RAMAIAH
Institute of Technology

M.S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of Computer Science and Engineering

Course Name: Object Oriented Programming
Course Code: CS36
Credits: 3:0:0

Term: September – December 2020

Faculty:
Dr. Geetha J
Hanumantharaju R

RAMAIAH
Institute of Technology

Unit-3

Inheritance, Packages & Interfaces

Inheritance Basics, Creating a Multilevel Hierarchy, Using super,, When Constructors Are Called, Method Overriding, Dynamic Method Dispatch, Using Abstract Classes, Using final with Inheritance, Packages, Access Protection, Importing Packages, Interfaces, String and StringBuffer Handling.

Inheritance

- The process of obtaining properties of one entity in another entity is known as Inheritance.
- **“extends”** keyword is used for the purpose of inheritance.
- A class can only extend “**only one class**”.
- Inheritance can also be referred to the concept of **Generalization**, which is the process of extracting common characteristics (states and behavior) from two or more classes, and combining them into a generalized super class.

Inheritance

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- It is an important part of **OOPs.**
- create new **classes** that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use Inheritance?

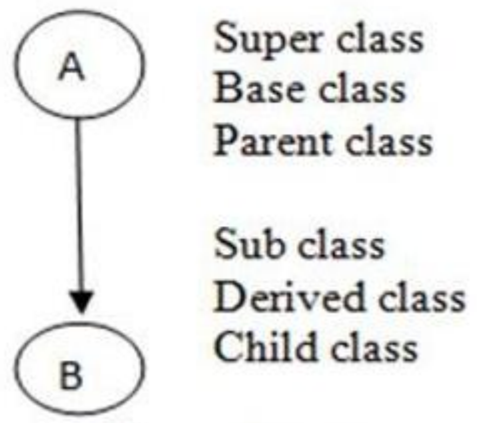
- For **Method Overriding** (so runtime polymorphism can be achieved).
- For **Code Reusability**.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

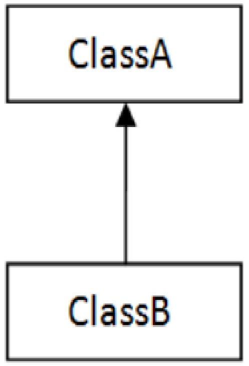
Types of Java Inheritance

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

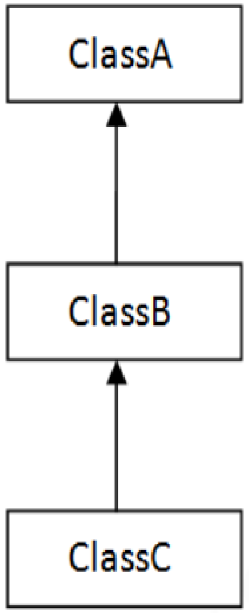


RAMAIAH Institute of Technology

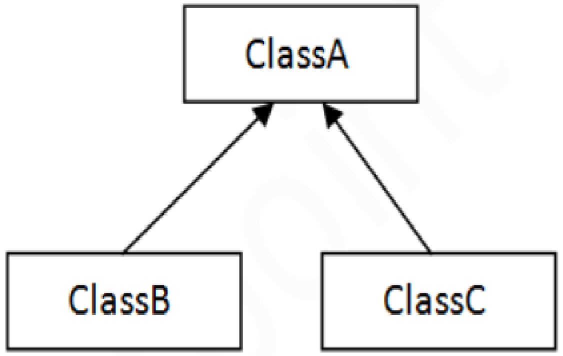
Syntax of Java Inheritance



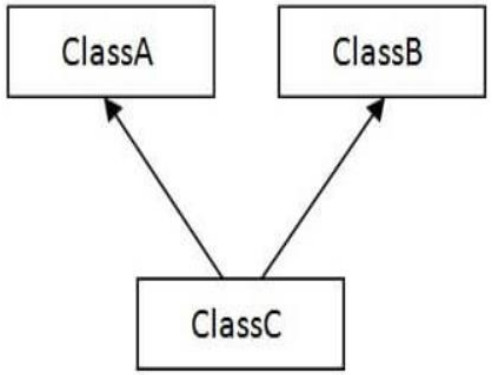
1) Single



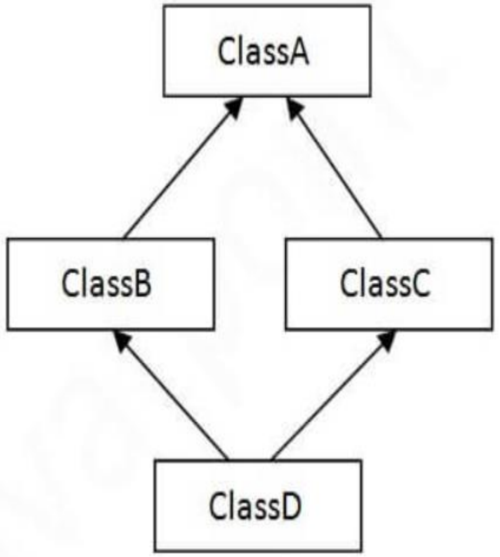
2) Multilevel



3) Hierarchical



4) Multiple



5) Hybrid

RAMAIAH
Institute of Technology

Examples of Inheritance

Single_Inheritance.java

Multilevel_Inheritance.java

Hierarchical_Inheritance.java

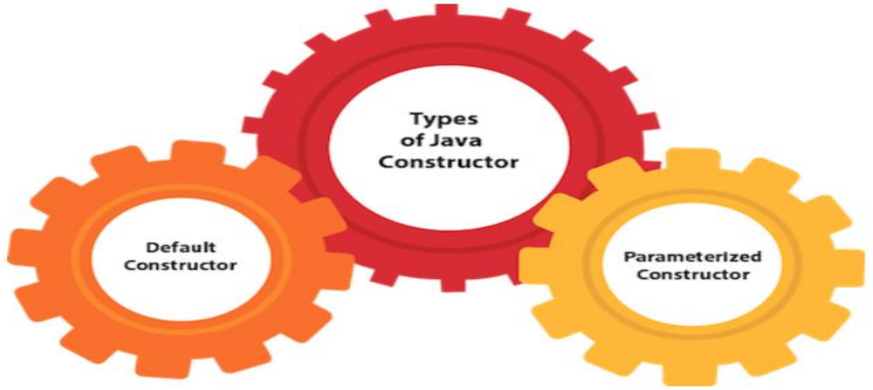
Why multiple inheritance not supported in java?

- To reduce the **complexity** and simplify the language
- 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 the same method and you call it from child class object, there will be ambiguity to call the 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.

Multiple_Inherit.java

RAMAIAH
Institute of Technology

Constructors in Java



- It is a special method will be **automatically called at the time of instantiating the class.**
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new() keyword, at least one constructor is called.
- **Rules:**
  - It must have same name as that of class name.
  - It doesn’t contain any return type even void also.
  - If we want we can provide access specifiers like default, protected and public other than private.
  - By using any parameterized constructor in our class, if we want to create instance for the class by using default constructor then we need to provide default constructor implementation explicitly in the class.

Constructor_Example.java

Order of execution of constructors in java

- Order of execution of constructors in inheritance relationship is from base /parent class to derived / child class.

Order_of_execution.java

RAMAIAH
Institute of Technology

Difference between constructor and method

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

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.

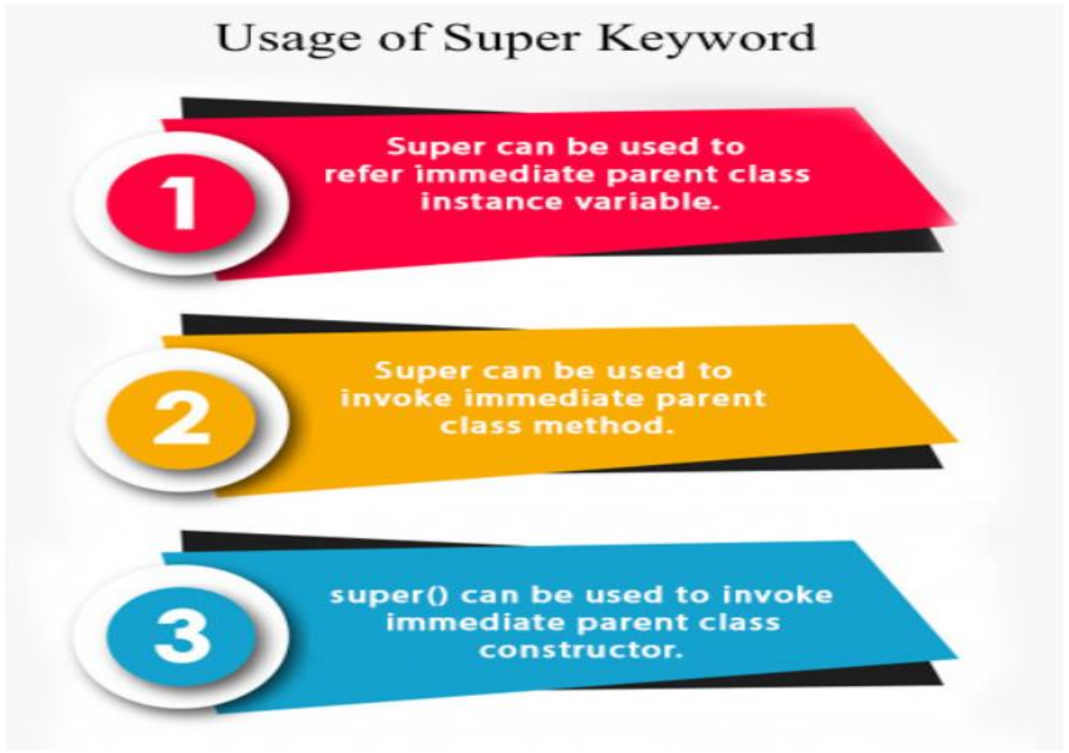
1. super is used to refer immediate parent class instance variable
2. super can be used to invoke parent class method
3. super is used to invoke parent class constructor

RAMAIAH
Institute of Technology

Super keyword in java

Super_Keyword.java

Super_employee.java


Usage of Super Keyword

1. Super can be used to refer immediate parent class instance variable.
2. Super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

Method Overloading

- Defining two or more methods in a same java class that share the same name but their parameter declarations are different.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.

Method_Overloading.java

Method Overloading

- When the exact match is not found, Java's automatic type conversion is used to find the match.

```
class OverloadDemo {
        void test(double a) {
                System.out.println("Inside test(double) a: " + a);
        }
}
class Overload {
        public static void main(String args[]) {
                OverloadDemo ob = new OverloadDemo();
                int i = 88;
                ob.test(i); // this will invoke test(double)
                ob.test(123.2); // this will invoke test(double)
        }
}
```

```
OUTPUT:
Inside test(double) a: 88
Inside test(double) a: 123.2
```

Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to ***override the method*** *in* the superclass.
- When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass.
- The version of the method defined by the superclass will be hidden.

Method Overriding

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

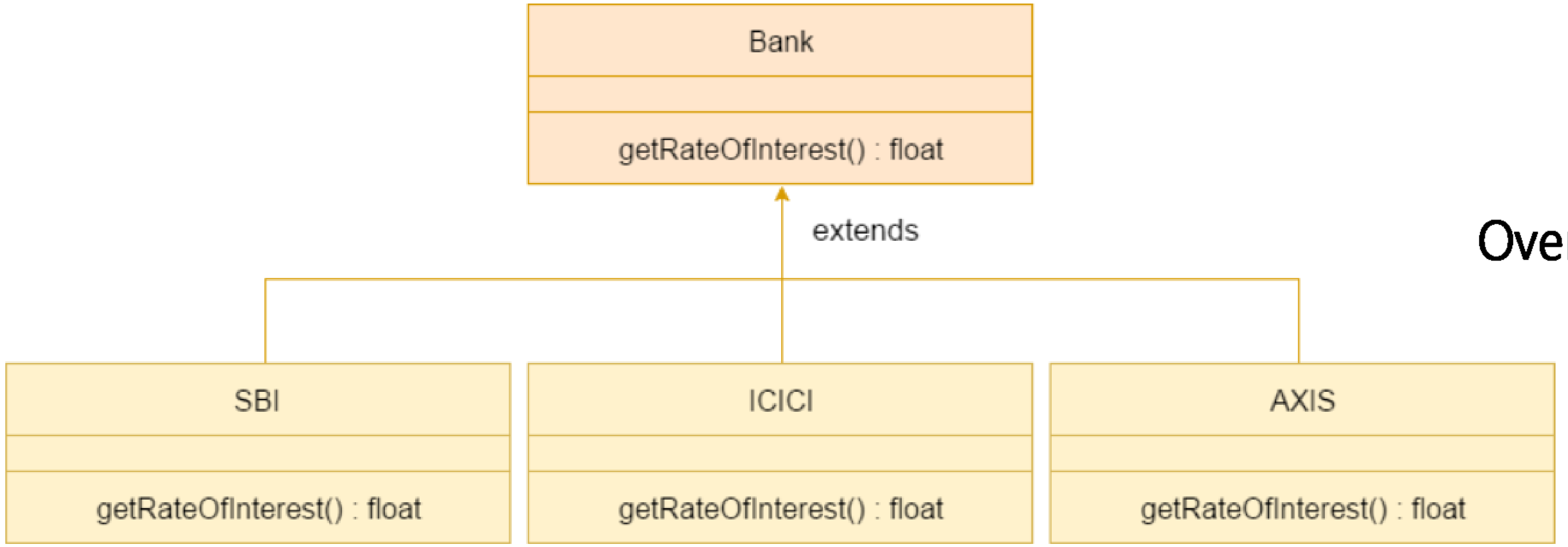
Overide.java

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

RAMAIAH
Institute of Technology

Method Overriding: Use case


Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

Overriding_Bank.java

Dynamic method dispatch

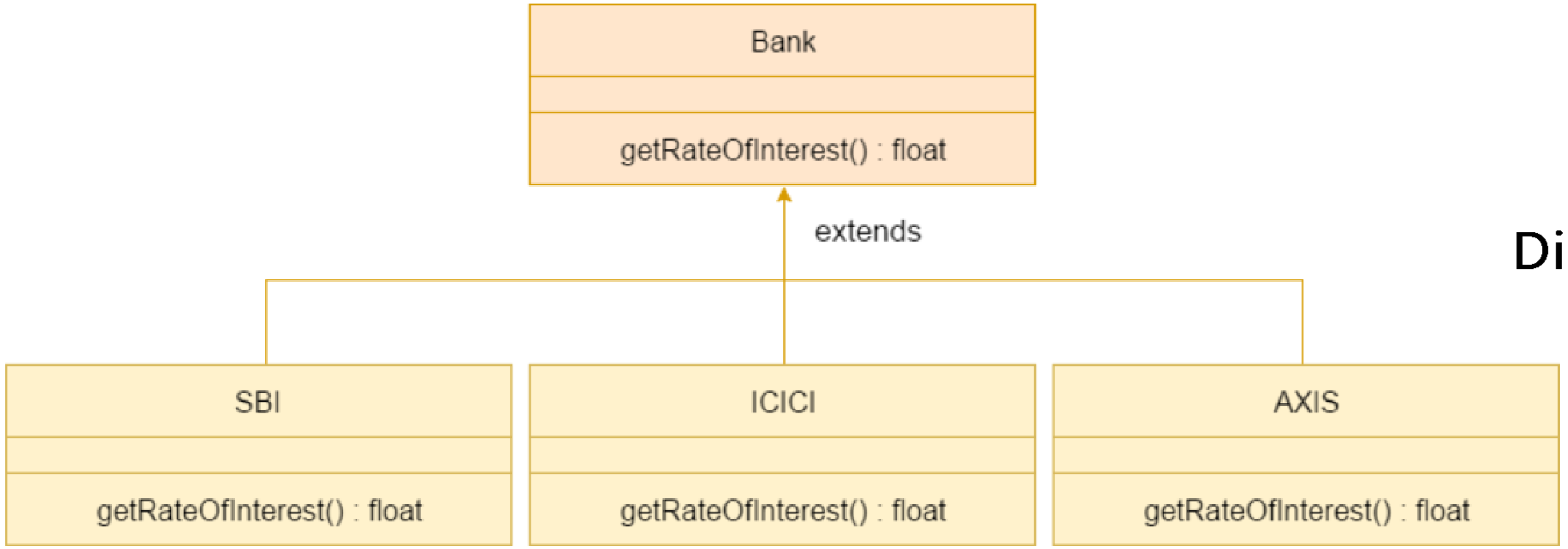
- Method overriding forms the basis for one of Java’s most powerful concepts: **Dynamic Method Dispatch.**
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Dynamic method dispatch

- Superclass reference variable can refer to a subclass object.
- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
- It is the type of the object being referred to that determines which version of an overridden method will be executed.

Dispatch.java

Dynamic method dispatch: Use case


Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

Dispatch_Bank.java

Using Abstract classes

- There are situations in which you will want to define a superclass that declares the structure of a given abstraction **without providing a complete implementation of every method.**
- **Create a superclass that only defines a generalized form** that will be **shared by all of its subclas**ses, leaving it to each subclass to fill in the details.
- Such a class determines the nature of the methods that the subclasses must implement.

RAMAIAH Institute of Technology

Using Abstract classes

- Any class that contains one or more abstract methods must also be declared abstract.
  - **abstract** class A{}
- There can be **no objects** of an abstract class.
- That is, an abstract class cannot be directly instantiated with the **new operator.**
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared **abstract itself.**

Abstract_Demo.java

Using Abstract classes

- Abstract classes **cannot be used to instantiate** objects.
- But, they can be used to create object references.
- Possible to create a reference to an abstract class so that it can be used to point to a subclass object.

Abstract_Areas.java

Test_Abstract_Bank.java

RAMAIAH
Institute of Technology

Using final with Inheritance

- **Using final to Prevent Overriding:**
- To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as final cannot be overridden.

```
class A {
  final void meth() {
    System.out.println("This is a final method.");
  }
}

class B extends A {
  void meth() { // ERROR! Can't override.
    System.out.println("Illegal!");
  }
}
```

Final_to_avoid_Inheritance.java

RAMAIAH
Institute of Technology

Using final with Inheritance

- Using final to Prevent Inheritance
- Sometimes you will want to prevent a class from being inherited.
- Declaring a class as final implicitly declares all of its methods as final, too.

```
final class A {
   //...
}

// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
   //...
}
```

Interfaces

- Using the keyword **interface**, you can fully abstract a class' interface from its implementation.
- Using interface, you can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, **but they lack instance variables.**
- Methods in interfaces are declared without any body and are by default **"public nonstatic and abstract"** in nature.

Interfaces

- All the variables present in the interface are by default **“public static and final”.**
- **“implements”** keyword is used for implementing interfaces to the class.
- **One class can implement any number of interfaces**.
- To implement an interface, **a class must implement all the methods of the interface.**
- If a class includes an interface but does not fully implement the methods required by that interface, then that class must be declared as **abstract.**

RAMAIAH Institute of Technology

Defining an interface

- An interface is defined much like a class. This is a simplified general form of an interface:

```
access interface name {
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);  type final-
        varname1 = value;
        type final-varname2 = value;
        //...
}
```

Implementing interfaces

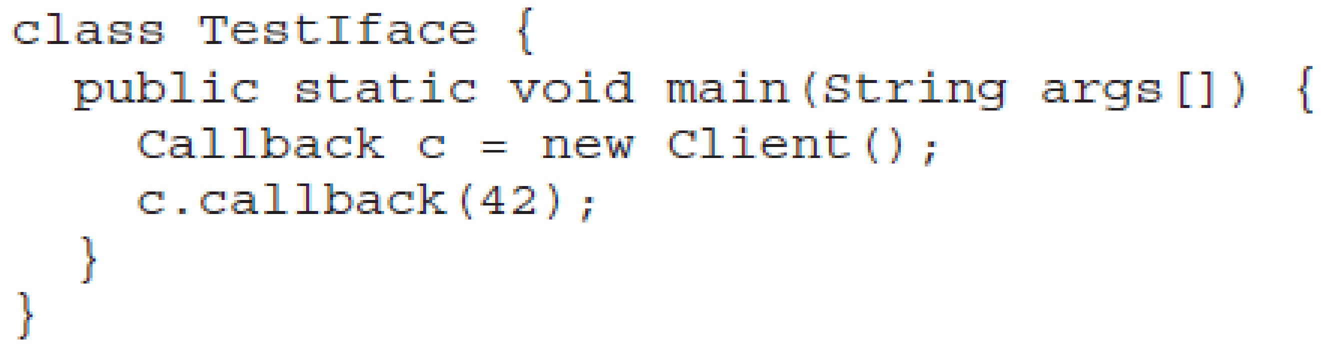
```
interface Callback {
  void callback(int param);
}
```

```
class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("callback called with " + p);
  }

  void nonIfaceMeth() {
    System.out.println("Classes that implement interfaces " +
                       "may also define other members, too.");
  }
}
```

Accessing Implementations Through Interface References

- You can declare variables as object references that use an interface rather than a class type.
- Any instance of any class that implements the declared interface can be referred to by such a variable.
- When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to.
- This is one of the key features of interfaces.

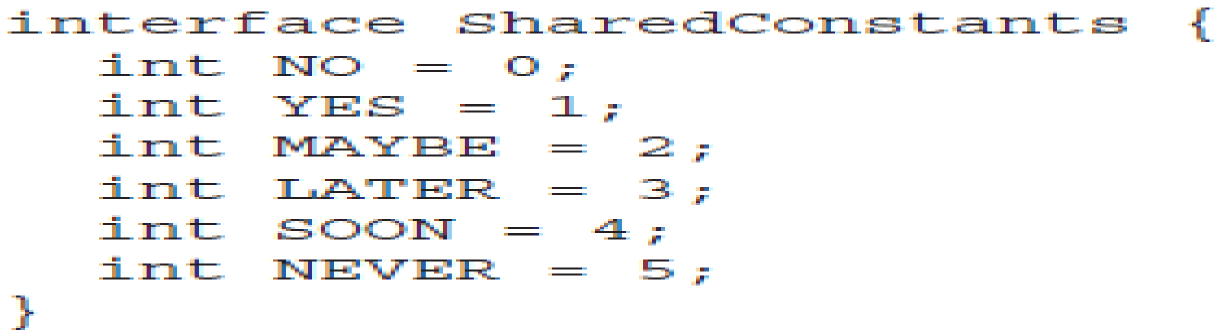
Accessing Implementations Through Interface References



```
class TestIface {
  public static void main(String args[]) {
    Callback c = new Client();
    c.callback(42);
  }
}
```

Variables in Interfaces

- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.
- When you include that interface in a class all of those variable names will be in scope as constants.



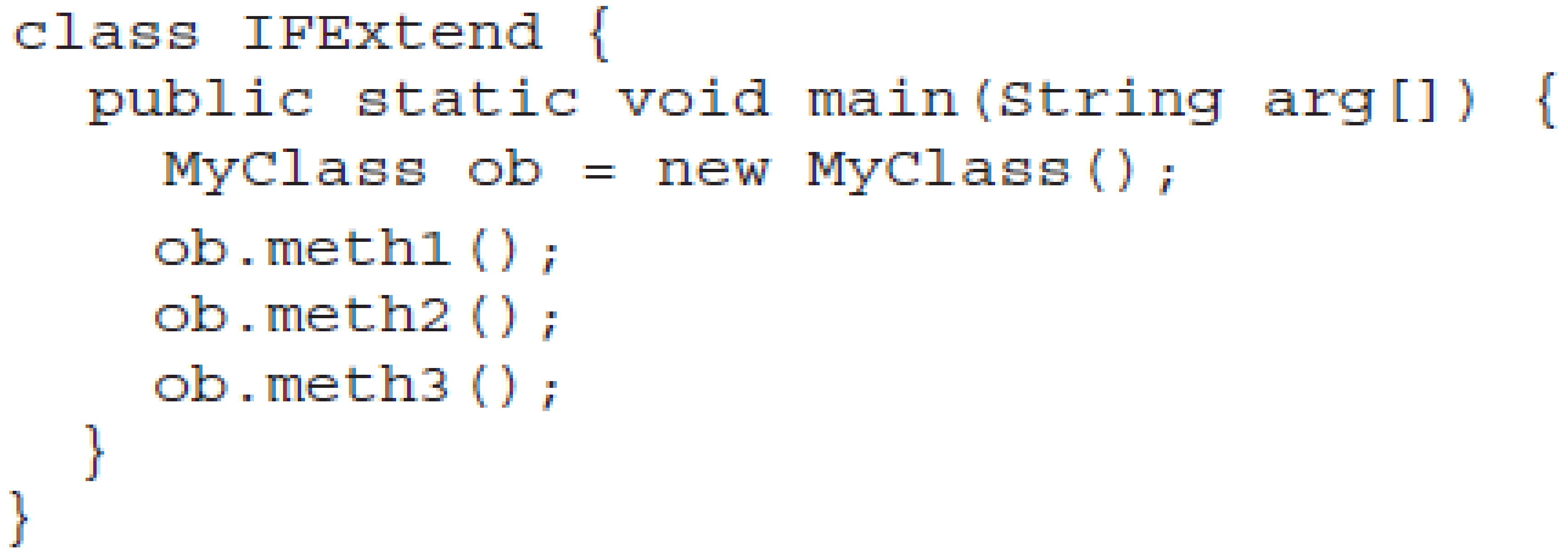
```
interface SharedConstants {
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;
}
```

Interfaces Can Be Extended

```
// One interface can extend another.
interface A {
  void meth1();
  void meth2();
}

// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
  void meth3();
}

// This class must implement all of A and B
class MyClass implements B {
  public void meth1() {
    System.out.println("Implement meth1().");
  }

  public void meth2() {
    System.out.println("Implement meth2().");
  }

  public void meth3() {
    System.out.println("Implement meth3().");
  }
}
```

RAMAIAH
Institute of Technology

Interfaces Can Be Extended

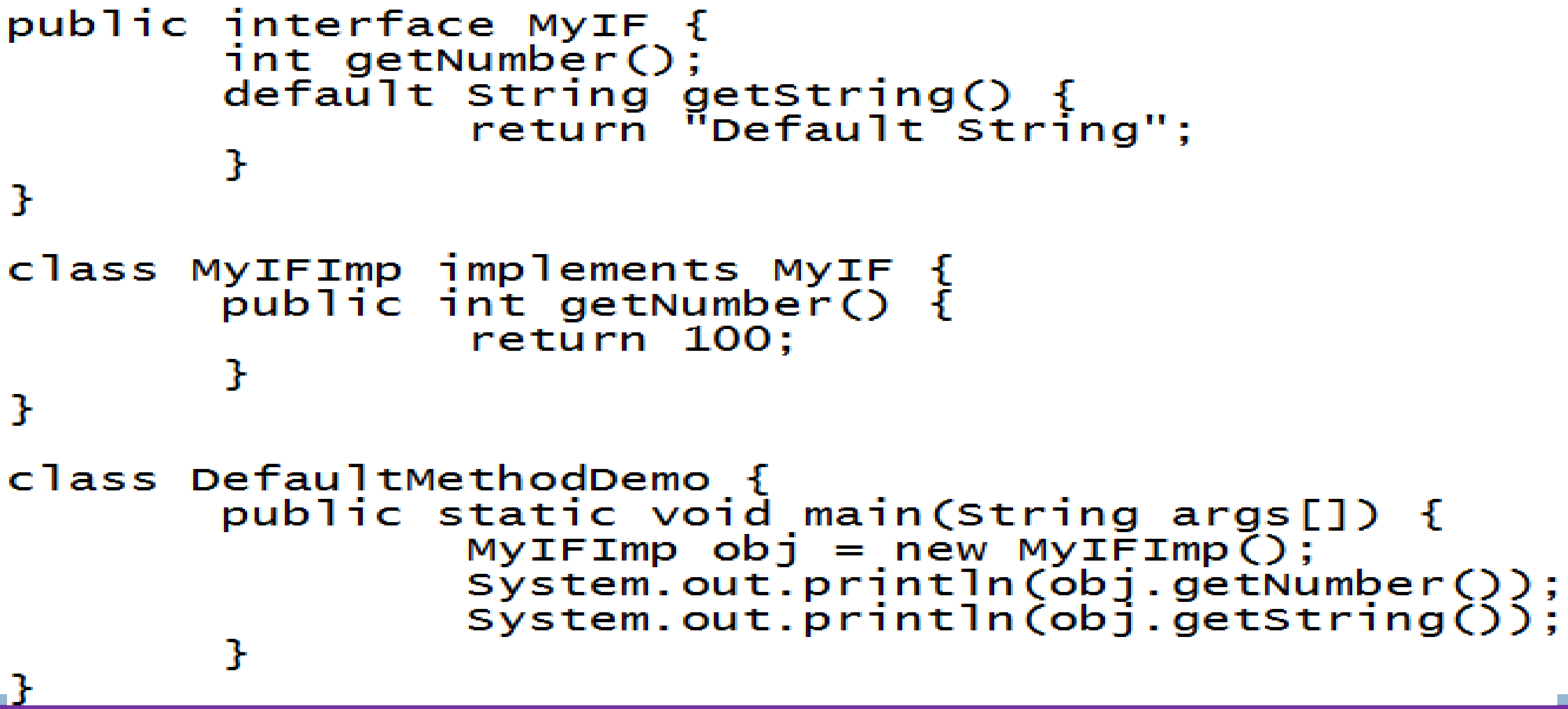


```
class IFExtend {
  public static void main(String arg[]) {
    MyClass ob = new MyClass();
    ob.meth1();
    ob.meth2();
    ob.meth3();
  }
}
```

Interfaces in Java 8

- By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.
- Java 8 allows the interfaces to have **default and static methods**.
- Allows the developers to add new methods to the interfaces without affecting the classes that implements these interfaces.
- **Static methods** in interfaces are similar to the default methods except that we cannot override these methods in the classes that implements these interfaces.

Interfaces in Java 8



```
public interface MyIF {
        int getNumber();
        default String getString() {
                return "Default String";
        }
}

class MyIFImp implements MyIF {
        public int getNumber() {
                return 100;
        }
}

class DefaultMethodDemo {
        public static void main(String args[]) {
                MyIFImp obj = new MyIFImp();
                System.out.println(obj.getNumber());
                System.out.println(obj.getString());
        }
}
```

RAMAIAH
Institute of Technology

Interfaces in Java 8

- JDK 8 supports the ability to define one or more static methods in interfaces.
- Like static methods in a class, a static method defined by an interface can be called independently of any object.
- No implementation of the interface is necessary.
- No instance of the interface is required.
- Instead, a static method is called by specifying the interface name:
  - **InterfaceName.staticMethodName**

Interfaces in Java 8

```
public interface MyIF {
        int getNumber();
        default String getString() {
                return "Default String";
        }
        static int getDefaultNumber() {
                return 0;
        }
}
```

- The getDefaultNumber( ) method can be called, as shown here:

int defNum = **MyIF.getDefaultNumber();**

RAMAIAH
Institute of Technology

Stack Interface Example

```
// Define an integer stack interface.
interface IntStack {
        void push(int item);
        int pop();
}
class FixedStack implements IntStack {
        public int stck[];
        public int tos;
        FixedStack(int size) {
                stck = new int[size];
                tos = -1;
        }
        public void push(int item) {
                if(tos==stck.length-1) // use length member
                        System.out.println("Stack is full.");
                else
                        stck[++tos] = item;
        }
        public int pop() {
                if(tos < 0) {
                        System.out.println("Stack underflow.");
                        return 0;
                }
                else
                        return stck[tos--];
        }
}
```

RAMAIAH Institute of Technology

Interface Example

```
public class IFTest {
      public static void main(String args[]) {
             FixedStack mystack1 = new FixedStack(5);
             for(int i=0; i<5; i++)
                    mystack1.push(i);
             System.out.println("Stack in mystack1:");
             for(int i=0; i<5; i++)
                    System.out.println(mystack1.pop());
      }
}
```

Interface Examples

Interface_Example.java

TestInterfaceStatic.java

TestInterfaceDefault.java

TestInterface4.java

TestInterface3.java

TestInterface2.java

TestInterface1.java

Multiple_Interface.java

Difference between Abstraction and interfaces

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

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

When to use abstract class

- ❑ An abstract class is a good choice if we are using the inheritance concept since it provides a common base class implementation to derived classes.
- ❑ An abstract class is also good if we want to declare non-public members. In an interface, all methods must be public.
- ❑ If we want to add new methods in the future, then an abstract class is a better choice. Because if we add new methods to an interface, then all of the classes that already implemented that interface will have to be changed to implement the new methods.

RAMAIAH
Institute of Technology

When to use abstract class

- If we want to create multiple versions of our component, create an abstract class. Abstract classes provide a simple and easy way to version our components. By updating the base class, all inheriting classes are automatically updated with the change. Interfaces, on the other hand, cannot be changed once created. If a new version of an interface is required, we must create a whole new interface.
- Abstract classes have the advantage of allowing better forward compatibility. Once clients use an interface, we cannot change it; if they use an abstract class, we can still add behavior without breaking the existing code.
- If we want to provide common, implemented functionality among all implementations of our component, use an abstract class. Abstract classes allow us to partially implement our class, whereas interfaces contain no implementation for any members.

When to use abstract class

```
abstract class Car {
  public void accelerate() {
    System.out.println("Do something to accelerate");
  }
  public void applyBrakes() {
    System.out.println("Do something to apply brakes");
  }
  public abstract void changeGears();
}
```

```
class Alto extends Car {
  public void changeGears() {
    System.out.println("Implement changeGears()
method for Alto Car");
  }
}
class Santro extends Car {
  public void changeGears() {
    System.out.println("Implement changeGears()
method for Santro Car");
  }
}
```

Now, any Car that wants to be instantiated must implement the changeGears () method.

When to use interface

- If the functionality we are creating will be useful across a wide range of disparate objects, use an interface. Abstract classes should be used primarily for objects that are closely related, whereas interfaces are best suited for providing a common functionality to unrelated classes.
- Interfaces are a good choice when we think that the API will not change for a while.
- Interfaces are also good when we want to have something similar to multiple inheritances since we can implement multiple interfaces.
- If we are designing small, concise bits of functionality, use interfaces. If we are designing large functional units, use an abstract class.

When to use interface

```
public interface Actor
{
  void perform();
}
public interface Producer
{
  void invest();
}
```

```
public interface ActorProducer extends Actor, Producer
{
  // some statements
}
```

Nowadays most of the actors are rich enough to produce their own movie. If we are using interfaces rather than abstract classes, we can implement both Actor and Producer. Also, we can define a new ActorProducer interface that extends both.

https://www.mindstick.com/Articles/12100/interfaces-in-java-real-life-example

RAMAIAH
Institute of Technology

Packages

- Packages are containers for classes.
- You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers.
- The package is both a naming and a visibility control mechanism.

Defining a Package

- To create a package is quite easy: simply include a package command as the first statement in a Java source file.
- The package statement defines a name space in which classes are stored.
- If you omit the package statement, the class names are put into the default package, which has no name.
- This is the general form of the package statement:

  package pkg;

Defining a Package

- Java uses file system directories to store packages.
- For example, the .class files for any classes you declare to be part of MyPackage must be stored in a directory called MyPackage.
- More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong.
- You can create a hierarchy of packages.

RAMAIAH
Institute of Technology

Defining a Package

- To do so, simply separate each package name from the one above it by use of a period.
- The general form of a multileveled package statement is shown here:
- **package pkg1[.pkg2[.pkg3]];**
- A package hierarchy must be reflected in the file system of your Java development system.
- For example, a package declared as **package java.awt.image;** needs to be stored in **java\awt\image** in a Windows environment.

Advantages of Packages

- Advantages of using a package in Java
  - Reusability
  - Better Organization
  - No Name Conflicts
  - Access Protection

RAMAIAH
Institute of Technology

Importing a Packages

- import *pkg1* [.*pkg2*].(*classname* | *);
- Once imported, a class can be referred to directly, using only its name.
- **Note: Import statement is optional.**
- Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy

RAMAIAH
Institute of Technology

Importing a Packages

```
import java.util.*;
class MyDate extends Date
{
}
```

- The same example without the **import** statement looks like this:

class MyDate **extends java.util.Date**
{
}

RAMAIAH
Institute of Technology

Importing a Packages

- All of the standard Java classes included with Java are stored in a package called **java**.
- The basic language functions are stored in a package inside of the **java** package called **java.lang**.
- The package **java.lang**, it is implicitly imported by the compiler for all programs

Package Example

```
package MyPack;

public class Balance {
        String name;
        double bal;

        public Balance(String n, double b) {
                name = n;
                bal = b;
        }

        public void show() {
                if(bal<0)
                        System.out.print("--> ");
                System.out.println(name + ": $" + bal);
        }
}
```

Balance.java

Package Example

```
import MyPack.*;

class TestBalance {
        public static void main(String args[]) {
                Balance test = new Balance("ABC", 99.88);
                test.show();
        }
}
```

TestBalance.java

Package Example

- Call this file **Balance.java and put it in a directory called MyPack.**
- Next, compile the file.
- Make sure that the resulting **.class file is also in the MyPack** directory.
- Compile and Run **TestBalance.java** file.
- javac **TestBalance.java**
- java **TestBalance**

RAMAIAH
Institute of Technology

Access Protection

- Java addresses four categories of visibility for class members:
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses
- The three access modifiers, **private, public, and protected**.

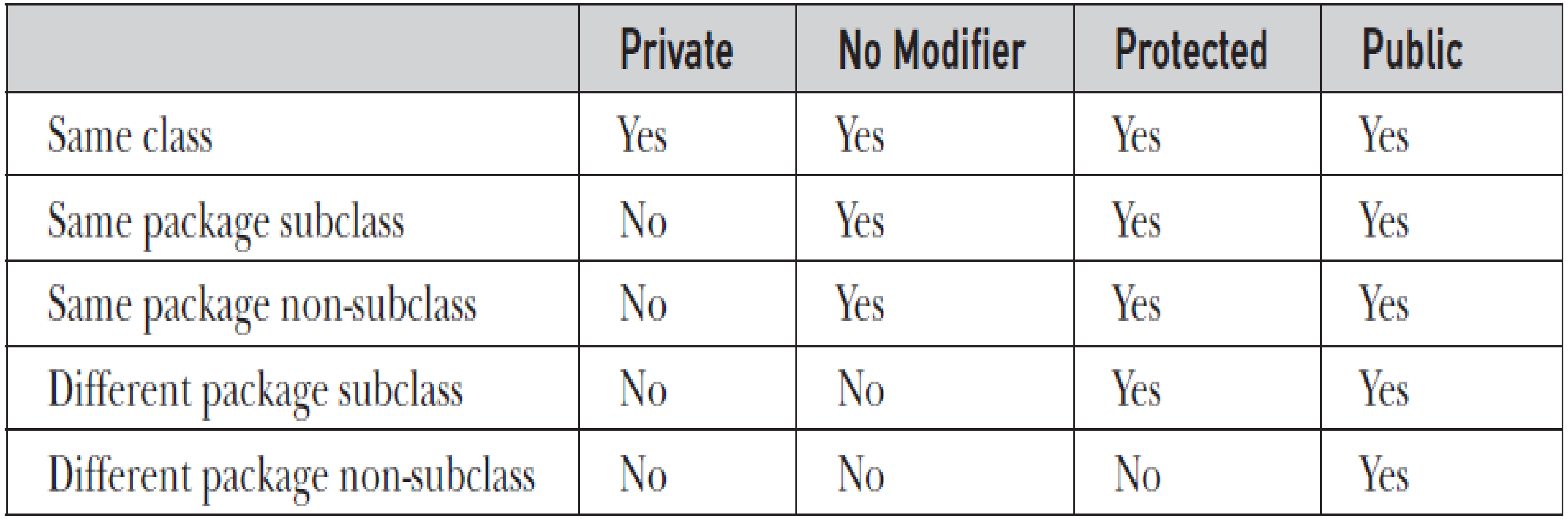
RAMAIAH
Institute of Technology

Access Protection

- Anything declared public can be accessed from anywhere.
- Anything declared private cannot be seen outside of its class.
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.
- If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.

RAMAIAH
Institute of Technology

Access Protection



	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Strings in java

- String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.
- Strings are **immutable**

How to create a string object?

- There are two ways to create String object:
  1. By string literal
  2. By new keyword

Creating string using String Literal

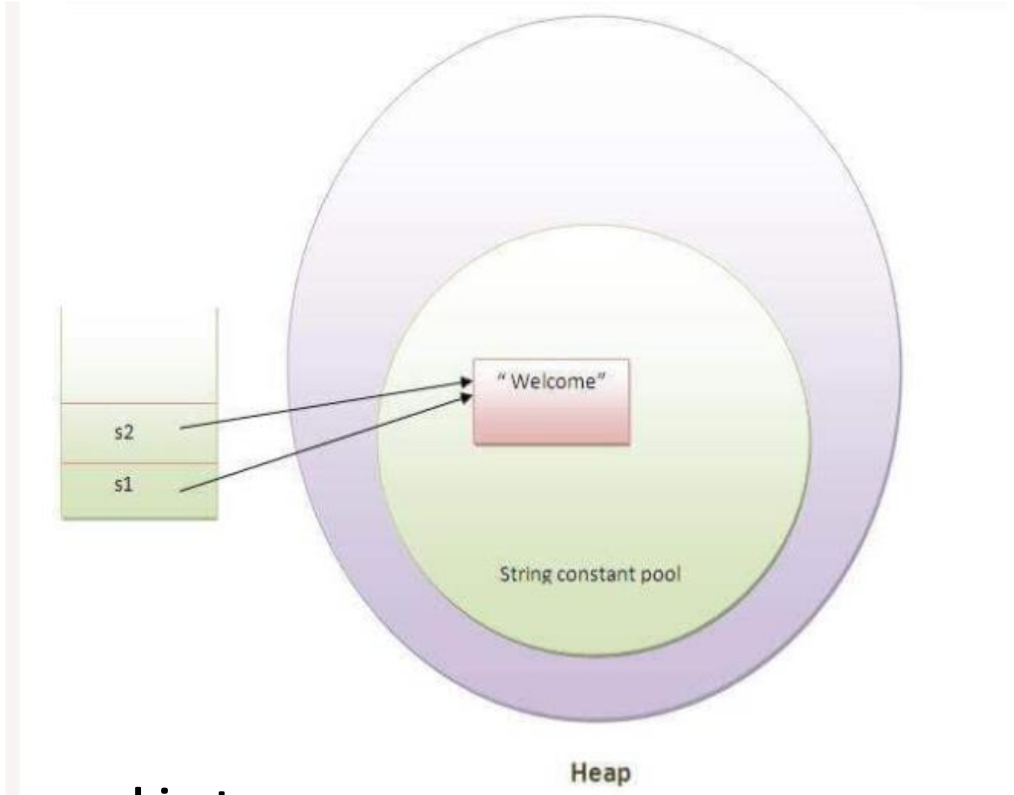
- Java String literal is created by using double quotes. For Example:

String s="welcome";

- Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

String s1="Welcome";

String s2="Welcome";//It doesn't create a new instance



To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

RAMAIAH
Institute of Technology

Creating string using new keyword

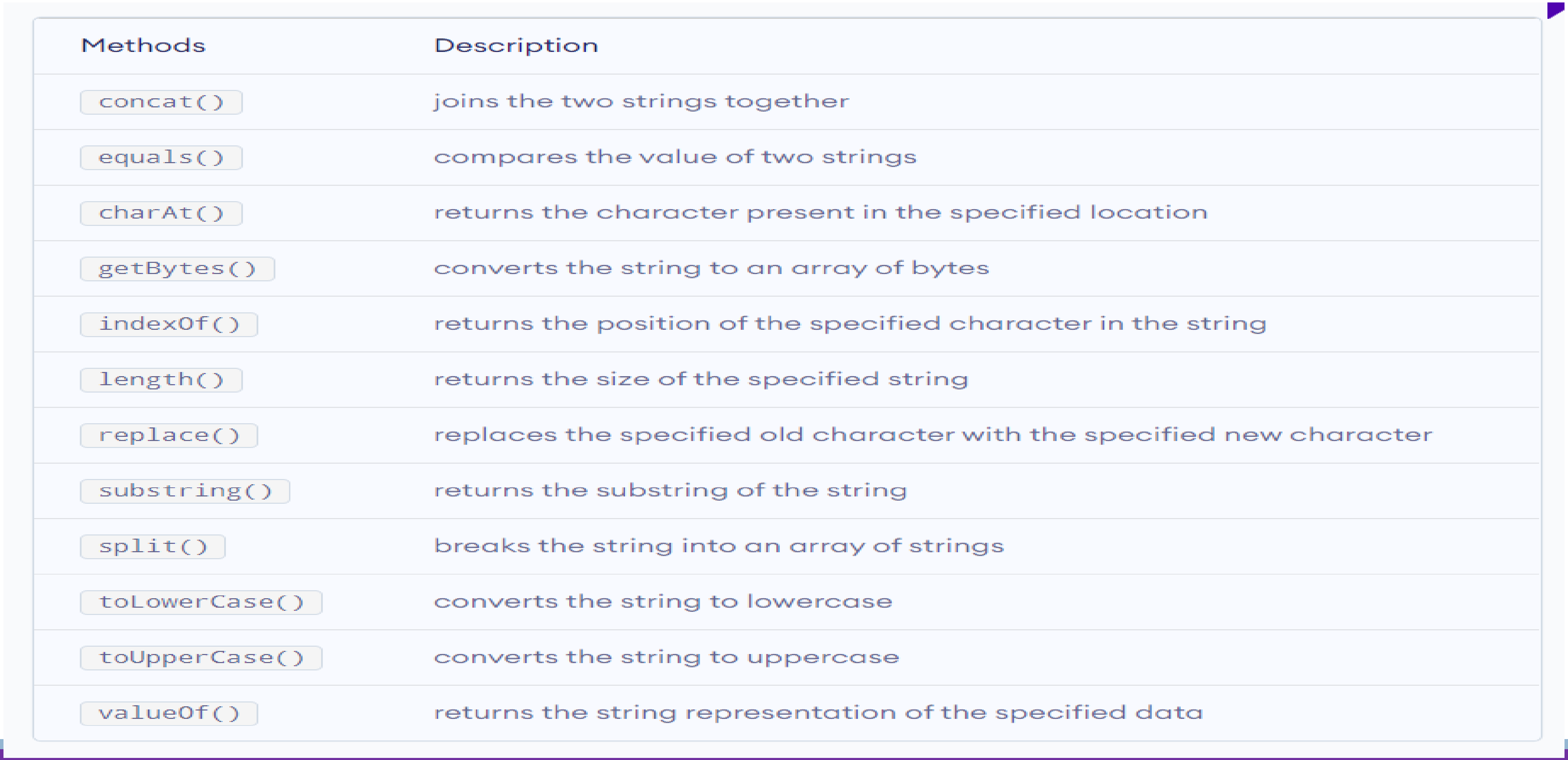
String s=new String("Welcome");//creates two objects and one reference variable

In such case, **JVM** will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

StringExample.java

String_Methods_Example.java

String methods



Methods	Description
`concat()`	joins the two strings together
`equals()`	compares the value of two strings
`charAt()`	returns the character present in the specified location
`getBytes()`	converts the string to an array of bytes
`indexOf()`	returns the position of the specified character in the string
`length()`	returns the size of the specified string
`replace()`	replaces the specified old character with the specified new character
`substring()`	returns the substring of the string
`split()`	breaks the string into an array of strings
`toLowerCase()`	converts the string to lowercase
`toUpperCase()`	converts the string to uppercase
`valueOf()`	returns the string representation of the specified data

RAMAIAH
Institute of Technology

StringBuffer class

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

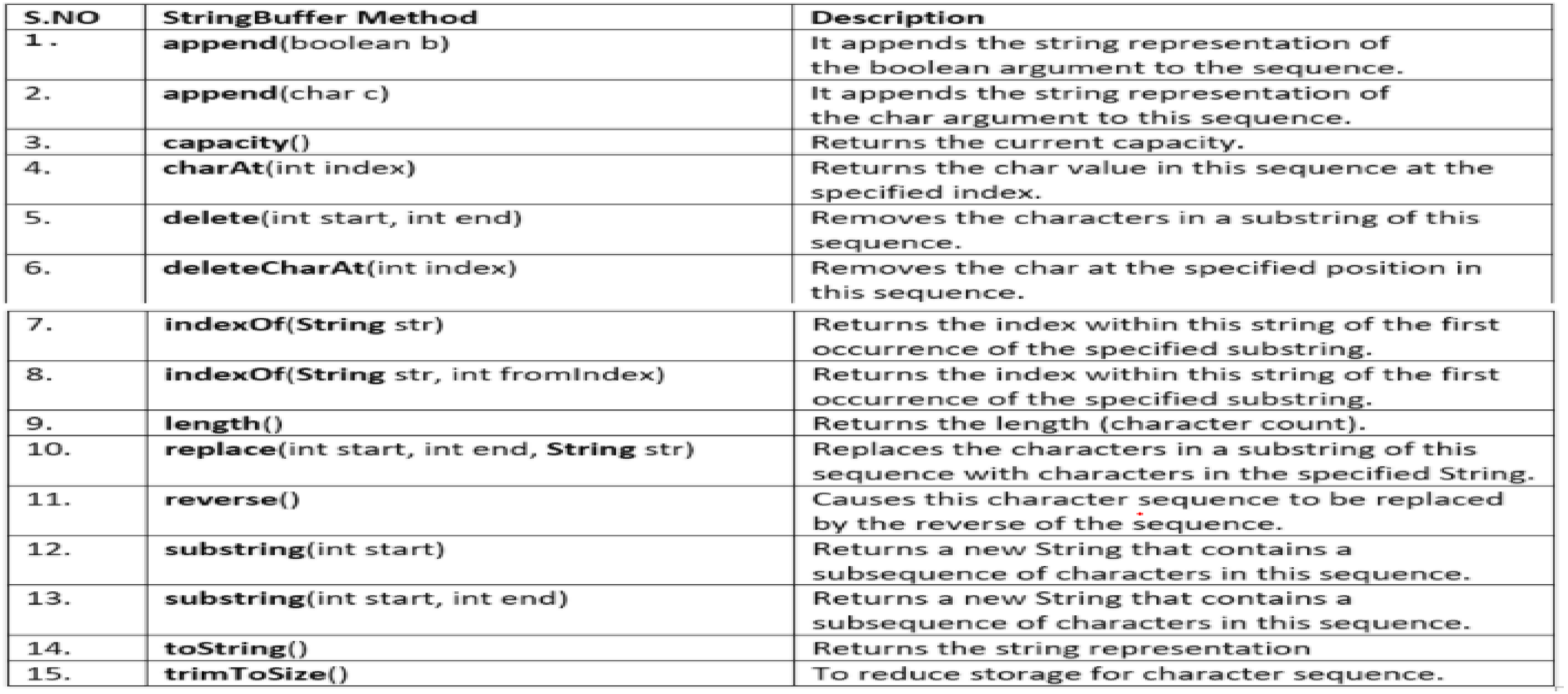
Important Constructors of StringBuffer class

StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

RAMAIAH
Institute of Technology

Methods of StringBuffer class

StringBufferExample.java



S.NO	StringBuffer Method	Description
1 .	**append**(boolean b)	It appends the string representation of the boolean argument to the sequence.
2.	**append**(char c)	It appends the string representation of the char argument to this sequence.
3.	**capacity**()	Returns the current capacity.
4.	**charAt**(int index)	Returns the char value in this sequence at the specified index.
5.	**delete**(int start, int end)	Removes the characters in a substring of this sequence.
6.	**deleteCharAt**(int index)	Removes the char at the specified position in this sequence.
7.	**indexOf**(**String** str)	Returns the index within this string of the first occurrence of the specified substring.
8.	**indexOf**(**String** str, int fromIndex)	Returns the index within this string of the first occurrence of the specified substring.
9.	**length**()	Returns the length (character count).
10.	**replace**(int start, int end, **String** str)	Replaces the characters in a substring of this sequence with characters in the specified String.
11.	**reverse**()	Causes this character sequence to be replaced by the reverse of the sequence.
12.	**substring**(int start)	Returns a new String that contains a subsequence of characters in this sequence.
13.	**substring**(int start, int end)	Returns a new String that contains a subsequence of characters in this sequence.
14.	**toString**()	Returns the string representation
15.	**trimToSize**()	To reduce storage for character sequence.

RAMAIAH
Institute of Technology

Thank you