Agenda:

Object class

Object cloning

Call by Value

Call by Reference

Math Class

Wrapper class

Command Line Arguments

Recursion

Object class:

The Object class is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

Object obj=getObject();//we don't know what object will be returned from this method

The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

Methods of Object class:

public final Class getClass() : returns the Class class object of this object. The Class class can further be used to get the metadata of this class.

public int hashCode() : returns the hashcode number for this object.

public boolean equals(Object obj) : compares the given object to this object.

protected Object clone() throws CloneNotSupportedException : creates and returns the exact copy (clone) of this object.

public String toString() : returns the string representation of this object.

public final void notify() : wakes up single thread, waiting on this object's monitor.

public final void notifyAll() : wakes up all the threads, waiting on this object's monitor.

public final void wait(long timeout)throws InterruptedException : causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).

public final void wait(long timeout,int nanos)throws InterruptedException : causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).

public final void wait()throws InterruptedException : causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).

protected void finalize()throws Throwable : is invoked by the garbage collector before object is being garbage collected.

Object cloning:

The object cloning is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.

The java.lang.Cloneable interface must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates CloneNotSupportedException.

The clone() method is defined in the Object class.

Signature:

protected Object clone() throws CloneNotSupportedException

Two types of cloning.

1. shallow cloning
2. Deep cloning

Shallow cloning:

- The default implementation of the clone method creates a shallow copy of the source object, it means a new instance of type Object is created, it copies all the fields to a new instance and returns a new object of type ‘Object’. This Object explicitly needs to be typecast in object type of source object.
- This object will have an exact copy of all the fields of source object including the primitive type and object references. If the source object contains any references to other objects in field then in the new instance will have only references to those objects, a copy of those objects is not created. This means if we make changes in shallow copy then changes will get reflected in the source object. Both instances are not independent.
- The clone method in Object class is protected in nature, so not all classes can use the clone() method. You need to implement Cloneable interface and override the clone method. If the Cloneable interface is not implemented then you will get CloneNotSupportedException.super.clone () will return shallow copy as per implementation in Object class.

Example:

```
public class Employee implements Cloneable {
        int empId;
        String firstName;
        String lastName;
        Address address;

        public Employee(int empId, String firstName, String lastName, Address address) {
                super();
                this.empId = empId;
                this.firstName = firstName;
                this.lastName = lastName;
                this.address = address;
        }
        protected Object clone() throws CloneNotSupportedException {
                return super.clone();
        }
}
```

```
public class Address {
	String houseNo;
	String streetName;
	String city;
	String state;
	String zip;

	public Address(String houseNo, String streetName, String city, String state, String zip) {
		super();
		this.houseNo = houseNo;
		this.streetName = streetName;
		this.city = city;
		this.state = state;
		this.zip = zip;
	}
}
```

```
public class ShallowCopyTest {
	public static void main(String[] args) {
	 Address addr = new Address("123","timberland","Atlanta","GA","30003");
	Employee emp1 = new Employee (100, "Suresh","Reddy", addr);
	Employee emp2 = null;
	try {
	  // Creating a clone of emp1 and assigning it to emp2
	   emp2 = (Employee) emp1.clone();
	} catch (CloneNotSupportedException e) {
	   e.printStackTrace();
	}
	// Printing the zip code of 'emp1'
	System.out.println(emp1.address.zip); //30003

	// Changing the zip code of 'emp2'
	emp2.address.zip = "30002";

	// This change will be reflected in original Employee 'emp1'
	System.out.println(emp1.address.zip); // 30002
   }
}
```

Deep cloning:

The deep copy of an object will have an exact copy of all the fields of source object like a shallow copy, but unlike sallow copy if the source object has any reference to object as fields, then a replica of the object is created by calling clone method.

This means that both source and destination objects are independent of each other. Any change made in the cloned object will not impact the source object.

Add the following changes in Address.java
1. implements Cloneable interface

2. Override clone() method.

```
protected Object clone() throws CloneNotSupportedException {
    return super.clone();
}
```

Replace below clone() method in Employee.java

```
protected Object clone() throws CloneNotSupportedException {
        Employee emp = (Employee) super.clone();
        emp.address = (Address) address.clone();
        return emp;
}
public class DeepCopyTest {
                public static void main(String[] args) {
                        Address addr = new Address("123", "timberland", "Atlanta", "GA", "30003");
                        Employee emp1 = new Employee(100, "Suresh", "Reddy", addr);
                        Employee emp2 = null;
                        try {
                                emp2 = (Employee) emp1.clone();
                        } catch (CloneNotSupportedException e) {
                                 e.printStackTrace();
                        }
                        System.out.println(emp1.address.zip); // 30003
                        emp2.address.zip = "30002";
                        System.out.println(emp1.address.zip); // 30003
                }
}
```

Call by Value:

It is a process in which the function parameter values are copied to another variable and instead this object copied is passed. This is known as call by Value.

Call by Reference:

It is a process in which the actual copy of reference is passed to the function. This is called by Reference.

```
public class CallByValue {
        int a = 10;
        void change(int a) {
                a = a + 50;
        }
        public static void main(String args[]) {
                CallByValue cv = new CallByValue();
                System.out.println("before change() invocation :" + cv.a);
                cv.change(500);
                System.out.println("after change() invocation " + cv.a);
        }
 }
```

output:

before change() invocation :10

after change() invocation: 10

```
public class CallByValueTest {
        int a = 20;

        void change(CallByValueTest cvt) {
                cvt.a = cvt.a + 100;// changes will be in the instance variable
        }
        public static void main(String args[]) {
                CallByValueTest cvt = new CallByValueTest();
                System.out.println("before change() invocation :" + cvt.a);
                cvt.change(cvt);// passing object
                System.out.println("after change() invocation :" + cvt.a);
        }
}
```

output:

before change() invocation :20

after change() invocation :120

Math class:

Java Math class provides several methods to work on math calculations like min(), max(), avg(), sin(), cos(), tan(), round(), ceil(), floor(), abs() etc.

Some of the methods:

Math.abs() : It will return the Absolute value of the given value.

Math.max() : It returns the Largest of two values.

Math.min() : It is used to return the Smallest of two values.

Math.round() : It is used to round of the decimal numbers to the nearest value.

Math.sqrt() : It is used to return the square root of a number.

Math.cbrt() : It is used to return the cube root of a number.

Math.pow() : It returns the value of first argument raised to the power to second argument.

Math.signum() : It is used to find the sign of a given value.

Math.ceil() : It is used to find the smallest integer value that is greater than or equal to the argument or mathematical integer.

Math.copySign() : It is used to find the Absolute value of first argument along with sign specified in second argument.

Math.nextAfter() : It is used to return the floating-point number adjacent to the first argument in the direction of the second argument.

Math.nextUp() : It returns the floating-point value adjacent to d in the direction of positive infinity.

Math.nextDown() : It returns the floating-point value adjacent to d in the direction of negative infinity.

Math.floor() : It is used to find the largest integer value which is less than or equal to the argument and is equal to the mathematical integer of a double value.

```
public class CalculatorExample {

  public static void main(String[] args) {
        int x = 6;
        int y = 4;
        // return the maximum of two numbers
        System.out.println("Maximum number of x and y is: " +Math.max(x, y));

        // return the square root of y
        System.out.println("Square root of y is: " + Math.sqrt(y));

        //returns 6 power of 4 i.e. 6*6*6*6
        System.out.println("Power of x and y is: " + Math.pow(x, y));

        // return the logarithm of given value
        System.out.println("Logarithm of x is: " + Math.log(x));
        System.out.println("Logarithm of y is: " + Math.log(y));
   }

}
```

Wrapper class:

The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.

Since J2SE 5.0, autoboxing and unboxing feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes:

Change the value in Method: Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.

Serialization: We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.

Synchronization: Java synchronization works with objects in Multithreading.

java.util package: The java.util package provides the utility classes to deal with objects.

Collection Framework: Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

Primitive Type and corresponding Wrapper class.

boolean -> Boolean

char ->Character

byte ->Byte

short ->Short

int ->Integer

long ->Long

float ->Float

double ->Double

Autoboxing:

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

Example:

```
int a=10;

Integer i=Integer.valueOf(a);//converting int into Integer explicitly

Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+i+j);
```

Unboxing:

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

```
Integer a=new Integer(3);

int i=a.intValue();//converting Integer to int explicitly

int j=a;//unboxing, now compiler will write a.intValue() internally

System.out.println(a+i+j);
```

Recursion:

A method in java that calls itself is called recursion.

It has two important condition.

1. Base condition
2. recursive statement

```
public class FactOfNumber {
	static int factorial(int n) {
		if (n == 1)
			return 1;
		else
			return (n * factorial(n - 1));
	}

	public static void main(String[] args) {
		System.out.println("Factorial of 4 :: " + factorial(4));
	}
}
```

Command Line Argument:
The java command-line argument is an argument i.e. passed at the time of running the java program.

The arguments passed from the console can be received in the java program and it can be used as an input.

Example:

```
public class CommandLineExample {
        public static void main(String[] args) {
                int a = Integer.parseInt(args[0]);
                int b = Integer.parseInt(args[1]);
                System.out.println("sum::" + (a + b));
        }
}
```