Cloning

. Cloning is a process of creating a duplicate object from an exisitng object.

. When creating complex objects, cloning is used.

. Instead of creating an object from scratch, we can clone exisiting objec to create a duplicate object.

. Once an object is cloned, the original and duplicate objects acts independently. The changes made to the original object, doesn’t effect on duplicate object and vice-versa.



. For example, in a gaming application, we need multiple enemies with the same characterstics. So, we create one enemy object, and we clone this object to create multiple enemies.

How to clone an object?

 . first implement a class, whose objects requires cloning from java.lang.Cloneable marker interface.

 . we have to override clone() method of Object class, in that class.

//example

**package** com.ashokit.cloning;

**class** ClassA **implements** Cloneable {

 **private** **int** x;

 **private** **int** y;

 ClassA(**int** x, **int** y) {

// System.out.println("ClassA(int,int): constructor");

 **this**.x = x;

 **this**.y = y;

 }

 //Here, you are overriding clone() method of Object class

 @Override

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

 **return** **super**.clone();

 }

 //Here, you are overriding toString() method of Object class

 @Override

 **public** String toString() {

 **return** "ClassA [x=" + x + ", y=" + y + "]";

 }

 **public** **int** getX() {

 **return** x;

 }

 **public** **void** setX(**int** x) {

 **this**.x = x;

 }

 **public** **int** getY() {

 **return** y;

 }

 **public** **void** setY(**int** y) {

 **this**.y = y;

 }

}

**public** **class** Main {

 **public** **static** **void** main(String[] args) **throws** CloneNotSupportedException {

 ClassA ca = **new** ClassA(10, 20);

 ClassA clonedCa = (ClassA) ca.clone();

 System.***out***.println("original object : " + ca);

 System.***out***.println("cloned object : "+ clonedCa);

 //make changes to original object

 ca.setX(30);

 ca.setY(40);

 System.***out***.println("==============================");

 System.***out***.println("After making changes in original object");

 System.***out***.println("original object : " + ca);

 System.***out***.println("cloned object : "+ clonedCa);

 }

}

output:

original object : ClassA [x=10, y=20]

cloned object : ClassA [x=10, y=20]

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

After making changes in original object

original object : ClassA [x=30, y=40]

cloned object : ClassA [x=10, y=20]

Types of cloning:

1. shallow cloning
2. deep cloning

. In shallow cloing, when a dependent object is cloned, its dependency object will not be cloned. It means, both original and cloned dependent objects will share the same dependency object.

. In deep cloning, when a dependent object is cloned, its dependency object is also cloned. So, both original and cloned dependent objects have separate dependency objects.

. In shallow cloning, dependent class implements Cloneable interface, but dependency class doesn’t.

. In deep cloning, both the classes implements Cloneable interface.



// shallow cloning example

**package** com.ashokit.cloning;

//dependent class

**class** ClassA **implements** Cloneable {

 **private** **int** x;

 **private** **int** y;

 **private** ClassB classB;

 ClassA(**int** x, **int** y) {

// System.out.println("ClassA(int,int): constructor");

 **this**.x = x;

 **this**.y = y;

 classB = **new** ClassB(100, 200);

 }

 @Override

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

 **return** **super**.clone();

 }

 @Override

 **public** String toString() {

 **return** "ClassA [x=" + x + ", y=" + y + ", classB=" + classB + "]";

 }

 **public** **int** getX() {

 **return** x;

 }

 **public** **void** setX(**int** x) {

 **this**.x = x;

 }

 **public** **int** getY() {

 **return** y;

 }

 **public** **void** setY(**int** y) {

 **this**.y = y;

 }

 **public** ClassB getClassB() {

 **return** classB;

 }

 **public** **void** setClassB(ClassB classB) {

 **this**.classB = classB;

 }

}

//dependency class

**class** ClassB {

 **private** **int** a;

 **private** **int** b;

 ClassB(**int** a, **int** b) {

 **this**.a = a;

 **this**.b = b;

 }

 @Override

 **public** String toString() {

 **return** "ClassB [a=" + a + ", b=" + b + "]";

 }

 **public** **int** getA() {

 **return** a;

 }

 **public** **void** setA(**int** a) {

 **this**.a = a;

 }

 **public** **int** getB() {

 **return** b;

 }

 **public** **void** setB(**int** b) {

 **this**.b = b;

 }

}

**public** **class** Main {

 **public** **static** **void** main(String[] args) **throws** CloneNotSupportedException {

 ClassA ca = **new** ClassA(10,20);

 ClassA clonedCa = (ClassA) ca.clone();

 System.***out***.println("original object : " + ca);

 System.***out***.println("cloned object : " + clonedCa);

 System.***out***.println("======================================");

 //making a change in the dependency object through original dependent object

 ca.getClassB().setA(99);

 ca.getClassB().setB(199);

 System.***out***.println("After changing dependency object, through original dependent object");

 System.***out***.println("original object : "+ ca);

 System.***out***.println("cloned object : " + clonedCa);

 }

}

output:

original object : ClassA [x=10, y=20, classB=ClassB [a=100, b=200]]

cloned object : ClassA [x=10, y=20, classB=ClassB [a=100, b=200]]

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

After changing dependency object, through original dependent object

original object : ClassA [x=10, y=20, classB=ClassB [a=99, b=199]]

cloned object : ClassA [x=10, y=20, classB=ClassB [a=99, b=199]]

//deep cloning example.

**package** com.ashokit.cloning;

//dependent class

**class** ClassA **implements** Cloneable {

 **private** **int** x;

 **private** **int** y;

 **private** ClassB classB;

 ClassA(**int** x, **int** y) {

// System.out.println("ClassA(int,int): constructor");

 **this**.x = x;

 **this**.y = y;

 classB = **new** ClassB(100, 200);

 }

 @Override

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

 ClassB clonedB = (ClassB) classB.clone();

 ClassA clonedA = (ClassA)**super**.clone();

 clonedA.setClassB(clonedB);

 **return** clonedA;

 }

 @Override

 **public** String toString() {

 **return** "ClassA [x=" + x + ", y=" + y + ", classB=" + classB + "]";

 }

 **public** **int** getX() {

 **return** x;

 }

 **public** **void** setX(**int** x) {

 **this**.x = x;

 }

 **public** **int** getY() {

 **return** y;

 }

 **public** **void** setY(**int** y) {

 **this**.y = y;

 }

 **public** ClassB getClassB() {

 **return** classB;

 }

 **public** **void** setClassB(ClassB classB) {

 **this**.classB = classB;

 }

}

//dependency class

**class** ClassB **implements** Cloneable {

 **private** **int** a;

 **private** **int** b;

 ClassB(**int** a, **int** b) {

 **this**.a = a;

 **this**.b = b;

 }

 @Override

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

 **return** **super**.clone();

 }

 @Override

 **public** String toString() {

 **return** "ClassB [a=" + a + ", b=" + b + "]";

 }

 **public** **int** getA() {

 **return** a;

 }

 **public** **void** setA(**int** a) {

 **this**.a = a;

 }

 **public** **int** getB() {

 **return** b;

 }

 **public** **void** setB(**int** b) {

 **this**.b = b;

 }

}

**public** **class** Main {

 **public** **static** **void** main(String[] args) **throws** CloneNotSupportedException {

 ClassA ca = **new** ClassA(10,20);

 ClassA clonedCa = (ClassA) ca.clone();

 System.***out***.println("original object : " + ca);

 System.***out***.println("cloned object : " + clonedCa);

 System.***out***.println("======================================");

 //making a change in the dependency object through original dependent object

 ca.getClassB().setA(99);

 ca.getClassB().setB(199);

 System.***out***.println("After making changes to the dependency object, through original dependent object");

 System.***out***.println("original object : "+ ca);

 System.***out***.println("cloned object : " + clonedCa);

 }

}

output:

original object : ClassA [x=10, y=20, classB=ClassB [a=100, b=200]]

cloned object : ClassA [x=10, y=20, classB=ClassB [a=100, b=200]]

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

After making changes to the dependency object, through original dependent object

original object : ClassA [x=10, y=20, classB=ClassB [a=99, b=199]]

cloned object : ClassA [x=10, y=20, classB=ClassB [a=100, b=200]]

 serialization & deserialization

* serialization is a process of converting a Java object into a stream of bytes, which can be easily saved in a file, transmitted over a network or stored in a database.
* deserialization is a reverse process of serialization. The stream of bytes are collected from a file, or network, or a database and converted back to a Java object.
* we serialize an object, when we want to save the state of an object into a file, or we want to send an object to another machine.
* To serialize a Java object, the class should implement a marker interface, java.io.Serializable.
* We use ObjectOutputStream class for serialization and ObjectInputStream class for deserialization.
* The writeObject() of ObjectOutputStream class converts a Java object into a stream of bytes.
* The readObject() of ObjectInputStream class converts a stream of bytes into a Java object.



// A demo on serialization

 **package** com.ashokit.demo;

**import** java.io.FileInputStream;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutputStream;

**import** java.io.Serializable;

**class** Employee **implements** Serializable {

 **int** empno;

 String ename;

 Employee(**int** empno, String ename) {

 **this**.empno = empno;

 **this**.ename = ename;

 }

 @Override

 **public** String toString() {

 **return** "Employee [empno=" + empno + ", ename=" + ename + "]";

 }

}

**public** **class** Main {

 **public** **static** **void** main(String[] args) {

 Employee emp = **new** Employee(7878, "Scott");

 //serialization

 **try** ( FileOutputStream fileOut = **new** FileOutputStream("D:\\employee.ser");

 ObjectOutputStream out = **new** ObjectOutputStream(fileOut)) {

 out.writeObject(emp);

 }

 **catch**(IOException ie) {

 ie.printStackTrace();

 }

 //deserialization

 Employee deserializedEmp = **null**;

 **try** ( FileInputStream fileIn = **new** FileInputStream("D:\\employee.ser");

 ObjectInputStream in = **new** ObjectInputStream(fileIn)) {

 deserializedEmp = (Employee) in.readObject();

 }

 **catch**(IOException | ClassNotFoundException ex) {

 ex.printStackTrace();

 }

 System.***out***.println("Deserialized Employee : "+ deserializedEmp);

 }

}

what is serialVersionUID?

 . It is a unique identifier for each class that implements Serializable interface.

 . It must be declared as a private static final long variable.

 private static final long serialVersionUID = 1L;

 . when an object is serialized, the serialVersionUID is also included in the stream of bytes.

 . At the time of deserialization, JVM checks the id of the class and the id in the stream of bytes. If they match then only deserialization will proceed.

 . Otherwise, JVM throws InvalidClassException.

what is transient keyword?

 . If you want to stop any attribute of a class from serialization then you have to declare that variable as transient.

 . when an object is serialized, the transient variable value will be stored as null, into the file.

 . transient keyword can be used with only variables. Not with classes, interfaces or methods.

For example,

 public class User implements Serializable {

 private String username;

 private transient String password;

 //methods

 }

( Please also refer Externalizable interface )

enum keyword:

-------------

 . enum stands for enumeration.

 . enum keyword is used to create a special data type.

 . When you want to create a variable and that should be allowed to

 store only one of the fixed set of constants as a value, then you

 should create a special data type with enum keyword.

 ex:

 public enum Day {

 MONDAY, TUESDAY, WEBNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY

 }

 . The constants in enum, are implicity static and final. So, we write the constants in upper case, but it is not a strict rule.

 . You can create a variable of enum type as,

 Day day;

 . The variable can store one of the constants.

 day = Day.MONDAY;

 . Enums can also have fields, constructors and methods.

ex:

 Gender.java

 **public** **enum** Gender {

 ***MALE***("Male Person"), ***FEMALE***("Female Person"), ***OTHERS***("Other persons");

 **private** String str; //field

 Gender(String str) { //constructor

 **this**.str = str;

 }

 **public** String getStr() { //method

 **return** str;

 }

 **public** String sayHello() { //method

 **return** "hello";

 }

}

TestClass.java

**public** **class** TestClass {

 **public** **static** **void** main(String[] args) {

 Gender gender = Gender.***MALE***;

 System.***out***.println(gender.sayHello()+ " : "+gender.getStr());

 Gender gender2 = Gender.***FEMALE***;

 System.***out***.println(gender2.sayHello()+ " : "+gender2.getStr());

 //values(): returns an array of all enum constants

 **for**(Gender g : Gender.*values*()) {

 System.***out***.println(g);

 }

 }

}

JVM internal components:

1. class loader
2. runtime data areas
3. execution engine
4. Java Native Interface(JNI)
5. Native Method libraries

. Class Loader is responsible for loading class files into JVM.

. Before JDK11, we have 3 types of class loaders.

1. Bootstrap class loader: loads the classes from Java API
2. Extension class loader: loads the classes from extensions

 directory( jre/lib/ext).

1. Application class loader: loads the classes from the

 application classpath

 . From JDK11, we have 2 types of class loaders.

1. Bootstrap class loader
2. Application class loader

 . Runtime data areas are,

 1.method area: stores the class level information.

 static variables, class structures and methods are

 stored in this method area.

 2.heap area: stores objects and arrays.

 3.stack area: stores local variables, partial results of threads.

 4.PC registers: Each thread has a Program Counter(PC), it holds the

 address of the current instruction being executed by

 the thread.

 5.Native method stack: Manages native(non-Java) method calls.

 . Execution Engine: It is responsible for reading byte code and executing it.

 . Execution engine contains,

1. interpreter
2. Just-In-Time compiler(JIT)
3. Garbage Collector(GC)

 . JNI allows the JVM to interact with native applications written in other languages like C/C++.

 . Native method libraries are OS specific libraries and these will perform some tasks related system resources. JVM will access these libraries via JNI.