Inheritance and constructor chaining

- The reason behind inheritance is reusability , if something is in the parent class , just write a child class extending it with added features/functionalities . You save time by doing this as the parent class was already a tested and functional class and you need not write everything from scratch.
- The keywords and and understanding of inheritance has also been disucssed in Inheritance basics in slide 2.
- If there are two classes A and B and class A is extending class B , then first constructor of class A gets executed and then the constructor of class B gets executed. This is because compiler provides super() as the first statement as an implicit statement in all the constructors which makes a call to the constructor of the parent class.

exercise

- Execute and tweak the code ConstructorCallingSequenceInInheritanceDemo.java and InheritanceDemo.java inside com.musigma.javatraining.basic package from the github repo : https://github.com/Mu-Sigma/poc.git in JavaTraining branch.

- Try to change print statements and see the output and understand.

Abstract class

- Classes which are not ready yet to be initialized are made abstract classes by prefixing keyword “abstract” before the class itself. A class marked as abstract can’t be instantiated.An abstract class is put to use / realised by the instances/objects of it’s subclasses.
- An animal class doesn’t stand sense if instantiated , can we visualise an animal entity walking around us ? How will they look , i can imagine a dog by my side playing around but picture gets blurred if i try to visualize an animal.

Code snippet for abstract class

- Sample code for abstract class , an abstract class can be having abstract or non abstract methods . Abstract methods are the methods with no body , only declarations.

```
package javatutorial;
// code snippet for an abstract class
abstract public class AbstractDemo {
abstract public void show();
}
```

- If we try to instantiate the above abstract class it will give us an error.

Problem with abstract class

- Suppose we want to have an abstract class in which we want to override methods of two different classes , so we need to extend them . But java stops us to extend two different classes at a time . Reason being if there is same method in both the classes and we call that common method on the object of the child , then jvm gets confused.
- What is the solution to it ? Interface . In interface we can extend two interfaces and then we can implement the declared functions of both the interfaces in the implementing class.

Design issues between interface and abstract

- If interface is better than abstract class as we just discussed , then why do we need abstract classes ? At times we also want to reuse the existing methods of a class and also want to override it’s any function as per our need then we go for abstract classes.
- If we don’t have inheritance related needs as mentioned above , then interface is a cleaner solution in terms of design issues.
- Interfaces and abstract classes are higly leveraged in dynamic polymorphism - a style that is very commonly used and we will be learning next.

Polymorphism

- By polymorphism we mean that same method name but different functionalities.

  There are **two types** of polymorphism :

  - 1. Static polymorphism / overloading (Type checking) : this means that what are the types and the number of the arguments of the method.
  - 2. Dynamic polymorphism / overriding (Content based checking) : this means that what is the content inside the reference of the parent class/interface.

Overloading

- Definition : The method name remains same but the signature of the method changes which include different types and number of arguments .

- Example :

```
public void run (String str)
public void run ( Integer intValue)
```

- Above two mentioned methods are overloaded methods.The type of argument is taken into account , thus we say type checking has happened.

Dynamic polymorphism

- At times it so happens that we are dissatisfied or want to improve the functionality provided in a method of the parent class . This situations arises the need of dynamic polymorphism (run time polymorphism).
- We extend the class and rewrite the same method with the same signature (ie. the name , number and type of arguments of a function) but with a different implementation (content/body/logic) of the method which is called as overriding.
- A reference is created for the base class and then the the object of the parent class is inserted to it . A content check is done on the reference that which object is it carrying at the run time . The method of the same object gets called.

Interface polymorphism

- A reference to the interface is created and then the objects of the implementing classes of the interface are created . Suppose there is a method declared in the interface , then we know now that the classes have to override that method.
- Then the object of one of the class is assigned to the reference of the interface and then when we call the method , the method defined in the implementing class whose object was assigned gets called.
- Similarly it would happen with the second implementing class and it’s object.

Inheritance polymorphism

- We create the reference to the base class and say there is a method in the base class . Then let’s we have have two classes extending the parent class and they are overriding the method of the base class in their own way .
- We create a reference to the base class and then assign the object of the child class . Whichever child object is injected to the reference of the parent class , that particular child class method implementation gets called.
- This happens by the virtue of content type checking , the type of the object which is assigned to the parent class reference is what determines which class’s methods are going to be called.

Exercise

- Run and tweak the codes : PolymorphismByInterface , PolymorphismByOverloading and PolymorphismByOverriding in com.musigma.javatraining.basic package on git repo : https://github.com/Mu-Sigma/poc.git in JavaTraining branch.
- Write one interface , a method to be overridden and override the method differently in two different implementing classes and call them.

Packages and keywords (reserved meanings)

- Classes are packaged logically inside packages . Ex : com.musigma.ird.bigdata
  can be the package name for java classes of some common type.
- There are four modifiers to instance variables and methods as follows :
- public : the instance variables/methods are visible acorss all the packages.
- private: the instance variables / methods are visible only inside the class.
- protected : the instance variables/methods are visible also in the extending class.
- default : the instance variables/methods are visible in the same package.
- If an instance variable is marked static , it has a value per class not per object of the class . To be more clear if the variable is static and assigned a value of 1 , then this would carry the same value across all the objects of the class.

keywords

- final : if a method is marked as ‘final’ , then the methods can no more be overridden. If a variable is marked final , similarly it’s value can’t be changed throughout the execution .

Exception handling (try - catch)

- At times it so happens that despite the write code , there are uncertainties that it might not work due unpredictable reasons like the file assuming which you wrote the code has been removed from the server just before run time.
- We write that suspicous code inside try block and do the handling in the catch block which needs to be written subsequently after the try block , else it would be an error. Sample code is written as follows :

```
try{
// suspicous code
}
catch{
// error handling occurs
}
```

finally

- We use finally keyword for some operations that are to be done anyways whether an exception is met or not . It’s applicable to usecases where you want to deallocate the resources which were withheld for sure shot use by other similar clients.
- Example: it we are writing a java database connectivity code or socket connection code , then we must close it so that it may release the resource after the operation is done.

flow control

- Instead of the program getting crashed while meeting a run time exception , the control goes back to the statement following the catch block when an exception is met during the execution of code inside try block. It saves from our application getting crashed in production environments and captures the error message in catch which can be emailed to the concerned team.
- throws : Any method in which the suspicous code has been written can declare it’ s risky behaviour by declaring on the top that it throws an exception. Sample code is as follows :

```
public void riskyMethod throws Exception(){

}
```

throw , Exception hierarchy

- throw keyword is used to throw back an object of the custom exception created. A custom exception can be created by extending Exception class . It can be caught and handled accordingly.Exception is the parent of all Exception classes.
- Exception hierarchy : after a try statement we can place a catch block which can catch Exception error or a catch block which can catch a more precise Exception. It’s a rule to place the catch with smallest Child class of Exception as the argument so that we can catch the precise exceptions as far as possible .

```
try{
}
catch(NullPointerException e){
}
catch(Exception exception) { } ;
```

Checked / unchecked exceptions.

- The compiler is intelligent enough to give a compile time error if it sees you doing a file i/o operation without any try or catch . This is called a checked exception where it forces you to write your code under try-catch.
- If some number arithmetic is there and inside somewhere you divide any number by zero . Compiler can’t suggest you to put this under try - catch . The developer is expected to write a try-catch for any suspicion , thus it’s called uncatched exception.

Passing the exception caught and ducking it.

- Any exception that has been thrown by a method needs to be the method calling it at some point of time.
- It can be handled by calling the erroneous method in a try block and then catching it or else you can again mark the calling method by declaring it throws an exception and pass the buck.
- This buck needs to be handled somewhere , if not handled it brings the jvm down.

Exercise

- Execute and tweak the code TryCatchDemo class inside com.musigma.javatraining.exception package at the github repository :

File Handling - buffer concept

Mu Sigma

- File input / output happens through java stream and goes through the buffer which is the heap memory allocated . Stream is the java object converted in bytestream which is a stream of bytes (series of 1s and 0s).
- Buffer concept : at times files are big enough so that it can’t be handled at a time in the memory . So chunk/portion of the file is brought to the memory and processed there which is called the buffer concept.
- Buffer concept is similar to that while shopping we use trolley to ship our huge shopping items from the billing desk to our car trunk . We make several trips using the trolley but don’t overload and crash ourselves . The trolley is the buffer here.

Reading from a file

Mu Sigma

- Reading from a file happens in the process that first the contents of file is read from the specified path and then read in buffer .
- Then it can be printed on the console. Sample code to read a file is as follows :

```
// sample code for reading a file content
try {
File file = new File("SongList.txt");
BufferedReader reader = new BufferedReader(new FileReader(file));
String line = null;
while ((line= reader.readLine()) != null) {
System.out.println(line);
}
} catch(Exception ex) {
ex.printStackTrace();
}
```

Writing to a file

Mu Sigma

- Contents to the file are first written to a buffer which is written to a file object .
- Then finally the file object is converted into a real file on the specified path .
  Following code snippet for the same is as follows :

```
try{
        // Create file
        FileWriter fstream = new FileWriter("D:/writeFile.txt");
        BufferedWriter out = new BufferedWriter(fstream);
        out.write("Hello Java"+"\n"+"Hello big data");
        // Close the output stream
        out.close();
  }catch(Exception e){//Catch exception if any
        System.err.println("Error: " + e.getMessage());
  }
```

Serialization

Mu Sigma

- The process of capturing the state of an object is called Serialisation . What happens is that the state of an object (ie. the value of the instance variables) of the object along with some additional information which is the class type gets stored in the serialized object which can be later used by the jvm to deserialize (reproduce) the same object on the heap.
- Underlying concept is that two objects of the same class only vary in their values of their instance variables while their class name/type remains the same.
- When you serialize a java object residing on the heap , the values of the instance variables are sucked into a byte stream along with class type/name which is persisted . The same serialized object which was persisted is referred and deserialized by the jvm using the information contained by the serialized object.

Serialized object - procedure and contents.

Mu Sigma

- JVM constructs another object using the information contained by the serialized object from the persisted file but at another/new address on the heap .
- Serialized object contains the data of the instance variables of the class.
- Code snippet to serialize an object

```
FileOutputStream fs = new FileOutputStream("file.ser"); // file.ser is the file in hard drive where serialization will happen.
ObjectOutputStream os = new ObjectOutputStream(fs); // the object steam which is ready to be persisted.
os.writeObject(importantFileOjbect); // the object which we want to persist
os.close()
```

Deserialize a file

Mu Sigma

- Serialization is done only with a point of view to deserialize it sometime and get our domain objects back in the same state in which they were persisted.
- Deserialization is getting back the objects with the same state in which they were serialized/persisted.

```
// if you want to deserialize a serialized object from file.ser
FileInputStream fileStream = new FileInputStream("file.ser"); // reading the contents from file.ser
ObjectInputStream os = new ObjectInputStream(fileStream); // getting a object input stream
Object one = os.readObject(); // getting the parent java Object ( parenet of all class )
ImportantFile importantFileObject = (ImportantFile) one; // typecasting to get our domain objects back.
```

Cascading in serialization

Mu Sigma

- Suppose if there is a Car class and internally it is having Seat class which in itself is a class , then the question arises that whether the Seat class will also be serialized.
- In serialization , either all or none happens . If it has to be serialized , then all the objects of the referred classes also gets persisted.

Exercise

Mu Sigma

- Refer to the WriteToFile code in com.musigma.javatraining.basic package in git repo : https://github.com/Mu-Sigma/poc.git in JavaTraining branch , then run and tweak it.
- Serialize an object into a text file on your hard drive and then deserialize the same and print the contents on the console.

Multithreading - why we need them ?

Mu Sigma

- At times it so happens that suppose we are working on a chat client and our java process is busy sending the messages that we are writing but there also has to be a parallel process to keep reading the incoming messages so they may appear on the screen.
- As we know that a method keeps on executing on a single stack , so for faster and parallel execution we need that our jvm running on heap may do method execution on multiple stacks instead of a single main stack dedicated to our java main method.
- Each of those stacks when executing different methods of the same jvm are called threads . Every distinct executing stack is a thread of execution.

How to spawn threads ?

Mu Sigma

- Thread thread = new Thread() is the way to create a user thread. But this dies out the moment it's created as it has no job to do.
- A class implements Runnable interface and overrides run method necessarily which is the job . An object of this class is passed to the Thread constructor which also gives it a job when an object of the Thread gets created.

```
Thread thread = new Thread(runnableObject);
```

- Multiple threads for the same job can be created for parallel execution of job.

Difference between thread object and executio...

- Thread object does not do anything , it just rests on the heap.
- When thread.start() is called , then actual execution starts and the job allotted to the thread/threads start getting executed.
- If there are two methods , one main method and any other method and a thread is allocated to the other method , then there will be two stacks getting executed , one stack for main() method and other stack for run() method of the job allocated to that thread.
- Suppose there are three threads allocated the same job , then the same method run() is on the top of the stack of all three stacks and keep getting executed.

Thread sleep

Mu Sigma

- When we want to guarantee that a thread should definitely run , then we have to make other thread sleep . This is the only way thread scheduler doesn’t work on the sleeping thread and allows other thread to get executed.
- There is no particular order in which thread scheduler tells the stacks to get executed.
- thread.sleep(2000) is used to make a thread to sleep and the time is mentioned in units of milliseconds.

Thread scheduler

Mu Sigma

- Thread scheduler internally decides that which thread or stack should be allowed to execute . There are no definite algorithms in the developer hand so that it can be controlled.
- thread.sleep() is the only way to handle this situation.

Dirty read/write due to sleep

Mu Sigma

- Suppose there is a method related to bank money withdrawal and there are two persons handling the same account (joint account say) . Both of them can be considered threads here.
- Suppose one person checked the balance , saw it ok , slept and the other person emptied the account . Now when the person wakes up , he/she withdraws as before going to sleep account was loaded with money . This gives over withdrawn crisis which is a result of dirty read/write.

synchronized keyword

Mu Sigma

- If we want to make the previous case thread safe , then we use ‘synchronized’ keyword before the withdrawal() method .
- What ‘synchronized’ keyword does is that it allows a thread to execute completely and doesn’t allow any other thread to interrupt execution in mid of the execution of method.
- It does not allow dirty read/write conditions.

Deadlock

Mu Sigma

- Suppose there are two threads A and B. One thread A was executing and it slept for some time . Say another thread B which was executing wanted the resource which thread A was withholding . So thread B stopped executing and went to sleep . Now when thread A wakes up it suppose wants a resource withheld by B but B is already sleeping waiting for A to release it’s resource .
- Above is a deadlock situation

Exercise

Mu Sigma

- What is the differnce between thread object and thread of execution ?
- Run MultipleThreadsDemo class inside com.musigma.javatraining.threads from the code at git repo : https://github.com/Mu-Sigma/poc.git in JavaTraining branch.