immediately after the declaration of data fields.
- The general form of a method declaration is:

```
type MethodName (parameter-list)
{
        Method-body;
}
```

Methods

- A method is a set of code which is referred to by name and can be called (invoked) at any point in a program simply by utilizing the method's name.
- Think of a method as a subprogram that acts on data and often returns a value.Each method has its own name.
- When that name is encountered in a program, the execution of the program branches to the body of that method.
- When the method is finished, execution returns to the area of the program code from which it was called,

Methods Declaration

- Modifiers—such as public, private, and others
- The return type—the data type of the value returned by the method, or void if the method does not return a value.
- The method name—the rules for field names apply to method names as well, but the convention is a little different.
- The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
- An exception list
- The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

Method Types

- **Built-in:** Build-in methods are part of the compiler package, such as System.out.println( ) and System.exit(0).
- **User-defined:** User-defined methods are created by you, the programmer. These methods take-on names that you assign to them and perform tasks that you create.

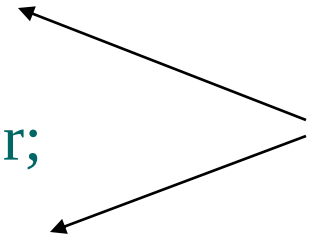
```
int number = abcObj.add(4,6);
```

Adding Methods to Class Circle

```
public class Circle {

    public double x, y; // centre of the circle
    public double r;    // radius of circle

    //Methods to return circumference and area
    public double circumference() {
                return 2*3.14*r;
    }
    public double area() {
                return 3.14 * r * r;
    }
}
```



Method Overloading

- Same method name, different type and/or number of parameters
- Return types do not differentiate between overloaded methods
- If no exact match found for methods, automatic type conversions will be applied to match the methods to be called
- public class DataArtist { ...

```
public void draw(String s) { ... }
public void draw(int i) { ... }
public void draw(double f) { ... }
public void draw(int i, double f) { ... }
}
```

Methods Using Variable Arguments

- Java provides a feature, called varargs or variable arguments, which enables the programmers to write a more generic method that share the same functionality.
- The following code snippet displays the use of variable arguments:

```
public class Statistics {
public float average(int... nums) {
int sum = 0;
for ( int x : nums ) {
sum += x;
}
return ((float) sum) / nums.length;
}
```

Legal and illegal vararg

- Void doStuff(int…x){}
- Void dostuff2(char c,int…x){}
- Void doStuff3(Animal…anmRef)

Illegal

- Void doStuff(int x…){}
- Void dostuff2(int…x,char c){}
- Void doStuff3(int…x, int…y)

Constructors

- A **java constructor** has the same name as the name of the class to which it belongs
- Constructor's syntax does not include a return type, since constructors never return a value.
- Constructor is called when the new keyword is used
- When no explicit constructors are provided. Java provides a default constructor which takes no arguments and performs no special actions or initializations
- Constructors may include parameters of various types. When the constructor is invoked using the new operator, the types must match those that are specified in the constructor definition.

Constructor Overloading

- Like methods, constructors can also be overloaded
- Constructor overloading allows you to declare multiple constructor

Constructor Chaining

- Calling one constructor from other is called *constructor chaining*
- this() can be used to call other constructor of same class
- The this() call in a constructor invokes the an other constructor with the corresponding parameter list within the same class.
- Java requires that any this() call must occur as the first statement in a constructor.

Garbage Collection

- In Java, process of deallocating memory is handled automatically by the garbage collector
- The garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects
- An in use object, or a referenced object, means that some part of your program still maintains a pointer to that object
- An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed

Unreferencing Object

- **1) By nulling a reference:**

  Employee e=new Employee();

  e=null;

- **2) By assigning a reference to another:**

  Employee e1=new Employee();

  Employee e2=new Employee();

  e1=e2;//now the first object referred by e1 is available for garbage collection

- **3) By annonymous object:**

  new Employee();

Garbage Collection: Methods

- **finalize() method:**

  The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in System class as:

  ```
  protected void finalize(){}
  ```

- **gc() method:**

  The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

  ```
  public static void gc(){}
  ```

Wrapper classes

Wrapper classes

Java is an object-oriented language and can view everything as an object. A simple file can be treated as an object (with **java.io.File**), an address of a system can be seen as an object (with **java.util.URL**), an image can be treated as an object (with **java.awt.Image**) and a simple data type can be converted into an object (with **wrapper classes**).

The primitive data types are not objects; they do not belong to any class; they are defined in the language itself. Sometimes, it is required to convert data types into objects in Java language. For example, upto JDK1.4, the data structures accept only objects to store. A data type is to be converted into an object and then added to a Stack or Vector etc. For this conversion, the designers introduced **wrapper classes**

What is wrapper class :-

• As the name says, a wrapper class wraps (encloses) around a data type and gives it an object appearance. Wherever, the data type is required as an object, this object can be used. Wrapper classes include methods to unwrap the object and give back the data type.
• It can be compared with a chocolate. The manufacturer wraps the chocolate with some foil or paper to prevent from pollution. The user takes the chocolate, removes and throws the wrapper and eats it

Wrapper classes

- The following conversion.

```
int k = 100;
Integer it1 = new Integer(k);
```

- The **int** data type **k** is converted into an object, **it1** using **Integer** class. The **it1** object can be used in Java programming wherever **k** is required an object.
- The following code can be used to unwrap (getting back **int** from **Integer** object) the object **it1**.

```
int m = it1.intValue();
System.out.println(m*m); // prints
10000
```

intValue() is a method of Integer class that returns an int data type.

Wrapper classes

- In the above code, **Integer** class is known as a wrapper class (because it wraps around a data type to give it an impression of object). To wrap (or to convert) each primitive data type, there comes a wrapper class. Eight wrapper classes exist in **java.lang** package that represent 8 data types.
- List of Wrapper classes

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

Java
Numbers Class

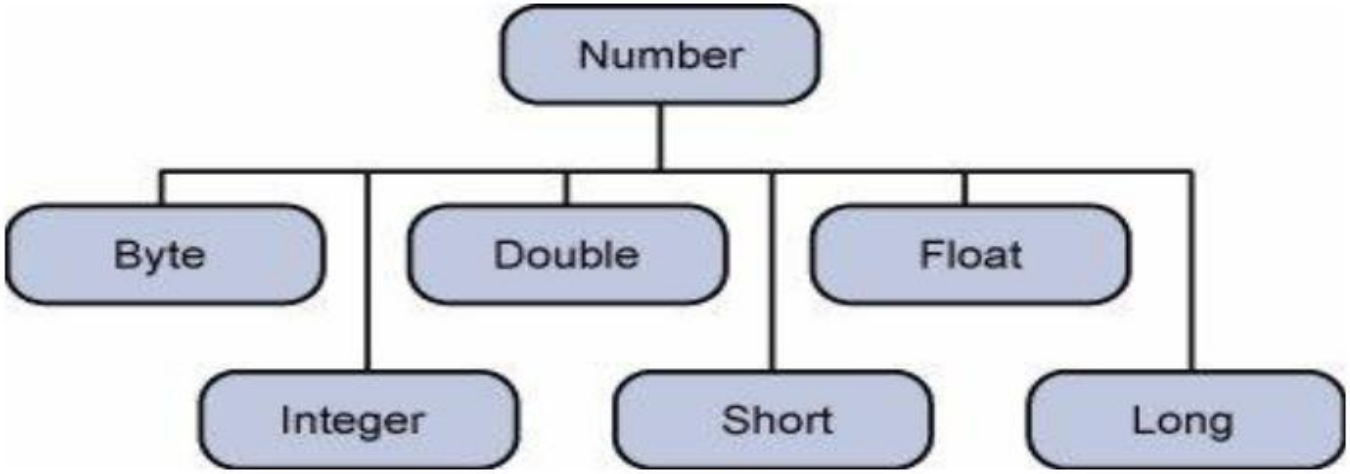
Java Numbers Class

- Normally, when we work with Numbers, we use primitive data types such as byte, int, long, double etc

```
int a = 5000;
float b = 13.65;
byte  c= 0xaf;
```

Java Numbers Class

- However in development we come across situations were we need to use objects instead of primitive data types. In-order to achieve this Java provides wrapper classes for each primitive data type.
- All the wrapper classes (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.



Number Methods:

Method	Description
byte byteValue() short shortValue() int intValue() long longValue() float floatValue() double doubleValue()	Converts the value of this Number object to the primitive data type returned.
int compareTo(Byte anotherByte) int compareTo(Double anotherDouble) int compareTo(Float anotherFloat) int compareTo(Integer anotherInteger) int compareTo(Long anotherLong) int compareTo(Short anotherShort)	Compares this Number object to the argument.
boolean equals(Object obj)	Determines whether this number object is equal to the argument. The methods return true if the argument is not null and is an object of the same type and with the same numeric value. There are some extra requirements for Double and Float objects that are described in the Java API documentation.

Number (Conversion Methods) : Integer

Method	Description
static Integer decode(String s)	Decodes a string into an integer. Can accept string representations of decimal, octal, or hexadecimal numbers as input.
static int parseInt(String s)	Returns an integer (decimal only).
static int parseInt(String s, int radix)	Returns an integer, given a string representation of decimal, binary, octal, or hexadecimal (radix equals 10, 2, 8, or 16 respectively) numbers as input.
String toString()	Returns a String object representing the value of this Integer.
static String toString(int i)	Returns a String object representing the specified integer.
static Integer valueOf(int i)	Returns an Integer object holding the value of the specified primitive.
static Integer valueOf(String s)	Returns an Integer object holding the value of the specified string representation.
static Integer valueOf(String s, int radix)	Returns an Integer object holding the integer value of the specified string representation, parsed with the value of radix. For example, if s = "333" and radix = 8, the method returns the base-ten integer equivalent of the octal number 333.

Java
String Class

String

- Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.
- The Java platform provides the String class to create and manipulate strings.

Creating Strings:

```
String greeting = "Hello sisoft!"
```

String Length:

```
public class Stringlength
 {
 public static void main(String args[])
{
   String str= "Dot saw I was Tod";
   int len = str.length();
System.out.println( "String Length is : " +
   len );
} }
```

Concatenating Strings:

```
Sting str1="hello";
String str2="sisoft";
   String str=string1.concat(string2);
SOP(str);
```

Converting Numbers to Strings

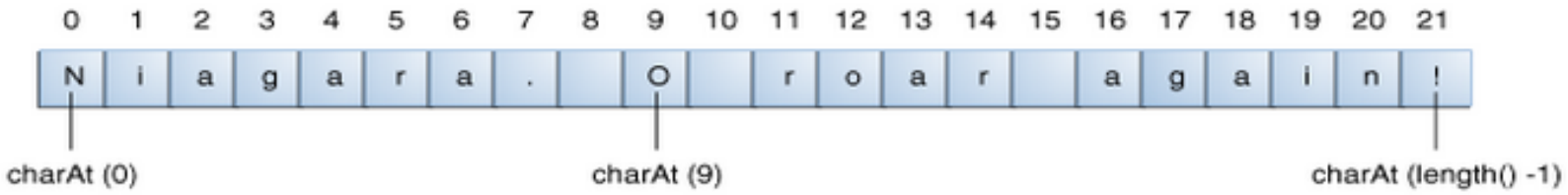
```
int i=68;
String s2 = String.valueOf(i); // The valueOf class
   method.
```

Converting Strings to Numbers

- Getting Characters and Substrings by Index

```
char aChar = str.charAt(9);
```

Indices begin at 0, so the character at index 9 is 'O', as illustrated in the following figure:



Here are several other `String` methods for manipulating strings:

Other Methods in the `String` Class for Manipulating Strings

Method	Description
`String[] split(String regex)` `String[] split(String regex, int limit)`	Searches for a match as specified by the string argument (which contains a regular expression) and splits this string into an array of strings accordingly. The optional integer argument specifies the maximum size of the returned array. Regular expressions are covered in the lesson titled "Regular Expressions."
`CharSequence subSequence(int beginIndex, int endIndex)`	Returns a new character sequence constructed from `beginIndex` index up until `endIndex` - 1.
`String trim()`	Returns a copy of this string with leading and trailing white space removed.
`String toLowerCase()` `String toUpperCase()`	Returns a copy of this string converted to lowercase or uppercase. If no conversions are necessary, these methods return the original string.

Replacing Characters and Substrings into a String

Method	Description
`String replace(char oldChar, char newChar)`	Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
`String replace(CharSequence target, CharSequence replacement)`	Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
`String replaceAll(String regex, String replacement)`	Replaces each substring of this string that matches the given regular expression with the given replacement.
`String replaceFirst(String regex, String replacement)`	Replaces the first substring of this string that matches the given regular expression with the given replacement.

Searching for Characters and Substrings in a String

The Search Methods in the `String` Class

Method	Description
`int indexOf(int ch)` `int lastIndexOf(int ch)`	Returns the index of the first (last) occurrence of the specified character.
`int indexOf(int ch, int fromIndex)` `int lastIndexOf(int ch, int fromIndex)`	Returns the index of the first (last) occurrence of the specified character, searching forward (backward) from the specified index.
`int indexOf(String str)` `int lastIndexOf(String str)`	Returns the index of the first (last) occurrence of the specified substring.
`int indexOf(String str, int fromIndex)` `int lastIndexOf(String str, int fromIndex)`	Returns the index of the first (last) occurrence of the specified substring, searching forward (backward) from the specified index.
`boolean contains(CharSequence s)`	Returns true if the string contains the specified character sequence.

StringBuilder and StringBuffer

- These are mutable form of String and contain methods which make String processing easy
- StringBuffer class is *exactly* the same as the StringBuilder class
- Each method in StringBuffer is synchronized hence StringBuffer is thread safe .
- StringBuilder is also not thread safe

String, StringBuilder & StringBuffer

	String	***StringBuffer***	***StringBuilder***
Storage Area	Constant String Pool	Heap	Heap
Modifiable	No (immutable)	Yes(mutable)	Yes(mutable)
Thread safee	Yes	Yes	No
Performance	Fast	Very slow	Fast

Autoboxing and UnBoxing

- *Autoboxing* is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.
- For example, converting an int to an Integer, a double to a Double, and so on.
- If the conversion goes the other way, this is called *unboxing*.
- Here is the simplest example of autoboxing:

```
Character ch = 'a';
Integer i1 = 22 ;
```