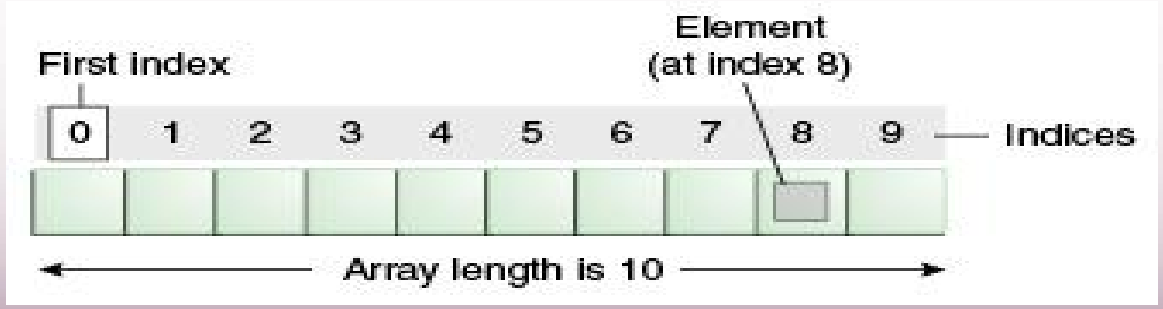
Arrays

Introduction to Arrays

- Array is one variable is used to store a list of data and manipulate
- This type of variable is called an array.
- It stores multiple data items of the same data type, in a contiguous block of memory, divided into a number of slots.
- The *new* keyword to create an array object.
- Need to tell the compiler how many elements will be stored in it.



Creating Arrays

- There are three steps to creating an array,
- **declaring**
  - int[] k;
  - float[] yt;
  - String[] names;
- **allocating**
  - k = new int[3];
  - yt = new float[7];
  - names = new String[50];
- **initializing**
  - int[ ] k = {1, 2, 3};
  - float[ ] yt = {0.0f, 1.2f, 3.4f, -9.87f, 65.4f, 0.0f, 567.9f};

Enhanced For Loop

- The enhanced for loop, simplifies looping through an array or a collection.
- Instead of having *three* components, the enhanced for has *two*.

- **for**(data_type variable : array | collection){}

- int [] a = {1,2,3,4};

- **for(int n : a)**

- **System.out.print(n);**.

Array Bounds

- All array subscripts begin with 0 and ends with n-1
- In order to get the number of elements in an array, can use the length field of an array.

- The length field of an array returns the size of the array.

```
int list [] = new int[10];

for (int i = 0; i< list.length; i++)
{
        System.out.println(list[i]);
}
```

- There is no array element arr[n]! This will result in an array-index-out-of bounds exception.

Array of Objects

```
public static void displaybooks(Book[] bks)
{
for(int i=0;i<bks.length;i++)
{
    System.out.println("Book Number :="+bks[i].getBookno());
    System.out.println("Book Name :="+bks[i].getBookname());
}
}
     public static void main(String args[])
     {
     Book[]  bk = new Book[2];
     Book b1 = new Book(100,"java");
    Book b2= new Book(101,"j2ee");
     bk[0]=b1;
     bk[1]=b2;
          ArrayofObject obj= new ArrayofObject();
      displaybooks(bk);

}
```

Copying Arrays

- System.arraycopy()
  - can be used to efficiently copy data from one array into another:

- public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

- The two Object arguments
  - the array to copy *from* and the array to copy *to*.

- The three int arguments
  - **starting position** in the **source** array,
  - **starting position** in the **destination array**,
  - number of array elements to copy.

STRINGS

Strings

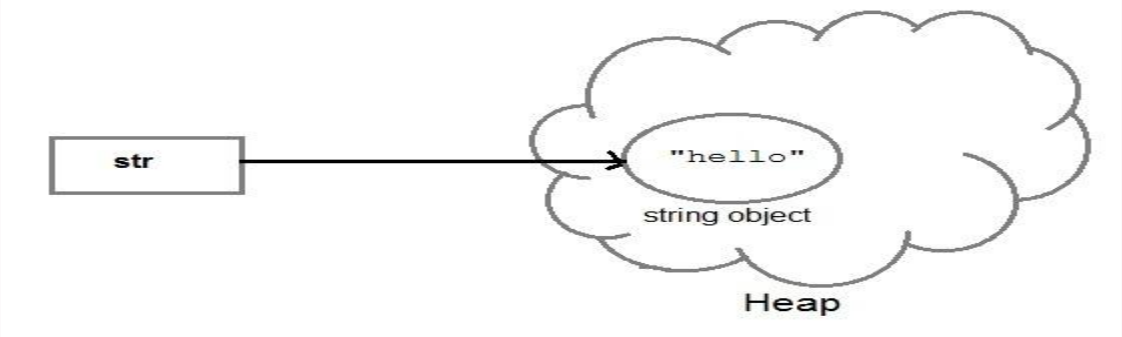
- Many characters join together to form a string.
  - Collection of characters bound by double quotes (“”).

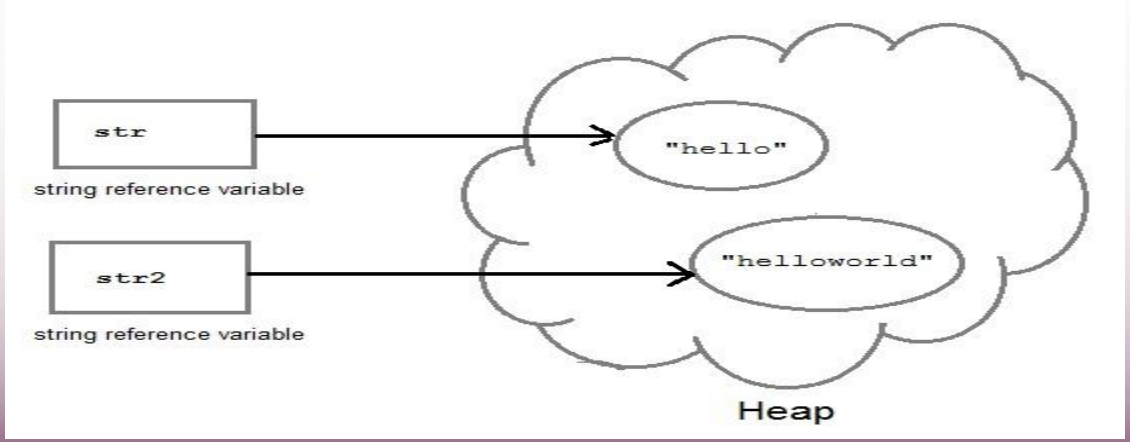
- String are created in two ways
  - **Direct Initialization**
    - String message1= "Hello World";
    - String message2 =“Hello World

- **No new objects are created.**
  - When the second line is executed the message2 instance points to the message1 object which was declared before.

String Creation

- String str= "Hello";
- String str2=str;





Strings

```
String a = "abcd";

String b = "abcd";

System.out.println(a == b); // True

System.out.println(a.equals(b)); // True
```

Strings

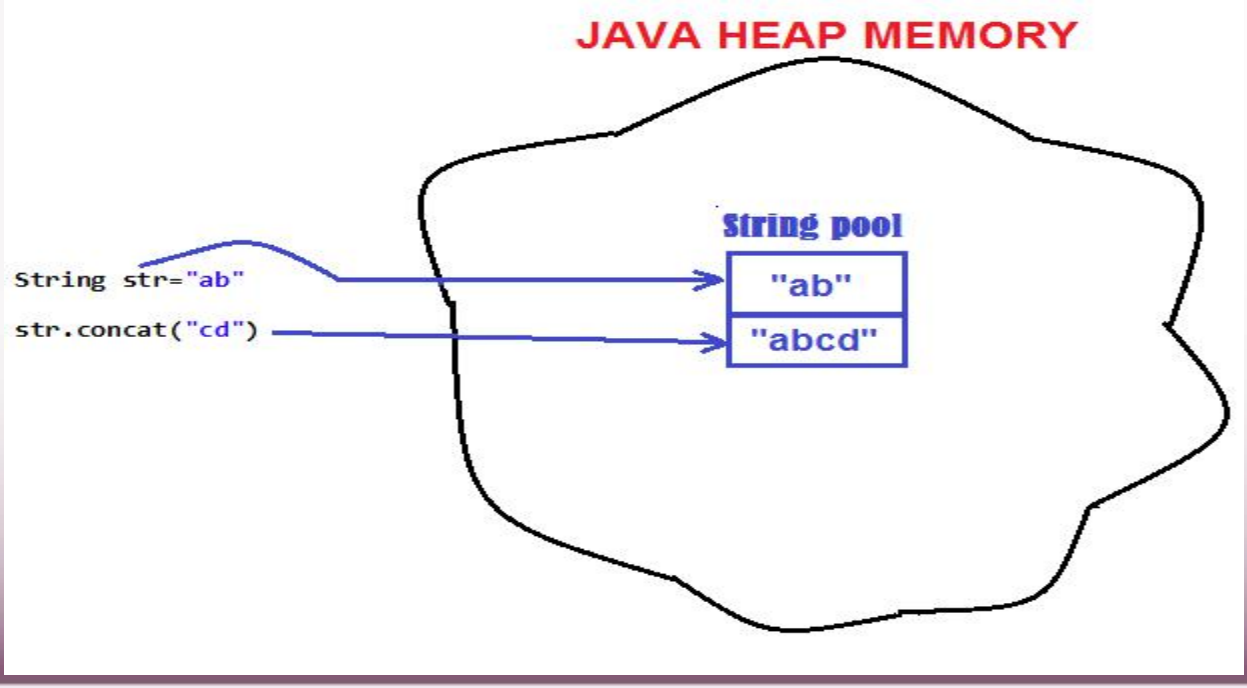
- **Using constructor**
  - String myString = new String("Hello World")

- A new space is allocated in heap regardless of the object exists in pool or not.
  - JVM is forced to create a new string every time its executed

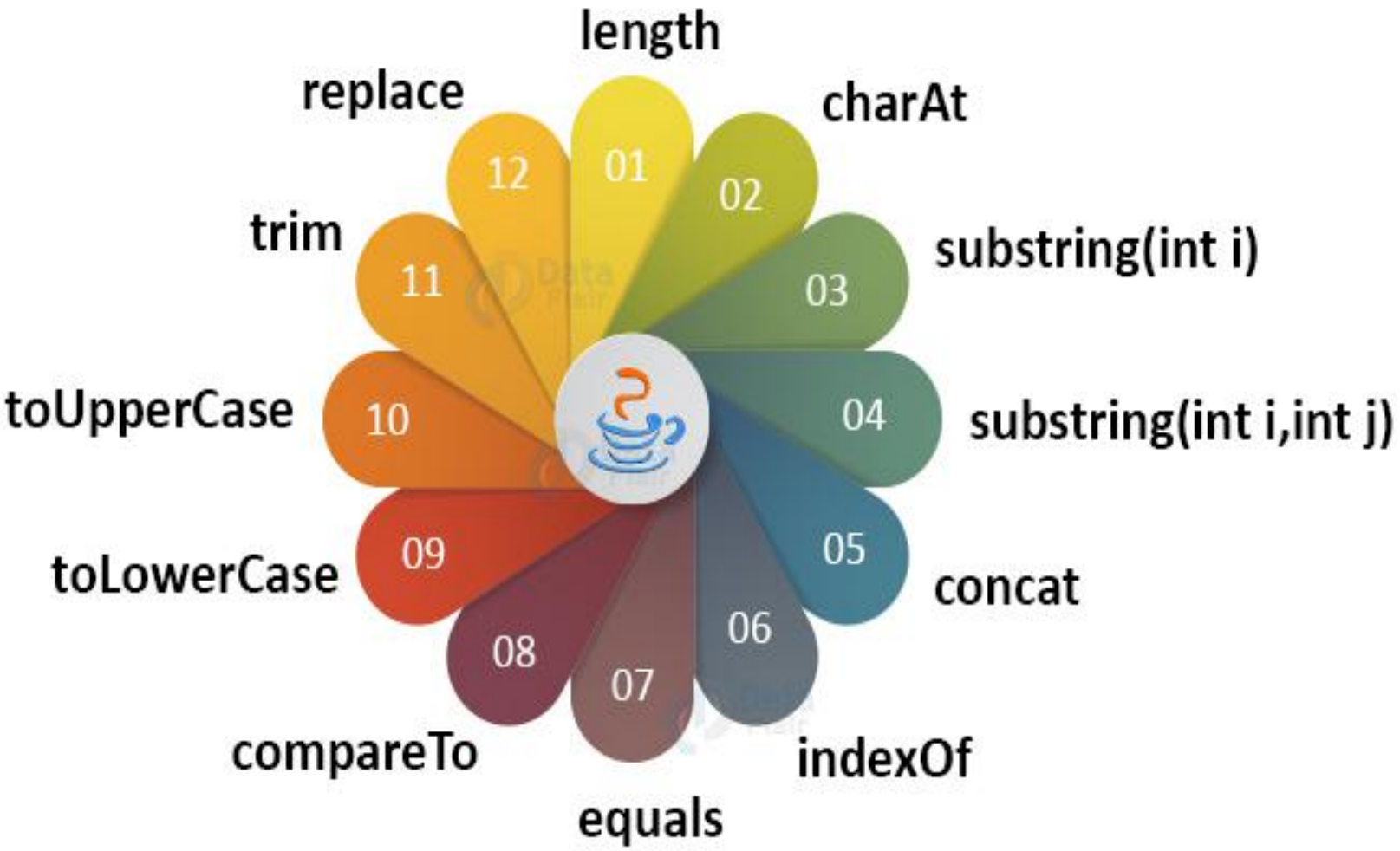
```
String c = new String("abcd");

String d = new String("abcd");

System.out.println(c == d); // False

System.out.println(c.equals(d)); // True
```

String are Immutable

- Java String objects are immutable.
  - Unmodifiable or unchangeable.
- Once string object is created its data or state can't be changed
  - A new string object is created.



String Methods in Java

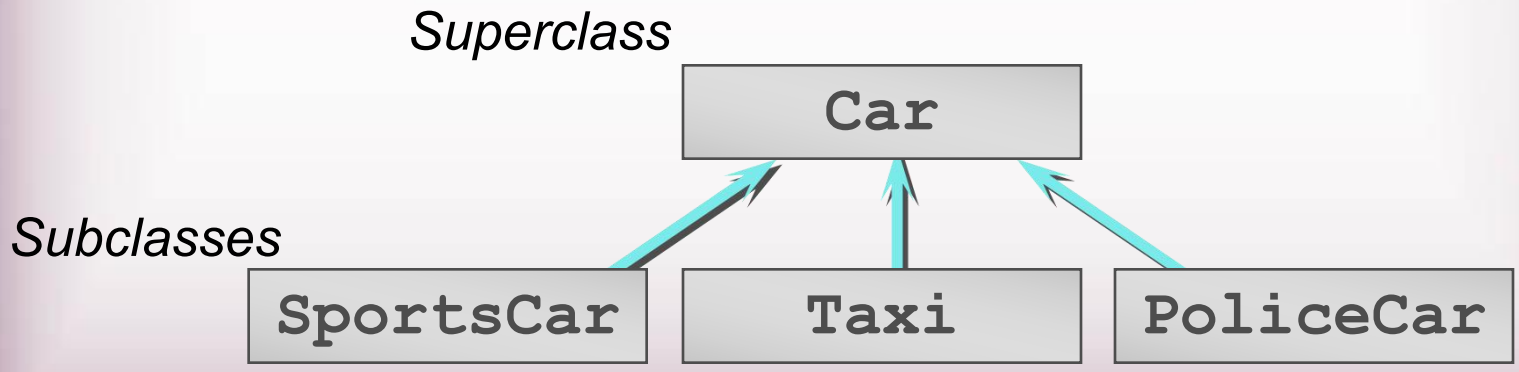


INHERITANCE

Overview

- A class can inherit from another class
  - Original class is the “superclass”
  - New class is called the “subclass”

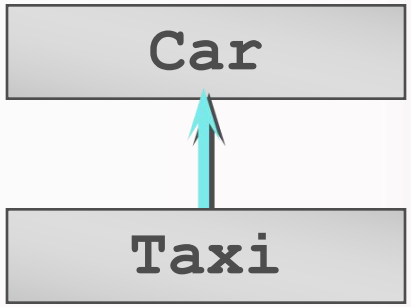
- Inheritance is a fundamental OO concept



Example of Inheritance

– The Car class defines certain methods and variables

– `Taxi` extends `Car`, and can:
  - Add new variables
  - Add new methods
  - Override methods of the `Car` class



Specifying Inheritance in Java

– Inheritance is achieved by specifying which superclass the subclass
"extends"

•Taxi inherits all the variables and methods of Car

```
public class Car {
   …
}
```

```
public class Taxi extends Car {
   …
}
```

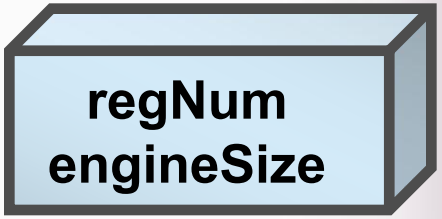
What Does an Object Look Like?

A subclass inherits all the instance variables of its superclass

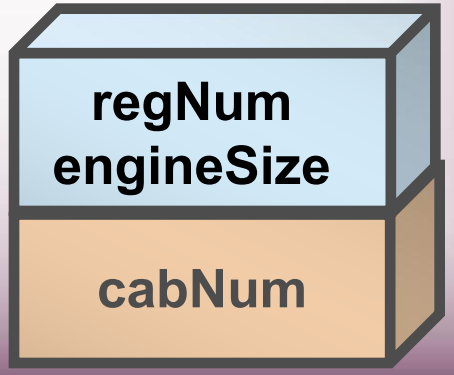
```
public class Car {
   String regNum;
    int engineSize; …
}
```

```
public class Taxi extends Car {
   private int cabNum; …
}
```

Car object



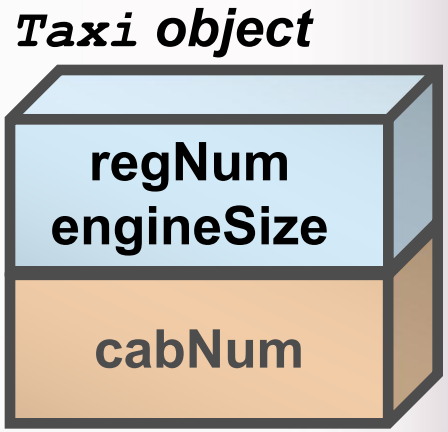
Taxi object



Default Initialization

– What happens when a subclass object is created?

```
Taxi taxi1 = new Taxi();
```



– If no constructors are defined:

– No-arg constructor is called in superclass

  • No-arg constructor called in subclass

Nondefault Initialization

Specific initialization can be performed as follows:

```
public class Car {
   Car(String r, int e) {
      regNum = r;
      engineSize = e;
   } …
```

```
public class Taxi extends Car {
   Taxi(String r, int e, int c) {
      super(r, e);
      cabNum = c;
   } …
```

Use `super()` to call superclass constructor

Specifying Additional Methods

- The superclass defines methods that are applicable for all kinds of Car
- The subclass can specify additional methods that are specific to Taxis

```
public class Car {
   public int getReg()…
   public void changeOwner()…
   …
```

```
public class Taxi extends Car {
   public void renewCabLicense()…
   public boolean isBooked()…
   …
```

Overriding Superclass Methods

- A subclass inherits all the methods of its superclass
- The subclass can override a method with a specialized version, if it needs to:

```
public class Car {
  public String details() {
    return "Reg:" + getReg();
  }
```

```
public class Taxi extends Car {
  public String details() {
    return "Reg:" + getReg() + cabNum;
  }
```

Overriding

- When a sub-class defines a
  - "method with ***same method name, argument types, argument order and return type as a method in its super class***, its called method overriding. "
- Methods declared as **final, static and private cannot be overridden**.
- An overriding method can be declared as final
- The method can't be less accessible
  - **Public => only public**
  - **Protected => Both Public and Protected**
  - **Default => default, public and protected**

Overriding with Compatible Return type

- Arguments must be same and return type must be compatible

```
class Base
{
   private int i = 5;

   public Number  getNumber()
   {
     return i;
   }
}

class Derived extends Base
{
   @Override
   public Integer getNumber()
   {
     return new Integer(10);
   }
}
```

Invoking Superclass Methods

- If a subclass overrides a method, it can still call the original superclass method
- Use super.method()to call a superclass method from the subclass
- Though keyword super is used to refer super class, method call super.super.method() is invalid.

```
public class Car {
  public String details() {
    return "Reg:" + getReg();
  }
```

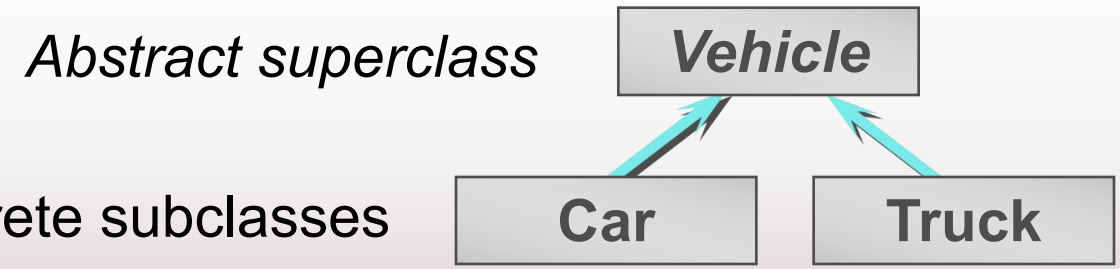
```
public class Taxi extends Car {
  public String details() {
    return super.details() + cabNum;
  }
```

super keyword

- The keyword *super* refers to the base class
  - *super( )*
    - invokes the base class constructor
    - base class constructors are automatically invoked
  - *super.method( )*
    - invokes the specified method in the base class
  - *super.variable*
    - to access the specified variable in the base class
- *super must be the first statement in a constructor*

Abstract Class

- An abstract class is a shared superclass
  - Provides a high-level partial implementation of some concept
  - Cannot be instantiated
- Use the abstract keyword to declare a class as abstract



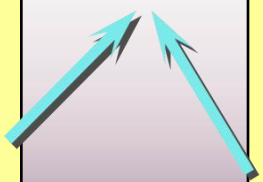
Defining Abstract Classes in Java

- Use the `abstract` keyword to declare a method as abstract
  - Only provide the method signature
  - Class must also be abstract
- Must be implemented by a concrete subclass
  - Each concrete subclass can implement the method differently

```
public abstract class Vehicle {
   private String regNum;
   public String getRegNum()…
   public abstract boolean
                       isRoadWorthy();
}
```

```
public class Car
        extends Vehicle{
 private int numSeats;
 public void fitRack()…
```

```
public class Truck
        extends Vehicle{
 private int axles;
 public void setLoad()…
```



The Abstract Class-Super Class

Note this

```
public abstract class BankAccount {

   public abstract void deposit(double  amount);

   public abstract void withdraw(double amount);

   public void sayHello() {

            System.out.println("Thanks-Come Again");
      }

}
```

Child Class –Its alsoAbstract

```
public abstract class Savingaccount extends BankAccount
   {

   public abstract void getDetails();
}
```

The Concrete Class

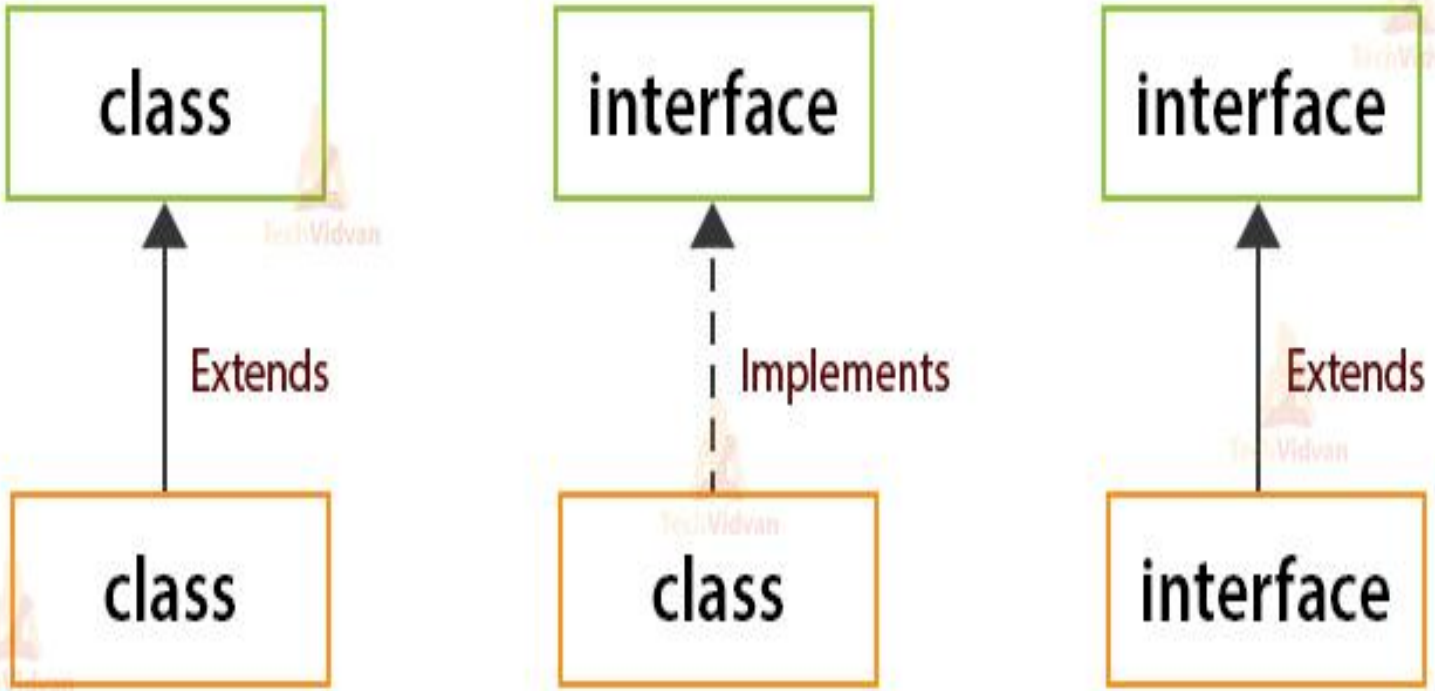
```
public class Supersavings extends Savingaccount {
  private double balance,amount ;
  private String custname;
  private int accno;
     public void deposit(double amt)
    {
      balance+=amt;
     }
    public void withdraw(double amt)
    {
     balance-=amt;
 }
    public void getDetails()
    {
     System.out.println(custname);
     System.out.println(accno);
     System.out.println(amount);
     System.out.println(balance);
    }
}
```

Interfaces & Polymorphism

Interfaces

- An interface is like a ***fully abstract class***
  - All the methods are **public and abstract- Till Java 8**
    - No instance variables if present they are **public static final**
- Defines a set of methods that other classes can implement
  - ***A class can implement many interfaces,***
    - ***but can extend only one class***
    - **can extend other interfaces** and extend multiple interfaces

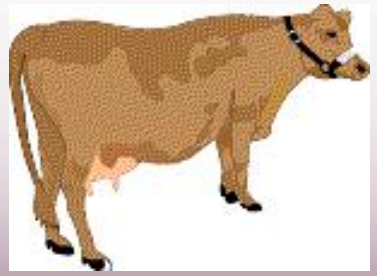
Relationship between Class and Interface in Java



Example of Interfaces

- Interfaces describe an aspect of behavior that different classes can support
  - Any class that supports the “Steerable” interface can be steered:

Steerable



Not Steerable

Interface

- Interfaces define types in an abstract form
  - Allows the specification of a reference type without providing an implementation
  - The class that implements the interface decides how to implement
  - **Use `interface` keyword instead of `class`**

```
public interface Function {
    public double apply(double arg);
}
```

Implement the Interface

```
public class CurrencyConverter implements Function {

 @Override
 public double apply(double arg) {
    return arg * 45;
  }
}
```

Function usdToInr = **new CurrencyConverter();**

System.***out.println(usdToInr.apply(100));***

Implementing an Interface

```
public interface Steerable {
   int maxTurn = 135;
   void turnLeft(int deg);
   void turnRight(int deg);
}
```

```
public class Car
        extends Vehicle
        implements Steerable {

   public void turnLeft(int deg)
   {…}

   public void turnRight(int deg)
   {…}
```

Partial Implementation of an Interface

- Declare the class as abstract if the class does not implement all the methods of an interface

```
Public abstract class Car
                  extends Vehicle
                     implements Steerable {

  public void turnLeft(int deg)
  {…}

}
```

Using `instanceof` with Interfaces

- The `instanceof` operator can be used to check if an object implements an interface
- Downcasting can be used if necessary, to call methods defined in the interface

```
public void aMethod(Object obj) {
   …
   if (obj instanceof Steerable)
      ((Steerable)obj).turnLeft();
}
```

Extending an Interface

- One Interface can extend another Interface

```
public interface Steerable {
   int maxTurn = 135;
   void turnLeft(int deg);
   void turnRight(int deg);
}
```

```
public interface Navigator
                    extends Steerable {
   void calcSpeed(int distance);

}
```

Polymorphism

- This OO Principle allows the programmer to program abstractly
- Objects of a subclass can be treated as objects of its superclass
- The base class of a hierarchy is usually an abstract class or an Interface
  - They define the common behavior (functionality)
- Classes and Subclasses inherit this behavior and extend it according to their own properties by overriding the methods of the super class
- **Subclass objects can be treated as super class** objects through references,
  - But, **super class objects are not subclass objects**

Polymorphism

- Allows for hierarchies to be highly extensible
- New classes can be introduced and can still be processed without changing other parts of the program
- Same method can do different things, depending on the class that implements it.
- Method invoked is associated with OBJECT and not with reference.
- Types of Polymorphism -
  - Method Overloading.
  - Method Overriding.

Polymorphism

```
public interface Conditional {

public boolean test();

}
```

Implementation -One

```
public class Professor implements Conditional {

private String qualification;

@Override
public boolean test() {
   return this.qualification.equalsIgnoreCase("phd");
}


}
```

Implementation -Two

```
public class Student implements Conditional {

private double markScored;

@Override
public boolean test() {
  return this.markScored>90;
}

}
```

Polymorphism

```
public static void print(Conditional poly) {

        System.out.println(poly.test());
}

public static void main(String[] args) {

 Student ramesh = new Student(101,"Ramesh",67);
 Professor prof = new Professor(201, "manish", "ece",
"phd");

    print(ramesh);
    print(prof);
}
```

Substitution of Object References in Method Calls

- A subclass object can be passed into any method that expects a superclass object

- The method that got invoked, is the version that's present in the object type and NOT the reference type.

```
public static void main(String[] args) {
   Taxi t = new Taxi("ABC123", 2000, 79);
   displayDetails(t);

public static void displayDetails(Car c) {
   System.out.println(c.details());
}
```

Dynamic Method Dispatch

```
class First
{
  public void show()
  {
      System.out.println("ShowFirst");
  }
}

class Second extends First
{
    public void show()
    {
System.out.println("Show Second");
    }
}
```

```
First fst = new Second();

Second sec = (Second)fst;

sec.show();

First fst2 = new First();

Second sec2 = (Second)fst2;

sec2.show();
```

subclass= (subclass)superclass

Valid at both Compile ,Run Time

subclass= (subclass) erclass

Valid at Compile exception at ,RunTi

Dynamic Method Dispatch

- **Related Through Inheritance**

- ***superclass*** **=** ***subclass***
  - always valid
- ***subclass*** **=** ***superclass***
  - not valid at compile time, needs a cast
- ***subclass*****=(*****subclass*****)*****superclass***
  - ***valid at compile time,*** **checked at runtime.**
  - **if INVALID then ClassCastException is thrown.**

- **Unrelated Classes – Not Allowed**

- someClass = someUnrelatedClass
  - not valid, won't compile
- somcClass = (someClass)someUnrelatedClass
  - not valid, won't compile