Core Java

Session2 By Saurabh Sharma

Agenda

In this session will cover the following two modules:

- Inheritance and Polymorphism.
- Classes in Java.

Module 1: Objectives

After completion of this module, you should be able to:

- Understand inheritance in Java.
- Understand inheriting classes.
- Define overriding methods.
- Explain interfaces and methods.
- Explain abstract classes and methods.

Understand Inheritance

- **Inheritance defines relationship among classes, wherein one class shares structure or behavior defined in one or more classes by Grady Booch.**
- The ability of a class of objects to inherit the properties and methods from another class, called its parent or super or base class.
- The class that inherits the data members and methods is known as subclass or child class.
- The class from which the subclass inherits is known as base / parent class.
- In Java, a class can only **extend** one parent.

Understand Inheritance

- Object Oriented Languages also implements reuse in the same way that we do in real life.
- Using
  - has-a.
  - is-a.
- Has-a or composition relationship is implemented by having a class having another class as its member, or rather an object having another object as its member.
  - **class Car{ Stereo s; …}**
  - **class College { Teacher[] ts; Student ss[]; … }**
- Is-a is implemented through what we call ***inheritance*** relationship .

Understand inheritance with example scenarios .

- **Student** and **Teacher** are **Person**. **Person** can be a super class and **Student** and **Teacher** can be subclass of **Person** class. **Student** is-a **Person**, **Teacher** is-a **Person.**
- **HOD** is-a **Teacher. Since Teacher** is-a **Person, HOD** is also a **Person.**
- **Theory and Lab are ClassRoomSession.**
- **SeminarHall** is a **ClassRoom.**

Inheriting classes.

- A Java class inherits another class using the extends keyword after the class name followed by the parents class name as below.

  Example:- public class childclassname extends superclassname

- The child class inherits all the instance variables and methods of the parent class.

No Multiple Inheritance In Java for classes

Example: Inheriting classes

Parent class

```
public class Person
{
        private String name;
        public void setName(String name)
        {
            this.name=name;
        }
        public String getName()
        {
            return name;
        }
}
```

Child class

The extends keyword indicates inheritance

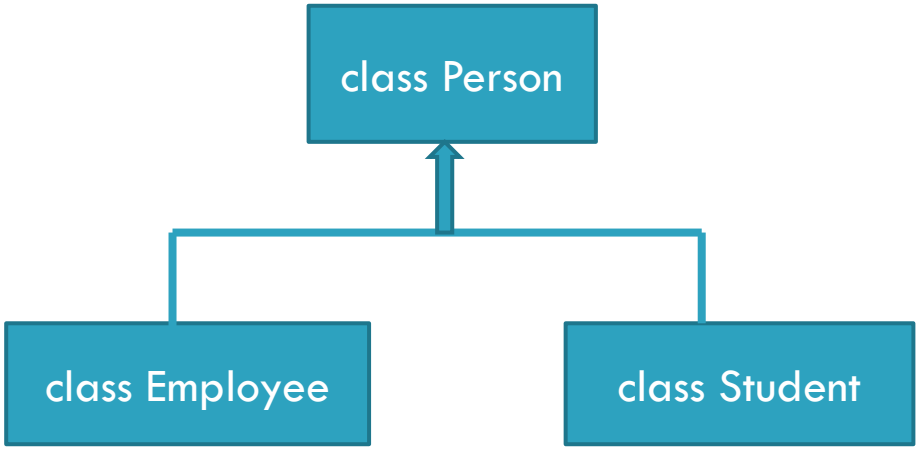
```
public class Student extends Person
{
        private int sid;
        public void setSid(int sid)
        {
            this.sid=sid;
        }
        public int getSid()
        {
            return sid;
        }
}
```

Members that are not accessible through inheritance

- All the features that are there in super class are there in subclass as well. But there are some restrictions with respect to what super class features (members) can be accessed from subclass.
- Super class members that cannot be accessed from subclass
  - **private** members .
  - Default members (if subclasses are in the different packages).
- For previous example, **Student** class cannot access **name** of the **Person** class since they are **private** but can access all of the methods since they are **public.**

Types of inheritance

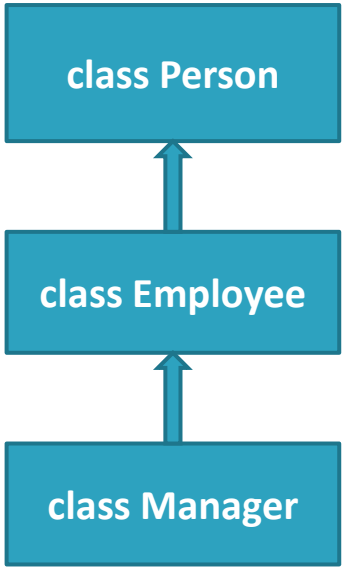
- Single Level Inheritance → Derives a subclass from a single superclass.



Types of inheritance(Continued)

- Multilevel Inheritance → Inherits the properties of another subclass.

Example:



super()

- Keyword **super() or super(< with parameters>)** is used to call the super class constructor.
- **super()** (like **this()**) can be called only from the constructor.
- It must be the first statement of the constructor.
- Which also implies that **super()** and **this()** cannot be used together because both must be first statement of constructor.
- Compiler inserts a **super()** statement in all constructor if subclass constructor does not explicitly call some form of **super()** that calls super class constructor.
- This is to ensure initialization of super class members because they are also part of subclass.

Example of super

- Assume that there is only a parameter constructor in class **Person** which is taking one parameter **name** type of **String** and **Student** is a child class of **Person class** then inside **Student** class we must have to override one constructor.
- Example:

```
1. public class Student extends Person
2. {
3.         public Student()
4.         {
5.                         super("Mark");   //Use for calling superclass constructor
6.         }
7. }
```

- For classes where super class does not have no-argument constructor, all subclass constructors must explicitly call appropriate super class constructor.
- **Note→** In this example calling super("Mark") becomes compulsory from constructor of Student class since only one constructor is defined in Person class which takes one argument.

Calling super class methods

- The **super** keyword can also be used to invoke super class methods from the subclass method.

  super.getName()

- This becomes necessary only when subclass has redefined the method in super class.

More on this in overriding session

Initializers

- Initializers are blocks of code used to initialize member variables.
  a) Non-Static Initializers
     - Used to initialize instance variables.
     - Invoked every time object is created.
     - Syntax
       - **{ <<statements>>}** **//Line 1**
  b) Static Initializers
     - Used to initialize static variables.
     - Invoked once when the class is loaded.
     - Syntax
       - **static { <<statements>>}** **//Line2**

Why we need initializers?

- The declarations and initialization of fields can be done in same line like this: **int var=1;**
- Initializers are required for initializations that require a set of java statements for computing the initialization variable.
- For instance if the initial value is to be read from a file, then the set of file statements can be put in the initialization block. Another place where this may be required in the use of for-loop for initializing arrays.
- The Compiler copies instance initializer block into every constructor. Therefore, this can be used to share a block of code between multiple constructors.
- The static initializer is the only place to initialize static fields in cases where initialization exceeds more than a statement.

Example of initialization block

```
1. public class Person
2. {
3.        public Person()
4.        {
5.             System.out.println("Constrictor!!!!!!");
6.        }
7.
8.        {
9.             System.out.println("Non-Static Initialization
Block");
10.      }
11.       static
12.       {
13.            System.out.println("Static Initialization Block");
14.       }
```

```
15.  public  static void main(String as[])
16.  {
17.          Person p=new Person();
18.           Person p1=new Person();
19.       }
20.}
```

Output:-

```
Static Initialization Block
Non-Static Initialization Block
Constrictor!!!!!!
Non-Static Initialization Block
Constrictor!!!!!!
```

Order of initializations when subclass instance is created

1. Static initializations of super class: Static variables are initialized and static blocks are executed in the order of their appearance in the code.
2. Static initializations of subclass class : Static variables are initialized and static blocks are executed in the order of their appearance in the code.
3. Instance initializations of super class : Instance variables are initialized and instance blocks are executed in the order of their appearance in the code.
4. Super class constructor is executed.
5. Instance initializations of subclass class : Instance variables are initialized and instance blocks are executed in the order of their appearance in the code.
6. Subclass class constructor is executed.

Only after the super class part, sub class part happens

Example: Order of initializations block in inheritance.

```
public class Person
{
        public Person()
        {
            System.out.println(“Constrictor Person”);
        }
        {
            System.out.println(“Non-Static Initialization Block Person”);
        }
        static
        {
            System.out.println(“Static Initialization Block Person”);
        }
}
```

Example: Order of initializations block in inheritance(Continued).

```
class Student extends Person
{
    public Student()
    {
        System.out.println("Constrictor Student");
    }
    {
      System.out.println("Non-Static Initialization Block  Student");
    }
    static
    {
         System.out.println("Static Initialization Block Student");
    }
    public  static void main(String as[])
    {
      Student s=new Student();
    }
}
```

Output:

Static Initialization Block Person
Static Initialization Block Student
Non-Static Initialization Block Person
Constrictor Person
Non-Static Initialization Block Student
Constrictor Student

Conversion and casting

- A subclass object reference can be converted to super class object reference automatically. But for vice versa, casting is required.
- Automatic conversion example.
  - Only members of **Person** class are accessible

    Person p= new Student(); **//Line 1**

    p.getName(); **//No Compilation Error** **//Line2**

    p.getSid(); **// Compilation Error** **//Line3**
- Casting conversion example.
  - We cast it back to **Student**

    Student s=(Student) p; **//Line4**

    s.getSid(); // No Compilation Error **//Line5**
- Dangers of casting: if the original object is not of subclass type then a runtime exception will be thrown on accessing subclass methods.

  Student h= (Student) new Person()); **//Line6**

  // Runtime error-**ClassCastException**

Overriding

- Many times we may have to redefine some of the existing features of a class in a subclass.
- For example, a big car inheriting from a small car may retain its features like steering wheel but needs to have interiors like seats, seat cover etc of bigger size.
- Redefinition of an inherited method declared in the super class by the subclass is called Overriding.
- When a method is redefined there is some flexibility in terms of not having exactly same method declaration as the super class method.
- They are defined by a set of rules.

Rules of Overriding

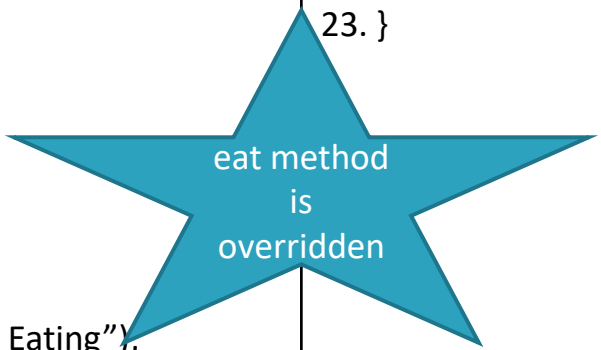
- The signature of the method(method name + argument list) must exactly match.
- The return type must be same or covariant type(a subtype of the return type of super class method).
- The access specifier can be same or be less restrictive.

  (List of access specifies in order of their restriction: **private→default→protected→public**)
- Instance methods can be overridden only if they are inherited/visible by the subclass.
- Exception thrown cannot be new exceptions or parent class exception. We will discuss more about exception in exception handling session.
- You cannot override a method marked **final.**
- You cannot override a method marked **static.**
- If a method can't be inherited , you cannot override it. Remember that overriding implies that you are reimplementing a method you inherited!

Example Of Overriding Method

```
1. public class Animal
2. {
3.   private String name;
4.    public void setName(String name)
5.    {
6.          this.name=name;
7.    }
8.    public String getName()
9.    {
10.          return name;
11. }
12.   public void eat()
13.  {
14.        System.out.println("Animal is Eating");
15.    }
16. }
```

```
17. public class Dog extends Animal
18. {
19.    public void eat()
20.   {
21.         System.out.println("Dog is Eating");
22.   }
23. }
```



Example Of Covariant Returns

```
1.public class Animal
2.{
3.   private String name;
4.   public void setName(String name)
5.   {
6.        this.name=name;
7.    }
8.    public String getName()
9.    {
10.         return name;
11.  }
12.   public Animal eat()
13.   {
14.       System.out.println("Animal is Eating");
15.          return new Animal();
16.    }
17.}
```

```
18. public class Dog extends Animal
19. {
20.   public Dog eat()
21.   {
22.       System.out.println("Dog is Eating");
23.        return new Dog();
24.   }
25.}
```

❑The overridden method's return type can also be a subtype of the original method return class subtype.

Example Of Access Specifier rule.

```
1. public class Animal
2. {
3.   private String name;
4.   protected void  eat()
5.   {
6.       System.out.println("Animal is Eating");

7.   }
8.   public void setName(String name)
9.   {
10.       this.name=name;
11. }
12. public String getName()
 13.{
14.        return name;
15. }

16.}
```

```
17. public class Dog extends Animal
18. {
19.   public void  eat()
20.   {
21.       System.out.println("Dog is Eating");
22.   }
23.}
```

Cannot have private or default access specifier for method eat() in class Dog.

Example Of Visibility Rule In Method Overriding

```
package foo;
public class Animal
{
  private String name;
          void  eat()
  {
     System.out.println("Animal is Eating");

  }
  public void setName(String name)
  {
      this.name=name;
   }
   public String getName()
   {
       return name;
    }

}
```

```
Package bar;
public class Dog extends Animal
{
  private void  eat()
   {
      System.out.println("Dog is Eating");
    }
}
```

Can have any access specifier here since default method can't be accessible out side the package. Same with private also private methods are not inherited /visible.

Restricting inheritance using final keyword

- We can prevent an inheritance of classes by other classes by declaring them as final classes.
- This is achieved in Java by using the keyword final as follows:

```
final class Person              //Line1
{
        //members
}
class Employee extends Person    //Compilation Error. Line 2
{
        //members
}
```

- Any attempt to inherit these classes will cause an error.

Final members: A way for preventing overriding of members in subclasses

- All methods and variables can be overridden by default in subclasses.
- This can be prevented by declaring them as final using the keyword “final” as a modifier. For example:
  - final int marks = 100;
  - final void eat();
- This ensures that functionality defined in this method cannot be altered any. Similarly, the value of a final variable cannot be altered.

@Override

- When overriding a method, **@Override** annotation could be used.
- This tells the compiler that you intend to override a method in the superclass.
- If, for some reason, the compiler detects that the method does not exist in one of the superclasses, it will generate an error.

```
@Override
public void eat()
{
        System.out.println("Dog  is Eating");
}
```

Polymorphism

- Polymorphism is an object-oriented language feature.
- Polymorphism refers to an object's ability to use a single method name to invoke one of different methods at run time – depending on where it is in the inheritance hierarchy.
- It exists only when there is inheritance and the compiler uses dynamic binding to implement it.
- Compiler resolves methods called on a object using
  - Static binding/Early binding: Compiler resolves the call at the compile time.
  - Dynamic binding: Compiler resolves the call at the runtime.
- Overloading is resolved at compile-time. Sometimes this is also referred to as compile-time polymorphism.
- Overriding uses run-time polymorphism or simply polymorphism.

Static Binding Example

- **Example 1:**

  Dog d = new Dog();

  d.getName();

  d.eat();

- **Example 2:**

  Animal a= new Animal();

  a.getName();

  a.eat();

- Since the type of object is known at compile time, to be either of type Dog in the first example or of type Animal in the 2nd example, compiler knows which method to call and so can statically bind the method.

Example of Dynamic Binding/Runtime polymorphism

```
1. public class Animal
2. {
3.    public void eat()
4.    {
5.          System.out.prrintln(“Animal  is Eating”);
6.    }
7. }
8. class Dog extends Animal
9. {
10.    public void eat()
11.    {
12.          System.out.prrintln(“Dog  Eating”);
13.    }
14. }
```

```
15. class TestAnimal
16. {
17.      public static void main(String as[])
18.       {
19.            Animal obj=new Animal();
20.            obj.eat();
21.            obj=new Dog();  //Animal Reference, but a Dog Object.
22.            obj.eat();
23.        }
24. }
```

Output:

Animal is Eating

Dog is Eating

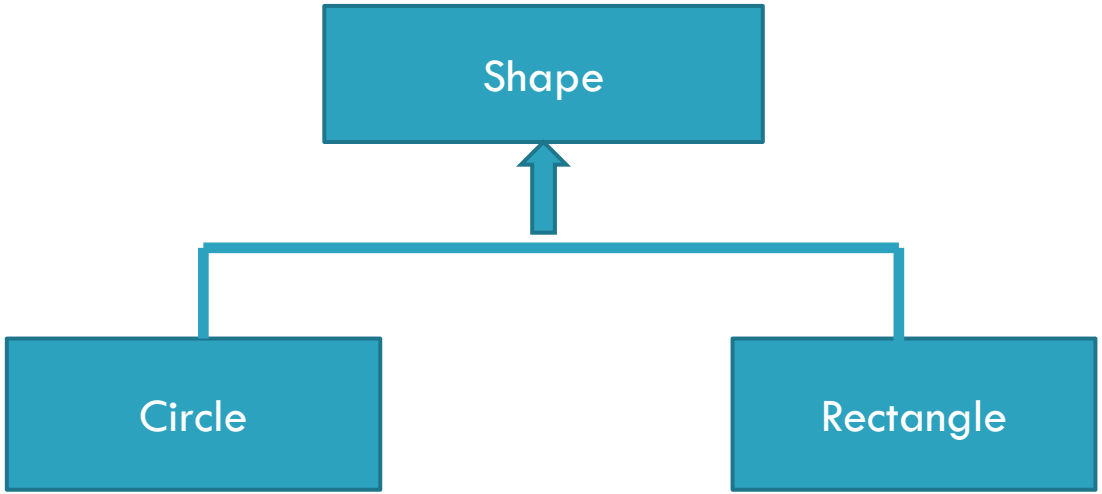
Abstract class

- An Abstract class is a conceptual class.
- An Abstract class cannot be instantiated – objects cannot be created.
- Abstract classes provides a common root for a group of classes, nicely tied together in a package.
- When we define a class to be “final”, it cannot be extended. In certain situation, we want properties of classes to be always extended and used. Such classes are called Abstract Classes.
- Abstract methods are the methods that don’t have the method body. They are just declarations.
- While an **abstract** class can have abstract methods, it could also NOT have any **abstract** methods.
- A class declared abstract, even with no abstract methods can not be instantiated.
- Bear in mind that even if a single method is **abstract**, the whole class must be declared **abstract**.
- Please note that the classes that we have created so far are called concrete classes (classes from which you could create instances.
- An **abstract** method must not be **static**.

Inheriting abstract class

- A class can inherit from abstract class in two ways:
  - By Complete Implementation.
    - A class that inherits from the **abstract** class and override all the abstract methods by providing implementations specific to that class.
    - This results in a concrete class
  - By Partial Implementation.
    - A class that inherits from the abstract class and does NOT override one or more **abstract** method.
    - Such class must be marked as an **abstract** class.

Creating abstract classes and methods



Example of Creating abstract classes and methods

```
public abstract class Shape
{
        public abstract double area();

        public void move()
        { // non-abstract method
          // implementation
        }
}
```

In this class **area()** is an abstract method. All the concrete child class of class **Shape** must have to override method **area().**

```
public Circle extends Shape {
                private double r;
                private static final double PI =3.1415926535;
                public Circle() { r = 1.0; }
                public double area() { return PI * r * r; }
   …
  }
 public Rectangle extends Shape {
                private double l, b;
                public Rectangle() { l = 0.0; b=0.0; }
                public double area() { return l * b; }
   ...
  }
```

In this example **Circle** and **Rectangle** is the child of **Shape.**

Interface

- Interface is a conceptual entity.
- An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.
- An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviours of an object. An interface contains behaviours that a class implements.
- Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.
- An interface is similar to a class in the following ways:
  - An interface can contain any number of methods.
  - An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
  - The bytecode of an interface appears in a **.class** file.
  - Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

Interface

- However, an interface is different from a class in several ways, including:
  - You cannot instantiate an interface.
  - An interface does not contain any constructors.
  - All of the methods in an interface are abstract.
  - An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
  - An interface is not extended by a class; it is implemented by a class.
  - An interface can extend multiple interfaces.

Syntax of Interface

Interface definition:

1. interface *interface_name*
2. {
3. [datatype variable_name=value;]
4. [returntype method_name();]
5. }

Class implementing interface:

6. class *class_name* [extends class] implements *interface_name_1 [, interface_name_2 … interface_name_n]*
7. {
8. // implements methods in the *interface_name*
9. }

Like classes, interfaces can also be defined inside packages.

Therefore they have public or package access.

Example of Interface

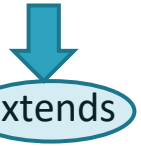
```
1. public interface Shape {
2.       double PI=3.14;
3.       void area();
4. }
5. class  Circle implements Shape {
6.        private double radius;
7.        Circle(double r) { radius=r; }
8.        public void area() {
9.             System.out.println(PI* radius* radius);
10.       }
11.        public static void main(String a[]) {
12.              Shape s= new Circle(10);// or Circle c= new Circle (10);
13.              s.area();
14.        }
15. }
```

Points to note

```
1. public interface Shape {
2. double PI=3.14;  //Compiler automatically inserts public, static and final
3. void area();  // Compiler automatically inserts public and abstract
4. }
5. class  Circle implements Shape {
6. private double radius;
7. Circle(double r){ radius=r; }
8. public void area() { System.out.println(PI* radius* radius);
9. }
10. public static void main(String a[]){
12. Shape s= new Shape(); // compilation error
13. System.out.println(Shape.PI+ " "+ Circle.PI);
// both prints 3.14
}}
```

Inheriting interfaces

- Like classes, interfaces can also be extended. The new sub-interface will inherit all the members of the super-interface in the manner similar to classes.
- This is achieved by using the extends keyword.



```
interface InterfaceName2 extends InterfaceName1
{
    // Body of InterfaceName2
}
```

instanceof

- **instanceof** returns **true** or **false** when interface names are used except in case of **final** classes. In case of **final** class, it results in compilation error.
- Below code never gives a compilation error.

```
1. Circle c= new Circle();
   System.out.println(c instanceof Shape );// prints true
2. Student s= new Student(); //assume that student is a class
   System.out.println(s instanceof Shape );//prints false
```

- Below code gives a compilation error.

```
3. class Square implements Shape{}
   Square sq= new Square();
   System.out.println(sq instanceof Circle);
   // gives compilation error
```

Interface and casting

- A class implementing an interface is automatically converted to the interface type.
- This is very obvious and we used it in the example as well.

  1. Shape s= new Circle(10);// no compilation error
- To convert an interface reference back to the original class type requires explicit casting

  2. Example: **Circle c =(Circle)s;**
- Any reference can be converted to interface type by explicit casting.

  3. Student s= new Student();

  4. Shape s1=(Shape) s; //no compilation error

  But

  5. Circle c=(Circle)s; // Compilation error

Uses Of Interface

- Interfaces are
  - ✓ used to share constants.
  - ✓ used to set standards/define contracts.
  - ✓ used just to tag a class, so objects of its class can represent another type.
  - ✓ used to overcome the issues that arise because Java does not support multiple inheritance.

Classes in Java: Agenda

- Object class.
- Overview of java.lang package.
- Overview of java.util package.

Object class

- All classes in java, by default, inherit from a predefined java class called **Object**.
- **Object** class is defined in **java.lang** package.
- This class is the root of the class hierarchy.
- All objects, including arrays, directly or indirectly inherit the methods of this class.
- Even if we do not explicitly write code to inherit from **Object**, compiler inserts **extends Object** to our class if it finds no **extends** clause specified in our class definition.
- **Object** class is a concrete class and has a no-argument constructor.

Object class (Continued)

- The methods in this class are:
  - equals(Object ref) - returns if both objects are equal.
  - finalize( ) - method called when an object's memory is destroyed.
  - getClass( ) - returns class to which the object belongs.
  - hashCode( ) - returns the hashcode of the class.
  - notify( ) - method to give message to a synchronized methods.
  - notifyAll( ) - method to give message to all synchronized methods.
  - toString() - return the string equivalent of the object name.
  - wait() - suspends a thread for a while.

equals()

- The **equals()** method compares two objects for equality and returns **true** if they are equal. The **equals()** method provided in the **Object** class uses the identity operator (**==**) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. The **equals()** method provided by **Object** tests whether the object *references* are equal—that is, if the objects compared are the exact same object.
- To test whether two objects are equal in the sense of *equivalency* (containing the same information), you must override the **equals()** method. Here is an example of a **Employee** class that overrides **equals()**:

Overriding equals()

```
1. public class Employee
2. {
3.     String name;
4.     public Employee(String name)
5.     {
6.                     this.name=name;
7.     }
8.     @Override
9.     public boolean equals(Object obj)
10.    {
11.        if(obj instanceof Employee &&
this.name==((Employee)obj).name)
12.        {
13.             return true;
14.        }
15.        else
16.        {
17.             return false;
18.        }
19.    }
20. }
```

```
21. class TestEmployee
22. {
23.     public static void main(String as[])
24.     {
25.         Employee e=new Employee("Raj");
26.         Employee e1=new Employee("Raj");
27.
28.         System.out.println(e.equals(e1));
29.     }
30. }
```

Output

true

If we will not override the equals method the output will be **"false".**

finilize()

- This methods is called just before the object is going to get garbage collector.
- A subclass will have to overrides the finalize method to dispose of system resources or to perform other cleanup.
- The finalize method is never invoked more than once by a Java virtual machine for any given object.

```
public class Test{
        public void finalize() throws Throwable{
        }
}
```

hashCode()

- This method returns a hash code value for the object. The implementation in Object class returns unique identifier for each object.
- If you override **equals**, you must override **hashCode**.
- HashCode values for equal objects must be same.
- **equals** and **hashCode** must use the same set of fields.
- Collection class like **Hashtable. HashSet** etc depend on this method heavily**.**

Overriding hashcode ()

```
1. public class Employee
2. {
3.     String name;
4.     public Employee(String name)
5.     {
6.             this.name=name;
7.     }
8.     @Override
9.     public boolean equals(Object obj)
10.     {……}
```

```
11. @Override
12. public int hashCode()
13.    {
14.            return name.hashCode();
15.     }
16.}
```

toString()

- What will happen when we try to print objects like primitives?

  Employee e=new Employee (“Raj");

  System.out.println(e);

  It prints: **Employee@4ea20232**

- This is because **print** methods (and many more methods) call **toString()** method on the **Object** to get the string representation of the object. It displays whatever the **toString()** method returns.
- **toString()** method that is inherited from the **Object** class prints class name and the unique hashcode of the object. (Hashcode is an integer value that is associated with an object.)
- If we are not happy with this result, then we should override **toString()** method.

Overriding toString()

```
1. class Employee
2. {
3.      String name;
4.      public Employee(String name)
5.      {
6.                  this.name=name;
7.      }
8.     @Override
9.     public String toString()
10.  {
11.                 return name;
12.     }
13.}
```

```
14. public static void main(String as[])
15. {
16.             Employee e=new Employee("Raj");
17.             System.out.println(e);
 18. }
19.}
```

Output:

Raj

Overview of java.lang package

- The java.lang package provides various classes and interfaces that are fundamental to Java programming.
- The java.lang package contains various classes that represent primitive data types, such as int, char, long, and double.

Overview of java.lang package(Continued)

class	Description
Class	Supports runtime processing of the class information of an object.
String	Provides functionality for String manipulation.
Integer	Provides methods to convert an Integer object to a String object.
Math	Provides functions for statistical, exponential operations.
Object	All the classes in the java.lang package and classes belongs to other package are the subclass of the Object class.
System	It provides a standard interface to input output and error devices, such as keyboard and VDU(Visual Display Unit)

String class

- The **java.lang.String** class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class. Strings are constant, their values cannot be changed after they are created.
- In Java, strings can be created without using char array.
- The String class has convenient methods that allows working with strings.

Constructors of String classs:

```
String()                //Line1
String(String s)        //Line2
```

Examples of creating String object:

```
String s="abc";                     //Line 3
String s= new String();             //Line 4
String s= new String("Hello");      //Line5
String s1= new String(s);           //Line6
```

Methods of String class:

- **int length()**
- Returns the length of this string
  - **String s= new String(“Hello”);**
  - **System.out.println(s.length()); //Line1**
  - **//prints 5**

- **char charAt(int index)**
- Returns the character at the specified index
  - **String s="Have a nice day";**
  - **System.out.println(s.charAt(0)); //Line2**
  - **// prints H**

Methods of String class (Continued):

- String concatenation can be done in two ways:
- **"+" operator**: Not a method but an operator that can be used with strings (Unlike C++, there no operator overloading in Java. However + is overloaded for strings for programmer convenience.)

  1. String s1="abc",s2="def";

  2. String s3=s1+s2; // returns abcdef

  3. String s4=s1+1; // returns abc1

  4. String s5=s1+"dd"; // returns abcdd

  5. String s4=s1+true; // returns abctrue

  6. String s4=s1+'d'; // returns abcd

  7. String s4=null+ s1; // returns nullabc

- **String concat(String str)**

  8. String s1="java".concat("c");//returns "javac"

Methods of String class (Continued):

- To compare if two strings are the same, equals methods are used.
- **boolean equals(Object object)**
- **boolean equalsIgnoreCase(String anotherString)**

Example:

1. String s1="abc";

2. String s2="sbc";

3. String s3="ABC";

4. s1.equals(s2) ;//returns false

5. s1.equalsIgnoreCase(s3) ;//returns true

Methods of String class (Continued):

- Why do we require equals() method to compare Strings ? Can we not compare using ==?
- == works fine with primitive data types.
- But with references, (since they are like pointers) , == will actually compare the addresses.
- What we want here is to check equality of value of strings.

```
1. String s1= “abc”;
2. String s2=new String(s1);
3. System.out.println(s1==s2); // returns false
4. System.out.println(s1.equals(s2)); // returns true
```

Methods of String class (Continued):

- To work with parts of a string, we have two methods
- **public String substring(int beginIndex)**
- **public String substring(int beginIndex, int endIndex)**
- Index begins from 0.

Example:

“icecream".substring(3); // returns “cream“

“icecream".substring(0,3); // returns “ice”

- To remove with leading and trailing whitespace

  String trim()

  Example:

  1. String str=“ Hello “;

  2. System.out.println(str.trim()+”java”); //print “Hellojava”

Methods of String class (Continued):

- **public int compareTo(String anotherString)**
- **public int compareToIgnoreCase(String str)**
  - Compares current String object with another String object. If the Strings are same the return value is 0 else the return value is non zero.
  - If str1 > str2 then return value is a positive number
  - If str1<str2 then return value is a negative number
  - Example:

    1. String s1="ABC"; String s2="acc";

    2. s2.compareTo(s1) // It will returns 32

    3. s2.compareToIgnoreCase(s1) // It will returns 1

Converting primitives to String

- **static String valueOf(XXX b)**

  where XXX includes all primitives like byte, short, int, long, float, double, char, boolean.

  Example:

  1. String i=String.valueOf(1224);

- Tokenizing string

  2. String[] split(String regex)

  Example :

  3. String str="apple,mango,banana";

  4. String list[]=str.split(",");

  5. for(String s:list) System.out.println(s);

Methods of String class (Continued):

- String toLowerCase()
- String toUpperCase()

- String replace(char oldChar, char newChar)
- String replaceAll(String reg,String replacement)

- public boolean startsWith(String prefix)
- public boolean endsWith(String suffix)

Immutability

- Immutability means something that cannot be changed.
- Strings are immutable object in Java. What does this mean?
- String literals are very heavily used in applications and they also occupy a lot of memory.
- Therefore for efficient memory management, all the strings are created and kept by the JVM in a place called string pool (which is part of Method Area).
- Garbage collector does not come into string pool.
- How does this save memory?

Immutability

Example1:

String s1="ABC";

String s2="ABC";

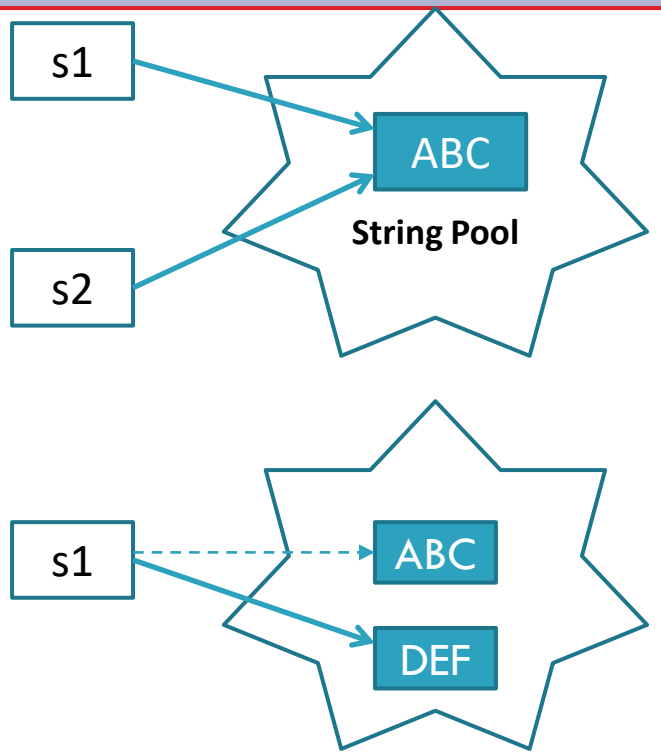
- When Strings are created this way by assigning literals to the variable straight, JVM checks if the string is available in the pool. If not it creates one. Otherwise it assigns it to the existing reference .

Example2:

String s1="ABC";

s1="DEF";

- When a value of a string reference is changed, a new string is created in the pool and that is assigned to the reference.
- Strings are Immutable Objects.
- That means that, once created, String object cannot be changed!



Immutability

- Assigning string references:

1. String s1="ABC";

2. String s2=s1;

3. s2=“DEF”;

4. System.out.println(s1);// prints ABC

Problem with String

- **String** class objects are immutable.
- In cases where we have lots of string manipulation we may end up with creating lot of strings in the string pool which are unnecessary.
- Therefore, in such cases we need to go for **StringBuffer** or **StringBuilder**

StringBuilder

- This **final** class can be used in the situations where we require lot of string manipulations.
- **StringBuilder** objects are mutable.
- This class is added in java5.
- Constructors
  - **StringBuilder()**
  - **StringBuilder(String str)**

Methods Of StringBuilder

Methods that are common in both String and StringBuilder classes:

- **char charAt(int index)**
- **int length()**
- **String substring(int start)**
- **String substring(int start, int end)**
- **int indexOf(String str)**
- **int indexOf(String str,int fromIndex)**
- **int lastIndexOf(String str)**
- **int lastIndexOf(String str, int fromIndex)**

Methods Of StringBuilder(Continued)

- Concatenation

  StringBuilder append(String str)
  StringBuilder append(StringBuffer str)
  StringBuilder append(char[] c)
  StringBuilder append(xx b)

  where **xx** is **boolean, char, int, long float and double**

  Example: **s1= new StringBuilder("Now");** **//Line1**

  s1.append(" Showing"); // Now Showing **//Line2**

  How do you achieve this in String class?

- **Replacing characters**

  StringBuilder replace(int start, int end, String s)

  Example: **StringBuilder s1= new StringBuilder("now");** **//Line3**

  s1.replace(0,0,"S");// Snow **//Line4**

  s1.replace(0,1,"S");// Sow **//Line5**

  s1.replace(0,2,"S"); //Sw **//Line6**

- Compare this to the **replace()** method in **String** class

Methods Of StringBuilder(Continued)

- Insertion and deletion of characters

  StringBuilder insert(int offset, Object str) **//Line1**

  StringBuilder insert(int offset, String str) **//Line2**

  StringBuilder insert(int offset, xx b) **//Line3**

  where **xx is boolean, char, int, long float** and **double**

- In case of **Object** as 2nd argument the string that is returned by **toString()** method is inserted.

  Example: **StringBuilder s1= new StringBuilder("Teacher():");**

  s1.insert(8, new Teacher("Tom"));

  // Teacher(Tom (1)):

- **StringBuilder delete(int start, int end)** **//Line4**

  Example: **StringBuilder deleteCharAt(int index)**

  StringBuilder s1= new StringBuilder("Teacher():");

  s1.delete(7, s1.length());// Teacher //Line5

Methods Of StringBuilder(Continued)

- Reverse:

  StringBuilder reverse()

- Check if a string is a palindrome.. Try to do this with String class. And then compare your code with the code here. It turns out that the code here is far simpler!

```
1. public class Palindrome {
2.     public static void main(String[] args)
3.     {
4.             String palindrome = "MalayalaM“;
5.             StringBuilder sb = new StringBuilder(palindrome);
6.             System.out.println(sb.equals(sb.reverse()));
7.             System.out.println(sb);
8.     }
9.}
```

Note

- Unlike **String, StringBuilder and StringBuffer** (we discuss in the next slide) does not override **equals()** method.
- That is,

  StringBuilder sb = new StringBuilder(“Hi”);

  StringBuilder sb3 = new StringBuilder(“Hi”);

  System.out.println(sb3.equals(sb));

  return **false!**
- Then how did previous example work?
- Recall that by default **equals()** method of **Object** class functions the same as the that of the **==**. Having said this, since **sb.reverse()** changes string in that same location, **sb** before and after calling reverse are same!
- Also note that **StringBuilder and StringBuffer** is not **Comparable (**unlike **String)**

StringBuffer

- **StringBuffer** has same methods as **StringBuilder** and it can also be used to create mutable Strings. It is also a **final** class.
- Only difference between both the classes is.
  - **StringBuffer** is thread-safe while **StringBuilder** is not thread-safe.
- Thread-safe class have methods that are synchronized.
- **synchronized** makes only one thread at a time access an object's **synchronized** methods. This may impact performance.
- Thinking of this issue, JSE built **StringBuilder** class that does not have **synchronized** methods. So, now it is up to programmers to make sure of the consistency of strings from **StringBuilder** class by providing **synchronized** blocks locking the object.
- We discuss more about synchronization in multithreading session.

Overview of java.util package

- The java.util package provides various utility classes and interfaces that support date and calendar operations, String manipulations and Collections manipulations. Classes provided by the java.util package

class	Description
Date	Encapsulates date and time information.
Calendar	Provides support for date conversion. It is recommended that **Calendar** class be used whenever possible because most of the methods in **Date** class are deprecated.
GregorianCalendar	It is a subclass of Calendar class, provides support for standard calendar used worldwide.

Date Class

- Date: Date class represent date and time. There are several constructors for Date objects.
- Constructors :
  - Date() : produces the current date and time.
  - Date(int year, int month, ind dayofmonth)
  - Date(int year, int month, ind dayofmonth,int hours, int mins)
  - Date(int year, int month, ind dayofmonth,int hours, int mins,int secs)
  - Date(long milliseconds) : no of milliseconds from January 1, 1970 midnight
  - Date(String strdate) : Converts the string representation of date into a Date object.
- Methods
  - boolean after(Date pdate) - returns true if the current date is after pdate.
  - boolean before(Date pdate) - returns true if the current date is before pdate.
  - boolena eqauls(Date pdate) - returns true if the current date same as pdate.
  - int getDay()
  - int getMonth()
  - int getYear()
  - void setDay(int dayno)
  - void setMonth(int monthno)
  - void setYear(int year)

Example of Date Class

- Example : Usage of Date class. SourceFile : TestDate.java

```
1. import java.util.*;
2. public class TestDate{
3.     public static void main(String args[])
4.     {
5.       Date today = new Date();
6.       System.out.println("Today's date is  "+today.toString());
7.       System.out.println("Current time is  "+today.getTime());
8.       Date aday = new Date(1998,10,9);
9.       Date bday = new Date(1998,11,10);
10.      Date cday = new Date(1998,9,23);
11.      Date tday = new Date(1998,9,23,12,20);
12.      System.out.println("A day is  "+aday.toString());
13.      System.out.println("B day is  "+bday.toString());
14.      System.out.println("C day is  "+cday.toString());
15.      System.out.println("T day with time is "+tday.toString());
16. if (aday.before(bday))
17.         System.out.println(" a is before b");
18.     if (cday.after(bday))
19.         System.out.println(" c is after b");
20.     System.out.println("Time of aday is "+aday.getTime());
21.     System.out.println("Time of today is "+tday.getTime());
22.     Date today1 = new Date();
23.     if (today1.equals(today))
24.          System.out.println(" today is same as today1");
25. }}
Output:
Today's date is  Wed Jul 31 11:24:11 IST 2013
Current time is  1375250051022
A day is  Wed Nov 09 00:00:00 IST 3898
B day is  Sat Dec 10 00:00:00 IST 3898
C day is  Sun Oct 23 00:00:00 IST 3898
T day with time is Sun Oct 23 12:20:00 IST 3898
 a is before b
Time of aday is 60868780200000
Time of today is 60867355800000
```

Formatting Dates

- **java.text.SimpleDateFormat** class is used for formatting and parsing dates in a locale-sensitive manner.
- In the constructor the date format can be specified using predefined letters that correspond to some meaning.
- Constructors**:**
  - **SimpleDateFormat()**
    - uses the default pattern and date format symbols for the default locale.
  - **SimpleDateFormat(String pattern)**
    - uses the given pattern and the default date format symbols for the default locale.

Methods In SimpleDateFormat

- **final String format(Date date)**
  - Formats a Date into a date/time string as per specification in the constructor.
- **Date parse(String source) throws ParseException**
  - Parses text from the beginning of the given string to produce a date based on the format specification in the constructor.
  - **java.text.ParseException** is a checked exception which is thrown if the expected string is not matching specified format.
- **Calendar getCalendar()**
  - Gets the calendar associated with this date/time formatter.

Example: SimpleDateFormat

```
import java.text.*;
import java.util.Date;
public class TestDateFormat {
public static void main(String[] args) throws ParseException {
        Date now = new Date( );
        SimpleDateFormat ft = new SimpleDateFormat ("E dd MMM yyyy 'at' hh:mm:ss a zzz");
        System.out.println(t.format(now));              //Line1
        SimpleDateFormat ft1 = new SimpleDateFormat ("dd.mm.yyyy");                 //Line2
        Date d= ft1.parse("10.7.1967");                 //Line3
        System.out.println(t.format(d));
}}
```

Output

Wed 31 Jul 2013 at 12:06:36 PM IST

10.07.1967

Calendar and GregorianCalendar

- **Calendar** is an abstract class.
- **GregorianCalendar** is a concrete subclass of **Calendar**. This class provides the standard calendar system used by most of the world.
- To create an instance of **Calendar** class **getInstance()** static method is used. This a **Calendar** object with the system's date and time.
- Internally the value is stored as time in millisecond represented by January 1, 1970 00:00:00.000 GMT (Gregorian).
- Constructor:
  - **GregorianCalendar()**
  - **GregorianCalendar(int year, int month, int dayOfMonth,[int hourOfDay, int minute, int second])**

Example Of Calendar

```
1. import java.util.Calendar;
2. class CalendarDemo
3. {
4.      public static void main(String args[])
5.      {
6.      String months[]= {"Jan", "Feb", "Mar", "Apr",
        "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov",
        "Dec"};
7.      Calendar cal = Calendar.getInstance();
8.      System.out.println("The Date is: ");
9.      System.out.print(months[cal.get(Calendar.
        MONTH)]);
10.     System.out.print(" " + cal.get(Calendar.DATE) + "
        ");
11.     System.out.println(cal.get(Calendar.YEAR));
        // Setting Time
12.        cal.set(Calendar.HOUR, 10);
13.     cal.set(Calendar.MINUTE, 27);
14.     cal.set(Calendar.SECOND, 0);
15.     System.out.print("Time is: ");
16.     System.out.print(cal.get(Calendar.HOUR) + ":");
17.     System.out.print(cal.get(Calendar.MINUTE) +
        ":");
18.     System.out.print(cal.get(Calendar.SECOND));
19.     }
20. }
```

Output

```
The Date is:
Jul 31 2013
Time is: 10:27:0
```

Example Of GregorianCalendar

```
import java.util.*;
public class GregorianCalendarDemo
{
        public static void main(String args[])
        {
                Calendar calendar = new GregorianCalendar();
                System.out.println(calendar.get(Calendar.YEAR));
                System.out.println(calendar.get(Calendar.MONTH+1));
                System.out.println(calendar.get(Calendar.DAY_OF_MONTH));
        }
}
```

Converting object to string

- Any Java reference type or primitive that appears where a String is expected will be converted into a string.
  - System.out.println("1 + 2 = " + (1 + 2)); // Line1
- For an reference type (object)
  - this is done by inserting code to call String toString() on the reference.
  - All reference types inherit this method from java.lang.Object and override this method to produce a string that represents the data in a form suitable for printing.
- To provide this service for Java's primitive types, the compiler must wrap the type in a so-called wrapper class, and call the wrapper class's toString method.
  - String arg1 = new Integer(1 + 2).toString();

Converting object to string(Continued)

- For user defined classes they will inherit the standard Object.toString() which produces something like "ClassName@123456" (where the number is the hashcode representation of the object).
- To have something meaningful the classes have to provide a method with the signature public String toString().

```
public class Car
{
    ----
    ---
    public String toString()                    //Line1
    {
        return brand + ” : ” + color + ” : ”+wheels+” : “+ speed;
    }
}
```

Converting String To Number

- Many data type wrapper classes provide the valueOf(String s) method which converts the given string into a numeric value.
- The syntax is straightforward. It requires using the static Integer.valueOf(String s) and intValue() methods from the java.lang.Integer class.
- To convert the String "22" into the int 22 you would write
  - int i = Integer.valueOf("22").intValue(); **//Line1**
- Doubles, floats and longs are converted similarly. To convert a String like "22" into the long value 22 you would write
  - long l = Long.valueOf("22").longValue(); **//Line2**
- To convert "22.5" into a float or a double you would write:
  - double x = Double.valueOf("22.5").doubleValue(); **//Line3**
  - float y = Float.valueOf("22.5").floatValue(); **//Line4**
- If the passed value is non-numeric like “Four," it will throw a **NumberFormatException.**

Example: Converting string to numbers

```
class Test{
public static void main (String args[])
{
String str="12";
String str1="10";
int a = Integer.valueOf(str).intValue();          //line1
int b = Integer.valueOf(str1).intValue();         //line2
System.out.println(a+b);                          //line3
}
}
```

Here's the output: 22

Regular expression

- A *regular expression* defines a pattern for a String. Regular Expressions can be used to search, edit or manipulate text. Regular expressions are not language specific but they differ slightly for each language. Java regular expressions are most similar to Perl.
- Java Regular Expression classes are present in ***java.util.regex*** package that contains three classes: **Pattern**, **Matcher** and **PatternSyntaxException**.
- Pattern object is the compiled version of the regular expression. It doesn't have any public constructor and we use it's public static method ***compile*** to create the pattern object by passing regular expression argument.
- Matcher is the regex engine object that matches the input String pattern with the pattern object created. This class doesn't have any public constructor and we get a Matcher object using pattern object ***matcher*** method that takes the input String as argument. We then use ***matches*** method that returns boolean result based on input String matches the regex pattern or not.
- PatternSyntaxException is thrown if the regular expression syntax is not correct.

Example Of Regular expression

```
import java.util.regex.*;
public class Demo {
	public static void main(String[] args) {
		String input = "I have a cat, but I like my dog better.";

		Pattern p = Pattern.compile("(mouse|cat|dog|wolf|bear|human)");
		Matcher m = p.matcher(input);

		while (m.find()) {
			System.out.println("Found a " + m.group() + " on position "+m.start());

		}
	}
}
```

Compile the REGEX pattern

Get matcher from static method on Pattern class

Seed the matcher with the string that you want to find matches

Find the group of chars that matched (as a String) and where those chars started and ended in the string to be matched

Output:

Found a cat on position 9

Found a dog on position 28

Regular expression(Continued)

- **Character classes**
  - \d A digit: [0-9]
  - \D A non-digit: [^0-9]
  - \s A whitespace character: [\t\n\x0B\f\r]
  - \S A non-whitespace character: [^\s]
  - \w A word character: [a-zA-Z_0-9]
  - \W A non-word character: [^\w]
- **Quantifiers**
  - * Match 0 or more times
  - + Match 1 or more times
  - ? Match 1 or 0 times
  - {n} Match exactly n times {n,} Match at least n times
  - {n,m} Match at least n but not more than m times

Regular expression(Continued)

- **Meta-characters**
  - **** Escape the next meta-character (it becomes a normal/literal character).
  - **^** Match the beginning of the line .
  - **.** Match any character (except newline).
  - **$** Match the end of the line (or before newline at the end).
  - **|** Alternation (‘or’ statement).
  - **()** Grouping.
  - **[]** Custom character class.

Regular expression: Matcher methods

- A matcher is created from a pattern by invoking the pattern's matcher method. Once created, a matcher can be used to perform three different kinds of match operations:
  - The matcher method attempts to match the entire input sequence against the pattern.
  - The find method scans the input sequence looking for the next subsequence that matches the pattern.

Questions