**Abstract Method:** A method it **doesn’t have any implementation** and forces child class to implement the method.

**Abstract class:** A class that has 0 or more abstract and non-abstract methods.

Also To restrict the object creation for the class. **What is the use of constructor then? :** To **initialize parent abstract class instance variables** when **child class object gets created**.

**Ex: HttpServlet**:all are concrete methods still we can declare abstract to restrict object creation.

Child class for an abstract class should provide implementation for all abstract methods otherwise the child class also should be declared as abstract.

**Note**: **It should be parent class** not a normal class bcz object creation not possible.

**Restrict Abstract with others**: final, static, private, synchronized, native, strictfp.

* Final classes can’t contain abstract methods but abstract classes can contain final methods.
* For Static methods compulsory we should provide complete implementation but for Abstract methods we can’t provide implementation hence abstract and static combination is illegal for methods.

**Final Method**: To restrict the overriding of parent class method in child class (inheriting is possible).

**Final class:** To restrict the inheriting of a class.

**Final variable(constant**): To restrict the reassignment(change) of a variable. Need to be initialized in constructor (instance block) or static block (final static)

**Access Modifiers:**

**Private:** private members can be accessed **with in the class.**

**Default:** can be accessed with in the package

**Protected:** can be accessed from the same package, if different package then only child class can access using child class reference only. but in the same package we can access protected members either by using parent class reference or by child class reference.

**Public:** public members can be accessed from anywhere.

**Variables: instance, static, local**

**Note**: The default instance and static variables initialization happens on **compilation time.**

{int a} --> local variables compulsory we should perform initialization before using, if not used then no need to initialize

int i\_v; --> instance variable by default initialized.

static int s\_v; --> static variables by default initialized

final int f\_v; --> **not initialized by default** must initial in instance block or constructor(initialization before completion of constructor)

final static int f\_s\_v; --> **not initialized by default** must initial in static block.

{final int bb;} -> If we are not using local variable then no need of perform initialization even though it is final.

{final int kk;} -> For the local variables the only applicable modifier is final.(no other like static, private, public, synchronized)

**Static (Memory and time):** Can applywith methods and variables. cannot declare static on top of a class but can declare as inner class.

**Static variable:** only one memory allocated for the variable irrespective of number of objects created for the class. So memory and time save.

**Static methods:** We can directly call static methods with class Name. No need to create instance for calling the static methods. So memory and time save.

* **Instance variables and methods cannot be accessed from static area. But Static members can be accessed from any where**
* Static blocks called at the time of class loading at that time instance members wont be available.
* Overloading, inheritance is possible for static methods.
* **Method hiding:** overriding is not applicable for static methods but seems to be overriding is possible, but it is method hiding.
* **Data hiding:** Restricts the access to the data members, and it can be accessible only through the member functions of the class (private, public)

**Synchronized: a**pplicable only for methods and block. **If a method declared** as synchronized **at a time only one thread is allowed** to execute the method at the same time on the given object. (critical-section)

* Advantage of synchronized keyword is to overcome **data inconsistency problem** and other thread interferences.
* Use of synchronized keyword may effect performance of the system.
* Synchronized and abstract combination is illegal for the methods.

**Transient:** applicable **only for variables**. Transient variables **won’t participate in the serialization** and **deserialization**. Ex: i=10==>10,transient int i = 10;==>0

Transient means not to serialize. (cannot be saved to file or transferred over network when serialized an object)

**JVM** **ignores the value** of transient variable and **saves it’s default value.** Transient int i=10; => 0

**Serialization:** converting an object from java supported format to network or file supported format. (Object ---🡪 file or n/w supported file.)

By using FileOutPutStream("filePath"), ObjectOutPutStream.writeObject(Object)

* Static variables never part of object state hence they won’t participate in serialization process.
* Declaring a static, final variable with transient there is no impact.

transient static int i = 10;=>10 , transient final int j = 20;=>20

**Deserialization:** Converting an object from network supported format or file supported format to java supported format. (File or n/w -🡪 object)

By using FileInputStream("path"), Object=ObjectInputStream.readObject() we can achieve deserialization.

* We can perform serialization only for serialized objects (So need to implement Serializable)
* Serializable interface doesn’t contain any method; it is marker interface.

1. private static final long serialVersionUID=1L:warning
2. It is 64 bit hash of class name ,interface, methods and field name
3. all the parameters taken into consideration to generate the serial versionID.
4. bydefalt we get even if you dont add ,serialized stream has :name of the class and serialVersionUID
5. InvalidClassException == if any changes in the class after serialization (object stream serialId and deserilized object Serialid compared)
6. **If drastic change in class then chnage serialId in class and then serialize .**
7. **if only small chnge** then **dont change serialId.**(should be present in class otherwise new one created and above exception)

**Marker interface**: **It is an empty interface (no field or methods).** We are giving indication to JVM that to treat the marked classes specially in order to support some special functionality.

Ex:Serializable, Clonnable and Remote,ThreadSafe interface.

Serializable==>To support serialization and deserialization

Clonable==>To support object cloning

* Annotation is better choice than marker interface and JUnit is a perfect example of using Annotation e.g. @Test for specifying a Test Class.
* A marker interface will mark all subclasses of the marked class. E.g. if we have to mark a class non serializable then we have to specifically mark it is as transient
* By implementing an interface if our objects will get some ability then Marker

Ex:Serializable, Clonable interfaces are marked for some ability.

* Even though an interface contains some methods by implementing that interface if our objects will get some ability.

**Ex**: Comparable, Runnable

**Volatile:** applicable **only for variables**. **if value of a variable keeps on changing then we have to declare that variable as volatile.** Why ? **The values of volatile variable will never be cached and all writes and reads will be done to and from the main memory.**

* Normally JVM will create a separate local copy for every thread, All the intermediate calculation performed by that thread will be referred in local copy instead of master copy, once the value got finalized just before terminating the thread the local copy value updated in the master copy. So that remaining threads will always get a stable value.
* To overcome “Data Inconsistency” problems.
* Volatile and final is illegal combination for the variables.

**Without Volatile:** If two threads run on different processors each thread may have its own local copy of shared Variable. If one thread modifies its value, the change might not reflect in the original one in the main memory instantly. This depends on the write policy of cache. Now the other thread is not aware of the modified value which leads to data inconsistency.

**guarantees that all reads of a volatile variable are read directly from main memory, and all writes to a volatile variable are written directly to main memory.**

volatile is enough only when there is one writer thread and remaining all are reader thread, but if we have **multiple writes** then **volatile+synchronized**

**Singleton-multi-threaded problems:**

An instance variable modification in a non-synchronized method leads to data inconsistency issue.

1.Don't use instance variables in multithreaded environment. OR

2.Use synchronized block/keyword on methods wherever the instance variables are modified to avoid unexpected results.

**Interface**: can have abstract, default & static methods, can declare object methods too.

Can achieve 100% abstraction, as all the methods declared inside the interfaces are abstract.

allowed modifiers for interface: public, abstract, default

**Interface methods:**

* a class implementing an interface should provide the implementation for all the interface methods or declare it as abstract class
* Whenever we are implementing an interface method we should declare that method as public otherwise compile time error.
* A class can extends only one class at a time. But an interface can extends any no of interfaces simultaneously.
* an interface can’t implement another interface, it extends.
* every Interface method is **by default public** and **abstract** whether we are declaring or no
* Not allowed: private, protected, static, final, native, strictfp, synchronized.
* If two interfaces contain a method with same signature and same return type in the implementation class, only one method implementation is enough.
* If two interfaces contain a method with the same name but different arguments. In the implementation class compulsory we should provide implementation for both methods and these methods act as overloads method.

**Interface variables:** An interface can contain variables also every interface variable is **by default public static and final** whether we r declaring or not.

public static final int i = 10<===> int i=10

From the implementation classes we r **allowed to access** interface variables but we r **not allowed to change** their values i.**e. reassignment is not possible because these are final**

**Abstract vs Interface:** Abstract classes was considered easy to evolve as it could add new methods and provide default implementations to those methods

|  |  |
| --- | --- |
| 0 or more abstract methods and non-abtracts methods | Only abtract, default, static and can declare Object methods |
| Normal Instance and static variables | Should be public static final variables(constants) |
| Has Constructor to initialize abstract class instance variables | No constructor |

* Abstract class can have final, non-final, static and non-static variables. The interface has only static and final variables.
* Before 1.8 Interfaces was not considered easy to evolve as, in the case of adding new methods to an interface, all the implementing classes had to change to provide implementation for the new methods.
* **With java 8 interfaces can have default methods so that issue has been addressed(solved).**

**Exceptions:**

**Exception**: is an event that disturbs the normal flow of execution. It is an object which is thrown at runtime.

* java.lang.Throwable is super class for all exception and errors.
* Whether the exception is checked or unchecked it always occurs at runtime only.
* If there is no chance of raising an exception in try statement then we r not allowed to maintain catch block for that exception violation leads to compile time error but this is applicable only for fully checked exceptions. Ex: IOException
* Statements after finally will be executed if exception handled with proper exception Handler.
* The whole idea of checked exception is to give a chance to recover from raised exception. just ignoring with empty braces in catch block or just printing tack trace or logging then continue with code is not beast practice.
* if user can take some action to recover from the expected error then make custom exception as checked
* just log an error unchecked exception when programming bugs invalid input, incorrect method implementation nothing application can do in mid of application running.
* if user cannot do anything then better to log and wait dev to fix it later so unchecked exception
* dont just catch and throw exception heavily instead return a boolean if an exception and caller will work accoding to the returned boolean.
* just catching & logging is not good .validate the input and throw exception to the caller to handle.
* instead of fileNotfound give proper message like illegalArgumentException("file name is empty") by validating file name
* in for loop raising and handling exception is not good:should not rely on exception handling do input validation

**Exception Handling**: handling of runtime errors (Exceptions) using try, catch, finally to maintain the normal flow execution.

**Types of Java Exceptions**: Checked, Un-checked, error (un-checked).

**1.** **Checked Exception** (Compile Time Exceptions): compiler enforces you to handle the exception at compile time. **Ex**: IOException, SQlException, ClassNotfoundException.

* Classes which directly inherit Throwable class except RuntimeEx and Error are known as Checked Exception.
* Ex: IOException, SQLException etc.Checked exceptions are checked at compile-time.
* Need to resolve at compile time, utill that compiler will not allow us to compile.

**2. Unchecked Exception**: compiler won't enforce you to handle the exception at compile time.

**Ex:**RuntimeEx(ArithmaticEx,NullPointerEx,NumberFormatEx,IndexOutOfBoundsE=>String,Array)

* Classes which inherit **RuntimeException** are known as unchecked exceptions

**3. Error**: it is irrecoverable. Exception due to program code but Error due to lack of system resources. **Ex**:StackOverflowError,OutofMemoryError,VirtualMachineError, Class def not found.

**Custom Checked Exception:** you need to **extend the Exception** class.

**Custom Un-checked Exception:** you need to **extend the RuntimeException** class.

**Exceptional handling using**:

**try**: Enclose the code that might throw an exception.

* Can’t use try block alone must be followed by either catch or finally.

**catch**: To handle the exception.

* Can't use catch block alone must be preceded by try block and can be followed by finally block.
* You can use multiple catch block with a single try.

**finally**: finally block will be executed whether an exception is handled or not.To execute the important code (cleanup) of the program.

**Ex**: closing connection, stream.

* If any Exception raised with no valid catch then finally {} is helpful to close the resources then print : ExceptionDesc==>stacktrace==>terminate.

**Internal working of Exception**: Exception Object thrown ==> Prints exceptionDesc ==> prints stacktrace==>terminate the program.

* **At a time only one Exception** is occurred and at a time only one catch block is executed.
* **All catch blocks must be ordered from most specific to most general** EX:catch for ArithmeticException must come before catch for Exception
* **If you don't handle exception,** before terminating the program, **JVM executes** **finally.**
* **For each try block there can be zero or more catch blocks**, but **only one finally block.**
* The **finally block will not be executed** if program exits (either by calling **System.exit ()** or by causing a fatal error that causes the process to abort).

**throw**: To throw an exception **explicitly**.

* Can throw either checked or unchecked exception. mainly used to throw custom exception.
* **Ex**: throw new IOException("sorry device error); throw new CustomException("custom ex description")
* Java Exception propagation: call stack order
* By default Unchecked Exceptions are forwarded in calling chain (propagated).
* By default Checked Exceptions are not forwarded in calling chain (propagated).use throws in declaration otherwise compile Time error.

**throws**: To declare exceptions at the method signature. It specifies that there may occur an exception in the method.

**Syntax:** return\_type method\_name() throws exception\_class\_name.You can declare multiple exceptions

* If you are calling a method that declares an exception, you must either caught or declare the exception again.

**final keyword** vs **finalize method** vs **finally block:**

final is a keyword used to apply restrictions on class, method and variable.

**final class** can't be inherited**, final method** can't be overridden **and final variable** value can't be changed.

**finally block is** used to place important code, it will be executed whether exception is handled or not.

**finalize method is** used to perform clean up processing just before object is garbage collected.

**Exception Handling with Method Overriding:**

* **If the super** class **method does not declare an exception**, **subclass** overridden **method** **cannot declare** the **checked exception** but it **can declare unchecked** exception.
* **If the super** class **method** declares **an exception,** **subclass** overridden **method** can **declare same, subclass exception** or **no exception** but **cannot** declare **parent exception.**
* **If the parent** class **constructor throws** some **checked exception** then the **child** class **constructor** must throw **same checked exception** **or** its **parent** **otherwise compiler error.**

**Custom Checked Exception Class:**

Class InvalidAgeException extends Exception {

InvalidAgeException(String s) {

super(s);

}

}

**ClassNotFoundException vs NoClassDefFoundError:**

**ClassNotFoundException** is an exception that occurs when you try to load a class at run time using **Class.forName("abc")** methods and mentioned classes are not found in the classpath.

**NoClassDefFoundError** is an error that occurs when a particular class is present at compile time, but was missing at run time.(if class missing in jar)

**Throwable** **class contains the following methods to display error information:**

**printStackTrace**: It displays error in format<==>Name of Exception Class : Description,StackTace

**toString()**: it displays error in the following format<==>Name of Exception Class : Description

**getMessage()**: it displays error information in the following format<==>Description

**Log Levels:** OFF**,** FATAL**,** ERROR**,** WARN**,** INFO**,** DEBUG**,** TRACE**,** ALL

**OOPS**:

**Object**: Any entity that has state and behavior is known as an object. An instance of a class .It can be physical and logical.

* An object contains an address and takes up some space in memory.
* Object’s state is stored in fields and behavior is shown via methods.
* **Ex**: A dog is an object because it has states i.e. color, name, breed etc. as well as behaviors i.e. barking, eating etc.

**Class**: A class is a template/blueprint that describes the state and behavior of an object.

It is a logical entity.

**Abstraction**: **Hiding** internal implementation details from the user and showing functionality is known as abstraction.

**Ex**: phone call, we don't know the internal processing.

**Advantage**: improves maintainability and security as we r not highlighting internal implementation.

**How do we achieve in java**: we use abstract classes and interfaces to achieve abstraction. We can achieve **100% abstraction using interfaces.**

* If we **don’t know about implementation** just we have to represent the specification then we should go for **interface**
* If **we don’t know about complete implementation** just we have **partial implementation** then we should go for **abstract**.
* If **we know complete implementation** and if we r ready to provide service then we should go for **concrete class.** Ex: ATM machine for cash withdrawal.

**Encapsulation**: **Binding** the data (variables) and functions together into a single unit in order to protect from un-authorized access is known as encapsulation.

**Advantage**: security, enhancement, maintainability.

**How do we achieve in java**: Declare the variables of a class as private and provide public setter and getter methods to modify and view the variables values. Ex: Java Bean.

In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

Encapsulation = Data Hiding + Abstraction

A class is said to be tightly encapsulated if all the data members declared as private.

**Data Hiding**: Restricts the access to the data members, and it can be accessible only through the member functions of the class (private, public methods)

**Advantage**: Security

**How**: Declare the variables of a class as private and provide public setter and getter methods to modify and view the variables values.Ex: Java Bean.

**Ex**: If your balance variable is declared as a public variable in the bank software, the person who has to see his account balance will have to access only private members through methods defined inside that class and this method will ask your account holder name or user Id, and password for authentication.

Ex: cola vending machine

**IS – A Relationship**: **Inheritance**

By using extends keyword we can implement inheritance.

**Advantage**: Reusability.

* Parent class reference can be used to hold child class object but by using that reference we r not allowed to call child class specific methods but can access overridden methods.
* A class can extend only one class at a time but an interface can extend any no of interfaces simultaneously.

**Inheritance**: When one object acquires all the state and behaviors (methods and fields) of parent object, it is known as inheritance.

**Why**: It provides code reusability. It is used to achieve runtime polymorphism.

**Note**: The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as super class (base class, parent class).

**Types of inheritance:**

**Single**: 1 extends 1

**Multilevel**: 1 extends 2, 2 extends 3

**Hierarchical**: 1 extends 3, 2 extends 3 (same parent for both classes)

**Multiple**: multiple parents and single child. 1 extends 2, 1 extends 3 .If both 2 and 3 has same methods then it will get ambiguity (which class method can be called)

**Hybrid**: Hierarchical and multiple

* Multiple inheritance is not supported in Java through class but can be through interfaces.
* Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes.
* So whether you have same method or different, there will be compile time error.
* ex: car inherits vehicle functionalities and human ->animal

**Polymorphism**: When one task performed in many ways then that is polymorphism.

**Example**: Draw something ==> shape or rectangle etc.

shape.draw() will draw square, rectangle based on the object you pass to the shape reference variable.

**Ex**: add integers, floats, and doubles.

**How do we achieve in java:** we use method overloading and method overriding to achieve polymorphism.

**The most common use of polymorphism in OOPs**: when a parent class reference is used to refer to a child class object.

public class Deer extends Animal implements Vegetarian{ }

A Deer IS-A Animal

A Deer IS-A Vegetarian

A Deer IS-A Deer

A Deer IS-A Object

Deer d = new Deer();

Animal a = d;

Vegetarian v = d;

Object o = d;

HAS – A Relationship: Composition or Aggregation.

**Advantage:** Reusability.

**Ex:** Employee has an Address, Car has a engine.

Code reuse is also best achieved by aggregation when there is no is-a relationship.

Inheritance should be used only if the relationship (is-a) is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

**Tight coupling:** classes tightly dependent on one another

class Subject {

Topic t = new Topic();

public void startReading()

{

t.understand();

}

}

**Loose coupling:**mostly independent

class Topic1 implements Topic

class Topic2 implements Topic

class Subject {

Topic t;

Subject( Topic t){

this.t=t;

}

public void startReading()

{

t.understand();

}

main(){

Subject s= new Subject(new Topic1());

s.startReading();

}

}

**Method Overloading**: defining the functions with **same name, different number of arguments or different type of arguments**

**Advantage**: Method overloading increases the readability of the program.

There are two ways to overload the method in java

By changing number of arguments

By changing the data type

Note: In java, **Method Overloading** is **not possible by** changing the **return type** of the method only. Return type is not part of method signature.

**Compile-time polymorphism or EarlyBinding:** method calling decided by compiler at compile time.

**How:** by method overloading

**Runtime-time polymorphism**: method call decided at runtime based on the object that it get.

**How:** by method overriding

**Automatic promotion in overloading:** In the case of overloading if there is no method with the required argument then the compiler won’t raise immediately compile time error. First it will promote arguments to next level and checks is there any

Matched method with promoted arguments, if there is no such method compiler will promote the argument to the next level and checks for the matched method.

After all possible promotions still the compiler unable to find the matched method then it raises compile time error

In the case of overloading the more specific version will get the chance first.

Object-->String or StringBuffer So if public void m1(String s),public void m1(StringBuffer sb)==> Ambiguity Error.

m1(double d)==can accept any type of input (byte,short,int,char,long,float,double)

**var-arg** **method** will always get **least priority** i.e if no other method matched then only var-arg method will get chance for execution

**Overloading** method resolution will always take care by **compiler** **based on the reference type** but not based on runtime object.

Animal a1 = new Monkey(); m1(Animal a) but not m1(Monkey m)

**Method Overriding:** Defining same function in subclass which is already defined or declared in super class. Should have **same function name and same number of arguments and same type of arguments and same or covariant return type.**

Covariant: sub type of parent return type

**Usage of Java Method Overriding:**

* Method overriding is used to provide child specific implementation of a method that is already provided by its super class.
* Method overriding is used for runtime polymorphism

**Problem without method overriding:** I have to provide a specific implementation of run() method in subclass.

* If the child class is not satisfied with the parent class implementation then the child is allowed to overwrite that parent class method to provide its own specific implementation.
* Final method can’t be overridden in child classes.
* Private methods are not visible in the child classes. Hence they won’t participate in overriding.
* Based on our requirement we can take exactly same declaration in child class, But It is not overriding.
* We can override abstract methods, synchronized methods. We can override a non-abstract method as abstract also.
* While overriding we are not allowed to decrease access privileges. Otherwise compile time error but we can increase access privileges. The following is the list of valid with respect to access privileges.
* While overriding the size of checked exceptions we are not allowed to increase in throws class but we can decrease the size of checked exceptions. But there are no restrictions on unchecked exceptions
* A static method can’t be overridden as non-static.
* abstract<-->non-abstract, synchronized<-->non-synchronized, final-->final,
* static<-->static(but it is method hiding)
* method hiding ==static polymorphism=compile time polymorphism=early binding
* Overriding in the case of Variable: Overriding concept is not applicable for variables.
* And it is applicable only for methods. Variable resolution always takes care by compiler based on reference type.

**Static control flow in parent and child classes**: All the static variable assignments & static blocks execution done from parent to child at the time of class loading.

**Instance Control Flow:** All the instance variable assignments, instance black and constructor execution from parent to child at the time of object creation with new.

* **Static control flow is only one time activity and that will execute at the time of class loading.**
* **But instance control flow is not one time activity. And it will execute for every object creation separately.**

**Constructor:** Objective of constructor is to perform initialization

* The name of the constructor and name of the class must be same.
* The only allowed modifiers for the constructors are **public, private, protected, <default>.** If we are using any other modifier we will get C.E(Compiler Error)
* **Return type** is **not allowed** for the constructor’s **even void** also. If we r declaring return type then the compiler treats it as a method and hence there is no C.E and R.E(RuntimeError)

**Default Constructor**: no-arg constructor contains only one – line super();

* If we r not writing any constructor then the compiler always generate default constructor.
* If we r writing at least one constructor then the compiler won’t generate any constructor.
* Hence every class contains either programmer written constructor or compiler generated default constructor but not both simultaneously
* The modifier of the default constructor is same as class modifier (either public or default).
* **Class** can have **public** or **default** modifiers.

**super() & this() in constructor**: can use any one as first statement in the constructor.

* We can overload constructor. Inheritance concept is not applicable for constructor and hence overriding is also not applicable.
* If the parent class constructor throws some checked exception. Compulsory the child class constructor should throw the same checked exception or its parent otherwise compile time error.

**Coupling:** The degree of dependency between the components is called coupling.

The main advantages of loosely coupling are:

1) It improves maintainability.

2) It makes enhancements easy.

3) It produces reusability.

**Cohesion:** Design or define separate components for each single purpose.(1 design -1 purpose)

For every task we can define a separate component like login.jsp for displaying login page, validate servlet for validation purpose, inbox servlet, For displaying mails etc…

The main advantages of High Cohesion are:

1) It improves maintainability.

2) Enhancements is easy.

3) Promotes reusability.

Ex: mobile with seperate battery

**Typecasting**: Casting from one type to other.

**Ex1:**Object o = new String(“raju”);

StringBuffer sb = (StringBuffer)o;//ClassCastException-unChecked

String sb = (String)o;//no error

**Ex2:**Object o = new StringBuffer (“raju”);

StringBuffer sb = (StringBuffer)o;//no error

String sb = (String)o; //ClassCastException-unChecked

**Wrapper class (java.lang)**: A wrapper class in java provides the mechanism to convert primitive into object and object into primitive.

* Wrapper & String classes overridden compareTo , equals and hashCode methods

**Autoboxing**: automatic conversion of primitive into object is known as autoboxing.

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

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

**Unboxing**: object into primitive.

Integer a=new Integer(3);

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

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

**String**: sequence of chars = an array of characters works same as Java string.

Java String is **immutable** which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

char[] ch={'j','a','v','a','t','p','o','i','n','t'}; String s=new String(ch); <====>String s="javatpoint";

Methods: concat(), equals(), split(), length(), replace(), compareTo(), **intern**()(**points to String constant pool**), substring()

java.lang.String class implements Serializable, Comparable and CharSequence interfaces.

CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in java by using these three classes.

**There are two ways to create String object:**

By string literal==>String s="lll"

By new keyword==>String s=new String("llll");

**Immutable**:

Cannot be changed, if you concat some thing to it then it will create a new memory in string constant pool, you need to reassign to some variable after concat to point to the new memory.

**Why String is Immutable:** more than one Stringpoints tosame memory in constant pool, If you change the value of a variable it should not affect other. To make Java more **memory efficient** (because no new objects are created if it exists already in the string constant pool).

**Custom Immutable class:** final class, final private variable, dont provide setters, constructor has new copy, getter should return copy

1. Make your class final, so that no other classes can extend it.
2. Make all your fields final, so that they’re initialized only once inside the constructor and never modified afterward.
3. Don’t expose setter methods.
4. When exposing methods which modify the state of the class, you must always return a new instance of the class.

**If the class holds a mutable object:**

Inside the constructor, make sure to use a clone copy of the passed argument and never set your mutable field to the real instance passed through constructor, this is to prevent the clients who pass the object from modifying it afterwards.

Make sure to always return a clone copy of the field and never return the real object instance.

final class Student {

// Member arrributes of final class

private final String name;

private final int regNo;

private final Map<String, String> metadata;

// Constructor of immutable class

// Parameterized constructor

public Student(String name, int regNo, Map<String, String> metadata)

{

// This keyword refers to current instance itself

this.name = name;

this.regNo = regNo;

// Creating Map object with reference to HashMap

// Declaring object of string type

Map<String, String> tempMap = new HashMap<>();

// Iterating using for-each loop

for (Map.Entry<String, String> entry :

metadata.entrySet()) {

tempMap.put(entry.getKey(), entry.getValue());

}

this.metadata = tempMap;

}

public Map<String, String> getMetadata()

{

// Creating Map with HashMap reference

Map<String, String> tempMap = new HashMap<>();

for (Map.Entry<String, String> entry :

this.metadata.entrySet()) {

tempMap.put(entry.getKey(), entry.getValue());

}

return tempMap;

}

}

**String constant pool:**

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned.

If the string doesn't exist in the pool, a new string instance is created and placed in the pool.

Ex:; String s1="Welcome"; String s2="Welcome";//It doesn't create a new instance

Note: String objects are stored in a special memory area known as the "string constant pool".

**Why Java uses the concept of String literal?**

To make Java more **memory efficient** (because no new objects are created if it exists already in the string constant pool).

String s=new String("Welcome");//**creates two objects and one reference variable**

In such case, JVM will create a new **string object in normal (non-pool) heap memory,** and the **literal "Welcome" will be placed in the string constant pool**. The variable s will **refer** to the **object** in a **heap** (non-pool).

s1==s2 reference(address) comparison

s1.equals(s2) content comparison

substring(0,4),startWith/end,charAt,compareTo,indexOf,lastIndexOf,contains,toCharArray,split(',')

**StringBuffer** === **mutable**, **thread-safe(methods synchronized)**,capacity(default-16),length(no of chars)

append, insert(2,"n"), reverse, setCharAt(1,'n')

**StringBuilder** === **mutable**, **not-thread safe but faster than string buffer**

append, insert(2,"n"), reverse, setCharAt(1,'n')

**String vs String Builder vs String Buffer:**

1. String is immutable and String buffer and builder are mutable.
2. String buffer( thread safe) or builder preferred if there are more concatenation operations.

**Nested inner**: OuterClass.InnerClass a=new OuterClass().new InnerClass();<==>from outside , outclass method can call inner class members by inner class instance

Nested inner classes can access outer class var/methds directly

Method local: In a method class present

**Static nested**: non static members of outer class cant be accessed in inner, from outside of class OuterClass.InnerClass o=new OuterClass.InnerClass()

**Collections**

**CopyOnWriteArralist**: uses ReentrantLock for thread safety no synchronized methods like stringBuffer, every update will create new Array with +1 increased size. theread safe alternative for arraylist.

Iterator only has remove() but list iterator has set(),add(),remove()

Hashmap: rehashing happens for all the stored elements once it reached its loadfactor, just doubling the capacity not help so need rehashing

HashTable: methods are syncronized

**Collection**.**synchronizedMap**(): require whole collection object is locked

**ConcurrentHashMap**: uses ReentrantLock for thread safety, no null key or value, method compute(key,(k,v)->null?1:v+1) for atomic operation

**Cloning**: shallow, deep

**Aggrigation**: loose whole and part relation

**Composition**: tight whole and part relation

int compute(){

if (n <= 1)

return n;

Fibonacci f1 = new Fibonacci(n - 1);

f1.fork();

Fibonacci f2 = new Fibonacci(n - 2);

return f2.compute() + f1.join();

}

**Running multiple futures in parallel:**

CompletableFuture.allOf (t1, t2, t3) waits for all the tasks to complete .get() and also. get(10) waits for 10 sec

Thread Local:1 object per thread instead of per task, So in thread pool only fixed number of instances. So memory efficiency and thread safety.

Reentrant, semaphores, blocking Queue: give specific time to get lock in order to overcome dead lock. So give time-out for blocking natures

Dead locks difficult to detect due to multiple lock types and thread sources can be done by thread dumps

**4 refernce:** grabage collection depends on these 4 references

strong, soft, weak, phanton

Phanton: we want to wait until the old image is collected before loading a new one. Here The reference of the old image will be enqueued in the ReferenceQueue once the old image object is finalized. After receiving that reference, we can load the new image in to memory.

**Comparable** provides a **single sorting sequence**, affects the original class

The **Comparator** provides **multiple sorting sequences,** doesn’t affect the original class so for api classes

**JVM:**

**Class loader subsystem:**

**Memory areas:**

**Execution engine**:

**Java Memory Model:**

**Stack** (local methods variables, ref), **Heap**, **Method Area** (permGen)

**Method Area** is a part of **permGen** and in java 8 it has replaced with **metaspace**(Native reg)

**Heap Area** divided into young generation and old generation

Young gen has Eden, surviviour from and survivor to

New object allocated in Eden.

**when Eden gets full** then **minor gc** get started and **marks all the reachable objects** and then **sweep to surviour1** and then compact the existing objects after deleting unreachable objects.

**Again after sometime Eden gets full** then marks the **Eden and surviour1 objects** and move these active objects to **surviour2** and delete the unreachable object in both Eden and suviviour1

The objects which got survived after few(16) cycles of GC then moved to old gen.

if old gen gets filled then major gc will gets triggered in entire heap and the application functionality will be paused for few sec.

GC type: serial, parallel, GC1

**Why String is immutable:**

2. Thread Safe:

As the String object is immutable we don't have to take care of the synchronization that is required while sharing an object across multiple threads.

3. Security:

As we have seen in class loading, immutable String objects avoid further errors by loading the correct class. This leads to making the application program more secure. Consider an example of banking software. The username and password cannot be modified by any intruder because String objects are immutable. This can make the application program more secure.

4. Heap Space:

The immutability of String helps to minimize the usage in the heap memory. When we try to declare a new String object, the JVM checks whether the value already exists in the String pool or not. If it exists, the same value is assigned to the new object. This feature allows Java to use the heap space efficiently.

Why String class is Final in Java?

The reason behind the String class being final is because no one can override the methods of the String class. So that it can provide the same features to the new String objects as well as to the old ones

**Java Basics**

1. Code editor tools useful to compile, debug and run Ex: STS, Eclipse

JVM, JRE and JDK are platform dependent But, Java is platform independent.

2. The compiled byte code is platform independent .So application developed on one platform can be run any platform like in UNIX, Linux, windows.

3. JRE(Java Runtime Environment) is to run java aplications. It is plotform dependent(seperate installation file for seperate OS)

A.java after compilation compiler generates A.class file

Javac A.java ====> A.class contains byte code

**Java Applications:**

1. standalone java application :runnable(Executable) file .jar
2. web applications :runable(Executable) file .war

Need to deploy runnable(Executable) files in JRE to run the applications.

4. Compile Time errors can be handled by developer while developing. Editor tools show such compile time errors.(ex:syntax errors)

5. Runtime errors cannot be handled at compile time .

**Java Advantages**: Java is a high level, simple, portable, robust(we can change the code), secured and object-oriented programming language. -

Class Simple {

Public static void main(String args[]){

System.out.println("Hello Java");

}

}

**Where it is used:**

Desktop Applications such as acrobat reader, media player, antivirus etc.

Web Applications such as irctc.co.in, javatpoint.com etc.

Enterprise Applications such as banking applications.

Mobile Embedded System, Smart Card, Robotics, Games etc.

JVM: contains class loader, memory area, execution engine etc.

1) **Class loader**: Class loader is a subsystem of JVM that is used to load class files.

2) **Class (Method) Area**

Class (Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3) **Heap**

It is the runtime data area in which objects are allocated.

4) **Stack**

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

5) Program Counter Register

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

6) Native Method Stack

It contains all the native methods used in the application.

**7) Execution Engine**

It contains:

1) A virtual processor

2) **Interpreter**: Read byte code stream then execute the instructions.(It changes Byte code to Machine Undertandable code)

3) **Just-In-Time (JIT**) compiler: It is **used to improve the performance**. It compiles parts of the byte code **that have similar functionality** at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

**Memory:**

When Object Created:

1. Memory allocated to hold instance variables in Heap.

2. Objects are initialized to their default values.

3. Constructor chain until Object constructor call.

4. Before the body of the constructor executed the instance block executes.

constructor()

{

super();or this();

instance block

{

}

constructor body;

}

**Memory:**

**Method/class Area**: Class structure, static variables, constants, runtime constant pool

**Stack**: contains Primitive data types and object reference variables. Method call and its method local primitive variables memory in the call stack frame (in case of Objects in method frame :only reference variables of object not whole object).

**Heap**: Objects and String objects (String constant pool) stored in Heap memory. and its reference in stack. Garbage collector runs on heap memory to remove unreferenced and null referenced objects.

**Instance (primitive) variables** memory allocated in object's memory in heap.

**Instance (object) variables** memory allocated in heap and its reference in object's memory in heap.

Java 8

1.Functional interfaces

2.Lambda expressions

3.Method references

4.Stream API

5.ForEach method: list.forEach()

6.Optional

7.Collectors: Collectors.toList(),toset(),toMap(),counting(),groupingBy(),mapping(),maxBy(),summingInt(),summarizingDouble(),

8.Base64 Encode Decode,

9.Default and Static methods in Interface

1.**Functional interfaces(@FunctionalInterface)**:An interface which has only one abstract method and can have any number of default, static methods but can contain only one abstract method.It can also declare methods of object class.

Ex:Runnable run() and Comparable compareTo(),callable call().

**JavaSE 8 included four main kinds of functional interfaces:**

1.**Predicate**: takes arguments & return boolean.Ex: filtering

{**boolean test(oneArg)**{return **boolean**;}}

2.**Consumer**:Accepts arguments and wont return any value. {**void accept(oneArg)**{}}

Consumer<Integer> consumer = (value) -> System.out.println(value);

3.**Functional: Object apply(arg1**){return}} takes input and returns output.

4.**Supplier:O bject get()**{return}: wont take input and returns output.

Example:

@FunctionalInterface //It is optional

interface Drawable{

public void draw(); // abstract method

// It can contain any number of Object class methods.

int hashCode();

String toString();

boolean equals(Object obj);

}

public class LambdaExpressionExample2 {

public static void main(String[] args) {

//with lambda

Drawable d2=(width)->{

System.out.println("Drawing "+width);

return width+5;

};

int wd = d2.draw(10);

}

}

2.**Java Lambda Expressions:** It **provides implementation for a Functional interface**,Less coding.

lambda expressions can be used to represent the instance of a functional interface

Ex: Runnable r2=()->System.out.println("Thread2 is running...");

Thread t2=new Thread(r2);

Java lambda expression is treated as a function, so compiler does not create .class file.

It helps to iterate,filter and extract data from collection.

Syntax:(argument-list) -> {body} == lambda expression

Ex: Collections.sort(list,(p1,p2)->{

return p1.name.compareTo(p2.name);

});

**Note: For sort objects should be comparable . So need to give comparator.**

3.**Java Method References:** ::**referring to a method that provides implementation of a functional interface**, It is compact and easy form of lambda expression.

Referring to some method that has implementation of functional interface.

Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

interface Sayable{

void say(int);

}

public class MethodReference {

public static void saySomething(int a){

System.out.println("Hello, this is static method."+a);

}

public static void main(String[] args) {

// Referring to a static method which is written in some class

Sayable sayable = MethodReference::saySomething; instead of Sayable sayable = (a) ->System.out.println("Hello, this is static method."+a);

// Calling interface method

sayable.say(5);

}

}

import java.util.function.BiFunction;

class Arithmetic{

public static int add(int a, int b){

return a+b;

}

}

public class MethodReference3 {

public static void main(String[] args) {

BiFunction<Integer, Integer, Integer>adder = Arithmetic::add;

int result = adder.apply(10, 20);

System.out.println(result);

like in forEach:

consumer c = (n) -> System.out.println("Hello, this is static method."+n); c.accept(n);

}

}

Note: we can use predifiened functional interfaces.

4.**Interfaces can have any number of default, static methods:It wont force child to implement the newly added default methods**.ex:List added with sort() method

Abstract Class vs Java 8 Interface:

An interface and an abstract class is almost similar except that you can create constructor in the abstract class whereas you can't do this in interface.

abstarct class with all abstratct methods == interface but multiple inheritance is not possible with abstract class and instances variables initialization done in abstract class constructor

normally if you have some partial functionality and froce the child class to override then abstract ,if you force child to fully implement then interface

5**.Optional:** Enforces/asks us to do null checks(isPresent()) before you use(get()), can add where ever the null check is required.

6.StreamAPI:

Operations performed on a stream does not modify it's source.Ex: filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.

**Function and BiFunction:Object apply()**

<R> Stream<R> map(Function<? super T, ? extends R> mapper)

IntStream mapToInt(ToIntFunction<? super T> mapper) – similarly for long and double returning primitive specific stream.

IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper) – similarly for long and double

<A> A[] toArray(IntFunction<A[]> generator)

<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)

**Predicate and BiPredicate: boolean test()**

Stream<T> filter(Predicate<? super T> predicate)

boolean anyMatch(Predicate<? super T> predicate)

boolean allMatch(Predicate<? super T> predicate)

boolean noneMatch(Predicate<? super T> predicate)

**Consumer and BiConsumer:accept(object)**

Stream<T> peek(Consumer<? super T> action)

void forEach(Consumer<? super T> action)

void forEachOrdered(Consumer<? super T> action)

**Supplier:object get()**

public static<T> Stream<T> generate(Supplier<T> s)

<R> R collect(Supplier<R> supplier,BiConsumer<R, ? super T> accumulator,BiConsumer<R, R> combiner)

Optional:isPresent() will return true and get() will return value

Optional<T> reduce(BinaryOperator<T> accumulator)

Optional<T> min(Comparator<? super T> comparator)

Optional<T> max(Comparator<? super T> comparator)

Optional<T> findFirst()

Optional<T> findAny()

Terminal methods:forEach, toArray, min, max, findFirst, anyMatch, allMatch ,noneMatch

A terminal operation is called short circuiting, if it may terminate in finite time for infinite stream:

Ex: anyMatch, allMatch, noneMatch, findFirst and findAny

Intermediate operations: filter and map.

An intermediate operation is called short circuiting, if it may produce finite stream for an infinite stream:

Ex:limit() and skip()

Stream<Integer> stream = Stream.of(1,2,3,4); create a stream from similar type of data.

we can’t pass primitive type array.

Stream.of(new Integer[]{1,2,3,4});array of Objects to return the stream. Note

Program – Stream API

import java.util.\*;

import java.util.concurrent.ConcurrentMap;

import java.util.function.BinaryOperator;

import java.util.function.Consumer;

import java.util.function.Function;

import java.util.stream.Collector;

import java.util.stream.Collectors;

import java.util.stream.DoubleStream;

import java.util.stream.IntStream;

import java.util.stream.LongStream;

import java.util.stream.Stream;

import java.math.BigDecimal;

class Product1{

int deptId;

String name;

float price;

public Product1(int deptId, String name, float price) {

this.deptId = deptId;

this.name = name;

this.price = price;

}

public int getId() {

return deptId;

}

public String getName() {

return name;

}

public float getPrice() {

return price;

}

}

public class Java8Features {

public static void main(String[] args) {

List<Product1> pList = new ArrayList<Product1>();

pList.add(new Product1(1,"HP Laptop",28000f));

pList.add(new Product1(1,"Dell Laptop",30000f));

pList.add(new Product1(1,"Lenevo Laptop",28000f));

pList.add(new Product1(4,"Sony Laptop",28000f));

pList.add(new Product1(5,"Apple Laptop",90000f));

List<String> list = Arrays.asList("HP Laptop","Dell Laptop","Lenevo Laptop","Sony Laptop","Apple Laptop","HP Laptop","HP Laptop","Lenevo Laptop");

List<Item> items = Arrays.asList(

new Item("apple", 10, new BigDecimal("9.99")),

new Item("banana", 20, new BigDecimal("19.99")),

new Item("orang", 10, new BigDecimal("29.99")),

new Item("watermelon", 10, new BigDecimal("29.99")),

new Item("papaya", 20, new BigDecimal("9.99")),

new Item("apple", 10, new BigDecimal("9.99")),

new Item("banana", 10, new BigDecimal("19.99")),

new Item("apple", 20, new BigDecimal("9.99"))

);

Optional<String> findFirst = list.stream().findFirst();

if(findFirst.isPresent()) {

System.out.println(findFirst.get());

System.out.println(findFirst.orElse(testMathods("Laptop")));

System.out.println(findFirst.orElseGet(()->testMathods("c")));//takes supplier

findFirst.ifPresent(testMathodss(a->a.toUpperCase()));//takes consumer

}

System.out.println(list.stream().allMatch(a->a.contains("testMathods")));//returns boolean

System.out.println(list.stream().anyMatch(a->a.contains("Sony Laptop")));

List<String> asList = list.stream().collect(Collectors.toList());

Map<Integer, List<Product1>> peopleByCity = pList.stream().collect(Collectors.groupingBy(Product1::getId));

Map<Integer, List<String>> peopleByCity1

= pList.stream().collect(Collectors.groupingBy(Product1::getId,Collectors.mapping(Product1::getName, Collectors.toList())));

Map<Integer, Map<Float, List<Product1>>> peopleByStateAndCity

= pList.stream().collect(Collectors.groupingBy(Product1::getId,Collectors.groupingBy(Product1::getPrice)));

Map<Integer, Optional<Product1>> peopleByStateAndCity2

= pList.stream().collect(Collectors.groupingBy(Product1::getId,Collectors.reducing(BinaryOperator.maxBy(Comparator.comparing(Product1::getPrice)))));

Map<Integer, Optional<Product1>> peopleByStateAndCity3

= pList.stream().collect(Collectors.groupingBy(Product1::getId,Collectors.maxBy(Comparator.comparing(Product1::getPrice))));

System.out.println("3333"+ peopleByStateAndCity3);

Map<String, Long> result =list.stream().collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));

Map<String, Long> counting = items.stream().collect(

Collectors.groupingBy(Item::getName, Collectors.counting()));

System.out.println(counting);

Map<String, Integer> sum = items.stream().collect(

Collectors.groupingBy(Item::getName, Collectors.summingInt(Item::getQty)));

Double x = pList.stream().collect(Collectors.averagingDouble(Product1::getPrice));

Long count = list.stream().collect(Collectors.counting());

Stream<String> s = Stream.of("apple", "banana", "orange");

Map<Integer, List<String>> map = s.collect(Collectors.groupingByConcurrent(String::length));//{5=[apple], 6=[banana, orange]}

List<String> collect = pList.stream().collect(Collectors.mapping(Product1::getName, Collectors.toList()));

Optional<Product1> collect2 = pList.stream().collect(Collectors.maxBy(Comparator.comparing(Product1::getPrice)));

List<Integer> intList = Arrays.asList(2, 4, 5, 6, 8);

Map<Boolean, List<Integer>> isEven = intList.stream().collect(

Collectors.partitioningBy(i -> i % 2 == 0));

List<Integer> list2 = isEven.get(true);

Collector<Product1, ?, Optional<Product1>> reducing = Collectors.reducing(BinaryOperator.maxBy(Comparator.comparing(Product1::getPrice)));

// pList.stream().collect(Collectors.groupingBy(Product1::getPrice,Collectors.reducing(Product1::getName,BinaryOperator.maxBy(Comparator.comparing(String::length)))));

DoubleSummaryStatistics collect4 = pList.stream().collect(Collectors.summarizingDouble(Product1::getPrice));

Double collect5 = pList.stream().collect(Collectors.summingDouble(Product1::getPrice));

Vector<String> empNames = pList.stream().map(Product1::getName).collect(Collectors.toCollection(Vector::new));

ConcurrentMap<String, Float> collect6 = pList.stream().collect(Collectors.toConcurrentMap(Product1::getName, Product1::getPrice));

List<String> collect7 = list.stream().collect(Collectors.toList());

Map<String, Float> collect3 = pList.stream().collect(Collectors.toMap(Product1::getName, Product1::getPrice));

Set<String> collect8 = list.stream().collect(Collectors.toSet());

// list.stream().collect(supplier, accumulator, combiner);

System.out.println(list.stream().count());

list.stream().distinct().forEach(a->System.out.println(a));;

list.stream().filter(a->a.contains("Laptop"));

Optional<String> findAny = list.stream().findAny();

List<List<String>> namesNested = Arrays.asList(

Arrays.asList("Jeff", "Bezos"),

Arrays.asList("Bill", "Gates"),

Arrays.asList("Mark", "Zuckerberg"));

List<String> namesFlatStream = namesNested.stream().flatMap(y->y.stream()).collect(Collectors.toList());

// List<String> namesFlatStream = namesNested.stream().flatMap(Collection::stream).collect(Collectors.toList());

long[][] data = {{1L,2L},{3L,4L},{5L,6L}};

LongStream ls1 = Arrays.stream(data).flatMapToLong(row -> Arrays.stream(row));

list.stream().forEach(a->System.out.println(a));

list.stream().forEachOrdered(a->System.out.println(a));

Stream.iterate(1,i->i+5).limit(5).forEach(a->System.out.println(a));

list.stream().map(a->a.toUpperCase()).forEach(a->System.out.println(a));

DoubleStream mapToDouble = pList.stream().mapToDouble(i->i.getId());

IntStream mapToInt = pList.stream().mapToInt(Product1::getId);

Optional<String> max = list.stream().max(Comparator.comparing(String::valueOf).reversed());

boolean noneMatch = list.stream().noneMatch(i->i.contains("a"));

list.stream().parallel().forEach(a->System.out.println(a));

list.stream().reduce("a",String::concat);

Optional<Float> sumSal = pList.stream().map(Product1::getPrice).reduce(Float::sum);

Optional<Float> reduce = pList.stream().map(Product1::getPrice).reduce((a,b)->a+b);

pList.sort(Comparator.comparing(Product1::getPrice).thenComparing(Product1::getName));

Collections.sort(pList,Comparator.comparing(Product1::getPrice).thenComparing(Product1::getName));

int reducedTwoParams =IntStream.range(1, 4).reduce(10, (a, b) -> a + b);

int reducedParams = Stream.of(1, 2, 3).reduce(10, (a, b) -> a + b, (a, b) -> {

System.out.println("combiner was called");//16 (10 + 1 + 2 + 3)

return a + b;

});

// The result will be the same as in the previous example (16) and there will be no login which means, that combiner wasn’t called. To make a combiner work, a stream should be parallel:

// Arrays.asList(1, 2, 3).parallelStream().reduce(identity, accumulator, combiner);

int reducedParallel = Arrays.asList(1, 2, 3).parallelStream().reduce(10, (a, b) -> a + b, (a, b) -> {

System.out.println("combiner was called");//12 + 13 = 25; 25 + 11 = 36

return a + b;

});

list.stream().sequential().forEach(a->System.out.println(a));;

list.stream().skip(5).forEach(a->System.out.println(a));;

list.stream().sorted(Comparator.reverseOrder());

pList.stream().sorted(Comparator.comparing(Product1::getPrice).reversed()).forEach(a->System.out.println(a));

Integer[] array = list.stream().toArray(Integer[]::new);

list.parallelStream().forEach(a->System.out.println(a));

}

private static Consumer<? super String> testMathodss(Consumer<String> conumer) {

// TODO Auto-generated method stub

return i->{

try {

conumer.accept(i);

} catch (Exception e) {

// TODO: handle exception

}

};

}

private static String testMathods(String string) {

System.out.println(" called for orElse when no data in the list:"+string);

return string.toUpperCase();

}

}

class Item {

private String name;

private int qty;

private BigDecimal price;

public Item(String string, int i, BigDecimal bigDecimal) {

this.name=string;

this.qty=i;

this.price=bigDecimal;

}

public String getName() {

return name;

}

public int getQty() {

return qty;

}

public BigDecimal getPrice() {

return price;

}

//constructors, getter/setters

}