

# **INTRODUCTION**

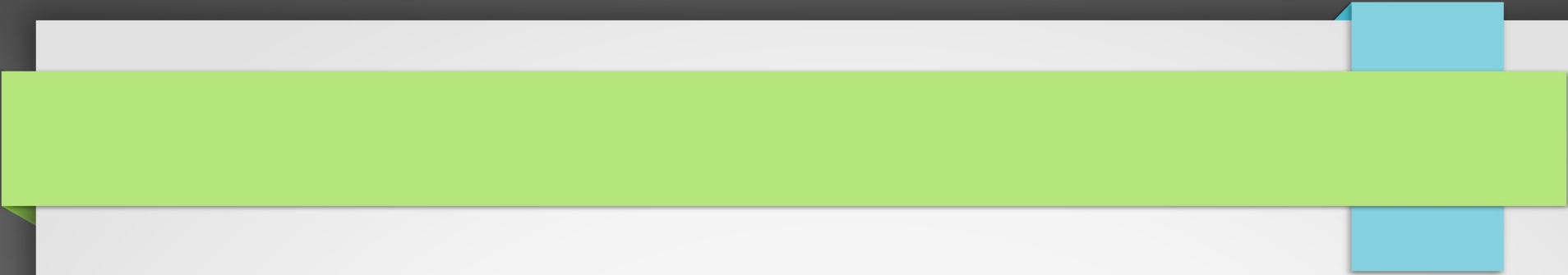
**210301301**  
**CORE JAVA**

**210301305**  
**PRACTICAL ON CORE JAVA**

**BY:**  
**Prof. (Dr.) Ankit Bhavsar**

# 210301301 CORE JAVA

UNIT	MODULES	WEIGHTAGE
1	Introduction to Java	20 %
2	Java Programming Constructs, Classes, Vectors and Arrays	20 %
3	Inheritance, Interface and Packages	20 %
4	Exception Handling and Multi - Threading	20 %
5	Applets	20 %



# **Unit – III**

## **Inheritance, Interface, Package**

# Index

- Inheritance
  - Introduction
  - Types of Inheritance
  - extend, super, final keyword
  - Overriding of Methods
  - Abstract Class
- Interface
  - Introduction
  - Variables in Interface
  - Extending Interface
  - Interface Vs Abstract Class

# Index

- Package
  - Introduction
  - Creating Package
  - Using Package
  - Access Protection
  - Java.lang package
    - Object class
    - Wrapper class
    - String class
    - String Buffer class

# Introduction

- Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.
- The idea behind inheritance in java is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.
- Inheritance represents the **IS-A relationship**, also known as parent-child relationship.

# Introduction

- Why use inheritance in java
  - For Method Overriding
  - For Code Reusability.

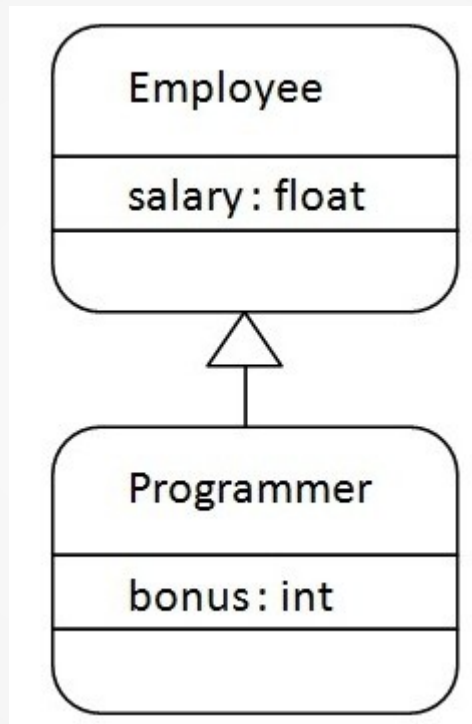
## Syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
    //methods and fields
}
```

The **extends** keyword indicates that you are making a new class that **derives from an existing class**. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called parent or super class and the new class is called child or subclass.

# Introduction



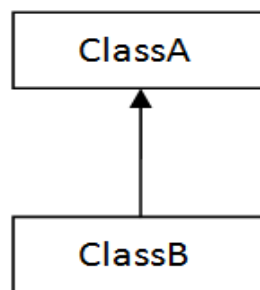
As displayed in the above figure, Programmer is the subclass and Employee is the superclass. **Relationship between two classes is Programmer IS-A Employee.**

It means that Programmer is a type of Employee.

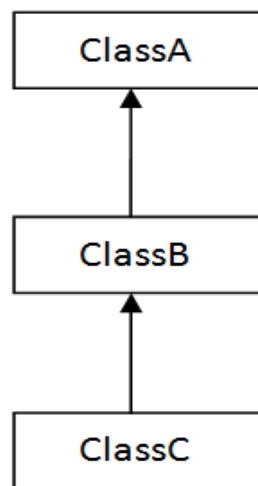


# Types of Inheritance

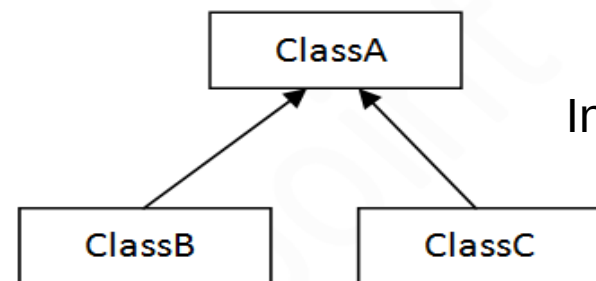
- On the basis of class, there can be **three types of inheritance in java**: single, multilevel and hierarchical.
- In java programming, multiple and hybrid inheritance is supported through interface only.
- **Multiple inheritance is not supported in java through class.**



1) Single



2) Multilevel



3) Hierarchical

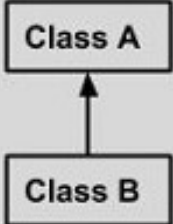
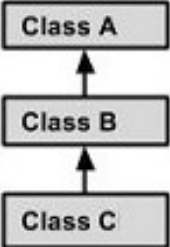
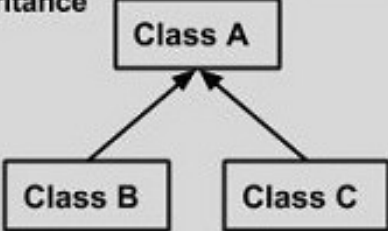
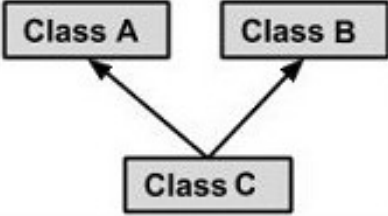
Employee1.java

Inheritance2.java

Inheritance3.java

Inheritance4.java

# Types of Inheritance

<b>Single Inheritance</b>  <pre> graph BT     B[Class B] --&gt; A[Class A]         </pre>	<pre> public class A {     ..... } public class B extends A {     ..... }         </pre>
<b>Multi Level Inheritance</b>  <pre> graph BT     C[Class C] --&gt; B[Class B]     B --&gt; A[Class A]         </pre>	<pre> public class A { .....} public class B extends A {.....} public class C extends B {.....}         </pre>
<b>Hierarchical Inheritance</b>  <pre> graph BT     B[Class B] --&gt; A[Class A]     C[Class C] --&gt; A         </pre>	<pre> public class A { .....} public class B extends A {.....} public class C extends A {.....}         </pre>
<b>Multiple Inheritance</b>  <pre> graph BT     C[Class C] --&gt; A[Class A]     C --&gt; B[Class B]         </pre>	<pre> public class A { .....} public class B {.....} public class C extends A,B {     ..... }         </pre> <p>// Java does not support mutiple Inheritance</p>

# Why multiple inheritance is not supported in java?

- To reduce the complexity and simplify the language, **multiple inheritance is not supported in java.**
- Consider a scenario where A, B and C are three classes. The C class inherits A and B classes.
- If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.
- Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes.
- So whether you have same method or different, there will be compile time error now.

Inheritance5.java

# Method Overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in java.
- In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

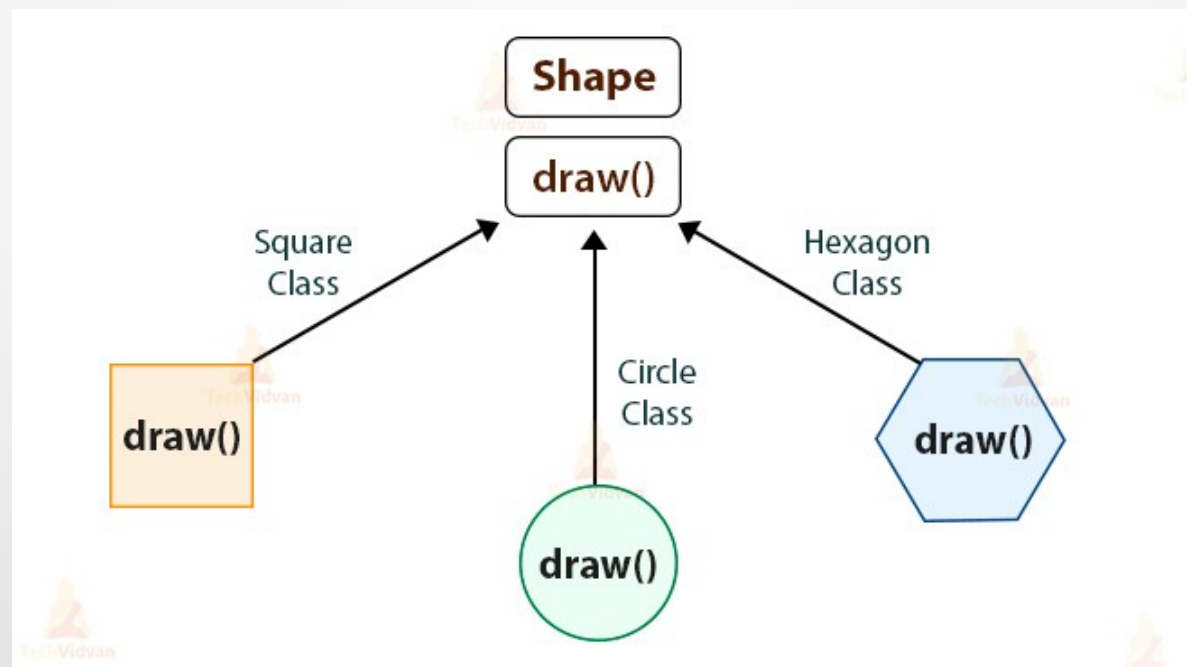
## Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

# Example of Method Overriding

## Rules for Java Method Overriding

- Method must have same name as in the parent class
- Method must have same parameter as in the parent class.
- Must be IS-A relationship (inheritance).



Inheritance6.java

# Super keyword in java

- 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.

## Usage of java super Keyword :

- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

Inheritance7.java   Inheritance8.java   Inheritance9.java

super() is added in each class constructor automatically by compiler if there is no super() or this().

Inheritance10.java

Inheritance11.java

# Final keyword in java

- The final keyword in java is **used to restrict the user.**
- The java final keyword can be used in many context.
- **Final can be:**
  - **Variable**
  - **Method**
  - **Class**
- The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable.
- **It can be initialized in the constructor only.**
- The blank final variable can be static also which will be initialized in the static block only.

# Final keyword

- Final Variable
  - If you make any variable as final, you cannot change the value of final variable(It will be constant).
- Final method
  - If you make any method as final, you cannot override it.
- Final class
  - If you make any class as final, you cannot extend it.



# Final Keyword

## Java Final Keyword

- ⇒ Stop Value Change
- ⇒ Stop Method Overriding
- ⇒ Stop Inheritance

[javatpoint.com](http://javatpoint.com)

Inheritance12.java

Inheritance13.java    Inheritance13\_1.java

Inheritance14.java    Inheritance14\_1.java

**Final Variable** → **To create constant variables**

**Final Methods** → **Prevent Method Overriding**

**Final Classes** → **Prevent Inheritance**

# Abstract keyword

- A class that is declared with abstract keyword, is known as abstract class in java.
- It can have abstract and non-abstract methods (method with body).
- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- There are two ways to achieve abstraction in java
  - Abstract class
  - Interface

# Abstract keyword

## Abstract class

- A class that is declared as abstract is known as abstract class.
- It needs to be extended and its method implemented.
- It cannot be instantiated.
- **Example**

```
abstract class A{
```

# Some rules of Abstract Class

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

# Some rules of Abstract Method

- If there is any abstract method in a class, that class must be abstract.
- If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.

# Abstract Method

## Abstract method

- A method that is declared as abstract and does not have implementation is known as abstract method.
- **Example**

`abstract void printStatus();`//no body and abstract

Inheritance16.java

Inheritance17.java

Inheritance18.java

Inheritance19.java

Inheritance20.java

# Why Interfaces?

```
public class extends Animal, Mammal{}
```

- However, a class can implement one or more interfaces, which has helped Java get rid of the impossibility of multiple inheritances.
- **The reason behind this is to prevent ambiguity.**
- Consider a case where class B extends class A and Class C and both class A and C have the same method display().
- Now java compiler cannot decide, which display method it should inherit. To prevent such situation, multiple inheritances is not allowed in java.

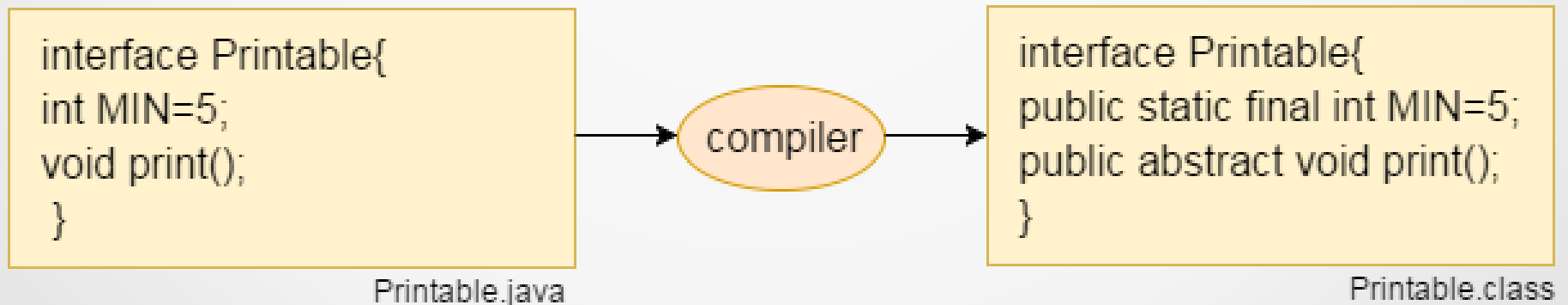
# Interfaces in Java

- An interface in java is a blueprint of a class.
- It has static constants and abstract methods.
- The interface in java is a mechanism to achieve abstraction.
- There can be only abstract methods in the java interface not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- Java Interface also represents IS-A relationship.
- It cannot be instantiated just like abstract class.
- By interface, we can support the functionality of multiple inheritance.



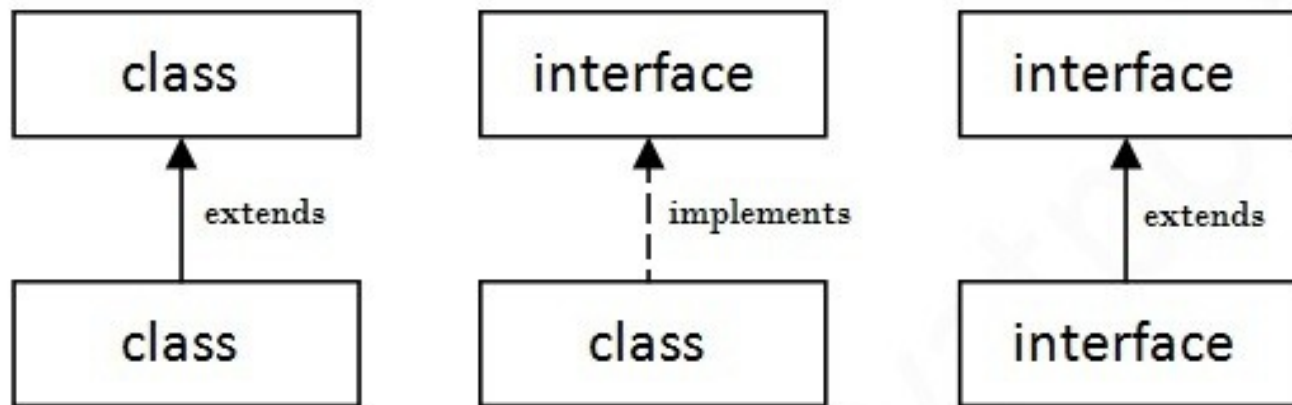
# Interfaces in Java

- The java compiler adds public and abstract keywords before the interface method. More, it adds public, static and final keywords before data members.
- In other words, Interface fields are public, static and final by default, and methods are public and abstract.



# Understanding relationship between classes and interfaces

- As shown in the figure given below, a class extends another class, an interface extends another interface but a class implements an interface.



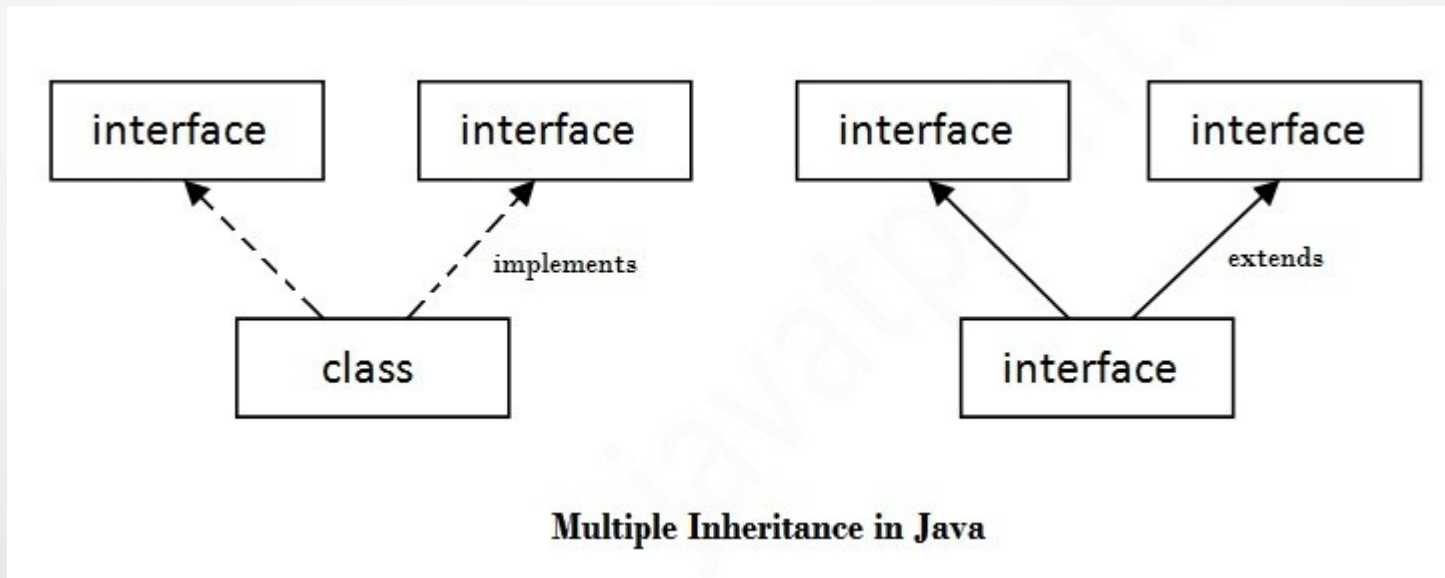
Inheritance21.java

Inheritance22.java

Unit - 3

# Multiple Inheritance in Java

- If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Inheritance23.java

Inheritance24.java

Inheritance25.java

# Interfaces in Java

- A class implements interface but one interface extends another interface .

# Interface v/s Abstract class

Interface	Abstract Class
Declared using the keyword interface.	Declared using the keyword abstract.
Multiple Inheritance is possible.	Multiple Inheritance is not possible.
<b>Implements</b> keyword is used to inherit an inheritance	<b>Extends</b> keyword is used to inherit a class.
By default, all methods in an interface are <b>public and abstract</b> ; need to tag it as public and abstract	Methods have to be tagged as <b>public or abstract</b> or both if required.
Interfaces have no implementation at all.	Abstract classes can have partial implementation.
All methods of an interface need to be overridden.	Only abstract methods need to be overridden.
All variables declared in an interface are by default public, static or final.	Variables have to be declared as public, static or final.
Interfaces do not have any constructors.	Abstract classes can have constructors.
Methods in an interface cannot be static.	Non-abstract methods can be static.

## Abstract class

## Interface

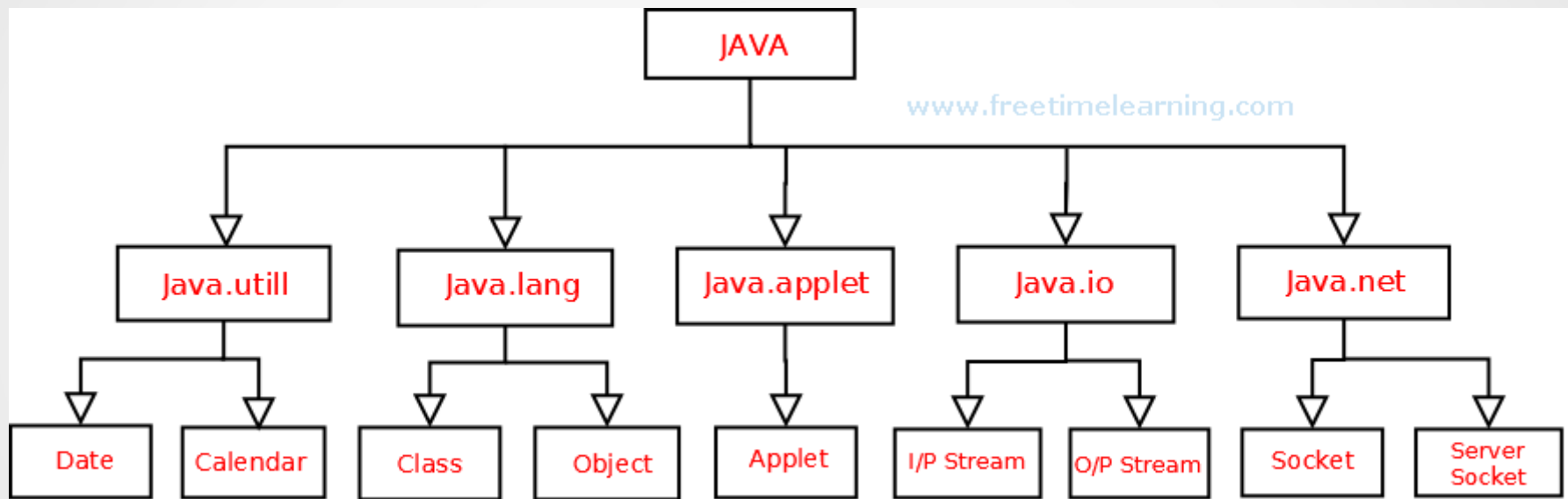
1) Abstract class can <b>have abstract and non-abstract</b> methods.	Interface can have <b>only abstract</b> methods. Since Java 8, it can have <b>default and static methods</b> also.
2) Abstract class <b>doesn't support multiple inheritance</b> .	Interface <b>supports multiple inheritance</b> .
3) Abstract class <b>can have final, non-final, static and non-static variables</b> .	Interface has <b>only static and final variables</b> .
4) Abstract class <b>can provide the implementation of interface</b> .	Interface <b>can't provide the implementation of abstract class</b> .
5) The <b>abstract keyword</b> is used to declare abstract class.	The <b>interface keyword</b> is used to declare interface.
6) An <b>abstract class</b> can extend another Java class and implement multiple Java interfaces.	An <b>interface</b> can extend another Java interface only.
7) An <b>abstract class</b> can be extended using keyword "extends".	An <b>interface</b> can be implemented using keyword "implements".
8) A Java <b>abstract class</b> can have class members like private, protected, etc.	Members of a Java interface are public by default.
9) <b>Example:</b> <pre>public abstract class Shape{     public abstract void draw(); }</pre>	<b>Example:</b> <pre>public interface Drawable{     void draw(); }</pre>

com...

# Introduction - PACKAGE

- A java package is a mechanism for organizing Java classes into groups.
- A java package is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form:
  - built-in package
  - user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- **Advantage of Java Package**
  - Java package is used to categorize the classes and interfaces so that they can be easily maintained.
  - Java package provides access protection.
  - Java package removes naming collision.

# Introduction - PACKAGE





# Packages in Java

- A package declaration resides at the top of a Java source file.
- All source files to be placed in a package have common package name.
- A package provides a unique namespace for the classes it contains.
- A package can contain:
  - Classes
  - Interfaces
  - Enumerated types
  - Annotations
- Two classes in different packages can have the same name.
- Packages provide a mechanism to hide its classes from being used by programs or packages belonging to other classes.

# Packages in Java

- **How to compile java package**

`javac -d directory javafilename`

- For example

`javac -d . Simple.java`

- The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

- **How to run java package program**

To Compile: `javac -d . Simple.java`

To Run: `java mypack.Simple`

Output: Welcome to package

# Packages in Java

How to access package from another package?

There are three ways to access the package from outside the package.

`import packagename.*;`

Package2.java    Package3.java

`import package.classname;`

Package4.java    Package5.java

`fully qualified name.`

Package6.java    Package7.java

1) Using packagename.\*

- If you use package.\* then all the classes and interfaces of this package will be accessible but not subpackages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

# Packages in Java

## 2) Using `package.name.classname`

- If you import `package.classname` then only declared class of this package will be accessible.

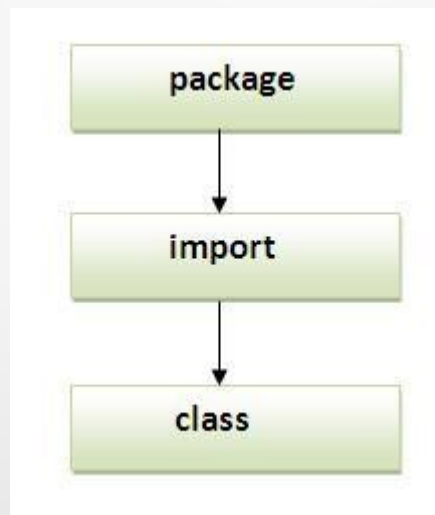
## 3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

- It is generally used when two packages have same class name e.g. `java.util` and `java.sql` packages contain `Date` class.

# Subpackages in Java

- If you import a package, subpackages will not be imported.
- If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages.
- Hence, you need to import the subpackage as well.
- Sequence of the program must be package then import then class.



# Subpackages in Java

- Package inside the package is called the subpackage. It should be created to categorize the package further.
- The standard of defining package is `domain.company.package`
- e.g. `abc.java.bean`

`Package8.java`

`Package9.java`

# Access Modifiers in Java

- There are two types of modifiers in java:
  - access modifiers
  - non-access modifiers
- The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.
- There are 4 types of java access modifiers:
  - Private
  - Default
  - Protected
  - Public
- There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc.

# Access Modifiers in Java

**Private access modifier:**      Access1.java      Access2.java

- The private access modifier is accessible only within class.
- If you make any class constructor private, you cannot create the instance of that class from outside the class.
- A class cannot be private or protected except nested class.

Access3.java      Access4.java

**Default access modifier:**

- If you don't use any modifier, it is treated as default by default.
- The default modifier is accessible only within package.

**Protected access modifier**      Access5.java      Access6.java

- The protected access modifier is accessible within package and outside the package but through inheritance only.
- The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.



# Access Modifiers in Java

## Public access modifier

Access8.java

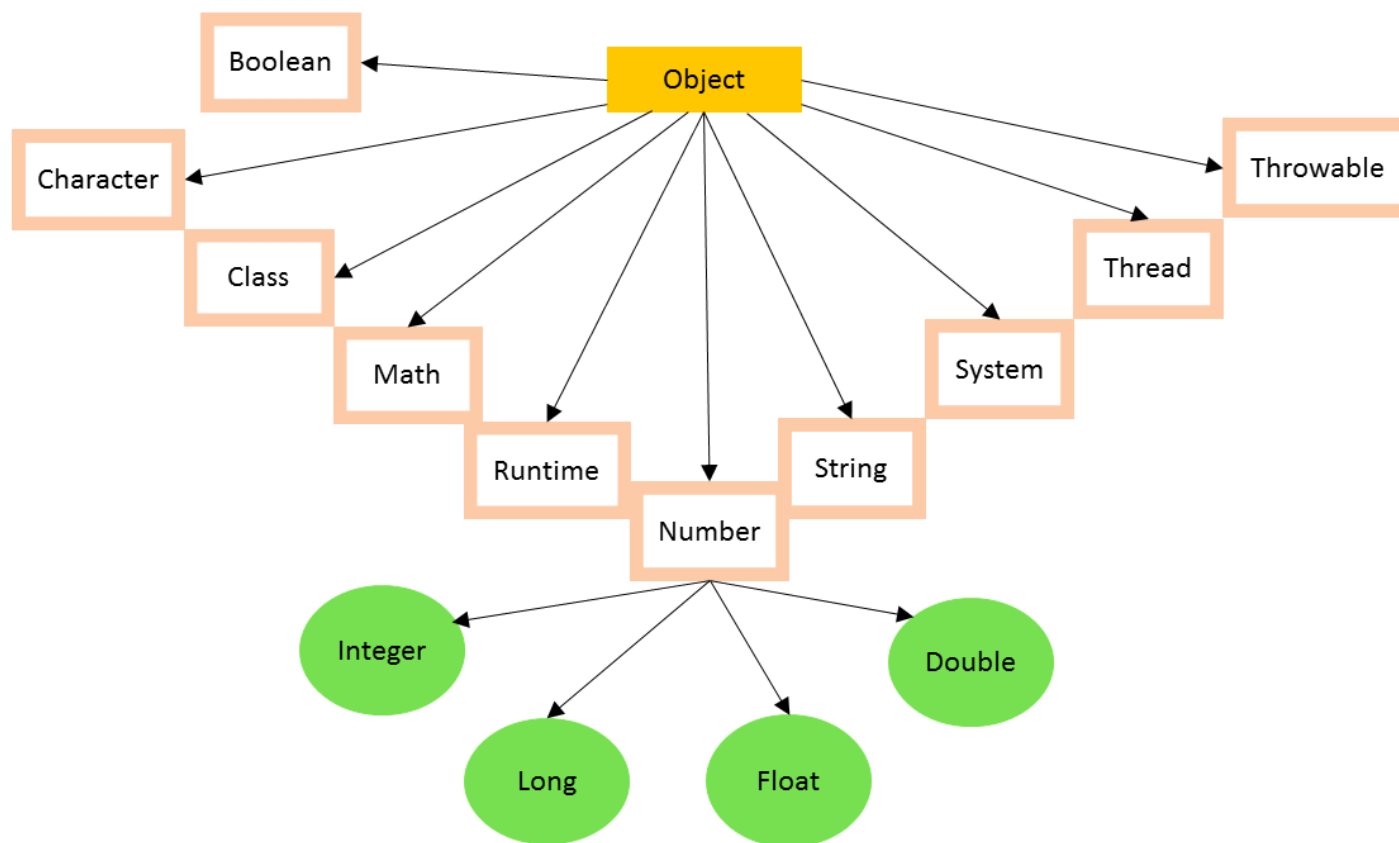
- The public access modifier is accessible everywhere.
- It has the widest scope among all other modifiers.

Access modifier	Within class	Within package	Outside package within subclass	Outside package
private	YES	NO	NO	NO
default	YES	YES	NO	NO
protected	YES	YES	YES	NO
public	YES	YES	YES	YES

# Java.lang package

- Java.lang is imported by default in all the classes that we create.
- There is no need to explicitly import the lang package.
- It contains classes that form the basic building blocks of Java.
- There are 37 classes in java.lang package.
- Some of them are as follows:
  - Boolean
  - Byte
  - Double
  - Float
  - Integer
  - Long
  - Object
  - Short

# Java.lang.Object class



# Java.lang.Object class

- The Object class is the parent class of all the classes in java by default.
- In other words, it is the topmost class of java.
- The Object class is beneficial if you want to refer any object whose type you don't know.
- There is no need to explicitly inherit the Object class.

# Java.lang.Object class

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object <i>object</i> )	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class<?> getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long <i>milliseconds</i> ) void wait(long <i>milliseconds</i> , int <i>nanoseconds</i> )	Waits on another thread of execution.

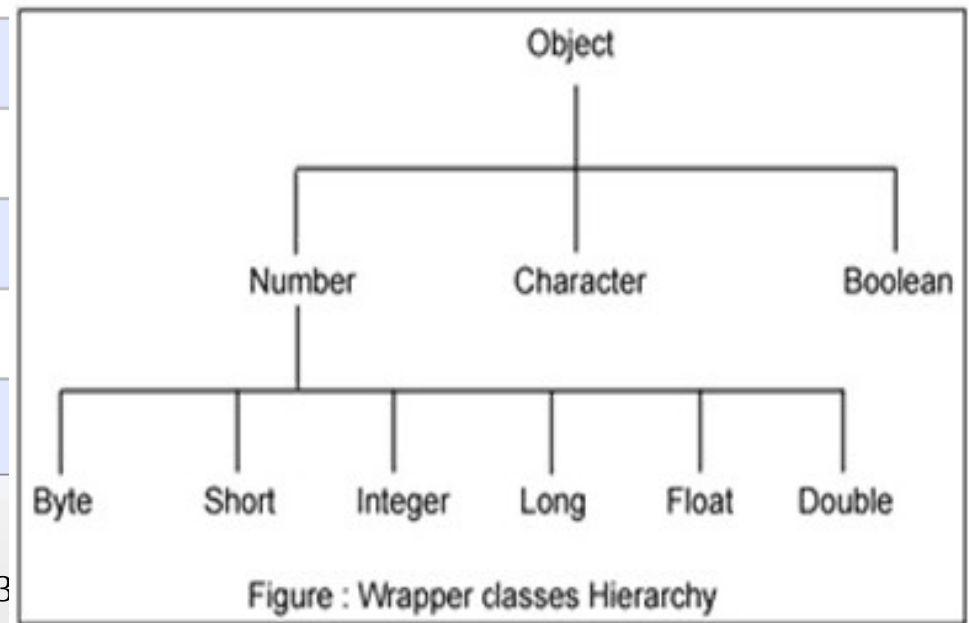
# Java Wrapper Classes

- Wrapper class in java provides the mechanism to convert primitive into object and object into primitive.
- For each primitive type, there is a corresponding wrapper class designed.
- An instance of wrapper contains or wraps a primitive value of the corresponding type.
- Wrappers allow for situations where primitives cannot be used but their corresponding objects are required.
- The eight classes of java.lang package are known as wrapper classes in java.
- The list of eight wrapper classes are given below:

# Java Wrapper Classes

## Wrapper Classes for Primitive Data Types

Primitive Data Types	Wrapper Classes
int	Integer
short	Short
long	Long
byte	Byte
float	Float
double	Double
char	Character
boolean	Boolean



# Java Wrapper Classes

- **Features of Wrapper Classes:**
  - All the wrapper classes except Character and Float have constructors. - one that takes the primitive value and another that takes the String representation of the value. Character has one constructor and float has three.
  - Just like strings, wrapper objects are also immutable. i.e once a value is assigned it cannot be changed.



# Wrapper classes

- The wrapper classes have a number of static methods for handling and manipulating primitive data types and objects.
- **Constructors:** converting primitive types to wrapper classes

```
Integer i = new Integer (10);
```

- **Methods:** converting wrapper objects to primitives

all numeric classes have methods to convert a numeric wrapper class to their respective primitive type.

byteValue(), intValue(), floatValue(), doubleValue(), longValue(), shortValue()

```
int v = a.intValue();
```

Class1.java

Class3.java

Class4.java

# Wrapper classes

- **Converting Primitives to String object:**

the method toString() is used to convert primitive number data types to String

```
String xyz = Integer.toString (v) //converting primitive integer to String
```

```
String xyz = Float.toString (x)
```

- **Converting Back from String Object to Primitives:**

the six parser methods are parseInt, parseDouble, parseFloat, parseLong, parseByte and parseShort.

```
int v = Integer.parseInt (xyz);
```

Class5.java

# Wrapper classes

- Wrapper classes are mainly used to wrap the primitive content into an object.
- This operation of wrapping primitive content into an object is called **boxing**.
- The reverse process i.e unwrapping the object into corresponding primitive data is called **Unboxing**.
- From JDK 1.5 onwards, **Auto-Boxing** is introduced. According to this feature, you need not to explicitly wrap the primitive content into an object. Just assign primitive data to corresponding wrapper class reference variable, **java automatically wraps primitive data into corresponding wrapper object.** Class6.java
- From JDK 1.5 onwards, **Auto-Unboxing** is introduced. According to this feature, you need not to call method of wrapper class to unbox the wrapper object. **Java implicitly converts wrapper object to corresponding primitive data if you assign wrapper object to primitive type variable.**

Class7.java

Unit - 3

Class8.java

# String class

- The `java.lang.String` class provides a lot of methods to work on string. By the help of these methods, we can perform operations on string such as **trimming, concatenating, converting, comparing, replacing strings etc.**
- Java String is a powerful concept because everything is treated as a string if you submit any form in window based, web based or mobile application.

# toUpperCase() and toLowerCase() method

- The java string `toUpperCase()` method converts this string into uppercase letter and string `toLowerCase()` method into lowercase letter.

Syntax:

`String toLowerCase()`

`String toUpperCase()`

Example:

```
String s="Sachin";
```

```
System.out.println(s.toUpperCase());//SACHIN
```

```
System.out.println(s.toLowerCase());//sachin
```

```
System.out.println(s);//Sachin(no change in original)
```

# String trim() method

- The string `trim()` method eliminates white spaces before and after string.

Syntax:

`String trim()`

Example:

```
String s=" Sachin ";
```

```
System.out.println(s);// Sachin
```

```
System.out.println(s.trim());//Sachin
```

String3.java

# String startsWith() and endsWith() method

- This method has two variants and tests if a string starts or ends with the specified prefix.

Syntax:

boolean startsWith(String prefix)

boolean endsWith(String prefix)

Example:

```
String s="Sachin";
```

```
System.out.println(s.startsWith("Sa")); //true
```

```
System.out.println(s.endsWith("n")); //true
```

String1.java

# String charAt() method

- The string `charAt()` method returns a character at specified index.

Syntax:

```
char charAt(int index)
```

Example:

```
String s="Sachin";
```

```
System.out.println(s.charAt(0));//S
```

```
System.out.println(s.charAt(3));//h
```



# String length() method

- The string **length() method returns length of the string**. The length is equal to the number of 16-bit Unicode characters in the string.

Syntax:

```
int length()
```

Example:

```
String s="Sachin";
```

```
System.out.println(s.length());//6
```

# String valueOf() method

- The string `valueOf()` method converts given type such as `int`, `long`, `float`, `double`, `boolean`, `char` and `char array` into string.

Syntax:

```
String valueOf(double d)
```

Example:

```
int a=10;
```

```
String s=String.valueOf(a);
```

```
System.out.println(s+10); //1010
```

# String replace() method

- The string `replace()` method replaces all occurrence of first sequence of character with second sequence of character.

Syntax:

`String replace(char oldChar, char newChar)`

Example:

```
String s1="Java is a programming language. Java is a  
platform. Java is an Island.";
```

```
String replaceString=s1.replace("Java","Kava");//replaces all  
occurrences of "Java" to "Kava"
```

String2.java

# String substr() method

- This method has two variants and returns a new string that is a substring of this string.
- The substring begins with the character at the specified index and extends to the end of this string or up to endIndex – 1, if the second argument is given.

## Syntax

```
public String substring(int beginIndex, int endIndex)
```

Example:

```
String Str = new String("Welcome to Java SY A");  
System.out.println(Str.substring(10, 13) ); //Java
```

# String compareTo() method

- This method compares two strings lexicographically.

Syntax

```
int compareTo(String anotherString)
```

The value 0 if the argument is a string equal to this string; a value less than 0 if the argument is a string greater than this string; and a value greater than 0 if the argument is a string less than this string.

```
String str1 = "Strings are immutable";  
String str2 = "Strings are immutable";  
String str3 = "Integers are not immutable";  
    int result = str1.compareTo( str2 ); //0  
System.out.println(result);  
result = str2.compareTo( str3 ); //10  
System.out.println(result);  
result = str3.compareTo( str1 ); //-10  
System.out.println(result);
```

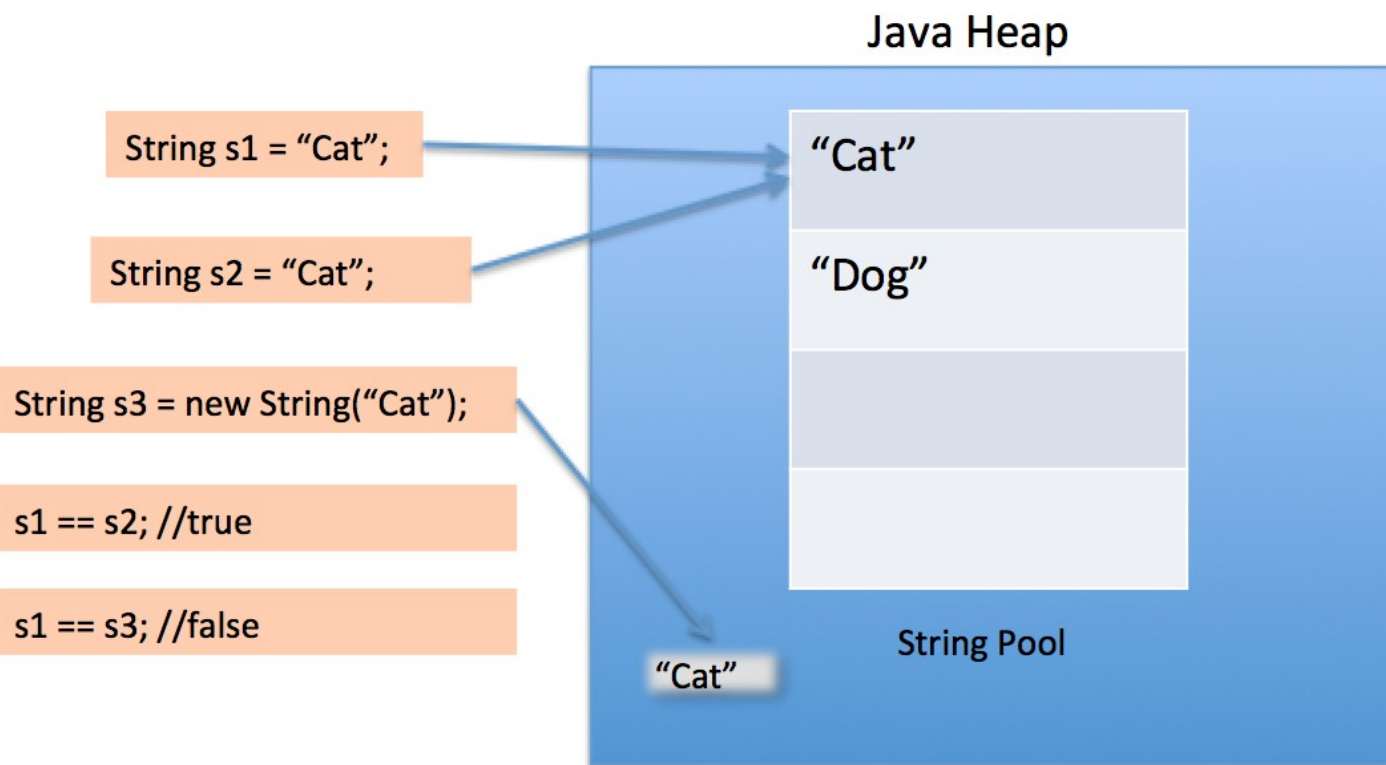
# String Buffer class

- The `java.lang.StringBuffer` class is a thread-safe, mutable sequence of characters.

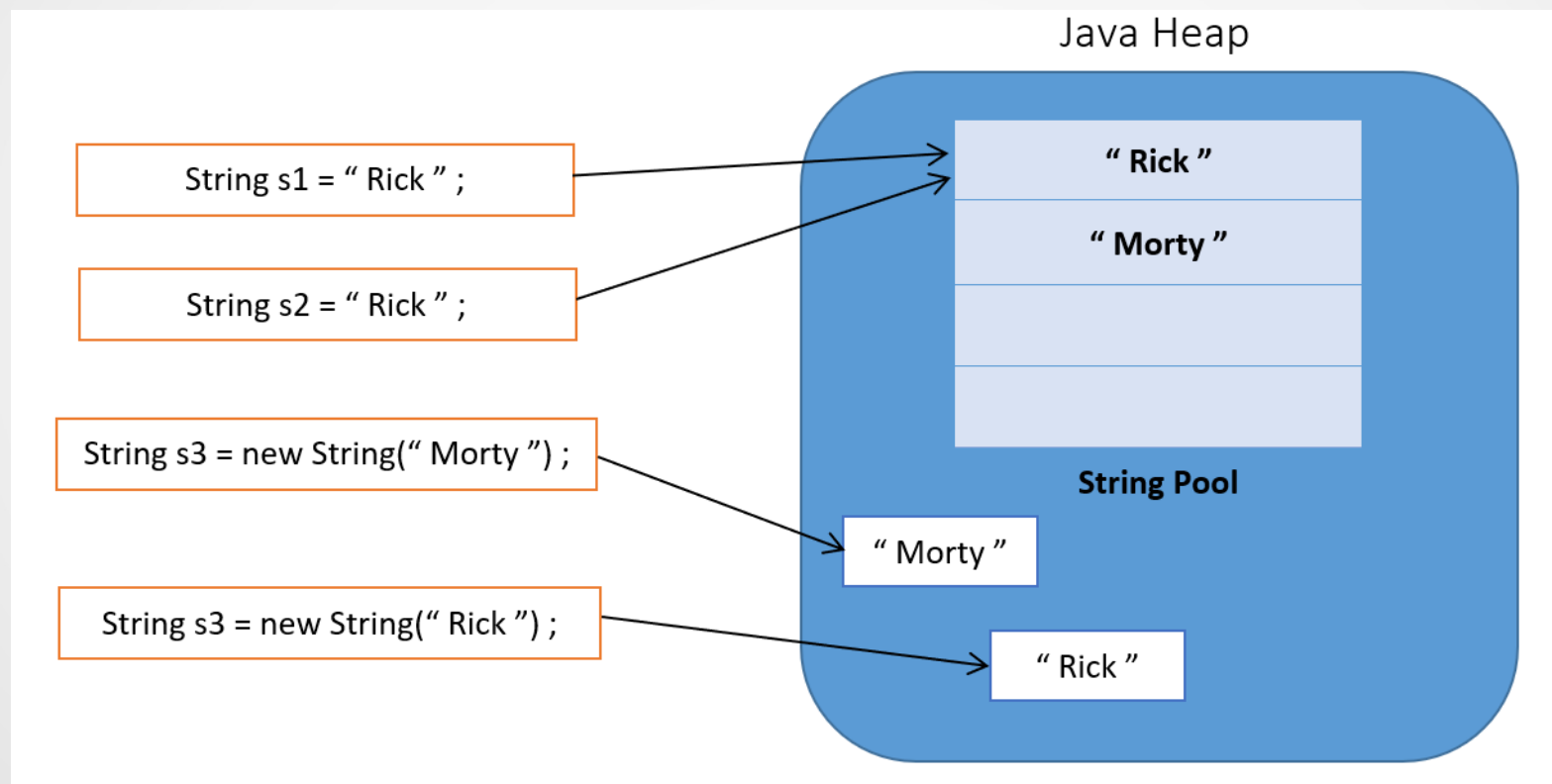
Following are the important points about StringBuffer:

- A string buffer is like a String, **but can be modified.**
- It contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.
- They are safe for use by multiple threads.
- **Every string buffer has a capacity.**

No.	String	StringBuffer
1)	The String class is immutable.	The StringBuffer class is mutable.
2)	String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when we concatenate t strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.
4)	String class is slower while performing concatenation operation.	StringBuffer class is faster while performing concatenation operation.
5)	String class uses String constant pool.	StringBuffer uses Heap memory







# StringBuffer.capacity() method

- The `java.lang.StringBuffer.capacity()` method returns the current capacity.
- The capacity is the amount of storage available for newly inserted characters, beyond which an allocation will occur.
- If the number of the character increases from its current capacity, it increases the capacity by  $(oldcapacity * 2) + 2$ .
- Syntax:

```
int capacity()
```

```
String s = new String("Hello");
```

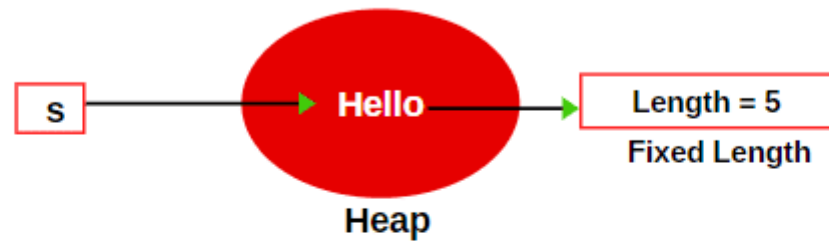


Fig 1: String object

```
StringBuffer sb = new StringBuffer("Hello");
```

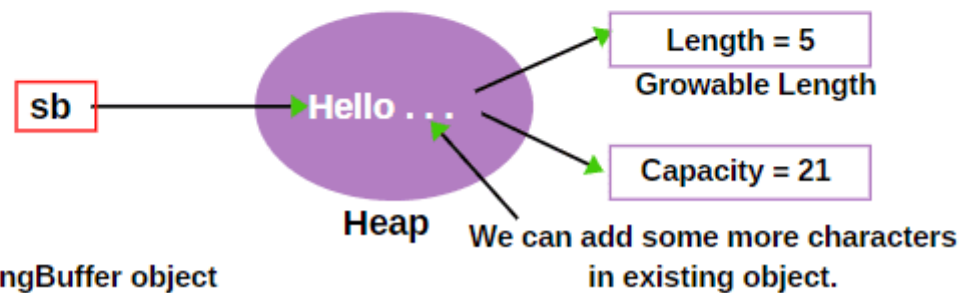


Fig 2: StringBuffer object

# StringBuffer.append()

- The `java.lang.StringBuffer.append(String str)` method appends the specified string to this character sequence.
- The characters of the String argument are appended, in order, increasing the length of this sequence by the length of the argument.
- If str is null, then the four characters "null" are appended.
- Syntax

`StringBuffer append(String str)`

# StringBuffer.replace()

- The `java.lang.StringBuffer.replace()` method replaces the characters in a substring of this sequence with characters in the specified String.
- The substring begins at the specified start and extends to the character at index end - 1 or to the end of the sequence if no such character exists.
- First the characters in the substring are removed and then the specified String is inserted at start.
- Syntax:

`StringBuffer replace(int start, int end, String str)`

# StringBuffer.reverse()

- The `java.lang.StringBuffer.reverse()` method causes this character sequence to be replaced by the reverse of the sequence.
- Syntax:  
`StringBuffer reverse()`

# StringBuffer.charAt()

- The `java.lang.StringBuffer.charAt()` method returns the char value in this sequence at the specified index.
- The first char value is at index 0, the next at index 1, and so on, as in array indexing.
- The index argument must be greater than or equal to 0, and less than the length of this sequence.
- Syntax:

`char charAt(int index)`

Access7.java

# StringBuffer.setCharAt()

- The `java.lang.StringBuffer.setCharAt()` method sets the character at the specified index to `ch`.
- This sequence is altered to represent a new character sequence that is identical to the old character sequence, except that it contains the character `ch` at position `index`.
- Syntax:

```
void setCharAt(int index, char ch)
```





# **UNIT 3 COMPLETED**