toString (), Inheritance,abstract

Reference: Liang, Y., 2015. Intro To Java Programming. 10th ed. Pearson Education Limited.

Java toString() Method

- If you want to represent any object as a string, toString() method comes into existence.
- The toString() method returns the String representation of the object.
- If you print any object, Java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depending on your implementation.
- Advantage of Java toString() method
- By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.

Understanding problem without toString() method

Let's see the simple code that prints reference.

Student.java

```
class Student{
 int rollno;
 String name;
 String city;
 Student(int rollno, String name, String city){
 this.rollno=rollno;
 this.name=name;
 this.city=city;
 }
 public static void main(String args[]){
   Student s1=new Student(101,"Raj","lucknow");
   Student s2=new Student(102,"Vijay","ghaziabad");
   System.out.println(s1);//compiler writes here s1.toString()
   System.out.println(s2);//compiler writes here s2.toString()
 }
}
```

Output:

```
Student@1fee6fc
Student@1eed786
```

Example of Java toString() method

Let's see an example of toString() method.

Student.java

```
class Student{
 int rollno;
 String name;
 String city;
  Student(int rollno, String name, String city){
 this.rollno=rollno;
 this.name=name;
 this.city=city;
 }
 public String toString(){//overriding the toString() method
  return rollno+" "+name+" "+city;
 }
 public static void main(String args[]){
   Student s1=new Student(101,"Raj","lucknow");
   Student s2=new Student(102,"Vijay","ghaziabad");

   System.out.println(s1);//compiler writes here s1.toString()
   System.out.println(s2);//compiler writes here s2.toString()
 }
}
```

Output:

```
101 Raj lucknow
102 Vijay ghaziabad
```

Super Keyword in Java

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

1) super is used to refer immediate parent class instance variable.

- We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields. In the example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

```
class Animal{
String color="white";
}

class Dog extends Animal{
 String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class }
 }

 class TestSuper1{
public static void main(String args[]){
 Dog d=new Dog();
 d.printColor();
}}
```

Output:
black
white

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden. In the example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local. To call the parent class method, we need to use super keyword.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
 void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
 d.work();
 }}
```

Output:
eating...
barking...

3) super is used to invoke parent class constructor

The super keyword can also be used to invoke the parent class constructor.
Let's see a simple example:

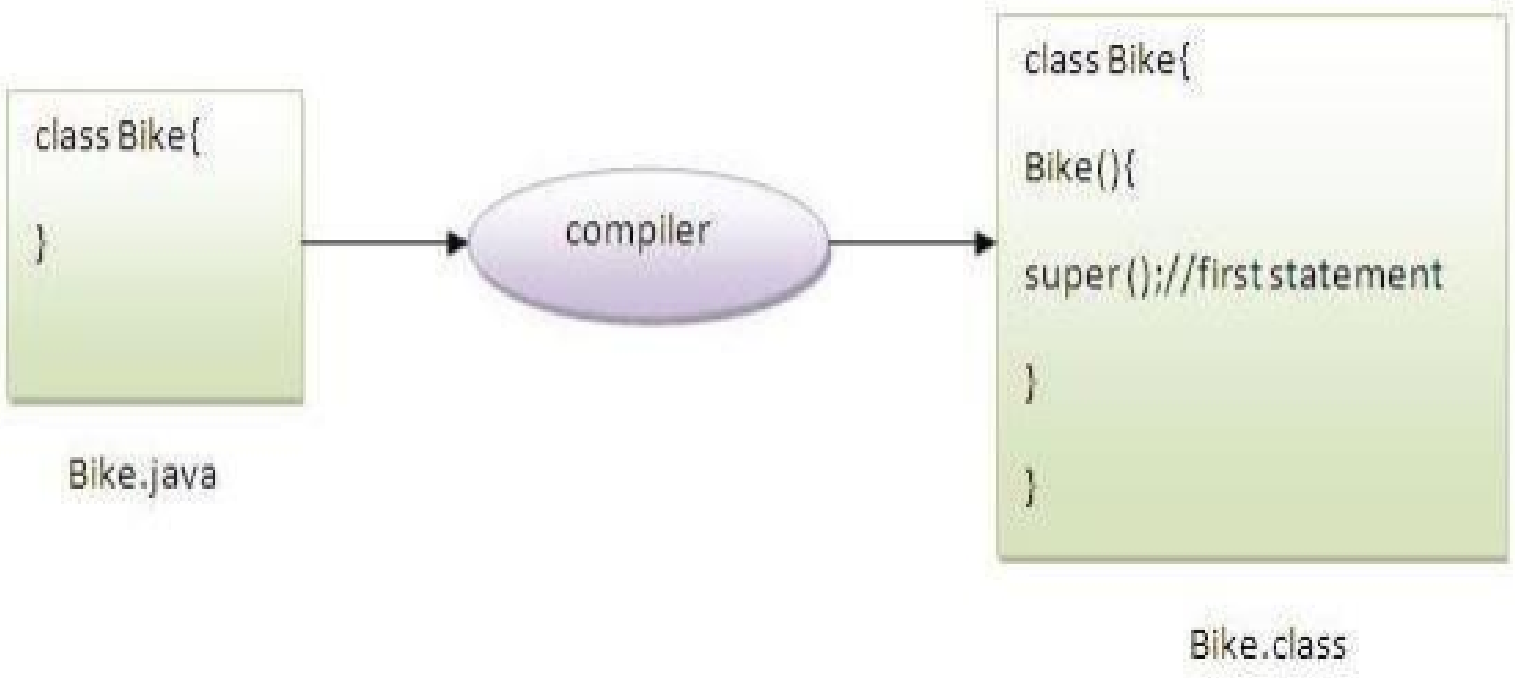
```
class Animal{
Animal(){System.out.println("animal is created");}
 }
 class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created"); }
 }
 class TestSuper3{
public static void main(String args[]){
 Dog d=new Dog();
 }}
```

Output:

animal is created
dog is created

As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

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



Another example of super keyword where super() is provided by the compi implicitly.

```
class Animal{
 Animal(){
System.out.println("animal is created");}
}
class Dog extends Animal{
 Dog(){
System.out.println("dog is created"); }
}
class TestSuper4{
public static void main(String args[]){
 Dog d=new Dog();  }}
```

Output:
animal is created
dog is created

super example: real use

- Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

```
class Person{
int id;
String name;
Person(int id,String name){
this.id=id;
this.name=name; } }
class Emp extends Person{
 double salary;
Emp(int id,String name,doublesalary){
 super(id,name);//reusing parent constructor
this.salary=salary; }
 void display(){System.out.println(id+" "+name+" "+salary);}  }
class TestSuper5{
public static void main(String[] args){
Emp e1=new Emp(1,"ankit",4500.50);
e1.display();  }}
```

Output:1 ankit 4500.50

IQRA UNIVERSITY IU

Abstract Class

- A class which contains the **abstract** keyword in its declaration is known as abstract class.
- Abstract classes may or may not contain *abstract methods*, i.e., methods without body (public void get();)
- But, if a class has at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

IQRA UNIVERSITY IU

Example

```
/* File name : Employee.java */
public abstract class Employee {
   private String name;
   private String address;
   private int number;

   public Employee(String name, String address, int number) {
      System.out.println("Constructing an Employee");
      this.name = name;
      this.address = address;
      this.number = number;
   }

   public double computePay() {
     System.out.println("Inside Employee computePay");
     return 0.0;
   }

   public void mailCheck() {
      System.out.println("Mailing a check to " + this.name + " " + this.address);
   }

   public String toString() {
      return name + " " + address + " " + number;
   }

   public String getName() {
      return name;
   }

   public String getAddress() {
      return address;
   }

   public void setAddress(String newAddress) {
      address = newAddress;
   }

   public int getNumber() {
      return number;
   }
}
```

- You can observe that except abstract methods the Employee class is same as normal class in Java. The class is now abstract, but it still has three fields, seven methods, and one constructor.
- Now you can try to instantiate the Employee class in the following way −

```
/* File name : AbstractDemo.java */
public class AbstractDemo {

   public static void main(String [] args) {
      /* Following is not allowed and would raise error */
      Employee e = new Employee("George W.", "Houston, TX", 43);
      System.out.println("\n Call mailCheck using Employee reference--");
      e.mailCheck();
   }
}
```

When you compile the above class, it gives you the following error –

```
Employee.java:46: Employee is abstract; cannot be instantiated
      Employee e = new Employee("George W.", "Houston, TX", 43);
                   ^
1 error
```

Inheriting the Abstract Class

- We can inherit the properties of Employee class just like concrete class in the following way –

Example

```
/* File name : Salary.java */
public class Salary extends Employee {
   private double salary;   // Annual salary

   public Salary(String name, String address, int number, double salary) {
      super(name, address, number);
      setSalary(salary);
   }

   public void mailCheck() {
      System.out.println("Within mailCheck of Salary class ");
      System.out.println("Mailing check to " + getName() + " with salary " + salary);
   }

   public double getSalary() {
      return salary;
   }

   public void setSalary(double newSalary) {
      if(newSalary >= 0.0) {
         salary = newSalary;
      }
   }
    public double computePay() {
        System.out.println("Computing salary pay for " + getName());
        return salary/52;
    }
}
```

Here, you cannot instantiate the Employee class, but you can instantiate the Salary Class, and using this instance you can access all the three fields and seven methods of Employee class as shown below.

```
/* File name : AbstractDemo.java */
public class AbstractDemo {

   public static void main(String [] args) {
      Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
      Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
      System.out.println("Call mailCheck using Salary reference --");
      s.mailCheck();
      System.out.println("\n Call mailCheck using Employee reference--");
      e.mailCheck();
   }
}
```

This produces the following result – Output

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

 Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0
```

IQRA UNIVERSITY IU

Abstract Methods

- If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract.
- **abstract** keyword is used to declare the method as abstract.
- You have to place the **abstract** keyword before the method name in the method declaration.
- An abstract method contains a method signature, but no method body.
- Instead of curly braces, an abstract method will have a semi colon (;) at the end.

Example

```
public abstract class Employee {
    private String name;
    private String address;
    private int number;

    public abstract double computePay();
    // Remainder of class definition
}
```

Declaring a method as abstract has two consequences –

- The class containing it must be declared as abstract.
- Any class inheriting the current class must either override the abstract method or declare itself as abstract.
- **Note** – Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

Suppose Salary class inherits the Employee class, then it should implement the **computePay()** method as shown below –

```
/* File name : Salary.java */
public class Salary extends Employee {
   private double salary;   // Annual salary

   public double computePay() {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }
   // Remainder of class definition
}
```

IQRA UNIVERSITY IU

Interface?

- The interface is a blueprint that can be used to implement a class. The interface does not contain any concrete methods (methods that have code). All the methods of an interface are abstract methods.
- An interface cannot be instantiated. However, classes that implement interfaces can be instantiated. Interfaces never contain instance variables but, they can contain public static final variables (i.e ., constant class variables)

IQRA UNIVERSITY IU

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

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();
	}

Difference No.1:

Abstract class can extend only one class or one abstract class at a time

```
class Example1{
   public void display1(){
      System.out.println("display1 method");
   }
}
abstract class Example2{
   public void display2(){
      System.out.println("display2 method");
   }
}
abstract class Example3 extends Example1{
   abstract void display3();
}
class Example4 extends Example3{
   public void display3(){
      System.out.println("display3 method");
   }
}
class Demo{
   public static void main(String args[]){
       Example4 obj=new Example4();
       obj.display3();
   }
}
```

Output:
display3 method

Interface can extend any number of interfaces at a time

```
interface Example1{
   public void display1();
}

interface Example2 {
   public void display2();
}

interface Example3 extends Example1,Example2{
}

class Example4 implements Example3{
   public void display1(){
      System.out.println("display1 method");
   }
   public void display2(){
      System.out.println("display2 method");
   }
}
class Demo{
   public static void main(String args[]){
      Example4 obj=new Example4();
      obj.display1();
   }
}
```

Output:
display1 method

Difference No.2:

Abstract class can be extended(inherited) by a class or an abstract class

```
class Example1{
  public void display1(){
    System.out.println("display1 method");
  }
}
abstract class Example2{
  public void display2(){
    System.out.println("display2 method");
  }
}
abstract class Example3 extends Example2{
  abstract void display3();
}
class Example4 extends Example3{
  public void display2(){
    System.out.println("Example4-display2 method");
  }
  public void display3(){
    System.out.println("display3 method");
  }
}
class Demo{
  public static void main(String args[]){
    Example4 obj=new Example4();
    obj.display2();
  }
}
```

Output:
Example4-display2 method

Interfaces can be extended only by interfaces. Classes has to implement them instead of extend

```
interface Example1{
  public void display1();
}
interface Example2 extends Example1{
}
class Example3 implements Example2{
  public void display1(){
    System.out.println("display1 method");
  }
}
class Demo{
  public static void main(String args[]){
    Example3 obj=new Example3();
    obj.display1();
  }
}
```

Output:
display1 method

Difference No.3:

Abstract class can have both abstract and concrete methods

```
abstract class Example1 {
   abstract void display1();
   public void display2(){
     System.out.println("display2 method");
   }
}
class Example2 extends Example1{
   public void display1(){
      System.out.println("display1 method");
   }
}
class Demo{
   public static void main(String args[]){
     Example2 obj=new Example2();
     obj.display1();
   }
}
```

Output:

display1 method

Interface can only have abstract methods, they cannot have concrete methods

```
interface Example1{
   public abstract void display1();
}
class Example2 implements Example1{
   public void display1(){
     System.out.println("display1 method");
   }
}
class Demo{
   public static void main(String args[]){
     Example2 obj=new Example2();
     obj.display1();
   }
}
```

Output:

display1 method

Difference No.4:

In abstract class, the keyword 'abstract' is mandatory to declare a method as an abstract

```
abstract class Example1{
   public abstract void display1();
}

class Example2 extends Example1{
   public void display1(){
      System.out.println("display1 method");
   }
   public void display2(){
      System.out.println("display2 method");
   }
}
class Demo{
   public static void main(String args[]){
       Example2 obj=new Example2();
       obj.display1();
   }
}
```

Output:
display1 method

In interfaces, the keyword 'abstract' is optional to declare a method as an abstract because all the methods are abstract by default

```
interface Example1{
   public void display1();
}
class Example2 implements Example1{
   public void display1(){
      System.out.println("display1 method");
   }
   public void display2(){
      System.out.println("display2 method");
   }
}
class Demo{
  public static void main(String args[]){
     Example2 obj=new Example2();
     obj.display1();
  }
}
```

Output:
display1 method

Difference No.5:

Abstract class can have protected and public abstract methods

```
abstract class Example1{
   protected abstract void display1();
   public abstract void display2();
   public abstract void display3();
}
class Example2 extends Example1{
   public void display1(){
       System.out.println("display1 method");
   }
   public void display2(){
      System.out.println("display2 method");
   }
   public void display3(){
      System.out.println("display3 method");
   }
}
class Demo{
   public static void main(String args[]){
      Example2 obj=new Example2();
      obj.display1();
   }
}
```

Output:
display1 method

Interface can have only public abstract methods

```
interface Example1{
   void display1();
}
class Example2 implements Example1{
   public void display1(){
      System.out.println("display1 method");
   }
   public void display2(){
      System.out.println("display2 method");
   }
}
class Demo{
   public static void main(String args[]){
       Example2 obj=new Example2();
       obj.display1();
   }
}
```

Output:
display1 method

Difference No.6:

Abstract class can have static, final or static final variables with any access specifier

```
abstract class Example1{
  private int numOne=10;
  protected final int numTwo=20;
  public static final int numThree=500;
  public void display1(){
    System.out.println("Num1="+numOne);
  }
}
class Example2 extends Example1{
  public void display2(){
    System.out.println("Num2="+numTwo);
    System.out.println("Num3="+numThree);
  }
}
class Demo{
  public static void main(String args[]){
    Example2 obj=new Example2();
    obj.display1();
    obj.display2();
  }
}
```

Output:

Num1=10

Num2=20

Num3=500

Interface can have only public abstract methods

```
interface Example1{
  int numOne=10;
}
class Example2 implements Example1{
  public void display1(){
    System.out.println("Num1="+numOne);
  }
}
class Demo{
  public static void main(String args[]){
    Example2 obj=new Example2();
    obj.display1();
  }
}
```

Output:

Num1=10

When to use an 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.
- 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 an 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.


THANK YOU