**Java OOPs Misc (JavaTpoint)**

**Topics:**

* **Object class**
* **Object Cloning**
* **Math class**
* **Wrapper Class**
* **Java Recursion**
* **Call By Value**
* **strictfp keyword**
* **javadoc tool**
* **Command Line Arg**
* **Object vs Class**
* **Overloading vs Overriding**

**Object class in Java**:

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.

Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:

Object obj=getObject();

|  |
| --- |
| The Object class provides many methods. They are as follows: |

|  |  |
| --- | --- |
| Method | Description |
| 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 in Java**

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.

**protected** Object clone() **throws** CloneNotSupportedException

**Advantage of Object cloning**

Although Object.clone() has some design issues but it is still a popular and easy way of copying objects. Following is a list of advantages of using clone() method:

You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.

It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.

Clone() is the fastest way to copy array.

**Disadvantage of Object cloning:**

* To use the Object.clone() method, we have to change a lot of syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() etc.
* We have to implement cloneable interface while it doesn't have any methods in it. We just have to use it to tell the JVM that we can perform clone() on our object.
* Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
* Object.clone() doesn't invoke any constructor so we don't have any control over object construction.
* If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.
* Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

class Student18 implements Cloneable{

int rollno;

String name;

Student18(int rollno,String name){

this.rollno=rollno;

this.name=name;

}

public Object clone()throws CloneNotSupportedException{

return super.clone();

}

public static void main(String args[]){

try{

Student18 s1=new Student18(101,"amit");

Student18 s2=(Student18)s1.clone();

System.out.println(s1.rollno+" "+s1.name);

System.out.println(s2.rollno+" "+s2.name);

}catch(CloneNotSupportedException c){}

}

}

=======================================================================

**Wrapper classes in Java: Javatpoint**

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 in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc.

**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. (Example in javatpoint)

**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.

The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

|  |  |
| --- | --- |
| Primitive Type | Wrapper class |
| boolean | [Boolean](https://www.javatpoint.com/java-boolean) |
| char | [Character](https://www.javatpoint.com/post/java-character) |
| byte | [Byte](https://www.javatpoint.com/java-byte) |
| short | [Short](https://www.javatpoint.com/java-short) |
| int | [Integer](https://www.javatpoint.com/java-integer) |
| long | [Long](https://www.javatpoint.com/java-long) |
| float | [Float](https://www.javatpoint.com/java-float) |
| double | [Double](https://www.javatpoint.com/java-double) |

**Q. why do we need wrapper classes?**

They convert primitive type to object and this is needed on internet to communicate between two application.

The classes in java.util package handle only objects and hence wrapper classes help in this case also.

=================================================================

**Call By value and call by reference in java**

There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value.

The changes being done in the called method, is not affected in the calling method.

class Operation2{

 int data=50;

 void change(Operation2 op){

 op.data=op.data+100;//changes will be in the instance variable

 }

   public static void main(String args[]){

   Operation2 op=new Operation2();

   System.out.println("before change "+op.data);

   op.change(op);//passing object

   System.out.println("after change "+op.data);   }  }

==========================================================

**Java Strictfp Keyword**

Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform. So, now you have better control over the floating-point arithmetic.

**Legal code for strictfp keyword:** The strictfp keyword can be applied on methods, classes and interfaces.

1. **strictfp** **class** A{}//strictfp applied on class
2. **strictfp** **interface** M{}//strictfp applied on interface
3. **class** A{

**strictfp** **void** m(){}//strictfp applied on method  }

**Illegal code for strictfp keyword:** The strictfp keyword **cannot** be applied on abstract methods, variables or constructors.

1. class B{

strictfp abstract void m();//Illegal combination of modifiers

}

1. class B{

strictfp int data=10;//modifier strictfp not allowed here

}

1. class B{

strictfp B(){}//modifier strictfp not allowed here

}

=============================================================

**Java Command Line Arguments**

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.

=================================================

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Difference between object and class**   |  |  |  | | --- | --- | --- | |  | **Object** | **Class** | | 1) | Object is an **instance** of a class. | Class is a **blueprint or template** from which objects are created. | | 2) | Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc. | Class is a **group of similar objects**. | | 3) | Object is a **physical** entity. | Class is a **logical** entity. | | 4) | Object is created through **new keyword** mainly e.g. Student s1=new Student(); | Class is declared using **class keyword** e.g. class Student{} | | 5) | Object is created **many times** as per requirement. | Class is declared **once**. | | 6) | Object **allocates memory when it is created**. | Class **doesn't allocated memory when it is created**. | | 7) | There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization. | There is only **one way to define class** in java using class keyword. | |
|  |

**Class:** Fruit **Object:** Apple, Banana, Mango, Guava wtc.

**Class:** Mobile phone **Object:** iPhone, Samsung, Moto

**Class:** Food **Object:** Pizza, Burger, Samosa

================================================================================

**Difference between method overloading and method overriding in java**

|  |  |  |
| --- | --- | --- |
| **No.** | **Method Overloading** | **Method Overriding** |
| 1) | Method overloading is used *to increase the readability* of the program. | Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class. |
| 2) | Method overloading is performed *within class*. | Method overriding occurs *in two classes* that have IS-A (inheritance) relationship. |
| 3) | In case of method overloading, *parameter must be different*. | In case of method overriding, *parameter must be same*. |
| 4) | Method overloading is the example of *compile time polymorphism*. | Method overriding is the example of *run time polymorphism*. |
| 5) | In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter. | *Return type must be same or covariant* in method overriding. |