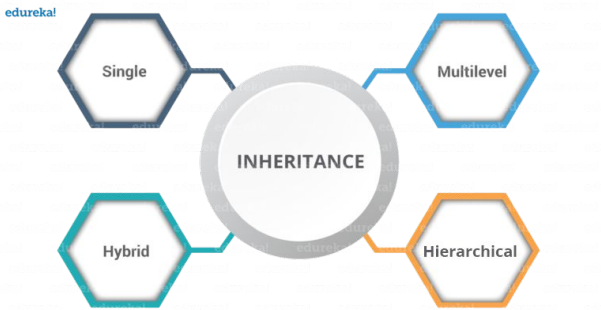
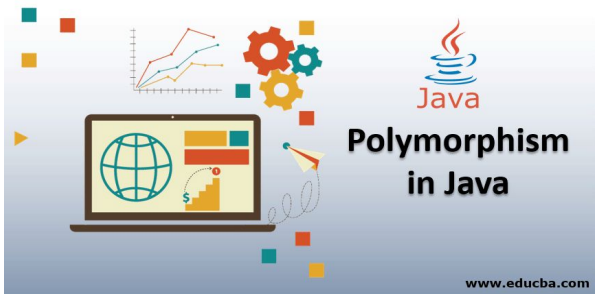
AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

CHAPTER 5
OBJECT ORIENTED THINKING





SUBJECT:OOP-I
CODE:3140705

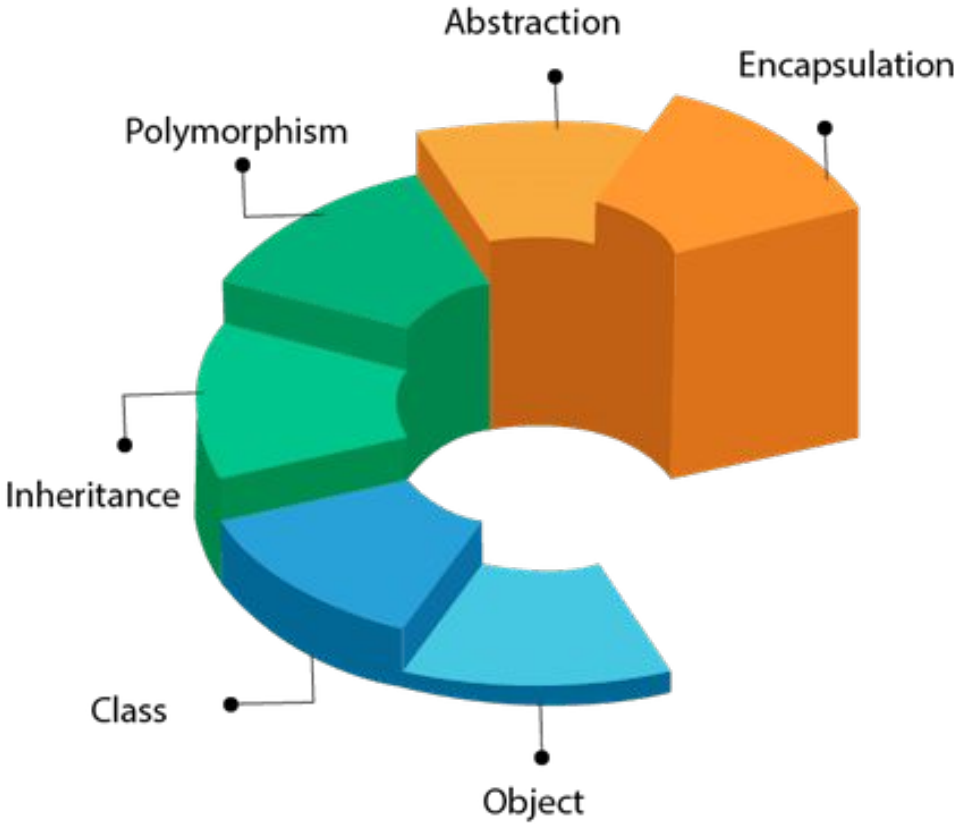
PREPARED BY:
ASST.PROF.NENSI KANSAGARA
(CSE DEPARTMENT,ACET)

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

PART I
CONCEPT OF CLASS AND OBJECT

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

OOPs (Object-Oriented Programming System)



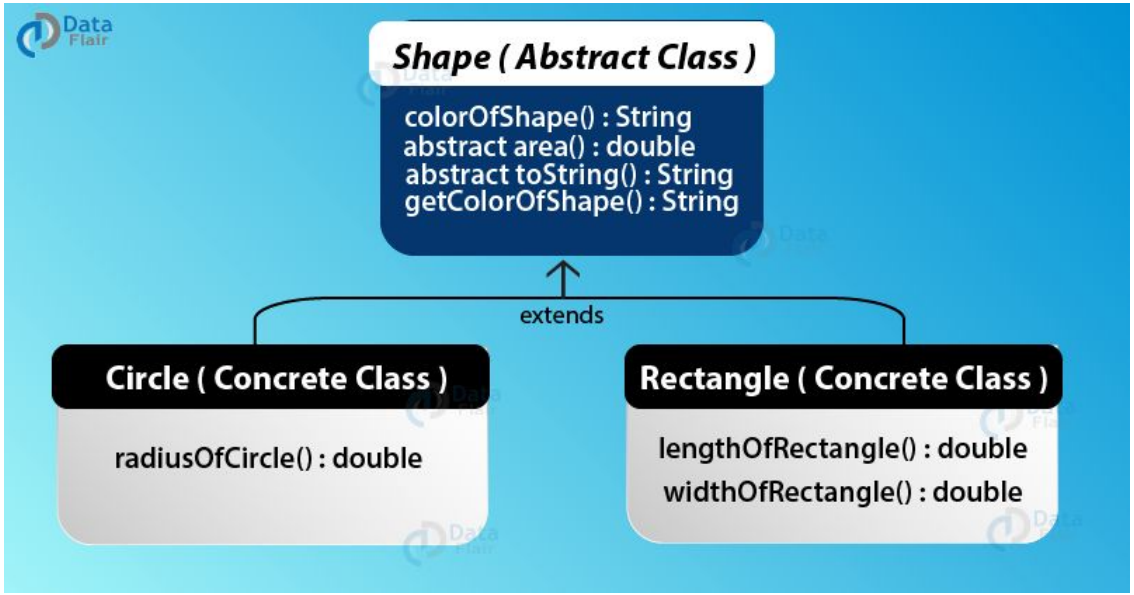
AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

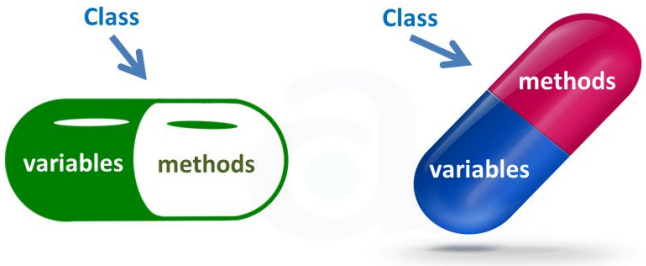
CLASS ABSTRACTION

- Abstraction is the quality of dealing with ideas rather than events. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the protocol your e-mail server uses are hidden from the user. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send.
- Likewise in Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

ENCAPSULATION

- Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.
- Encapsulation in Java is a mechanism for wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.
- To achieve encapsulation in Java −
  - Declare the variables of a class as private.
  - Provide public setter and getter methods to modify and view the variables values.



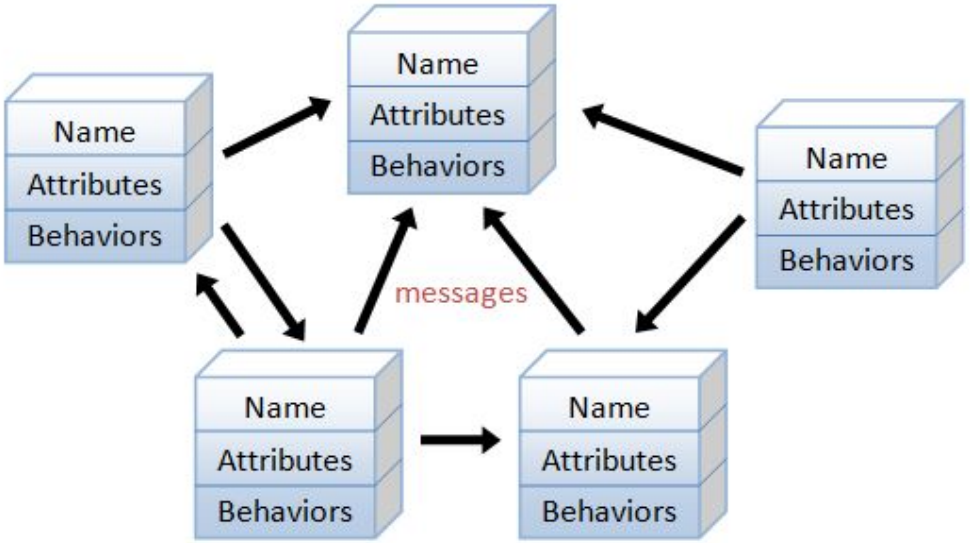


Encapsulation

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

THINKING OF OBJECTS

- The **object oriented Programming** Language is based upon the concept of "**objects**", which contains data as attributes in methods. Every **object** in **Java** has state and behavior which are represented by instance variables and methods. ... Here method is using instance variable values.

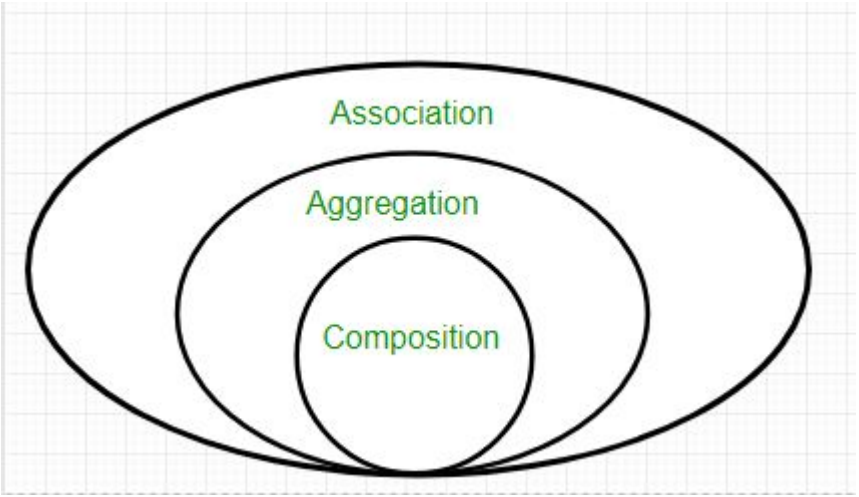


An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

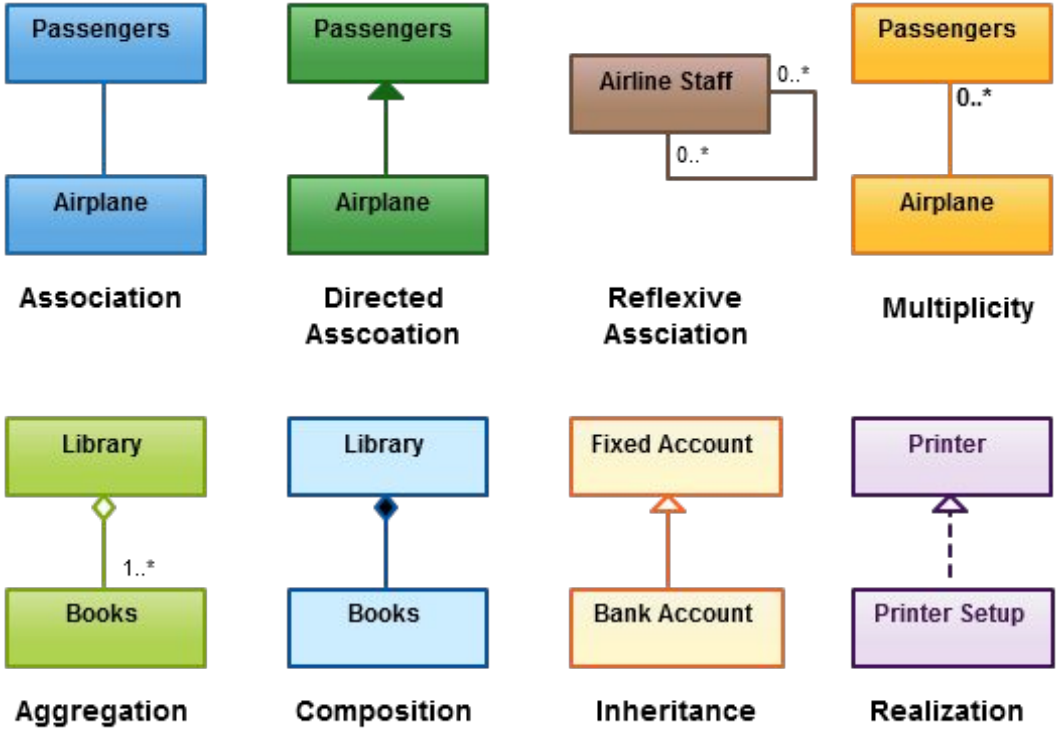
CLASS RELATIONSHIPS

- Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.
- In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.
- It is a special form of Association where:
  - It represents **Has-A** relationship.



AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

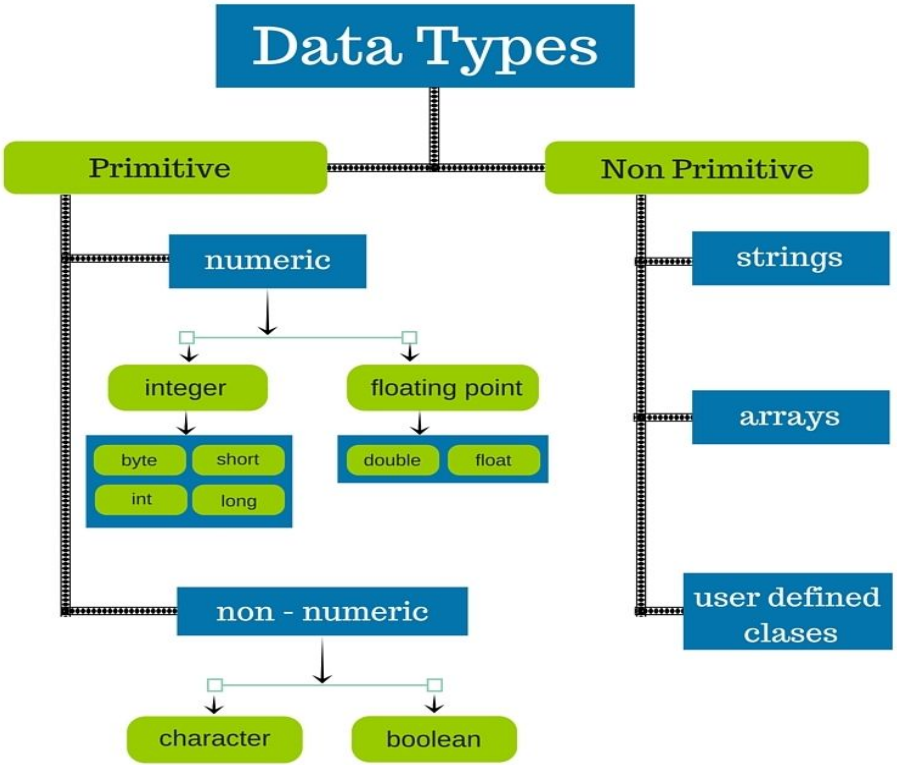
- It is a **unidirectional association** i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
- In Aggregation, **both the entries can survive individually** which means ending one entity will not effect the other entity



AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

- Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.
  - ➢ It represents **part-of** relationship.
  - ➢ In composition, both the entities are dependent on each other.
  - ➢ When there is a composition between two entities, the composed object **cannot exist** without the other entity.

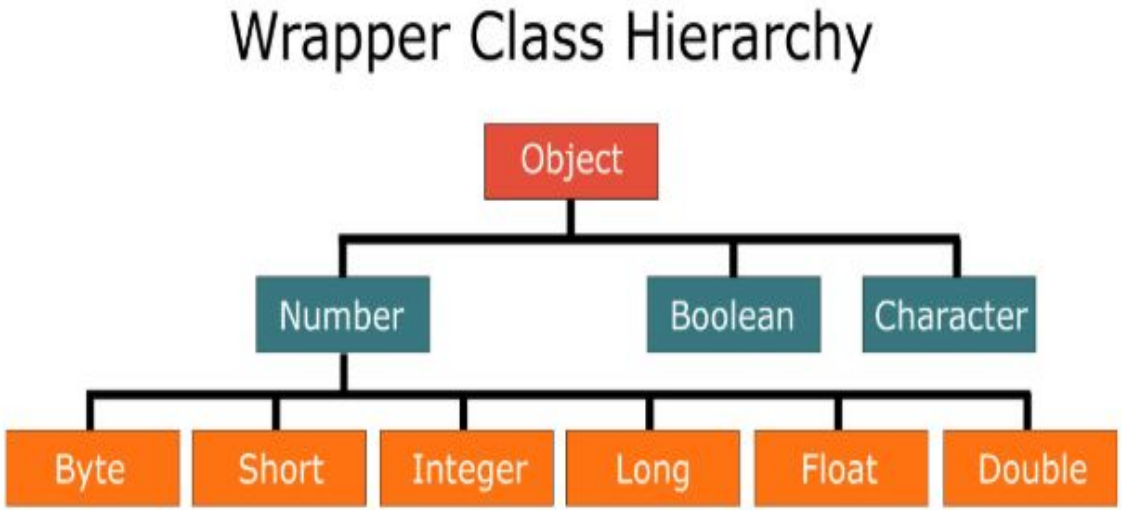
PRIMITIVE DATA TYPES



AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

WRAPPER CLASS TYPES

- A **Wrapper class** is a **class** whose object wraps or contains a primitive data types. When we create an object to a **wrapper class**, it contains a field and in this field, we can store a primitive data types. In other words, we can wrap a primitive value into a **wrapper class** object.



AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

NEED OF WRAPPER CLASS

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

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

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

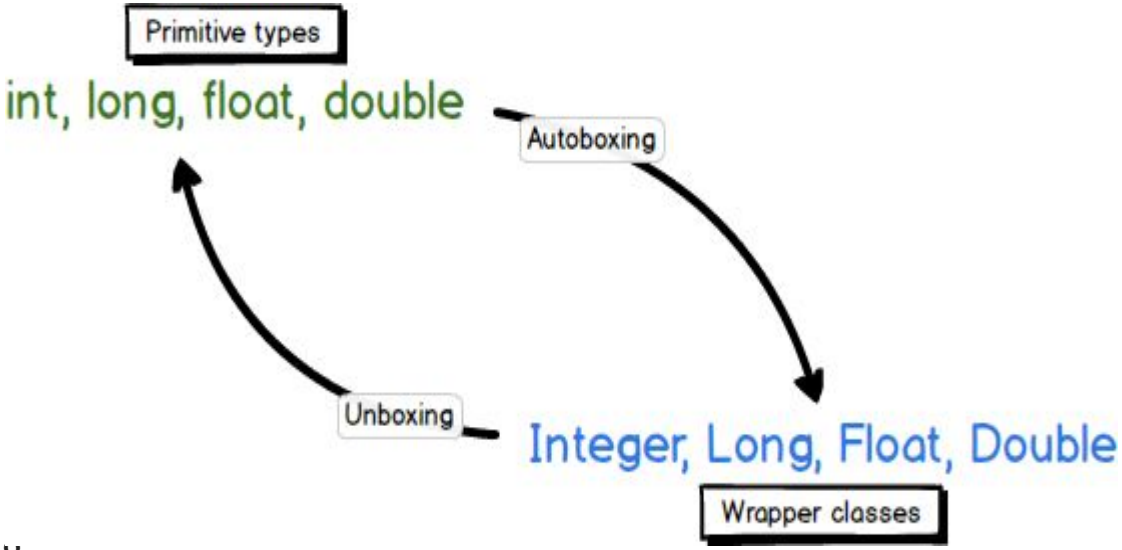
Advantages of Wrapper Class in Java

- Serialization
- Synchronization
- java.util package
- Collection Framework
- Changing the value inside a Method
- Polymorphism

TechVidvan

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**.



AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

BIGINTEGER CLASS

- BigInteger class is used for mathematical operation which involves very big integer calculations that are outside the limit of all available primitive data types.
- For example factorial of 100 contains 158 digits in it so we can't store it in any primitive data type available. We can store as large Integer as we want in it. There is no theoretical limit on the upper bound of the range because memory is allocated dynamically but practically as memory is limited you can store a number which has Integer.MAX_VALUE number of bits in it which should be sufficient to store mostly all large values.

```java
// Java program to find large factorials using BigInteger
import java.math.BigInteger;
import java.util.Scanner;

public class Example
{
    // Returns Factorial of N
    static BigInteger factorial(int N)
    {
        // Initialize result
        BigInteger f = new BigInteger("1"); // Or BigInteger.ONE

        // Multiply f with 2, 3, ...N
        for (int i = 2; i <= N; i++)
            f = f.multiply(BigInteger.valueOf(i));

        return f;
    }

    // Driver method
    public static void main(String args[]) throws Exception
    {
        int N = 20;
        System.out.println(factorial(N));
    }
}
```

Output:

```
2432902008176640000
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

BIGDECIMAL CLASS

- The BigDecimal class provides operations on double numbers for arithmetic, scale handling, rounding, comparison, format conversion and hashing. It can handle very large and very small floating point numbers with great precision but compensating with the time complexity a bit.
- A BigDecimal consists of a random precision integer unscaled value and a 32-bit integer scale. If greater than or equal to zero, the scale is the number of digits to the right of the decimal point. If less than zero, the unscaled value of the number is multiplied by 10^(-scale).

```java
// Java Program to illustrate BigDecimal Class

import java.math.BigDecimal;
public class BigDecimalExample
{
    public static void main(String[] args)
    {
        // Create two new BigDecimals
        BigDecimal bd1 =
               new BigDecimal("124567890.0987654321");
        BigDecimal bd2 =
               new BigDecimal("987654321.123456789");

        // Addition of two BigDecimals
        bd1 = bd1.add(bd2);
        System.out.println("BigDecimal1 = " + bd1);

        // Multiplication of two BigDecimals
        bd1 = bd1.multiply(bd2);
        System.out.println("BigDecimal1 = " + bd1);

        // Subtraction of two BigDecimals
        bd1 = bd1.subtract(bd2);
        System.out.println("BigDecimal1 = " + bd1);

        // Division of two BigDecimals
        bd1 = bd1.divide(bd2);
        System.out.println("BigDecimal1 = " + bd1);

        // BigDecimal1 raised to the power of 2
        bd1 = bd1.pow(2);
        System.out.println("BigDecimal1 = " + bd1);

        // Negate value of BigDecimal1
        bd1 = bd1.negate();
        System.out.println("BigDecimal1 = " + bd1);
    }
}
```

Output:-

```
BigDecimal1 = 1112222211.2222222211
BigDecimal1 = 1098491072963113850.7436076939614540479
BigDecimal1 = 1098491071975459529.6201509049614540479
BigDecimal1 = 1112222210.2222222211
BigDecimal1 = 1237038244911605079.77528397755061728521
BigDecimal1 = -1237038244911605079.77528397755061728521
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

PART II
STRINGS

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

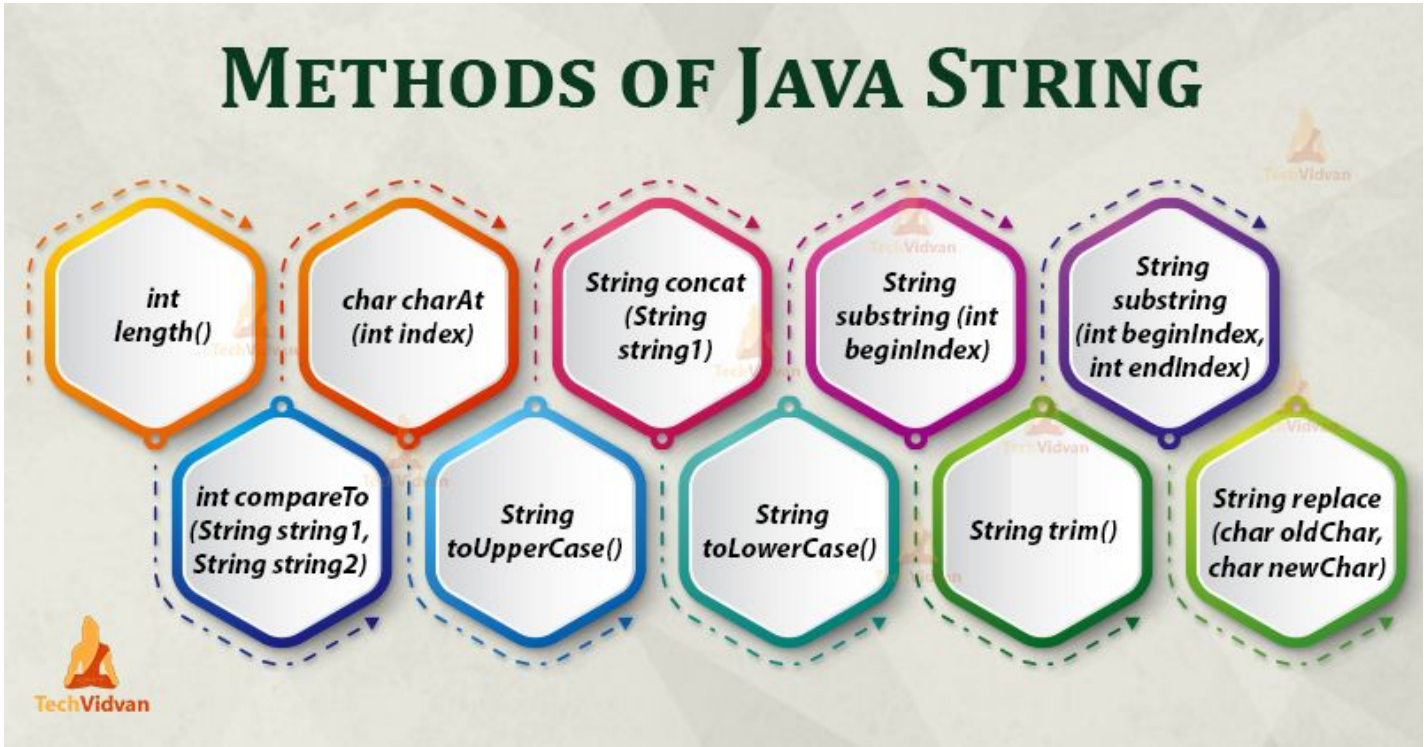
STRING CLASS

- In Java, a string is a sequence of characters. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.
- Unlike other programming languages, strings in Java are not primitive types (like int, char, etc). Instead, all strings are objects of a predefined class named String. For example,
- Here, we have created a string named type. Here, we have initialized the string with "java programming". In Java, we use double quotes to represent a string.
- The string is an instance of the String class.

```
// create a string
String type = "java programming";
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

STRING CLASS



AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

Example 1: Java find string's length

```java
class Main {
  public static void main(String[] args) {

    // create a string
    String greet = "Hello! World";
    System.out.println("The string is: " + greet);

    //checks the string length
    System.out.println("The length of the string: " + greet.length());
  }
}
```

Output

```
The string is: Hello! World
The length of the string: 12
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

Example 2: Java join two strings using concat()

```java
class Main {
  public static void main(String[] args) {

    // create string
    String greet = "Hello! ";
    System.out.println("First String: " + greet);

    String name = "World";
    System.out.println("Second String: " + name);

    // join two strings
    String joinedString = greet.concat(name);
    System.out.println("Joined String: " + joinedString);
  }
}
```

Output

```
First String: Hello!
Second String: World
Joined String: Hello! World
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

Example 3: Java join strings using + operator

```java
class Main {
  public static void main(String[] args) {

    // create string
    String greet = "Hello! ";
    System.out.println("First String: " + greet);

    String name = "World";
    System.out.println("Second String: " + name);

    // join two strings
    String joinedString = greet + name;
    System.out.println("Joined String: " + joinedString);
  }
}
```

Output

```
First String: Hello!
Second String: World
Joined String: Hello! World
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

Example 4: Java compare two strings

```
class Main {
  public static void main(String[] args) {

    // create strings
    String first = "java programming";
    String second = "java programming";
    String third = "python programming";

    // compare first and second strings
    boolean result1 = first.equals(second);
    System.out.println("Strings first and second are equal: " + result1);

    //compare first and third strings
    boolean result2 = first.equals(third);
    System.out.println("Strings first and third are equal: " + result2);
  }
}
```

Output

```
Strings first and second are equal: true
Strings first and third are equal: false
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

Example 5: Java get characters from a string

```java
class Main {
  public static void main(String[] args) {

    // create string using the string literal
    String greet = "Hello! World";
    System.out.println("The string is: " + greet);

    // returns the character at 3
    System.out.println("The character at 3: " + greet.charAt(3));

    // returns the character at 7
    System.out.println("The character at 7: " + greet.charAt(7));
  }
}
```

Output

```
The string is: Hello! World
The character at 3: l
The character at 7: W
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

STRINGBUILDER CLASS

- StringBuilder objects are like String objects, except that they can be modified. Hence Java StringBuilder class is also used to create mutable (modifiable) string object. StringBuilder is same as StringBuffer except for one important difference. StringBuilder is not synchronized, which means it is not thread safe. At any point, the length and content of the sequence can be changed through method invocations.
- StringBuilder class provides an API compatible with StringBuffer, but with no guarantee of synchronization. This class is designed for use as a drop-in replacement for StringBuffer in places where the string buffer was being used by a single thread. Where possible, it is recommended that this class be used in preference to StringBuffer as it will be faster under most implementations.
- Instances of StringBuilder are not safe for use by multiple threads. If such synchronization is required then it is recommended that StringBuffer be used.

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

CONSTRUCTOR'S OF STRINGBUILDER CLASS

- **StringBuilder ( ) :** Constructs a string builder with no characters in it and an initial capacity of 16 characters.
- **StringBuilder (int capacity) :** Constructs a string builder with no characters in it and an initial capacity specified by the capacity argument.
- **StringBuilder (String str) :** Constructs a string builder initialized to the contents of the specified string. The initial capacity of the string builder is 16 plus the length of the string argument.

APPEND()

- The append() method concatenates the given argument(string representation) to the end of the invoking StringBuilder object. StringBuilder class has several overloaded append() method. Few are:
  - StringBuilder append(String str)
  - StringBuilder append(int n)
  - StringBuilder append(Object obj)

```
StringBuilder strBuilder = new StringBuilder("Core");
strBuilder.append("JavaGuru");
System.out.println(strBuilder);
strBuilder.append(101);
System.out.println(strBuilder);
```

Output:

```
CoreJavaGuru
CoreJavaGuru101
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

INSERT()

- The insert() method inserts the given argument(string representation) into the invoking StringBuilder object at the given position.

```
StringBuilder  strBuilder=new StringBuilder ("Core");
strBuilder.insert(1,"Java");
System.out.println(strBuilder);
```

Output:

```
CJavaore
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

REPLACE

- The replace() method replaces the string from specified start index to the end index.

```
StringBuilder strBuilder=new StringBuilder("Core");
strBuilder.replace( 2, 4, "Java");
System.out.println(strBuilder);
```

Output:

```
CoJava
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

REVERSE()

- This method reverses the characters within a StringBuilder object.

```
StringBuilder strBuilder=new StringBuilder("Core");
strBuilder.reverse();
System.out.println(strBuilder);
```

Output:

```
eroC
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

CAPACITY()

- The capacity() method returns the current capacity of StringBuilder object. The capacity is the amount of storage available for newly inserted characters, beyond which an allocation will occur

```
StringBuilder strBuilder=new StringBuilder();
System.out.println(strBuilder.capacity());
strBuilder.append("1234");
System.out.println(strBuilder.capacity());
strBuilder.append("123456789112");
System.out.println(strBuilder.capacity());
strBuilder.append("1");
System.out.println(strBuilder.capacity()); //(oldcapacity*2)+2

StringBuilder strBuilder2=new StringBuilder("1234");
System.out.println(strBuilder2.capacity());
```

Output:

```
16
16
16
34
20
```

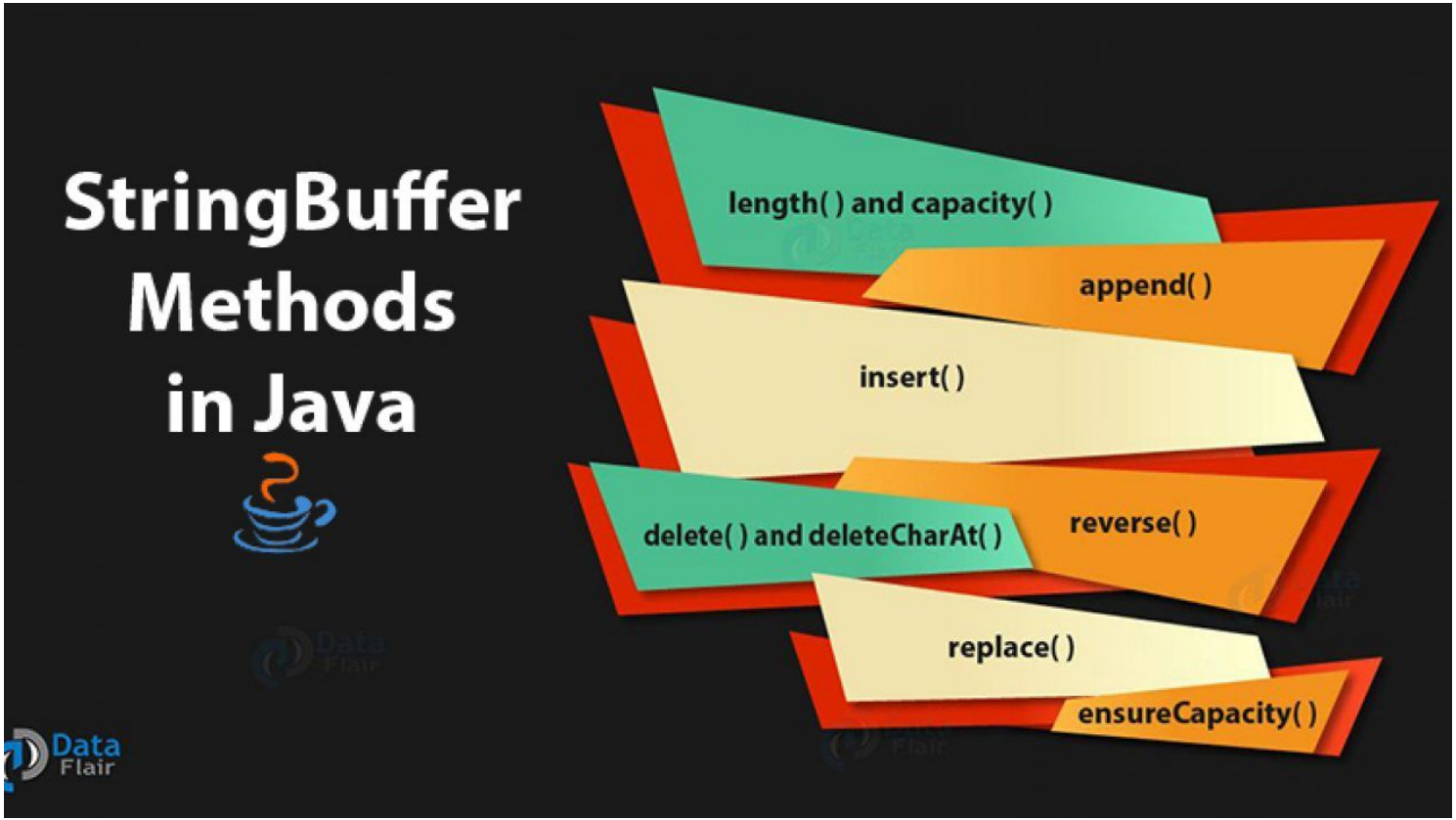
AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

STRINGBUFFER CLASS

- Java StringBuffer class is used to create mutable (modifiable) string object. A string buffer is like a String, but can be modified.
- As we know that String objects are immutable, so if we do a lot of modifications to String objects, we may end up with a memory leak. To overcome this we use StringBuffer class.
- StringBuffer class represents growable and writable character sequence. It is also thread-safe i.e. multiple threads cannot access it simultaneously.
- Every string buffer has a capacity. As long as the length of the character sequence contained in the string buffer does not exceed the capacity, it is not necessary to allocate a new internal buffer array. If the internal buffer overflows, it is automatically made large

CONSTRUCTOR OF STRINGBUFFER CLASS

- **StringBuffer ( ) :** Creates an empty string buffer with the initial capacity of 16.
- **StringBuffer (int capacity) :** Creates an empty string buffer with the specified capacity as length.
- **StringBuffer (String str) :** Creates a string buffer initialized to the contents of the specified string.
- **StringBuffer (charSequence[] ch) :** Creates a string buffer that contains the same characters as the specified CharSequence.



AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

APPEND()

- The append() method concatenates the given argument(string representation) to the end of the invoking StringBuffer object. StringBuffer class has several overloaded append() method.
  - StringBuffer append(String str)
  - StringBuffer append(int n)
  - StringBuffer append(Object obj)

```
StringBuffer strBuffer = new StringBuffer("Core");
strBuffer.append("JavaGuru");
System.out.println(strBuffer);
strBuffer.append(101);
System.out.println(strBuffer);
```

Output:

```
CoreJavaGuru
CoreJavaGuru101
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

INSERT()

- The insert() method inserts the given argument(string representation) into the invoking StringBuffer object at the given position.

```
StringBuffer strBuffer=new StringBuffer("Core");
strBuffer.insert(1,"Java");
System.out.println(strBuffer);
```

Output:

```
CJavaore
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

REPLACE()

- The replace() method replaces the string from specified start index to the end index.

```
StringBuffer strBuffer=new StringBuffer("Core");
strBuffer.replace( 2, 4, "Java");
System.out.println(strBuffer);
```

Output:

```
CoJava
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

REVERSE()

- This method reverses the characters within a StringBuffer object.

```
StringBuffer strBuffer=new StringBuffer("Core");
strBuffer.reverse();
System.out.println(strBuffer);
```

Output:

```
eroC
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

CAPACITY()

- The capacity() method returns the current capacity of StringBuffer object. The capacity is the amount of storage available for newly inserted characters, beyond which an allocation will occur.

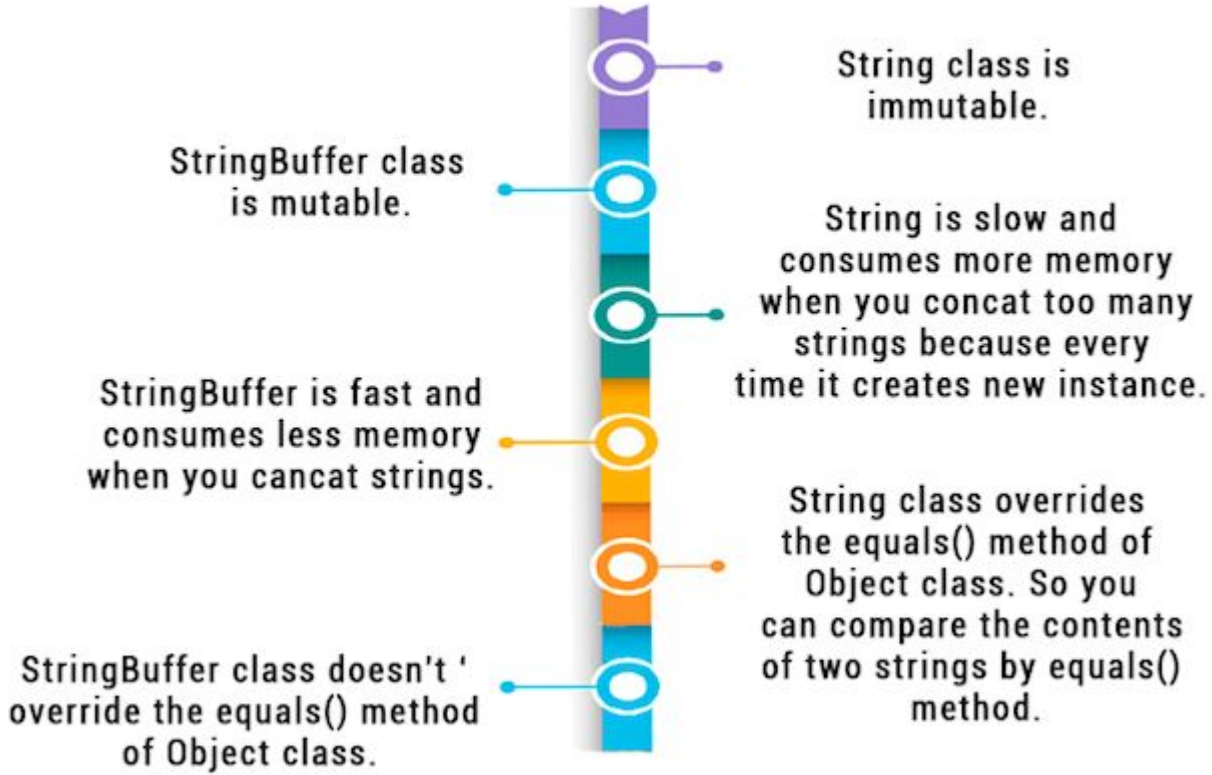
```
StringBuffer strBuffer=new StringBuffer();
System.out.println(strBuffer.capacity());
strBuffer.append("1234");
System.out.println(strBuffer.capacity());
strBuffer.append("123456789112");
System.out.println(strBuffer.capacity());
strBuffer.append("1");
System.out.println(strBuffer.capacity()); //(oldcapacity*2)+2

StringBuffer strBuffer2=new StringBuffer("1234");
System.out.println(strBuffer2.capacity());
```

Output:

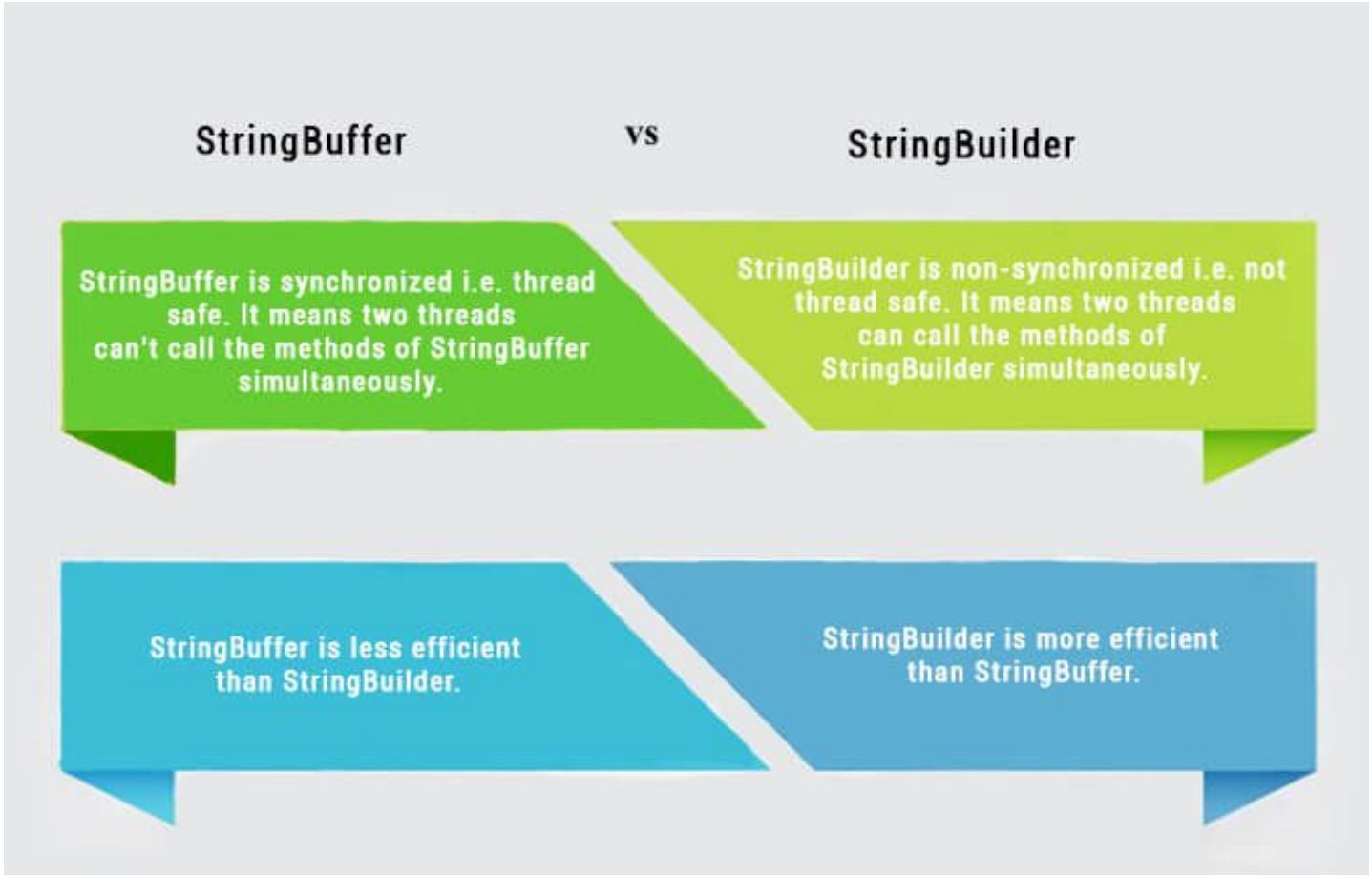
```
16
16
16
34
20
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

StringBuffer vs String



AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY



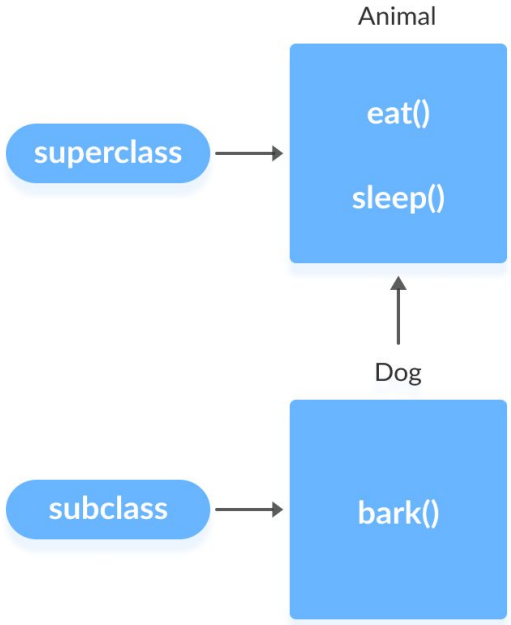
AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

	String	StringBuffer	StringBuilder
Storage	String pool	Heap	Heap
Modifiable	No(immutable)	Yes (mutable)	Yes (mutable)
Thread safe	Yes	Yes	No
Synchronized	Yes	Yes	No
Performance	Fast	Slow	Fast

PART III
INHERITANCE AND POLYMORPHISM

SUPERCLASS AND SUBCLASS

- **Java Inheritance** (**Subclass** and **Superclass**) In **Java**, it is possible to inherit attributes and methods from one class to another. ... **subclass** (child) - the class that inherits from another class. **superclass** (parent) - the class being inherited from.



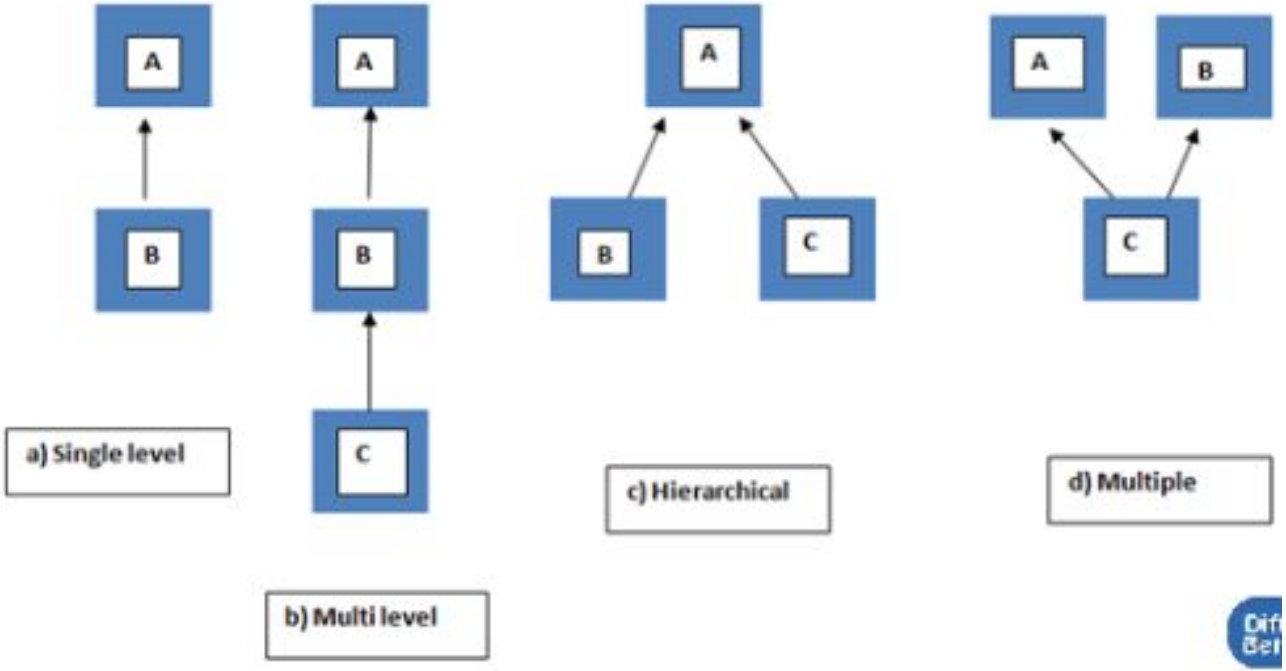
AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

DIFFERENCE BETWEEN SUPERCLASS & SUBCLASS

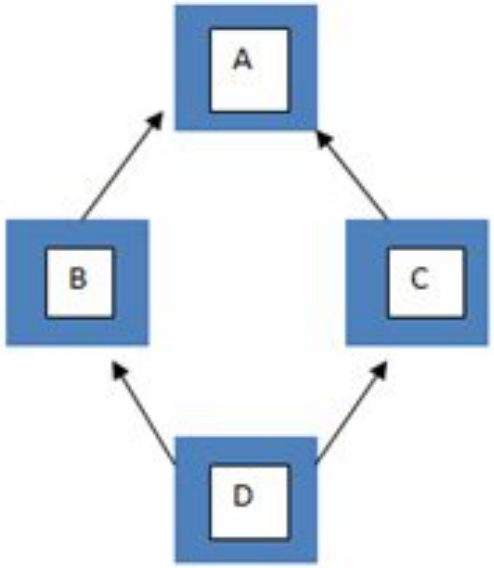
Superclass vs Subclass	
When implementing inheritance, the existing class from which the new classes are derived is the Superclass.	When implementing inheritance, the class that inherits the properties and methods from the Superclass is the Subclass.
Synonyms	
Superclass is known as base class, parent class.	Subclass is known as derived class, child class.
Functionality	
A superclass cannot use the properties and methods of the Subclass.	A subclass can use the properties and methods of the Superclass.
Single-Level-Inheritance	
There is one Superclass.	There is one Subclass.
Hierarchical Inheritance	
There is one Superclass	There are many Subclasses.
Multiple Inheritance	
There are many Superclasses.	There is one Subclass.

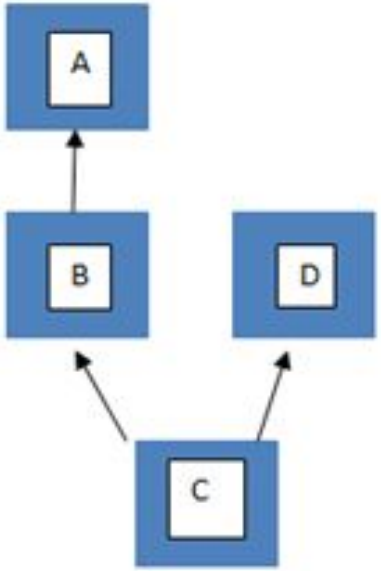
AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

TYPES OF INHERITANCE



AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY





e) Hybrid

- According to the above diagrams, Superclasses varies from each inheritance type. In single-level inheritance, A is the Superclass. In Multilevel inheritance, A is the Superclass for B and B is the Superclass for C. In Hierarchical Inheritance A is the Superclass for both B and C. In multiple inheritances both A and B are Superclasses for C.
- Hybrid inheritance is a combination of multi-level and multiple inheritances. In the left-hand side diagram, A is the Superclass for B, C and B, C are the Superclasses for D. In the right-hand side diagram, A is the Superclass for B and B, D are Superclasses for C.

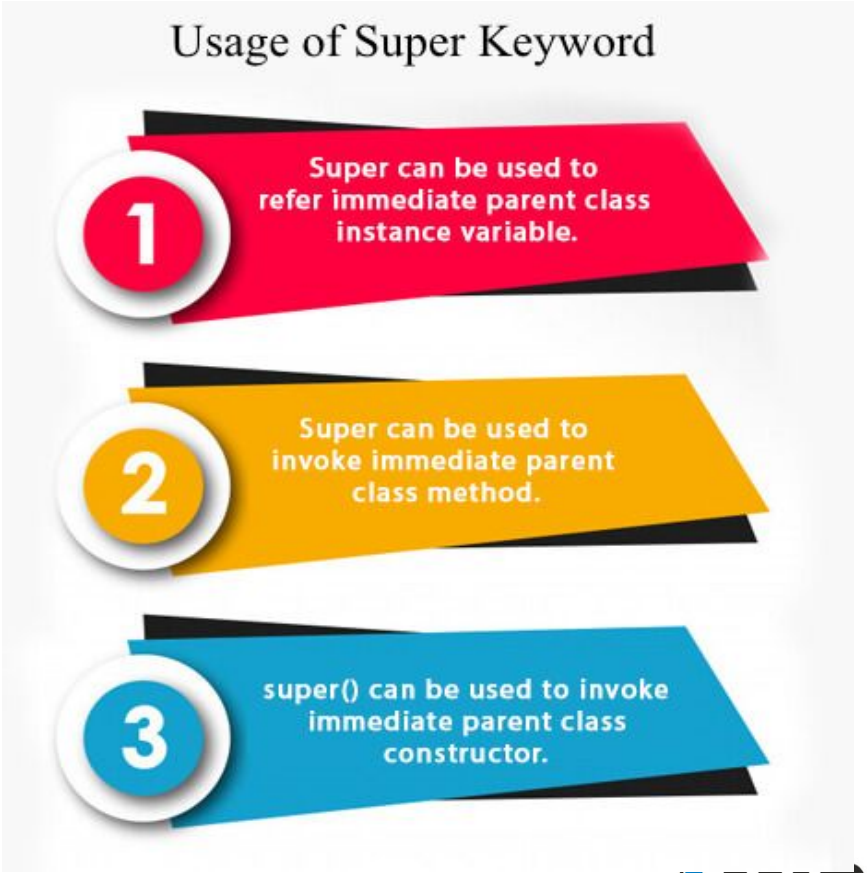
*SuperclassDemo.java

```java
public class SuperclassDemo {

    public static void main(String[] args) {
        B obj= new B();
        obj.multiply();
        obj.sub();
        obj.sum();
    }
}
class A{

    public void sum(){
        System.out.println("Summation");
    }

    public void sub(){
        System.out.println("Substraction");
    }
}
class B extends A{

    public void multiply(){
        System.out.println("Multiply");
    }

}
```

- According to the above program, class A have sum() and sub() methods. Class B has multiply() method. Class B is extending class A. Therefore, properties and methods of class A are accessible by class B. Therefore, class A is the Superclass. The reference type of class B is taken to create the object. So, all methods such as sum(), sub() and multiply() are accessible by the object. If Superclass reference type is used for object creation, the members of class B cannot be accessible. e.g. A obj = new B(); Therefore, Superclass reference cannot call the method multiply() because that method belongs to class B.

SUPER KEYWORD

- The **super keyword in Java** is a reference variable which is used to refer immediate parent class object. Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by **super** reference variable.



AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

1) How to use super keyword to access the variables of parent class

- When you have a variable in child class which is already present in the parent class then in order to access the variable of parent class, you need to use the super keyword.
- By calling a variable like this, we can access the variable of parent class if both the classes (parent and child) have same variable.
  - `super.variable_name`
- Let's take the same example that we have seen above, this time in print statement we are passing `super.num` instead of num.

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

```
class Superclass
{
   int num = 100;
}
class Subclass extends Superclass
{
   int num = 110;
   void printNumber(){
        /* Note that instead of writing num we are
         * writing super.num in the print statement
         * this refers to the num variable of Superclass
         */
        System.out.println(super.num);
   }
   public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printNumber();
   }
}
```

Output:

100

2) Use of super keyword to invoke constructor of parent class

- When we create the object of sub class, the new keyword invokes the constructor of child class, which implicitly invokes the constructor of parent class. So the order to execution when we create the object of child class is: parent class constructor is executed first and then the child class constructor is executed. It happens because compiler itself adds super()(this invokes the no-arg constructor of parent class) as the first statement in the constructor of child class.

```
class Parentclass
{
   Parentclass(){
        System.out.println("Constructor of parent class");
   }
}
class Subclass extends Parentclass
{
   Subclass(){
        /* Compile implicitly adds super() here as the
         *  first statement of this constructor.
         */
        System.out.println("Constructor of child class");
   }
   Subclass(int num){
        /* Even though it is a parameterized constructor.
         * The compiler still adds the no-arg super() here
         */
        System.out.println("arg constructor of child class");
   }
   void display(){
        System.out.println("Hello!");
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

```
   public static void main(String args[]){
        /* Creating object using default constructor. This
         * will invoke child class constructor, which will
         * invoke parent class constructor
         */
        Subclass obj= new Subclass();
        //Calling sub class method
        obj.display();
        /* Creating second object using arg constructor
         * it will invoke arg constructor of child class which will
         * invoke no-arg constructor of parent class automatically
         */
        Subclass obj2= new Subclass(10);
        obj2.display();
   }
}
```

Output:

```
Constructor of parent class
Constructor of child class
Hello!
Constructor of parent class
arg constructor of child class
Hello!
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

3) How to use super keyword in case of method overriding

- When a child class declares a same method which is already present in the parent class then this is called method overriding. We will learn method overriding in the next tutorials of this series. For now you just need to remember this: When a child class overrides a method of parent class, then the call to the method from child class object always call the child class version of the method. However by using super keyword like this: super.method_name you can call the method of parent class (the method which is overridden). In case of method overriding, these terminologies are used: Overridden method: The method of parent class Overriding method: The method of child class Lets take an example to understand this concept:

```
class Parentclass
{
   //Overridden method
   void display(){
        System.out.println("Parent class method");
   }
}
class Subclass extends Parentclass
{
   //Overriding method
   void display(){
        System.out.println("Child class method");
   }
   void printMsg(){
        //This would call Overriding method
        display();
        //This would call Overridden method
        super.display();
   }
   public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printMsg();
   }
}
```

Output:

```
Child class method
Parent class method
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

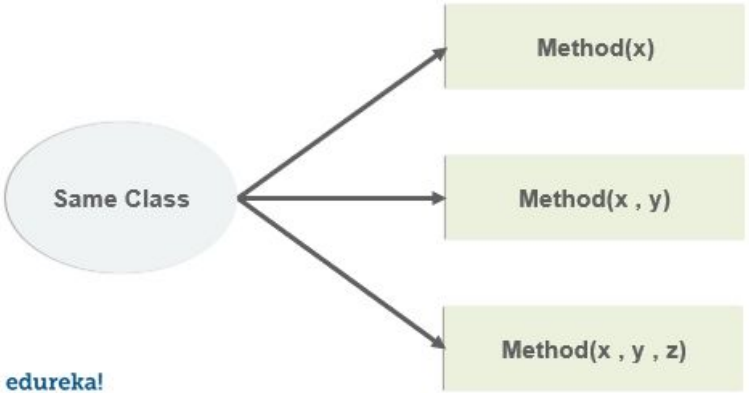
OVERRIDING AND OVERLOADING METHOD

- **Method overriding** is used to provide the specific implementation of the **method** that is already provided by its super class. ... In **java**, **method overloading** can't be performed by changing return type of the **method** only. Return type can be same or different in **method overloading**. But you must have to change the parameter.

METHOD OVERLOADING

- Method overloading allows the method to have the same name which differs on the basis of arguments or the argument types. It can be related to compile-time polymorphism. Following are a few pointers that we have to keep in mind while overloading methods in Java.
  - We cannot overload a return type.
  - Although we can overload static methods, the arguments or input parameters have to be different.
  - We cannot overload two methods if they only differ by a static keyword.
  - Like other static methods, the main() method can also be overloaded.

```java
public class Edureka{
public static void main(String[] args){
System.out.println("hello");
Edureka.main("edurekan");
}

public static void main(String arg1){
System.out.println(" welcome" + arg1);
Edureka.main("welcome" , "to edureka");
}

public static void main(String arg1 , String arg2){
System.out.println("hello" , +arg1 , +arg2);
}
}
```



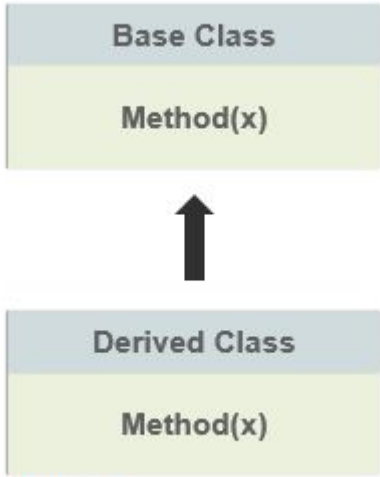
Output: hello welcome edurekan

hello, welcome to edureka

METHOD OVERRIDING

- Inheritance in java involves a relationship between parent and child classes. Whenever both the classes contain methods with the same name and arguments or parameters it is certain that one of the methods will override the other method during execution. The method that will be executed depends on the object.
- If the child class object calls the method, the child class method will override the parent class method. Otherwise, if the parent class object calls the method, the parent class method will be executed.

```java
class Parent{
void view(){
System.out.println("this is a parent class method);
}}
class Child extends Parent{
void view(){
System.out.println("this is a child class method);
}}
public static void main(String args[]){
Parent ob = new Parent();
ob.view();
Parent ob1 = new Child();
ob1.view();
```



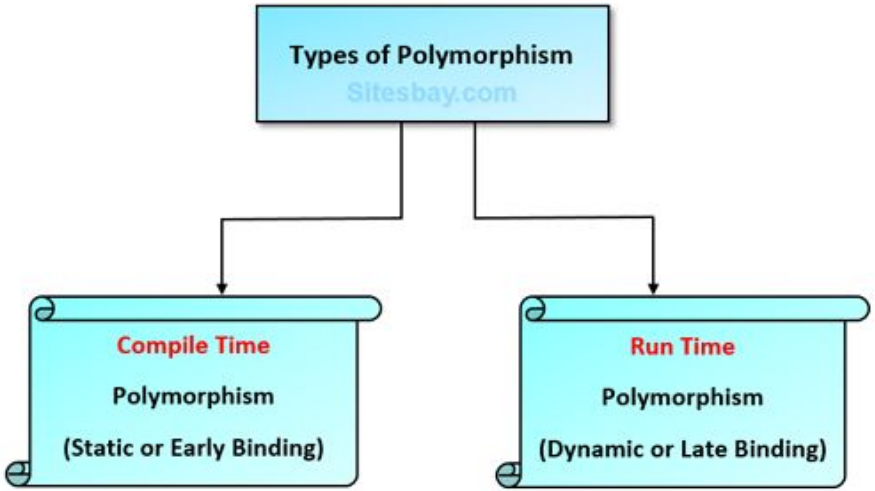
Output: this is a child class method

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

Method Overloading	Method Overriding
• It is used to increase the readability of the program	• Provides a specific implementation of the method already in the parent class
• It is performed within the same class	• It involves multiple classes
• Parameters must be different in case of overloading	• Parameters must be same in case of overriding
• Is an example of compile-time polymorphism	• It is an example of runtime polymorphism
• Return type can be different but you must change the parameters as well.	• Return type must be same in overriding
• Static methods can be overloaded	• Overriding does not involve static methods.

POLYMORPHISM AND DYNAMIC BINDING

- **Polymorphism in Java** is a concept by which we can perform a single action in different ways. ... So **polymorphism** means many forms. There are two types of **polymorphism in Java**: compile-time **polymorphism** and runtime **polymorphism**. We can perform **polymorphism in java** by method overloading and method overriding.



AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

Runtime Polymorphism example:

Animal.java

```
public class Animal{
   public void sound(){
      System.out.println("Animal is making a sound");
   }
}
```

Horse.java

```
class Horse extends Animal{
    @Override
    public void sound(){
        System.out.println("Neigh");
    }
    public static void main(String args[]){
        Animal obj = new Horse();
        obj.sound();
    }
}
```

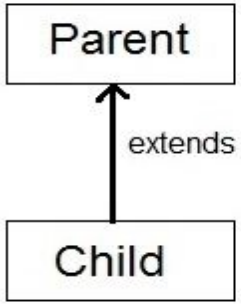
Output:

```
Neigh
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

DYNAMIC METHOD DISPATCH

- **Dynamic method dispatch** is a mechanism by which a call to an overridden **method** is resolved at runtime. This is how **java** implements runtime polymorphism. When an overridden **method** is called by a reference, **java** determines which version of that **method** to execute based on the type of object it refer to.




```
Parent p = new Parent( );

Child c = new Child( );

Parent p = new Child( );
```

Upcasting

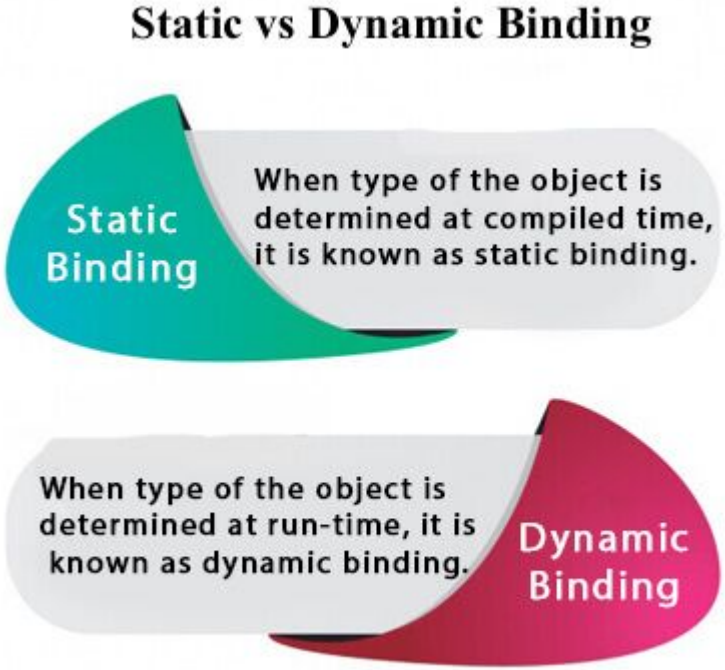
```
Child c = new Parent( );
```

incompatible type

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

DYNAMIC BINDING

- When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding. Method Overriding is a perfe example of dynamic binding as in overriding both parent and child classe have same method and in this case the **type of the object** determines which method is to be executed. The type of object is determined at the run time so this is known as dynamic binding.



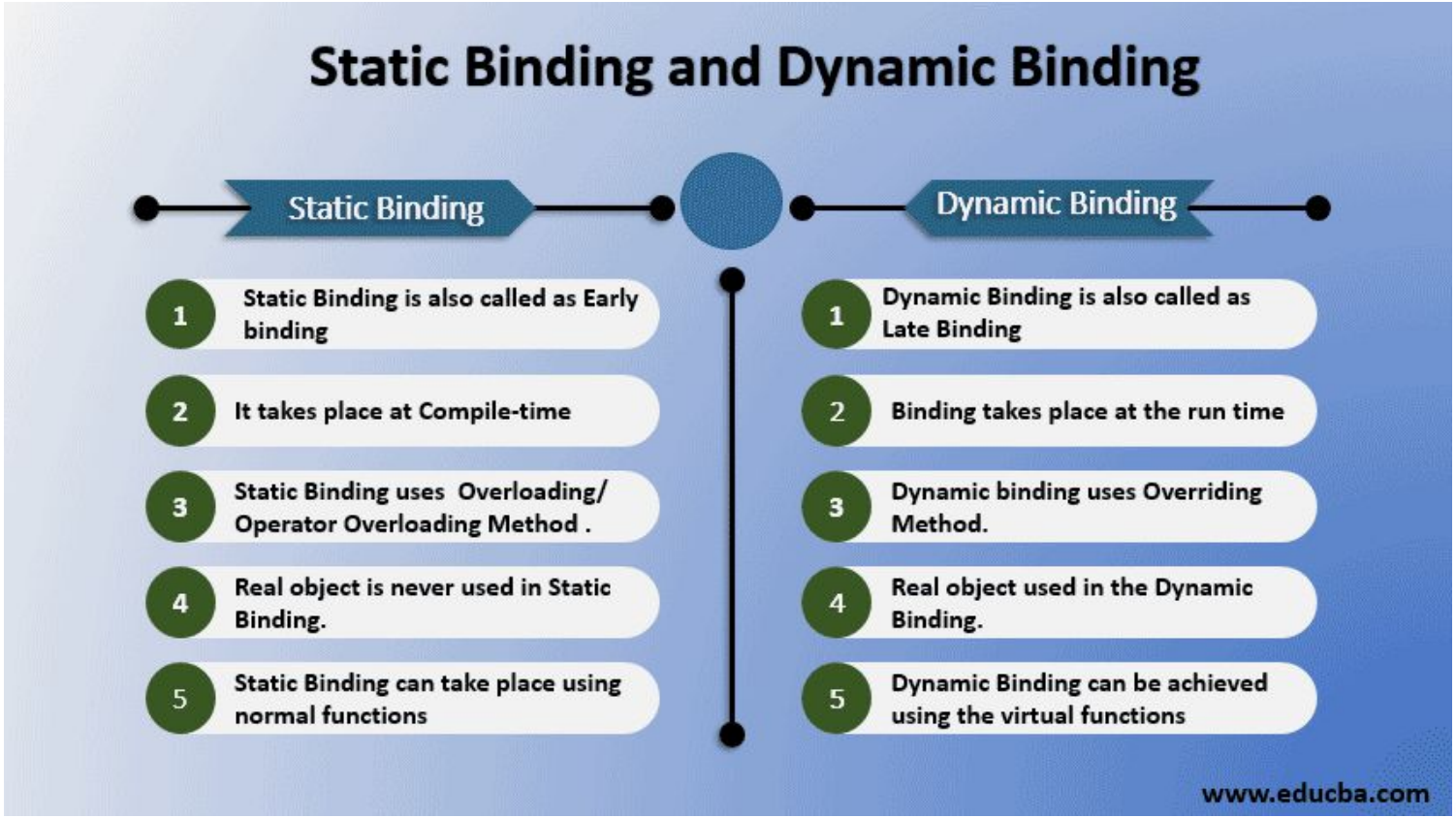
AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

```
class Human{
   //Overridden Method
   public void walk()
   {
       System.out.println("Human walks");
   }
}
class Demo extends Human{
   //Overriding Method
   public void walk(){
       System.out.println("Boy walks");
   }
   public static void main( String args[]) {
       /* Reference is of Human type and object is
        * Boy type
        */
       Human obj = new Demo();
       /* Reference is of HUman type and object is
        * of Human type.
        */
       Human obj2 = new Human();
       obj.walk();
       obj2.walk();
   }
}
```

Output:

```
Boy walks
Human walks
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY



AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

CASTING OBJECTS

- A cast, instructs the compiler to change the existing type of an object reference to another type.
- In Java, all casting will be checked both during compilation and during execution to ensure that they are legitimate.
- An attempt to cast an object to an incompatible object at runtime will results in a ClassCastException.
- A cast can be used to narrow or downcast the type of a reference to make it more specific

```java
class Animal {
  @Override
  public String toString() {
    return "I am an Animal";
  }
}

class Cow extends Animal {
    @Override
  public String toString() {
    return "I am a Cow";
  }
}

public class ObjectCasting {
  public static void main(String args[]) {
    Animal creature;
    Cow daisy = new Cow();
    System.out.println( daisy); // prints: I am a Cow
    creature = daisy;         // OK
    System.out.println(creature); // prints: I am a Cow
    creature = new Animal();
    System.out.println(creature); // prints: I am a Animal
//    daisy = creature;     // Compile-time error, incompatible type
    if (creature instanceof Cow) {
      daisy = (Cow) creature;   // OK but not an instance of Cow
      System.out.println( daisy);
    }
  }
}
```

The result of this is:

```
I am a Cow
I am a Cow
I am an Animal
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

FINAL METHOD AND CLASSES

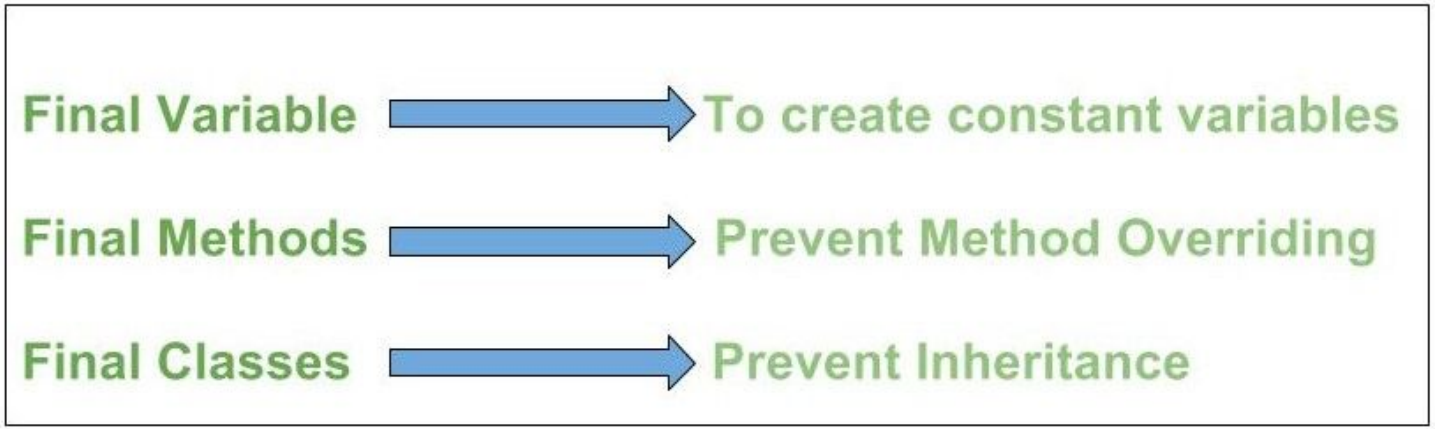
You can declare some or all of a class's methods *final*. You use the final keyword in a method declaration to indicate that the method cannot be overridden by subclasses. The Object class does this—a number of its methods are final.

You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object. For example, you might want to make the getFirstPlayer method in this ChessAlgorithm class final:

```
class ChessAlgorithm {

    enum ChessPlayer { WHITE, BLACK }

    final ChessPlayer getFirstPlayer() {

        return ChessPlayer.WHITE;

    }

}
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

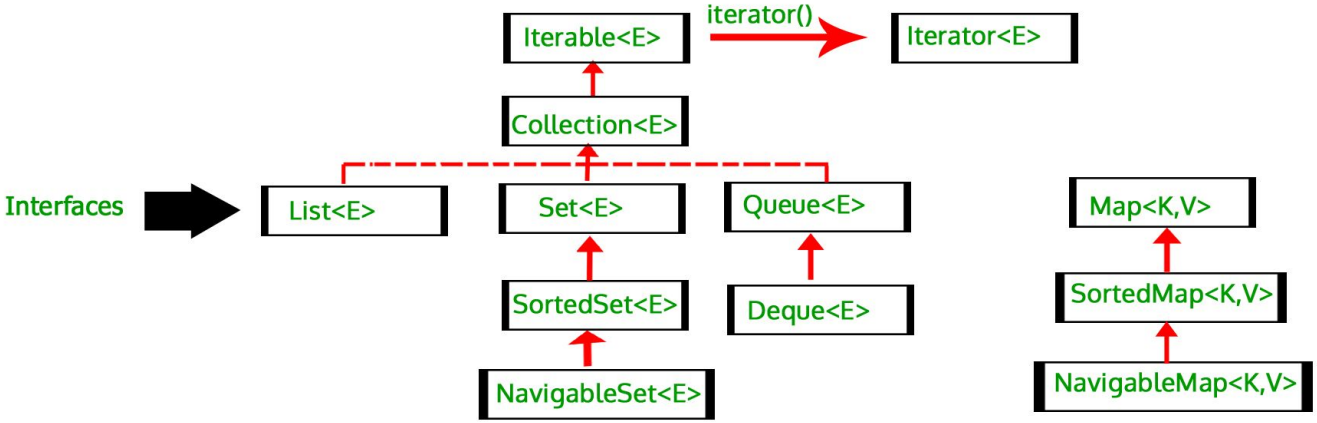
- Methods called from constructors should generally be declared final. If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results.
- Note that you can also declare an entire class final. A class that is declared final cannot be subclassed. This is particularly useful, for example, when creating an immutable class like the String class.



AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

ARRAYLIST CLASS AND METHODS

- Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.
- The important points about Java ArrayList class are:
  - Java ArrayList class can contain duplicate elements.
  - Java ArrayList class maintains insertion order.
  - Java ArrayList class is non synchronized.
  - Java ArrayList allows random access because array works at the index basis.
  - In Java ArrayList class, manipulation is slow because a lot of shifting needs to occur if any element is removed from the array list.



AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

`add(`**value**`)`	appends value at end of list
`add(`**index**`, `**value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index**`, `**value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

```java
import java.util.*;
 class ArrayList1{
 public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
      list.add("Ravi");//Adding object in arraylist
      list.add("Vijay");
      list.add("Ravi");
      list.add("Ajay");
      //Invoking arraylist object
      System.out.println(list);
  }
 }
}
```

```
[Ravi, Vijay, Ravi, Ajay]
```

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY

AMIRAJ
COLLEGE OF ENGINEERING & TECHNOLOGY