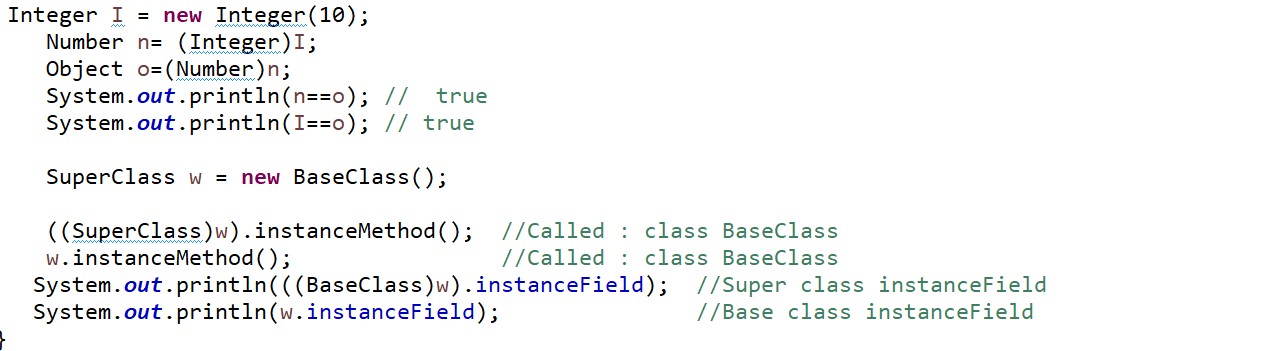
# Advanced Class designs

1. When a programmer does not define ANY constructor, the compiler inserts one automatically, the access modifier of which is same as that of the class.
2. **this.getClass():** Returns the runtime class of this Object. The returned Class object is the object that is locked by static synchronized methods of the represented class.
3. **Static Class:** It cannot be base class as it gives error : Illegal modifier for the class outerStaticClass; only public, abstract & final are permitted.
   1. Nested static class are used as a namespace.
4. Abstract Class: You want to define common method signatures in the class but force subclasses to provide implementations(by declaration abstract method) for such methods.
   1. It is ok to declared an abstract method protected and public not private as they give error as. The abstract method m1 in type A can only set a visibility modifier, one of public or protected
   2. For declaration of abstract method abstract keyword must be use in class but in interface not necessary.
   3. You want to pass different implementations for the same abstraction in method calls, that’s why abstraction is used.
5. For declaration of **non-static** **method** of static class, we must initialize the inner static class while for static method it is not necessary.
   1. **new** NonStaticClass.StaticClass().nonStaticMethod();
   2. NonStaticClass.StaticClass.staticMethod();
6. Protected method declared in super class to override not use(In Emergency) in base class like clonable(), finalize().
7. In type casting we cannot create new object, but we are providing new reference to the object.
   1. Method resolution is always based on **runtime** type object.
   2. In method hiding(static method) resolution is based on **reference** type
   3. Variable reference is always based on the **reference** type object.

Last sysout is wrong it must be super class instance field

1. The static method cannot override. So, if interface has **static method** so it will not override. So, it can be difference between default and static method in interface.

**interface** iB{

**static** **void** A() {

System.***out***.println(" static B");

}

}

**class** iC **implements** iB{

**public** **void** A() {

System.***out***.println(" ia ib");

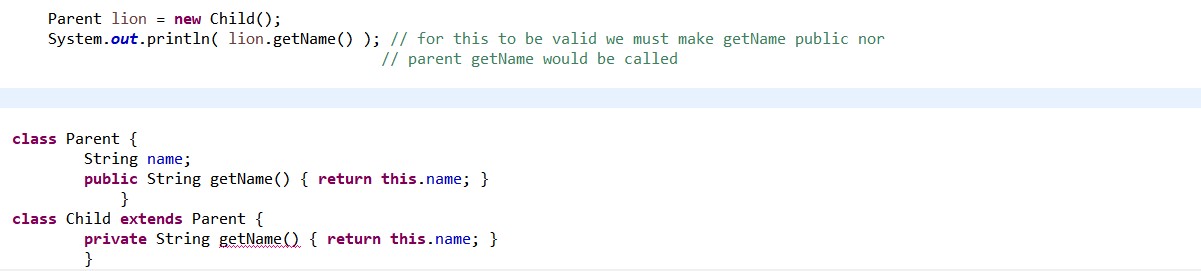
}

}

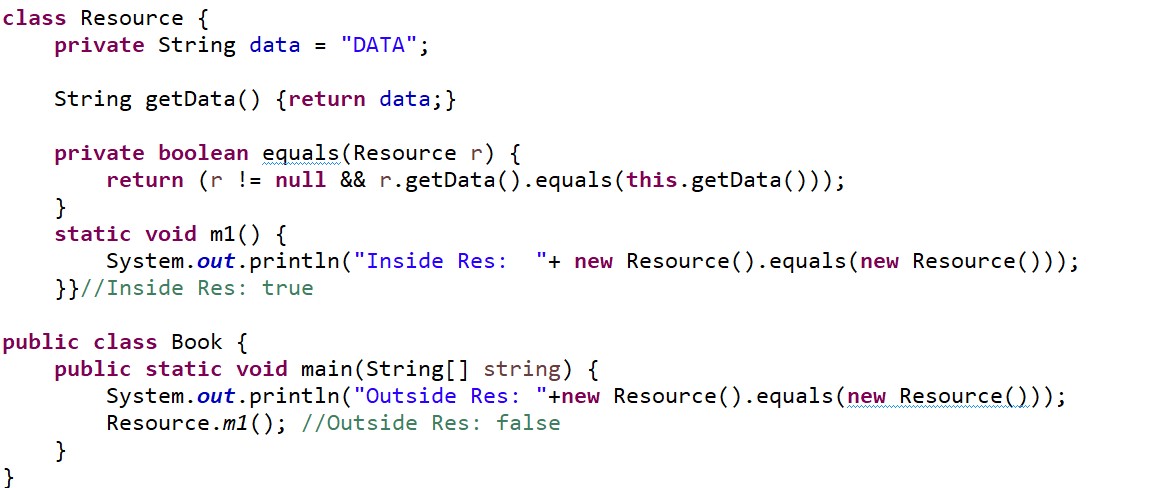
iB ic= **new** iC(); give error

ic.*A*(); This static method of interface iB can only be accessed as iB.A

1. Default method make the differences between interface and abstract classes.
2. The default methods were introduced from java 8 to provide backward compatibility so that existing interfaces can use the lambda expressions without implementing the methods in the implementation class. Default methods are also known as **defender methods**or **virtual extension methods**.
3. **Method Overriding**: Whatever method parents have by default available to the child through inheritance, if child class not satisfied with the parent class implementation then child can redefine that method based on is requirement this process is called method overriding.
   1. For overriding methods, the private method is not allowed if it is declared then it is **child specific** method.

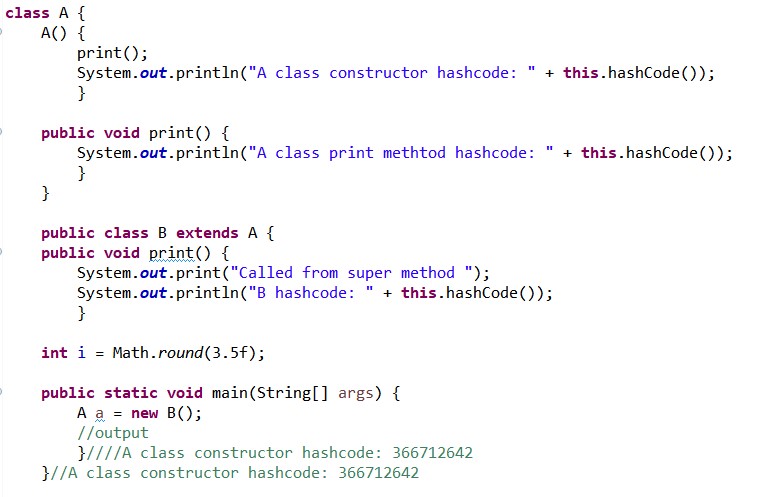


* 1. For overriding methods, the scope(**access modifier)** must be same or more accessible(**more open**) (if not then it give error Cannot reduce the visibility of the inherited method from Super) but return type must be same or more restrictive(child class) also known as covariant(same variant) return type(From 1.5 onwards, earlier only same return type are allowed).



* 1. Whenever we are creating child class objects , parent constructor will be executed but parent object won't be created. Here we got same hashcode so only b obj created.

Here we have called hashcode but we can also call this.getClass() to know which class is calling.



* 1. An instance method cannot override a **static method**, and a **static method** cannot hide an instance method. So super class method run a member of super class are printed.

Throw of exceptions:

**interface** iA{

**void** m1() **throws** IOException;

}

**interface** iB{

**void** m1() **throws** FileNotFoundException;

}

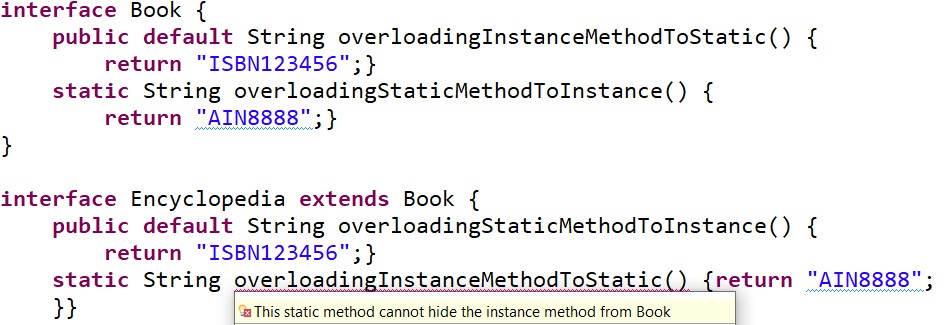
**class** iC **implements** iA, iB{

**public** **void** m1() **throws** IOException {

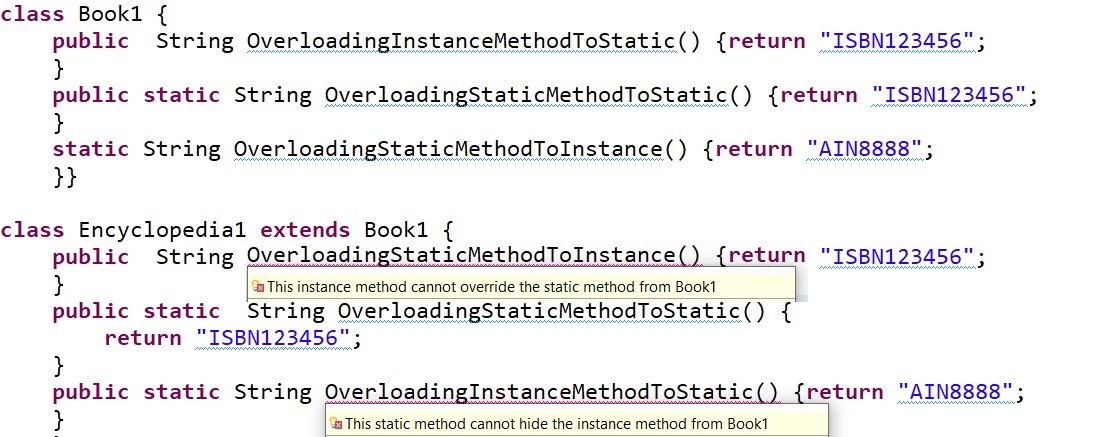
} // give error Exception IOException is not compatible with throws clause in iB.m1()

} // use throws FileNotFoundException

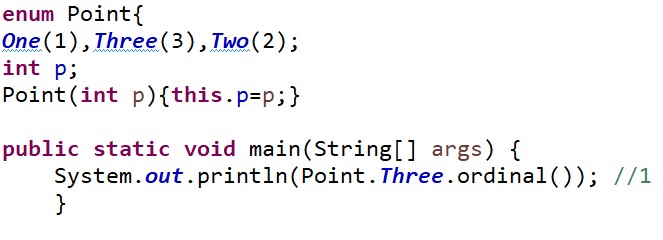
* 1. If in overriding any **checked exceptions** are thrown, only the **same exceptions** or subclasses of those exceptions are allowed to be thrown in subclass.
  2. A general rule for all: **static method cannot hide an instance method**( In super class).
     1. Interface: You cannot override a static method with a non-static method. You can, however, redeclare a static method of a super interface as a default method in the sub interface



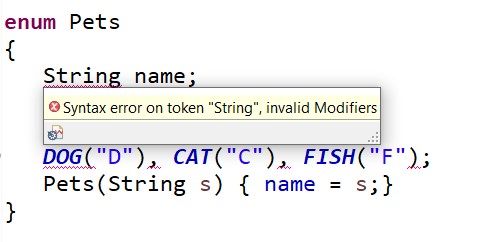
* + 1. Class: An instance method cannot override a static method, and a static method cannot hide an instance method



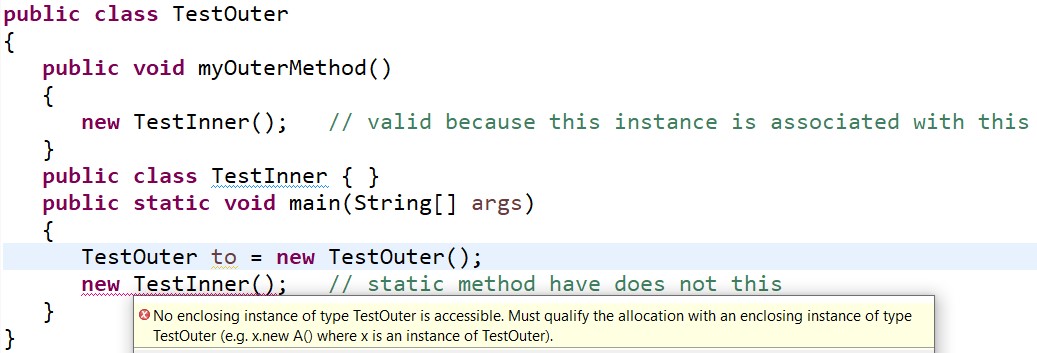
1. We cannot initiate the abstract class and interface, but we can create instance of an *anonymous subclass* of our abstract class. Ex **A a = new A(){}**;
2. **(access modifier )Private < Package-private<protected<Public**
3. **Default means package-private means within the package while protected means with-in-the-package plus outside package in child so more accessibility.**
4. **Instanceof** is an operator rather than a method. If a class is not subclass of it during Instanceof operator it gives the compile time error. ex :- anotherHippo instanceof Elephant; // DOES NOT COMPILE
5. **Object** wait() method throws **InterruptedException** and **IllegalMonitorStateException.**
6. If the equals() **method is overridden**, the hashcode() method should also be overridden such a way same hashCode is returned that are equal as per the equals method.
7. Joshua Bloch says on Effective Java:You must override hashCode() in every class that overrides equals(). Failure to do so will result in a violation of the general contract for Object.hashCode(), which will prevent your class from **functioning properly** in conjunction with all hash-based collections, including **HashMap, HashSet, and Hashtable**.
8. The hash code is used when storing the object as a key in a map.
9. The map put() methods **return** the previous value associated with **key**, or **null** if there was no mapping for key.
10. Enum:
    1. All enum implicitly extend java.lang.Enum . An enum is implicitly final, which means you cannot extend but you can implement it. Enum ordinal stands form the 0.
    2. Enum method
       1. Static values(),
       2. Static valueOf(String) if not found throws java.lang.IllegalArgumentException,
       3. name(), ordinal(), compareTo(Enum)
    3. The enum method can be marked abstract for the value and is used for calling the value methods.
    4. Enum implements comparable by their **natural order** of their **ordinal** as them implicitly implement comparable.

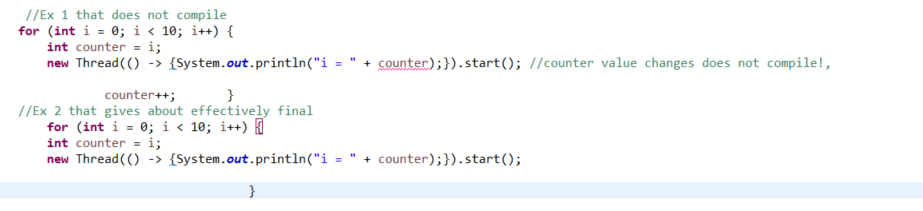


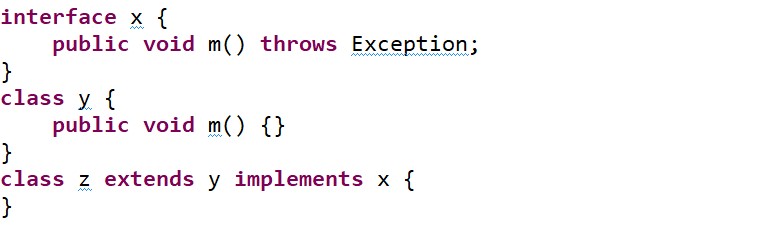
* 1. If enum constructor does not have access modifier then it is fine but there should be no **public** constructor.
  2. **Since enum maintains exactly one instance of its constants, you cannot clone it. You cannot even override the clone method in an enum because java.lang.Enum makes it final.**
  3. Enum constants (here, DOG, CAT, and FISH) must be declared before anything else.

 String name is invalid here.

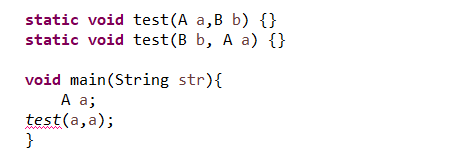
1. We can create a abstract method and then every enum value is required to implement this method or we can create a default implementation and override it. like   
   public void prinln(){SYSOUT(“DEFAULT”);
2. **Automatic variables** are called local variables in java.
3. Nested Class
   1. We can treat nested classes as the member inside the class.
   2. An Inner class can extend outer class.
   3. Type of nested class
      1. Static **nested class // it is not inner class**
         1. Inside static nested class superclass non static variables are not allowed while private static and public static variables are allowed.
      2. Inner class (Non static) inner class // Member Inner Classes
         1. The instance of member inner class can only be created by object of outer class.
         2. Can be declared public, private or protected or use default access
         3. Can extend any class and implement interfaces
         4. Cannot declare static fields or methods inside
         5. Can access members of the outer class.
      3. Locale inner class
         1. They do not have an access specifier (except final as same as method). ie private class A{}// invalid
         2. They cannot be declared static and cannot declare static fields or methods.
         3. They have access to all fields and methods of the enclosing class.
         4. They do not have access to local variables of a method unless those variables are final or effectively final. More on this shortly.
      4. Anonymous inner class
         1. An anonymous inner class must either extend a class by name or implement exactly one interface.
   4. Inner classes(not nested class) are not allowed to contain static methods or static variables because to initiate inner class we required the reference of class objects but static does not require instance of reference.
   5. Calling inner class method:



1. Method from the interface are always public as it gives error while making private or protected access modifier of subclass: Cannot reduce the visibility of the inherited method from myInterface.
2. A variable is not declared as final but whose value is never changed after initialization is effectively final.
3. We cannot use object function like toString() etc. on the primitive.
4. After declaration of default method in interface we cannot overload the object method in interface.
5. Functional interfaces are taken as method rather than interface so treat as it with return of method as return of functional interface. We must add function logic to it just.
6. Variable scope :- Members variable(Class Level) and Local Variable(Method Level) Local variable don’t exist after method execution is over.
7. **Which variable (or static method) will be used depends on the class that the variable is declared of. Which instance method will be used depends on the actual class of the object that is referenced by the variable.** **enthuware.ocpjp.v8.2.1530**
8. Following does not override cause error.



* 1. Same way if an interface has equals method like the comparator have and a concrete class does have to implement equal method because the class had already implemented the equals method.
  2. Remember here x interface throws Exception while class y does not throws exception.

1. The method test(A, B) is ambiguous for the type Class (Compile Error).
2. ActionListener is an interface which has only one method called ActionPerformed.
3. A file can have any number of classes with main method. And we must compile them with class name not the file name.
4. **Interface**
   1. **why public?**: to make the method available to every implementation class
   2. **why abstract**?:to implements class
   3. Interface method cannot have synchronized, final
   4. An interface can have variable, the main purpose of interface variables to define requirement level constants.
   5. Every interface variables are public static finals. They are static because without existing object class implementation they have access to variables.
   6. They are final because if a one class implementation changes it value then remaining implementation will be affected to restricts this every interface variable is always final.
   7. Inside implementation class we can access interface variables, but we cannot modify values.
   8. Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.
   9. **We can call default method using object reference.**
5. **Naming conflict:**
   1. If two interface contains a method with same signature and same return types then in the implementation class we must provide the implementation for only one method.
   2. If two interface contains a method with same name but different argument types then in the implementation class we have to provide implementation for both the methods and these method acts as overloaded methods.
   3. If two interface have same default name method we must define methods and return with interface\_name.**super**.Method\_name.
   4. If return types of two method i.e. int m1() and void m1() is different then we can not write any java class which implements both interface simultaneously.
6. Is a java class can implement any number of interface simultaneously?
   1. Yes, except a particular case, if two interface contains a method with same signature but different return types then it is impossible to implements both interface simultaneously.
7. **Variable conflict**
   1. It gives CE reference to variable is ambiguous.
   2. But we can use interface variables using interface name in the child class.
8. **Marker Interface:** If an interface does not contain any methods and by implementing the interface if our objects will get some ability(i.e. cloneable, serializability) such types of interfaces are called marker interfaces. Eg.
   1. Serializable interface
   2. Cloneable interface: For duplicating objects.
   3. RandomAccess
   4. SingleThreadModel
9. Without having any method how the objets will get ability in marker interfaces?
   1. Internally JVM is responsible to provide required ability.
10. Why JVM is providing required ability in marker interfaces?
    1. To reduce the complexity of programming and to make java language as simple.
11. Can we create our own Marker Interface?
    1. Yes, but customization of jvm is required.
12. Advantage of abstract class over interface.
    1. Abstract class have constructor, but interface doesn’t have so it is easy to create objects.
    2. If it is everything is abstract that is highly recommended to go with interface not abstract class, we can replace abstract class with interface, but it is not good practice. It is like recruiting IAS officer for sweeping purposes.
13. **Adapter Class** is a simple java class that implements an interface with only empty implementation. If we implement an interface for each and every method of interface compulsory we should provide implementation whether it is required or not required. The problem with this is that it increases the length of the code and reduces readability. We can solve this problem by using adapter classes. Instead of implementing interface if we extend Adapter class we have to provide implementation only for required method and we are not responsible to provide the implementation for each and every method of the interface. So the length of the code will be reduce.
14. C# 4 Language Specification § 8.9.5: If **the finally block** throws another exception, processing of the current exception is terminated

# Design pattern and principles

1. **Is-a** relationship also known as inheritance, it used for code reusability,
2. **Has-a** relationship is also known as Object Composition or Aggregation, it is mainly made by the new keyword. It can be thought of as an **alternate** to inheritance.
   1. E.g. Bird **has-a** beak rather than Bird **is-a** beak

public class Penguin {

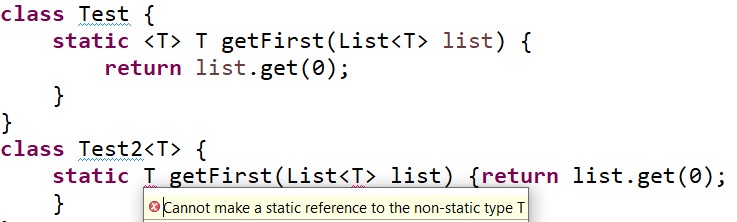
private final Flippers flippers;

private final WebbedFeet webbedFeet;}

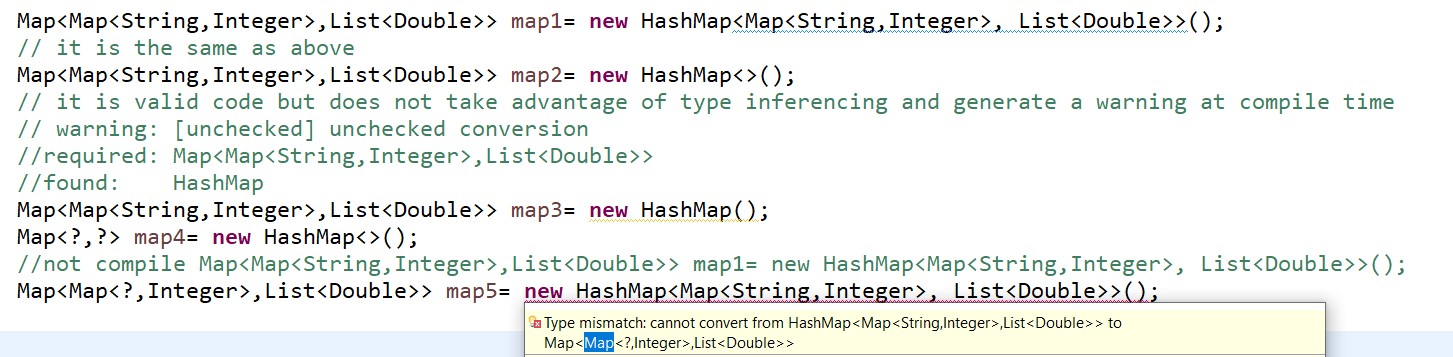
1. In case of composition whenever container object is destroyed all contained objects will be destroyed. But in the inheritance, it is not necessary that contained objects are also destroyed.
2. Encapsulation is design principle. A JavaBean is a design principle for encapsulating data in an object in Java. Getters for boolean properties may begin with is or get and get only for non-boolean properties.
3. For the singletons pattern : create 1 object
   1. Make the constructor private
   2. Ensure that object is created only once, used synchronized specifier where the object is created.
   3. Add a **private** static variable of the same class to the class and instantiate it.
      1. private static Console cons;
      2. private static Runtime currentRuntime = new Runtime();
   4. **Requires a *public static method Class getInstance* to retrieve the instance of the singleton**. as in Runtime run = Runtime.getRuntime() or Console con=System.console() or Calendar cal = Calendar.*getInstance*();
   5. Lazy inssitialization:
      1. Creating a reusable object, the first time it is requested is a software design pattern **known** as lazy instantiation.
      2. Don’t initialize as above
      3. Instantiate Singleton object in *public static synchronized Class getInstance* method if and only if Singleton object hasn’t created (reference of Singleton object == null)
      4. So, Singleton object can’t be *final* in this case as *final* variable have no default value and can’t be reassigned
4. For immutable strategy,
   1. use a **constructor** to set all properties of the object
   2. mark all the instance variable private final.
   3. Don't define any **setter** methods,
   4. Don't allow referenced mutable objects to be modified or accessed directly,
   5. Prevent methods from being overridden by making class or methods final or apply factory pattern.
5. Factory Pattern,
   1. Create a factory method that return instance or objects based on the set input
6. For Java Beans
   1. Private properties
   2. Getter for non-Boolean properties begins with get
   3. Getter for Boolean properties begins with is or get
   4. Setter begins with set
7. **Difference between immutable and Singleton vs Synchronized vs ThreadLocal**
8. Immutable
   1. The state of object doesn’t change after creation
   2. Since object state doesn’t change the hashcode assigned by JVM also remains same
   3. We can used immutable objects as keys in HashMap for faster retrieval
   4. Their state doesn’t change they can be used in concurrent programming without any need for synchronization .
   5. E.g. String, wrapper class like Integer, Long
9. Mutable
   1. ex **StringBuffer** any modification is done on object reflects.
   2. Since mutable object cannot be reused are just a use and throw so improper usage may lead to large amount of garbage.
   3. Make to return new object by new or clone to modify the default data. Ex new Date(data) or (date)dateData.clone
   4. The final class is immutable class but the variable need to be hard coded as it cannot be defined in setters. So, to avoid these to declare the final variable and initialize using the constructor
10. Singleton v/s Immutable
    1. **An immutable object is initialized by its constructor only while a singleton is instantiated or used by a static method.**
    2. If a singleton A provides a reference to mutable object B then B is singleton as well
11. Singleton v/s Synchronized
    1. Singleton restricts at class level while synchronized restricts access at method level only
    2. So, synchronized have better performance than singleton since in case of synchronized constructor cannot be synchronized , to solve this problem we must go for singleton only or keep the constructor as private
12. ThreadLocal in java is another way to achieve thread-safety apart from writing the immutable classes.
13. Singletons are used in situations where we need access to a single set of data throughout an application. For ex: application configuration data and reusable data caches are commonly implemented using singletons. Singletons may also be used to coordinate access to shared resources, such as coordinating write access to a file. **Java.io.Console** use singleton pattern.
14. **Native modifier** always applies on method and we cannot apply anywhere else. The method which are implemented in non-java{mostly c++} are called native method or foreigner methods’ public native int hashcode() as it must communicate with the memory level for the address.
    1. The main motive of native keyword to improve the performance of the system.
    2. To achieve machine level or memory level communication.
    3. To used already existing legacy non java code.
15. **Sudo code to use native code in java:**
    1. Load the native library.
    2. Declare a native method with native keyword.
    3. Invoke a native method.

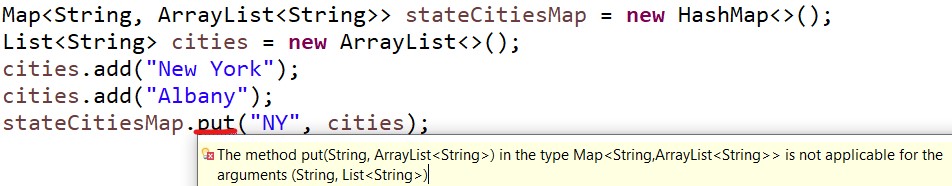
# Generics and collections

1. Java think that a generic type is an Object due to **Type Erasure**.( Type erasure is the technique using which the Java compiler translates generic / parameterized type to raw type in Java generics.)
2. Using generics, type parameter to be static. **static** T data;



1. In generic methods the formal type parameter is added before the return type.



Also remember type must be added as same i.e.

1. The super bound is not allowed in class definition. The below code does not even compile.



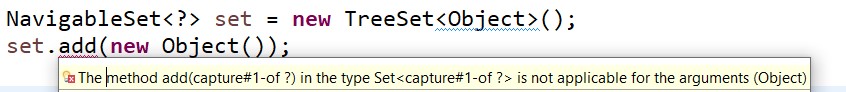
1. Super type does not have any wild cards.

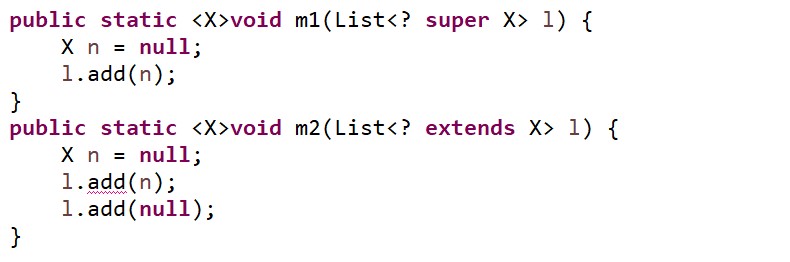


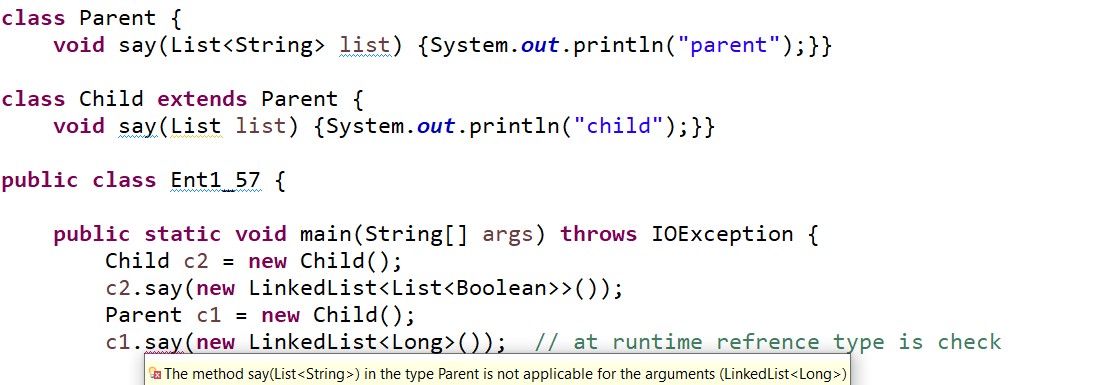
But below is valid:



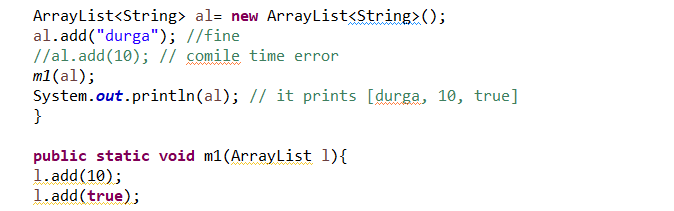
1. Generic can be defined on:
   1. Interface and class can use generics can of three types:-
      1. *class MySub extends GenericClass<String> implements GenericFace1<Integer>*
      2. *class MySub<T> extends GenericClass<T> implements GenericFace2<T, Integer>*
      3. *class MySub<T> extends GenericClass2 implements GenericFace2<T, Integer>*
   2. Generic on variable declaration
      1. List<? super Integer> list = new ArrayList<>();
2. There are three type of bound wildcards
   1. Unbounded Wildcard (**?**)
   2. Upper- bounded wildcards: (? extends type) ? extends type or type. Ex:-
      1. List<? **extends** Exception> list =**new** ArrayList<RuntimeException>();
      2. List<? **extends** Number> list = **new** ArrayList<Integer>();
      3. List<? **extends** Integer> list = **new** ArrayList<Integer>();
   3. Lower-bounded wildcards: (? super type) ? superclass of String or type itself ex: pg. 123
      1. List<? super Exception> list =new ArrayList<Object>();
      2. List<? extends A> list2 = new ArrayList<A>();
      3. boolean allMatch(Predicate<? super T> check)
3. Generics Methods:
   1. m1(ArrayList<String> l)
      1. Within the method we can add only string type of objects to the list. by mistake if we are trying to add another type. We will get compile time error.
   2. m2(ArrayList<?> l)
      1. We cannot add anything to except null it by l.add(), as we cannot add any type to it. But we can use it for printing purposes.
      2. Null is valid because it is any type.



* 1. m3(ArrayList<? extends x> l)
     1. Here we don't know which child we are calling so we cannot add except null. This type of method also best suitable read only operation.
     2. Since due to widening in java if we pass subclass then it automatically converted to higher class.
  2. m4(ArrayList<? super x> l)
     1. x can be either class or interface if x is class then we can call this method by passing ArrayList of either x type or its super classes. If x is interface then we can call this by method by passing ArrayList of x type or its super classes.
     2. Within the method we can add null as well as x (Because narrowing is explicit cast) but we cannot **add super of x** as we doesn’t know the exact type of the Class.
  3. some example:
     1. ArrayList<?> l= new ArrayList<?>(); ce: unexpected type, found ?, required class or interface without bounds.
     2. ArrayList<?> l= new ArrayList<? extends Number>(); ce: unexpected type, found ? extends Number, required class or interface without bounds.
     3. ArrayList<> l= new ArrayList<?>(); //<> are found in RHS only



* 1. We can define bounded types at method level also.
     1. <T extends Number> void m1(T t) // fine
     2. <T extends Number & Comparable & Runnable> // fine
     3. <T extends Runnable & Thread>//not fine First we have to take class then interface
     4. <T extends Number & Thread> //Not fine we cannot extend more than one class
  2. If we send generic object to non-generic area it behaves like a non-generic object. Similarly, if we send non generic object to generic area it start behaving like generic objects. The location on which object present the behavior will be defined.



* 1. This is because generics applicable only at compile time, so it does not work on runtime.
  2. The main purpose of generic is to provide type safety and to resolve type casting problems. Type safety and type casting both are applicable at compile time, hence generic concepts also only at compile time but not at run time. At compilation at last step generics syntax will be removed and hence for jvm generic syntax would not be available. Hence the following declaration are equals.
     1. ArrayList l= new ArrayList<String>();
     2. ArrayList l= new ArrayList<Integer>();
     3. ArrayList l= new ArrayList<Double>();
     4. ArrayList l= new ArrayList();
  3. The below ArrayList are equal to each other
     1. ArrayList<String> l= new ArrayList<String>();
     2. ArrayList<String> l= new ArrayList();
  4. The below method will give compile time error in declaring a block: name clash: Both methods have same erasure. While using as method overloading.
     1. void m1(ArrayList<Integer> l){}
     2. void m1(ArrayList<String> l){}
     3. At compile time jvm steps: Compile code using generics then remove the generic code using type erasure and then compile it.

1. Invalid Generic declaration:
   1. Map<String, List<String>> stateCitiesMap = new HashMap<String, List<>>();
      1. You cannot embed a diamond operator within another generic class instantiation.
   2. Map<String, List<String>> stateCitiesMap = new HashMap<String, ArrayList<String>>();
      1. Type mismatch: cannot convert from HashMap<String,ArrayList<String>> to Map<String,List<String>>
   3. List<Integer> list2 = new ArrayList<Integer>(Arrays.asList(p));
   4. List<?> list2 = new ArrayList<>(Arrays.asList(p));

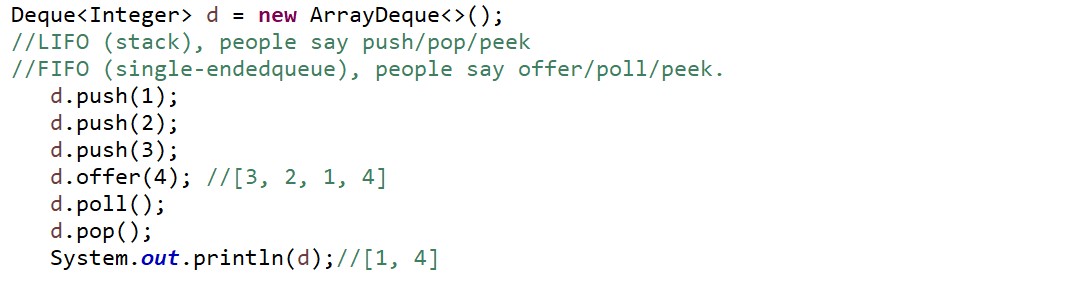
It is valid.

static <X> PlaceHolder<X, X> getDuplicateHolder(X x){ //3  
        return new PlaceHolder<X, X>(x, x); //4  
    }

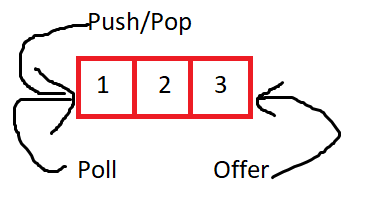
1. Arrays.asList(Arrays is [] to list) returns a List wrapper around an array. This wrapper has a fixed size and is directly backed by the array, and as such calls to set will modify the array, and any other method that modifies the list will throw an UnsupportedOperationException.



1. Java choose one name for **data class** and plural form of that name for the **factory or helper** class ex. Collection and Collections , Path and Paths
2. java.lang.Comparable use natural sorting with compareTo() and java.util.Comparator uses used define sorting using **compare**()
   1. Here positive means swapping between the value
   2. compareTo(T o): it returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. So o1.compateTo(o2) compare value in ascending order.
3. **Comparator** interface method:-
   1. Compare(T o1,T o2)
   2. Comparint()
   3. naturalOrder()
   4. thenComparing() : Returns a lexicographic-order comparator with a function that extracts a Comparable sort key.
4. **Sorting** **:** spaces < numbers < uppercase < lowercase
   1. try to sort a collection of class without passing Comparator or Comparable criteria will casue runtime exception, not compile time
   2. Sorting a collection containing not Comparable object will cause compile error
5. list.iterator().next()); // it gives the iterator element from the starting
6. For each method
   1. java.lang.Iterable.forEach(Consumer<? super String> action)
   2. forEach() is the **only** terminal operator that return void
   3. forEach() method of map use BiConsumer rather than consumer.
7. [NavigableMap](eclipse-javadoc:%E2%98%82=OcpBook/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava.util(TreeMap.class%E2%98%83TreeMap~subMap~TK;~Z~TK;~Z%E2%98%82java.util.NavigableMap):
   1. NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive,K toKey, boolean toInclusive):Returns a view of the portion of this map whose keys range from fromKey to toKey. If fromKey and toKey are equal, the returned map is empty unless fromInclusive and toInclusive are both true.
   2. NavigableMap<K,V> **tailMap**(K fromKey, boolean inclusive): Returns a view of the portion of this map whose keys are greater than (or equal to, if inclusive is true) fromKey.
8. Important **data structure** doesn’t allow nulls are:-
   1. Hashtable: no null keys or values
   2. TreeMap: no null keys
   3. TreeSet: no null element
   4. ArrayDeque: no null element b/c method like poll() use null as a special return value to indicate that the collection is empty. Since null has that meaning, Java forbids putting a null in there.
   5. ConcurrentHashMap: neither null key nor null **value gives** NullPointerException. It uses the bucket locking not the object locking method, so it is faster.
9. The **standard collection classes** that are thread safe are only Vector and HashTable.
10. **Arrays.parallelPrefix(ls, Long::sum);**
    1. **public static <T> void parallelPrefix(T[] array, BinaryOperator<T> op)**
    2. For example, if the array initially holds [2, 1, 0, 3] and the operation performs addition, then upon return the array holds [2, 3, 3, 6]
11. List interface methods
    1. void add(int index, E element)
    2. E get(int index)
12. **Set** interface
    1. It checks the element bucket by its hashCode() and equals() method to compare individual buckets.
13. NavigableSet
    1. E floor(E e)
    2. E ceiling(E e)
14. Queue interface When talking about LIFO (stack), people say push/pop/peek. When talking about Stack [FIFO] (single-ended queue), people say offer/poll/peek

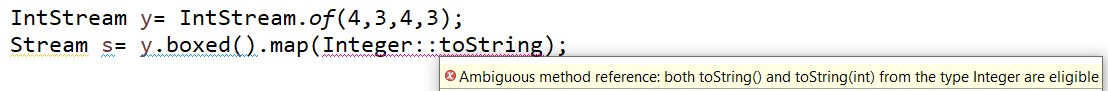


1. Deque data **[**Front Side |Backside**] -> [0,1,2,3…]** memory address as we see.

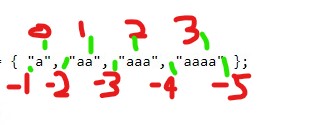


* 1. Removal occurs from the front side
  2. Boolean **Add**(E e) add elements to back Return true or exception (ClassCastException, NullPointerException, IllegalArgumentException)
  3. Boolean Offer(E e) add elements to back Return true or false
  4. void Push(E e) add element to front or exception (ClassCastException, NullPointerException, IllegalArgumentException)
  5. E **Element**() Return next element or exception
  6. E Peek() Return next element or null
  7. E Remove Return next element exception (Remove that element)
  8. E Poll Return next element or null (Remove that element)
  9. E Pop() Return next element or NoSuchElementException (Remove that element)

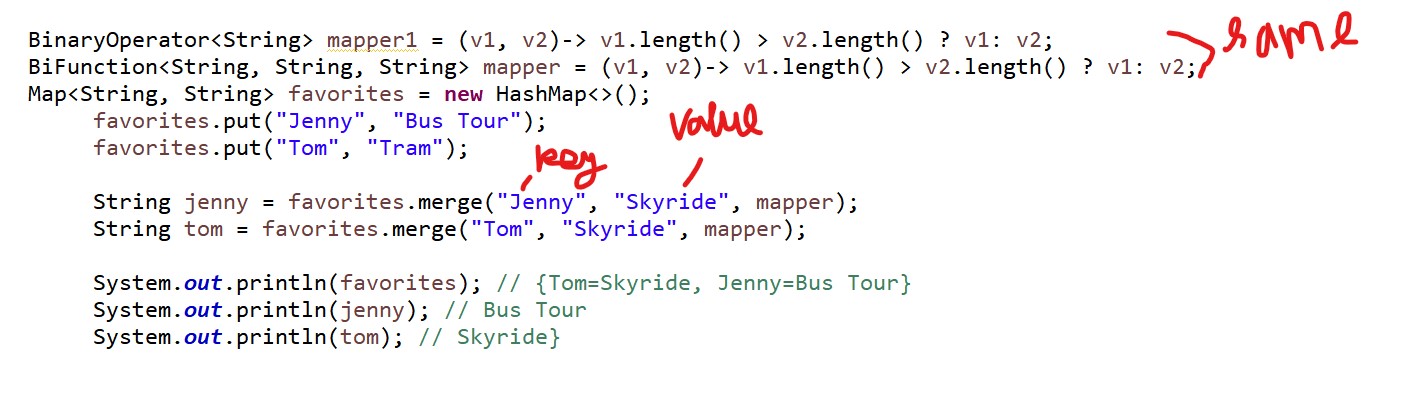
1. Public interface Deque<E> **extends** Queue<E> {}
   1. OfferFirst(E e) and offerLast element
   2. removeLast(E e) and removeFirst(E e)
2. String java.util.ArrayDeque
   1. peek() --> the head of the queue represented by this deque, or null if this deque is empty
   2. boolean java.util.ArrayDeque.**removeFirstOccurrence**(Object o) ;
   3. boolean [java](eclipse-javadoc:%E2%98%82=OcpBook/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava).[util](eclipse-javadoc:%E2%98%82=OcpBook/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava.util).ArrayDeque.**removeLastOccurrence**([Object](eclipse-javadoc:%E2%98%82=OcpBook/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava.util(Deque.class%E2%98%83Deque~removeFirstOccurrence~Ljava.lang.Object;%E2%98%82java.lang.Object) o)
3. **HashSet** allows null and it has no sorting and order does not maintain in the order.
4. **TreeSet** doesn’t allow null(throw NullPointerException) and it is sorted and maintains the order.
5. NavigableSet Interface eg.134
   1. E lower(E e) Returns greatest element that is < e, or null if no such element
   2. E floor(E e) Returns greatest element that is <= e, or null if no such element
   3. E ceiling(E e) Returns smallest element that is >= e, or null if no such element
   4. E higher(E e) Returns smallest element that is > e, or null if no such element
6. **Method reference :** it uses functional interface to input method as body of lambda and auto match the parameter list of lambda without explicitly written any parameter
   1. Comparator<Duck> byWeight = (d1, d2) -> DuckHelper.compareByWeight(d1, d2);
   2. Into with method reference
   3. Comparator<Duck> byWeight = DuckHelper::compareByWeight;
   4. :: tell that java must pass the tow parameter automatically and Java looks for a method that matches that description or context.
7. There are four examples:
   1. **Reference to a static method**: ContainingClass::staticMethodName
   2. **Reference to an instance method of a particular object:**
      1. containingObject::instanceMethodNameInstance
   3. **Reference to an instance method of an arbitrary object of a particular type** ContainingType::methodName
   4. **Reference to a constructor**: ClassName::new

****

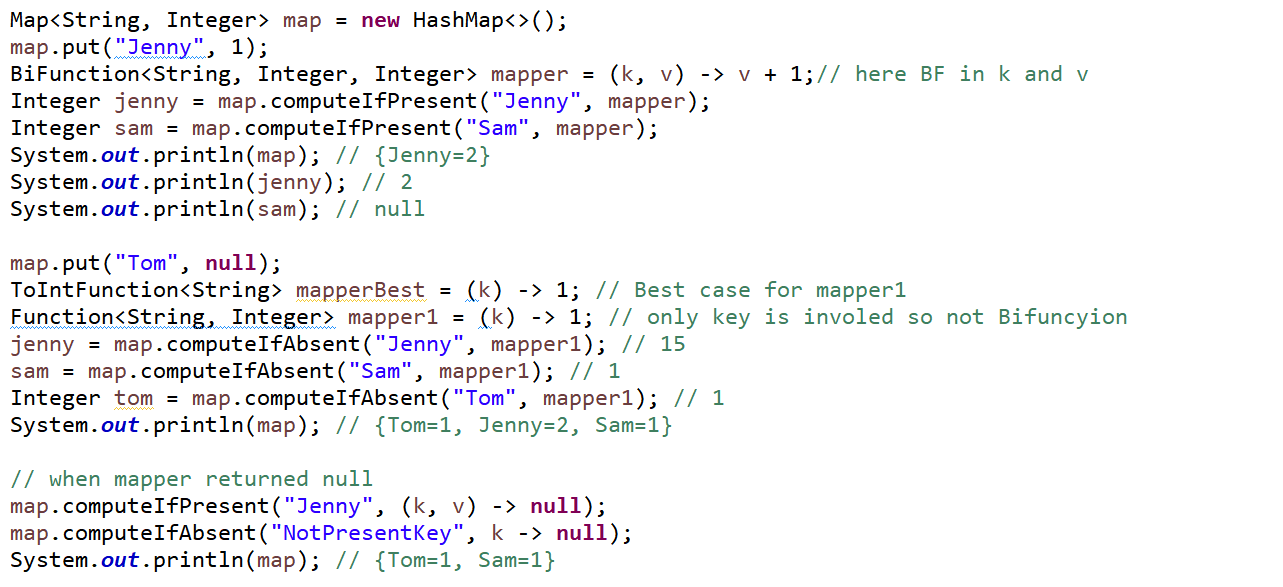
1. Collections.unmodifiableCollection() and various other method throw UnsupportedOperationException.
2. Various syntactically illegal method in method reference.
   1. map(x->MyProcessor::new(x)) // throws Syntax error on token(s), misplaced construct(s)
   2. forEach((x)->process( MyProcessor::new ));
3. some method ref link
   1. [moandjiezana.com](http://moandjiezana.com/blog/2014/understanding-method-references/)
   2. [baddotrobot.com](http://baddotrobot.com/blog/2014/02/18/method-references-in-java8/)
   3. [http://cr.openjdk.java.net/](http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html)
4. LinkedList means under List so ordered collection.
5. **Binary search**: It returns the index of the search key, if it is contained in the array; otherwise, (-(insertion point) – 1). The index start from 0.

So for a string arrary String[] sa = { "a", "aa", "aaa", "aaaa" }; we can get from -5 to 3 possible value as 

1. **Using New Java 8 Map APIs**
   1. **putIfAbsent**() method skips it if the value is already set to a non-null value.
   2. **Remember** putifPresent **is not present.**
   3. The merge() method allows adding logic to the problem of what to choose.
   4. Integer java.util.Map.merge(Integer key, Integer value, BiFunction<? super Integer, ? super Integer, ? extends Integer> remappingFunction)

The key is removed from the map when mapper returns null.

* 1. computeIfPresent and computeIfAbsent: computeIfPresent() calls the BiFunction if the requested key is found and opposite for computeIfAbsent.



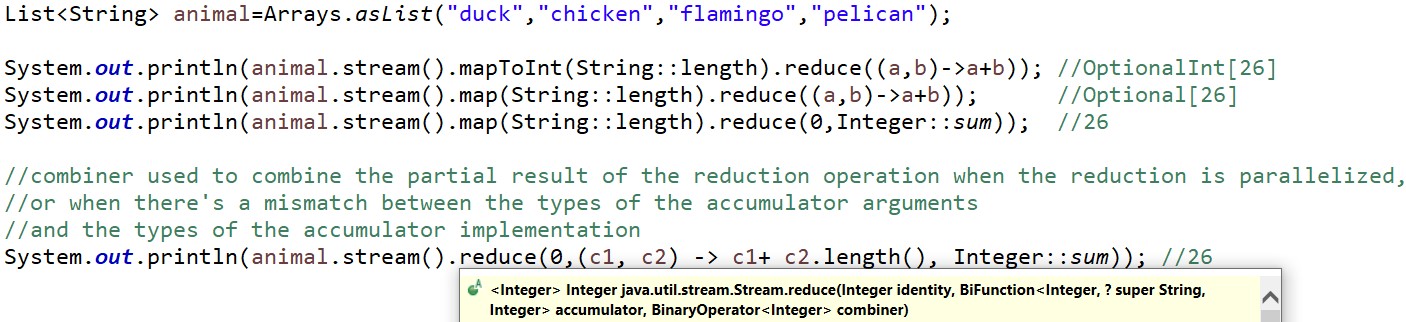
# Functional programming

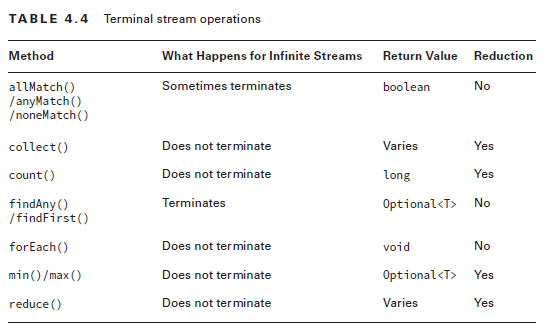
1. In Lambda expression without curly braces we cannot use return expression and vice versa.
   1. n->{n\*n;}; it depends upon the functional method, return statement is required with {} if the functional method needs to return a value.
   2. n->{return n\*n}; // invalid due to missing semicolon
   3. n-> // invalid no RHS
2. **All the functional interface present**

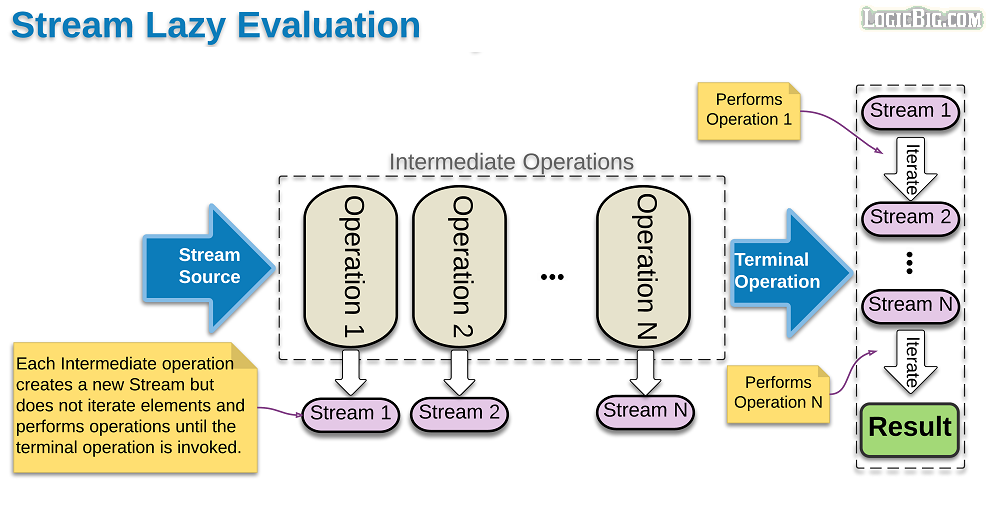
* public interface BiConsumer<T, U> {}
  + void accept(T t, U u­);
* public interface BiFunction<T, U, R> {}
  + - R apply(T t, U u);
* public interface BinaryOperator<T> extends BiFunction<T,T,T>{}
  + - T apply(T t, T t);
* public interface BiPredicate<T, U> {}
  + - boolean test(T t, U u);
* public interface BooleanSupplier {}
  + - boolean getAsBoolean();
* public interface DoubleSupplier {}
  + - double getAsDouble();
* public interface Consumer<T> {}
  + - void accept(T t)
    - default Consumer<T> **andThen**(Consumer<? super T> after) {}
      * It is present in all the interface that link like BiConsumer, DoubleConsumer, IntConsumer, LongConsumer
* public interface Function<T, R> {}
  + - R apply(T t);
    - default <V> Function<V, R> **compose**(Function<? super V, ? extends T> before){}
      * Returns a composed function that first applies the before function to its input, and then applies this function to the result.
      * square.compose(negate).apply(11)); gives 121
    - default <V> Function<T, V> **andThen**(Function<? super R, ? extends V> after) {}
      * Returns a composed function that first applies this function to its input, and then applies the after function to the result
      * Eg. square.andThen(negate).apply(11)); gives -121
    - static <T> Function<T, T> identity() {}
* public interface Predicate<T> {}
  + - boolean test(T t);
* public interface Supplier<T> {}
  + - T get();
* public interface UnaryOperator<T> extends Function<T, T> {}
  + - T apply(int t)
* public interface DoubleBinaryOperator {}
  + - double applyAsDouble(double left, double right);
* public interface DoubleConsumer {}
  + - void accept(double value);
* public interface DoubleFunction<R> {}
  + - R apply(Double d)
* public interface DoublePredicate {}
  + - boolean test(Double d)
* public interface DoubleToIntFunction {}
  + - int applyAsInt(double value);
* public interface DoubleToLongFunction {}
  + - long applyAsLong(double value);
* public interface DoubleUnaryOperator {}
  + - double applyAsDouble(double operand);
* public interface IntBinaryOperator {}
  + - int applyAsInt(int left, int right);
* public interface IntConsumer{}
  + - void accept(int value);
* public interface IntFunction<R> {}
  + - T apply(Int i)
* public interface IntPredicate {}
  + - boolean test(int i)
* public interface IntSupplier {}
  + - int getAsInt();
* public interface IntToDoubleFunction {}
  + - double applyAsDouble(int value);
* public interface IntToLongFunction {}
  + - long applyAsLong(int value);
* public interface IntUnaryOperator {}
  + - int applyAsInt(int operand);
* public interface LongBinaryOperator {}
  + - long applyAsLong(long left, long right);
* public interface LongConsumer {}
  + - void accept(long value);
* public interface LongFunction<R> {}
  + - R apply(long value);
* public interface LongPredicate {}
  + - boolean test(long value);
* public interface LongSupplier {}
  + - long getAsLong();
* public interface LongToDoubleFunction {
  + - double applyAsDouble(long value);
* public interface LongToIntFunction {}
  + - int applyAsInt(long value);
* public interface LongUnaryOperator {}
  + - long applyAsLong(long operand);
* public interface ObjDoubleConsumer<T> {}
  + - void accept(T t, double value);
* public interface ObjIntConsumer<T> {}
  + - void accept(T t, int value);
* public interface ObjLongConsumer<T> {}
  + - void accept(T t, long value);
* public interface ToDoubleBiFunction<T, U> {}
  + - double applyAsDouble(T t, U u);
* public interface ToDoubleFunction<T> {}
  + - double applyAsDouble(T value);
* public interface ToIntBiFunction<T, U> {}
  + - int applyAsInt(T t, U u);
* public interface ToIntFunction<T> {}
  + - int applyAsInt(T value);
* public interface ToLongBiFunction<T, U> {}
  + - long applyAsLong(T t, U u);
* public interface ToLongFunction<T> {}
  + - long applyAsLong(T value);
* public interface UnaryOperator<T> extends Function<T, T>{}
  + - T apply(T t);
    - static <T> UnaryOperator<T> identity() { return t->t;}

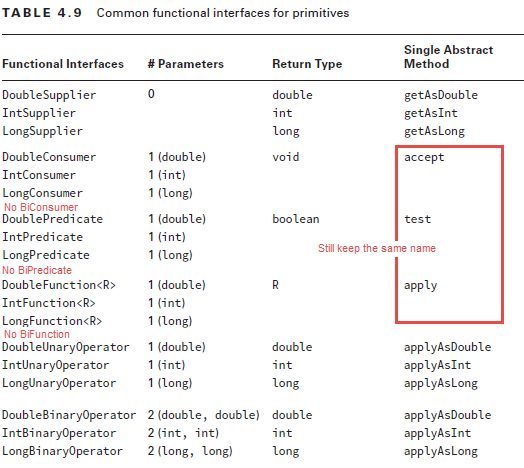
**To**(in function name) means there is not meaning of input type

1. Don't confuse Double with Binary here Double means Datatype not two whereas Binary means two
2. Generally, a function exists in **5** function class like test() occurs in Predicate,BiPredicate, primitive(int, double, long) but not necessary i.e. get() only in supplier other as getAsInt(), getAsDouble(), getAsLong().
3. In lambda expression if generic are not used then please not use there parameter
   1. Predicate even = (Integer i)-> i%2==0;  // give compile time error.
   2. Predicate<Integer> even = (Integer i)-> (Integer)i%2==0; // fine
   3. Predicate even = (Object i)-> (Integer)i%2==0; //fine
4. Stream does not implement iterable so can’t use for loop in it. It is same as assembly line in factory. The collection interface extends Iterable interface for iteration and it has method(Iterator<T> iterator()) while forEach() and Spliterator() are default method.
5. Common terminal operations :
   1. allMatch**/**anyMatch/noneMatch
      1. boolean java.util.stream.Stream.allMatch(**Predicate<? super Integer**> predicate)
      2. Unlike the anyMatch( ) method that returns false when the stream is empty, the allMatch( ) and noneMatch( ) methods return true if the stream is empty!
   2. collect
      1. <R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner)
   3. long count()
   4. findAny/ findFirst: Since element may not be present in Stream so they return Optional<T>
      1. Optional<T> findAny()
      2. Optional<T> findFirst()
   5. void forEach(Consumer<? super T> action)
   6. Optional<T> min(<? super T> comparator) Optional<T> max(<? super T> comparator)
      1. The stream take comparator while primitive does not take any parameter.
   7. reduce()
      1. identity: Initial value of the Stream and default value if Stream is empty.
      2. accumulator: a function that takes two parameters: a partial result of the reduction operation and the next element of the stream
      3. combiner: a function used to combine the partial result of the reduction operation when the reduction is parallelized, or when there's a mismatch between the types of the accumulator arguments and the types of the accumulator implementation
   8. some reduce functions:

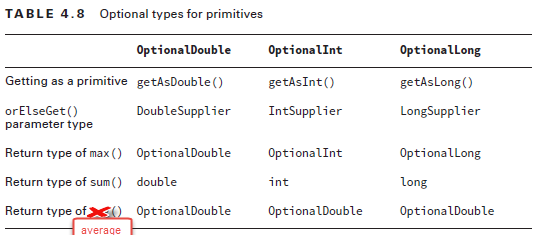




1. common intermediate operation
   1. filter() --> Stream<T> filter(Predicate<? super T> predicate)
   2. distinct()-> Stream<T> distinct()
   3. limit() -> Stream<T> limit(int maxSize)
   4. skip() -> Stream<T> skip(int n)
   5. map() -> <R> Stream<R> map(Function<? super T, ? extends R> mapper)
   6. flatMap()-> <R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
   7. sorted() ->
      1. Stream<T> sorted() : it uses the default sorted order
      2. Stream<T> sorted(Comparator<? super T> comparator)
   8. peek() ->Stream<T> peek(Consumer<? super T> action)
   9. To understand what flattening a stream consists in, consider a structure like [ [1,2,3],[4,5,6],[7,8,9] ] which has "two levels". Flattening this means transforming it in a "one level" structure : [ 1,2,3,4,5,6,7,8,9 ]
   10. **Boxed()** The method boxed() is designed only for streams of some primitive types (IntStream, DoubleStream, and LongStream) to box each primitive value of the stream into the corresponding wrapper class (Integer, Double, and Long respectively).
2. Those function in Stream whose return are Stream are lazy function while those that gives values are not lazy function. 
3. **Working with primitive versions of functional interfaces.**
   1. Performance can suffer when we use wrapper types: think of boxing and unboxing a few million integers in a stream. To avoid this performance problem, you can instead use relevant primitive versions of these functional interfaces.
   2. Generics are gone from some of the interfaces, since the type name tells us what primitive type is involved. In other cases, such as IntFunction, only the return type generic is needed.

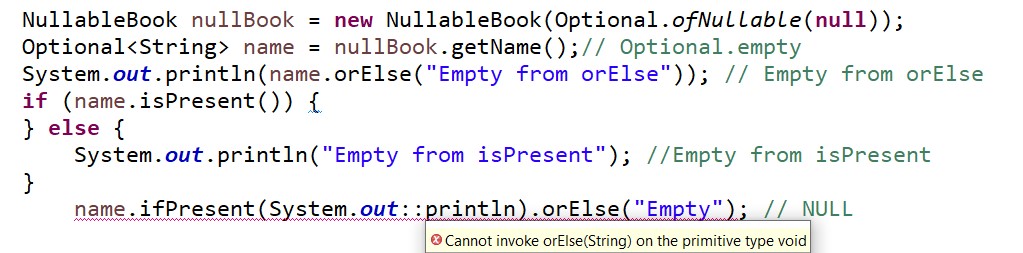


* 1. Primitive optional

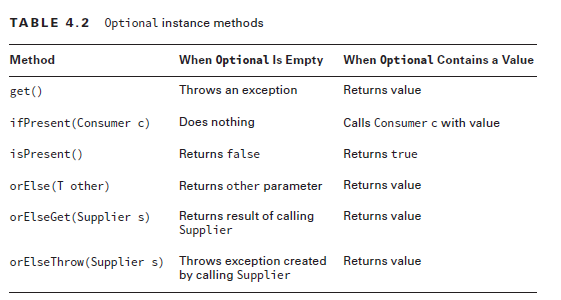


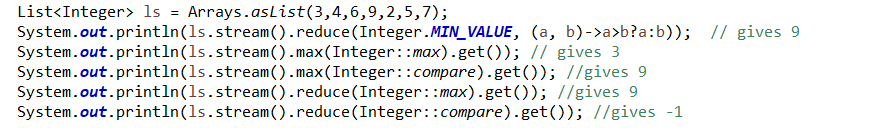
* 1. The primitive streams have math operations including average(), max(), and sum() and return type is terms of OptionalDouble for average and for further see above dia.
  2. Return type of **average**() in Optional primitive is **OptionalDouble**() and we have to use getAsDouble() for getting value rather than get(). It throws java.util.NoSuchElementException if it cannot find the value.
  3. While the return type of primitive stream collector function for average is double i.e. for averagingDouble (ToDoubleFunction f), averagingInt(ToIntFunction f), averagingLong(ToLongFunction f) is double.
  4. What is the only primitive functional interface that doesn’t involve double, int, or long? BooleanSupplier

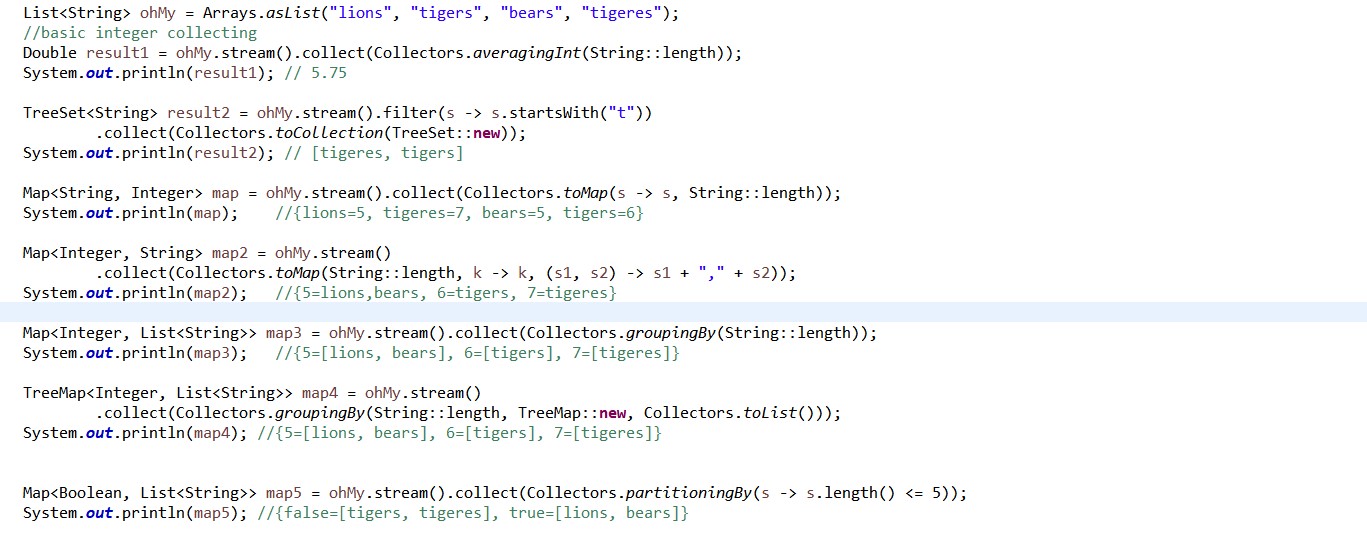
1. Optional class is a holder for value that can be **null**. Some methods are:-
   1. ifPresent(Consumer<? super T> consumer) {}



* 1. T orElse(T other) {} :if **T** has method in else it always run (Optional.isPresent() or not)
  2. T orElseGet(Supplier<? extends T> other) {} : Optional.isPresent() is false then other() method not run. Here get is used for Supplier as we have used in Supplier functional Interface.
  3. <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier) throws X{}



1. To create null objects in Optional we use:-
   1. Optional ops**=** Optional.**ofNullable**(null) // Optional.empty
   2. If we use Optional.of(null) It gives NullPointerException.
   3. Optional.ofNullable(value) and Optional.of(Value) give same output but of() give nullpointerexception if the value is null while other not that can be help in business.
2. Primitive Optional class
   1. java.util.OptionalDouble,
   2. java.util.OptionalLong,
   3. java.util.OptionalInt
      1. Empty()
      2. of()
      3. int **getAsInt**(){}
      4. boolean isPresent() { }
      5. void ifPresent(IntConsumer consumer) { }
      6. int orElse(int other) {}
      7. **int** orElseGet(IntSupplier other) {}
      8. <X extends Throwable> int orElseThrow(Supplier<X> exceptionSupplier) throws X {}
3. There is no mapToBoolean function in functional interface
4. The sum() method does not return an optional and by empty stream we got zero and min / max give optional .
5. **IntSummaryStatistics** stats = ints.**summaryStatistics**(); and stats never consume after termial operation.
6. **In supplier we have** for primitive we have getAsDouble, getAsInt, getAsLong
7. **In** primitive operator function like **IntUnaryOperator we use applyAsPrimitive like applyAsInt.**
8. Stream.*concat*(**stream1,stream2**): Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. The resulting stream is ordered if both input streams are ordered, and parallel if either of the input streams is parallel. When the resulting stream is closed, the close handlers for both input streams are invoked.
9. Stream.flatMap() : Returns a stream consisting of the results of replacing each element of this stream with the contents of a **mapped stream** produced by applying the provided **mapping function** to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an **empty** stream is used, instead.)
10. IntFunction<IntUnaryOperator> inFu = x -> y -> x\*y;
11. **Reduce and Max function:**
12. **Collecting using basic collections.**

**Collectors.groupingBy() more:**

****

1. **Joining Method:** joining method doesn’t takes any functional interface as argument.

****

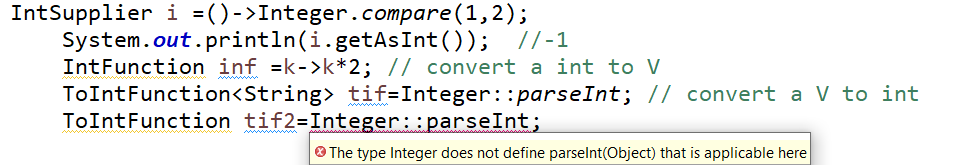
1. **Consumer<Integer> c =(Integer k) ->{k+=2;};**
   1. += is compound statement of assignment operator and it work only on primitive.



It is like a=5; and it does not return anything.

* 1. {} does not return anything else valid.

1. PrimitiveFunction IntFunction v/s ToIntFunction



# Dates, String and Localization

1. Two way to represent the time a. Human terms( year, month, day,min, sec) and machine time ( measuring time from origin called epoch). The Java 8 date and time API uses ISO 8601 as the default calendar format.
2. All immutable thread safe and use factory static method to instantiate:
   1. LocalTime :- ISO-8601 calendar system format: HH:MM:SS.nanosecond
   2. LocalDate
   3. LocalDateTime
   4. ZonedDateTime
   5. Duration :**PT**HMS
   6. Period : PYMD yamala pagala dewana
   7. Instant
3. method are :
4. plusYear/ minusYears
5. plusMonths/ minusMonths
6. plusWeeks /minusWeeks
7. plusDays / minusDays
8. plusHours /minusHours
9. plusMinutes /minusMinutes
10. plusSeconds/ minusSeconds
11. plusNanos / minusNanos

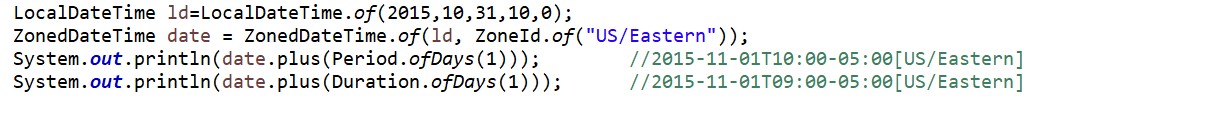
1. Local date methods :
   1. localdate.getDayOfMonth() -> give month value.
   2. localdate.**getDayOfWeek()** -> give weekday name.
2. All the classes in java.time are immutable and thread-safe.
3. This format follows the [ISO-8601](http://www.iso.org/iso/home/standards/iso8601.htm) standard for representing date and time.
4. UTC represents the time zone offset from zero
5. ChronoUnit is enum that implements the TemporalUnit interface and provides a set of standard units based on date and time, from milliseconds to millennia.
6. **between(start,end):** gives -ve if it is created with an end point that occurs before the start point.
   1. UnsupportedTemporalTypeException runtime exception throws if chronoFiled or ChronoUnit is not supported.
   2. ChronoUnit.HOURS.between: ChronoUnit.between is used to measure an amount in single unit of time.­­­
   3. Duration.between(start,end): it is in HMS not have any days or month
   4. Period.between(start,end):It gives value in YMD form have getMonths, getDays methods.
   5. Period take only LocalDate only as parameter while duration take Temporal as parameter.
7. The ChronoField enum, which implements the TemporalField interface, provides a rich set of constants for accessing date and time values. A few examples are CLOCK\_HOUR\_OF\_DAY, NANO\_OF\_DAY, and DAY\_OF\_YEAR. This enum can be used to express conceptual aspects of time, such as the third week of the year, the 11th hour of the day, or the first Monday of the month.
8. DayOfWeek enum has seven constant that describe days of week.
9. Month enum has 12 constants for each month.
   1. enum ChronoUnit implements TemporalUnit {}
10. class Duration implements TemporalAmount, Comparable<Duration>, Serializable {}
    1. java.time.temporal.TemporalUnit is interface
    2. **PT**nHnMnS : here second can comes in decimal like System.out.println(Duration.ofMillis(1100)) gives PT1.1S
11. class java.time.Period implements ChronoPeriod, Serializable {}
    1. format P1Y2M3Dv



b. System.out.println(p.negated()); // P-6D

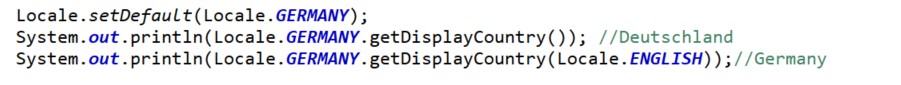
1. A Duration will add an exact number of seconds, thus a duration of one day is always exactly 24 hours. By contrast, a Period will add a conceptual day, trying to maintain the local time.

Ex in daylight saving:-

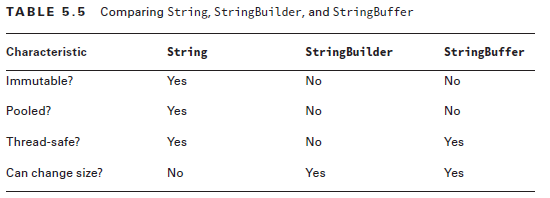


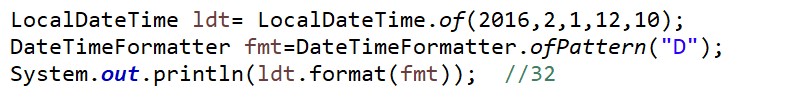
In other words period does not play with time or it does not have time component.

1. class java.time.Instant implements Temporal, TemporalAdjuster, Comparable< Instant>, Serializable {}
2. Suppose current time is 11:52:11:936 then Instant.now().**plusNanos(1) gives**
   1. 2019-06-05T11:52:11.936000001Z the last is milliseconds(1 ms=**10**9 ns).
3. java.util.Locale ->static Locale getDefault() {} have final class Builder {}that have Builder constructor -> Locale build() {}
4. **Four** ways to create *Locale*:
   1. Creating locale from constructor. Locale l1= new Locale(“en”,”US”);
   2. Creating locale from **local builder.** Locale l2= new Locale.Builder().setLanguage(“en”).setRegion(“US”).build();
   3. From forLanguage Tag method. Locale l3=Locale.forLangugeTag(“en-US”);
   4. From constants : Locale l4= Locale.US.
   5. getDisplayCountry(): Returns a name for the locale's country that is appropriate for display to the user.



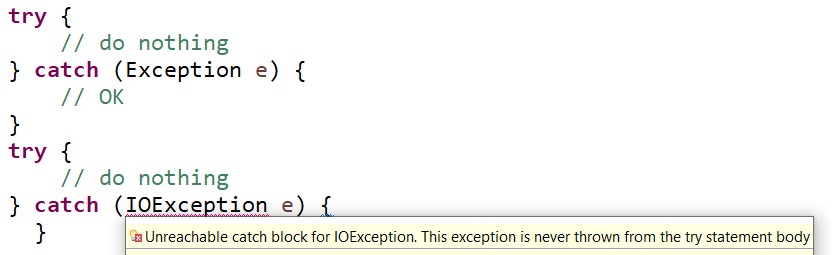
1. String is immutable



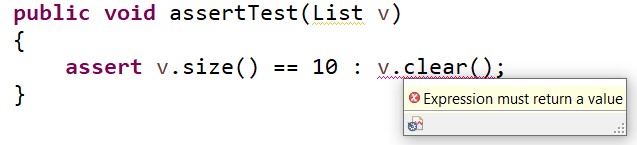
1. Time zone region is **not local** as for ex Russia has many time zone but same locale
2. Resource Bundles
   1. PropertyResourceBundle
      1. First .java resource file of nearest will run then property file are run including the .class files are as it’s what java looks for it in background
      2. Java docs for property loading ie. How it loads. <http://docs.oracle.com/javase/8/docs/api/java/util/ResourceBundle.html#getBundle-java.lang.String-java.util.Locale-java.lang.ClassLoader->
   2. class Properties extends Hashtable<Object,Object> {}
      * + 1. ->String getProperty(String key){}
          2. ->String getProperty(String key, String defaultValue) {}
   3. java.util.ResourceBundle ->have static final ResourceBundle getBundle(String baseName,Locale locale);
   4. ListResourceBundle
      1. abstract class java.util.ListResourceBundle extends ResourceBundle {} -->protected Object[][] getContents() {}
      2. If a key is not found then it prints null
3. Formatting:
   1. abstract class java.text.NumberFormat extends Format{}
      1. NumberFormat en = NumberFormat.getInstance(Locale.US);
   2. final class java.time.**format**.DateTimeFormatter{} -> DateTimeFormatter ofPattern("pattern")
      1. enum java.time.format.FormatStyle{}
      2. some pattern recognition:
         1. M ->1
         2. MM ->01
         3. MMM ->Jan
         4. MMMM >January
         5. d ->means day of month
         6. dd ->means day of month including zero for single digit day
         7. **D** ->for **day of year**
         8. ****
         9. , ->for out a comma
         10. u or y-> four digit year
         11. uu or yy-> two digit year
         12. uuuu or yyyy-> four digit year
         13. hh:mm-> hours(hh in 12 hours and HH for 24 hours) and minutes and single for single digit hours and minutes
   3. Generally, date follows yyyy-MM-dd HH:mm:ss
4. Conversion of local date to ZonedDateTime just add zone to it. If the time is same but they got difference by add zone to it.
5. Remember in daylight saving after 1:59AM 1:00AM comes then 2.00AM.So technically difference between 1am and 2 am is -2 hours.

# Exception

1. Unchecked exception or runtime exception
   1. java.lang.ArithmeticException extends RuntimeException {}
      1. ex: divide() it is use
   2. java.lang.ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException {} extends RuntimeException {}
      1. rangeCheck() it is ues to check it
      2. sort()
   3. java.lang.NegativeArraySizeException extends RuntimeException {}
   4. java.lang.ArrayStoreException extends RuntimeException {}
   5. java.lang.ClassCastException extends RuntimeException {}
   6. java.lang.IllegalStateException extends RuntimeException {}
   7. java.lang.NullPointerException extends RuntimeException {}
   8. java.lang.NumberFormatException extends java.lang.IllegalArgumentException extends RuntimeException {}
   9. java.lang.RuntimeException extend Exception{}
   10. java.lang.UnsupportedOperationException extends RuntimeException {}
   11. DateTimeParseException extends DateTimeException{}
   12. java.time.DateTimeException extends RuntimeException {}
   13. java.util.MissingResourceException extends RuntimeException {}
   14. **NoSuchElementException** **extends** RuntimeException {}
   15. ConcurrentModificationException **extends** RuntimeException {}
   16. UnsupportedTemporalTypeException extends DateTimeException
       1. This exception is used to indicate problems with creating, querying and manipulating date-time objects.
   17. DateTimeExceptionextends RuntimeException {}
2. Checked exceptions
   1. java.io.IOException extend Exception
   2. java.text.ParseException extends Exception {}
   3. java.io.**FileNotFoundException** extend IOException thrown by FileInputStream, FiltOutputStream and RandomAccessFile
   4. **NoSuchFileException** extends FileSystemException {} extends IOException{}
   5. java.io.NotSerializableException extends ObjectStreamException {} extends IOException {}
   6. java.sql.SQLException extends Exception implements Iterable<Throwable>{}
   7. java.nio.file.FileAlreadyExistsException **extends** FileSystemException {}
   8. NoSuchFieldException extends ReflectiveOperationException {}
3. Exception class has five constructors :
   1. Exception (){super();}
   2. Exception (String msg){super(msg);}
   3. Exception(String message, Throwable cause)
   4. Exception(Throwable cause)
   5. **protected** Exception(String message, Throwable cause, **boolean** enableSuppression, **boolean** writableStackTrace) {}
   6. We can also create Exception by using default i.e. no construction coding is required.
4. NoSuchElementException is runtime exception while NoSuchFileException is checked exception.
5. The compiler can't easily predict what kind of code can throw RuntimeException.IOException, however, is a checked exception - only method calls which are declared to throw it can do so. The compiler can be (reasonably) confident that it can't possible occur unless there are method calls which are declared to throw it.



1. Closeable restricts the type of exception thrown to IOException. Closeable requires implementations to be idempotent. If close() method declares a checked exception so try with resources must handle it. **Interface java.io.Closeable extends java.lang.AutoCloseable{}**
2. But before java 7 it was **Interface java.io.Closeable{}**
3. **Assertion** was used from java 1.4, since we normally use println for debugging that sometimes we forget to delete in production, which create performance problem and disturb server logs.
   1. For compile old version of java javac -source 1.3 Test.java
   2. Assert of two types:
      1. Simple type: assert (Boolean\_expression).
      2. Augmented Type: assert (Boolean\_expression): error\_message;
         1. error\_message is Expression2 is an expression that has a value mostly a String value.
         2. If error\_message is method ie m(); then it return must be not void ie it return something so useless it gives compile time error. :void type not allowed here.



1. Assertions are often used to check method post conditions, test control flow invariants, and validate class invariants, an assertion should never modify any data because it may be disabled at runtime, leading to unintended side effects.
2. Assertion used for validating post conditions at the end of any method. This means, after executing the business logic, you can use assertions to ensure that the **internal state of your variables or results is consistent** with what you expect. For example, a method that opens a socket or a file can use an assertion at the end to ensure that the socket or the file is indeed opened.
3. Command line flags for assertions:­-
   1. For nonsystem class: -enableassertions and –ea /-da that enable/disable assertion in all no system classes.
   2. For system class -esa/-dsa : to disable/enable assertion in system class.
4. We can use any number of flags that compile from left to right like java -ea -da -dsa test.java.
   1. For enabling assertion in two class b and d
      1. java -ea**:**pack1.B -ea:pack1.pack2.D;
   2. You can enable or disable assertions in the unnamed root (default)package (the one in the current directory) using the following commands:
      1. java -ea:... myPackage.myProgram
   3. Assertions can be enabled or disabled for specific packages or classes. To specify a class, use the class name. To specify a package, use the package name followed by **"..."** (three dots):
      1. java -ea**:**<class> myPackage.MyProgram
      2. java -da**:**<package>... myPackage.MyProgram

3. bad packages to enable assertions. **Java -ea:bad… Main**

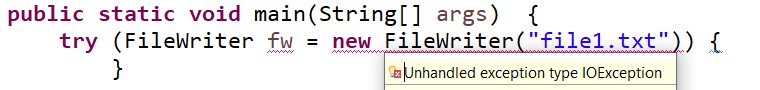
* 1. To enable assertion everywhere class of pack1 and disable in b class
     1. java -da:pack1.B -ea:pack1...
  2. To enable on main class and disable in good and bad packages :-java -ea:... -da:good... -da:bad... Main
  3. : importance
     1. java -ea**:**TestClass true // Enable assertion in TestClass and run also true class
     2. java -ea TestClass true // Enable assertion in TestClass and true as argument

1. The three possible outcomes of an assert statement are as follows:
   1. If assertions are disabled, Java skips the assertion and goes on in the code.
   2. If assertions are enabled and the boolean expression is true , then our assertion has been validated and nothing happens. The program continues to execute in its normal manner.
   3. If assertions are enabled and the boolean expression is false , then our assertion is invalid and a java.lang.AssertionError is thrown.
2. Exception thrown in the case of assertion failure is java.lang.AssertionError, which is derived from the Error class.
3. Assertion is used for validating input parameters of a **private method** not for public method(Should throw regular exceptions when passed bad parameter).
   1. The intent of assertions is to check your program logic -- an assertion failure is a "Stop everything -- there's a bug!" indication
4. **Appropriate and inappropriate use of assertion**
   1. It is always inappropriate to use business logic with assert statement because further execution of assert statement always at runtime.
   2. While performing debugging in our program if there is any place where the control is not allowed to reach that is the place to use the assertion, like default case in switch.
   3. It is always appropriate to validate private method argument and always inappropriate to validate public method argument in assertion as outer person does not know we are using assertion for validation. ex
      1. public static void main(String str){
      2. assert(str[0]=="00");} // so it is not appropriate as it is argument to main method which is public.
   4. We can catch assertion error with catch block, but it is stupid to do it.
   5. In web application in debug mode assert statement is executed automatically.
5. Throws keyword is in the function signature and used to declare as similar to try-catch block
6. Throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.
   1. Remember that throw means an exception is actually being thrown and throws indicate that the method merely has the potential to throw that exception.
   2. We can also understand throws means multiple can be while throw only one can be throw example uses both:

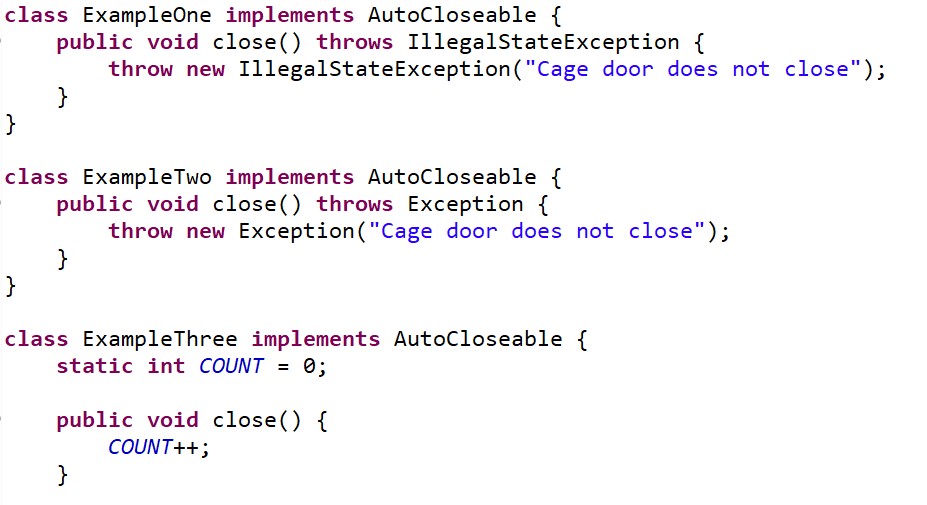
public String getDataFromDatabase() throws SQLException , IOException {

throw new UnsupportedOperationException();}

1. Multicatch is effectively final
   1. class Exception extends Throwable
   2. class Error extends Throwable
   3. class Throwable implements Serializable
2. Java also recommends making the close() method idempotent. **Idempotent** means that the method can called be multiple times without any side effects or undesirable behavior on subsequent runs. For example, it shouldn’t throw an exception the second time or change state or the like.
3. If catch block throw checked exception then it must be handled the Exception.
   1. Give error: Unhandled exception type Exception (add throws /try catch).
4. In try-with-resources catch block is optional if close() is not able to throw a checked exception.
   1. When a try-with-resources statement is used with a close() method that throws a checked exception, it must be handled by the method or caught within the try-with-resources statement.



* 1. ExampleOne is the best implementation. ExampleTwo throws Exception rather than a more specific subclass, which is not recommended. ExampleThree has a side effect. It changes the state of a variable. Side effects are not recommended.

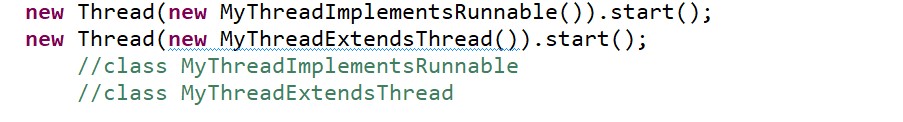


1. Throwable is the topmost class in exception hierarchy .
   1. Error is used for a serious system errors from which there can be no recovery.
   2. Exception is used for creating new exception.
2. Assertion are usually enabled in development and testing environment and disable in production environment. It is not good practice to write code that recovers from the assertion failure as assertion are for this purpose only.
3. **NullPointerException** is thrown when program attempts to use an object reference that has the null value.These can be:-
   1. Invoking a method from a null object.
   2. Accessing or modifying a null object’s field.
   3. Taking the length of null, as if it were an array.
   4. Accessing or modifying the slots of null object, as if it were an array.
   5. Throwing null, as if it were a Throwable value.
   6. When you try to synchronize over a null object.

# Concurrency (java.util.concurrent package)

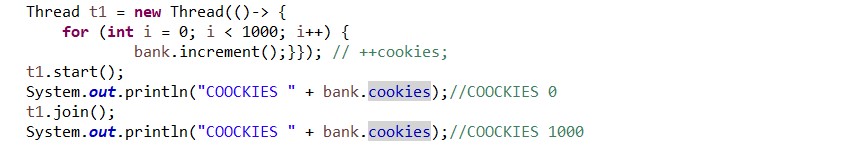
1. **JAVA THREAD POOL:** A **thread pool is a collection of pre-initialized threads**. Generally the size of collection is fixed, but it is not mandatory. It facilitates the execution of N number of tasks using same threads. If thread are more tasks than threads, then tasks need to wait in a queue like structure ([FIFO – First in first out](https://en.wikipedia.org/wiki/FIFO_and_LIFO_accounting#FIFO)).
2. A thread dies when run() method ends and does die when start() method ends as it starts the run().
3. Thread.sleep(int time) provides the main thread to sleep for in milli seconds and it throws the InterruptedException.
4. Implement runnable v/s extend thread

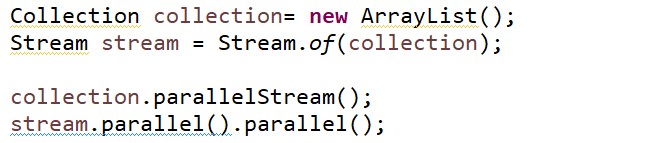
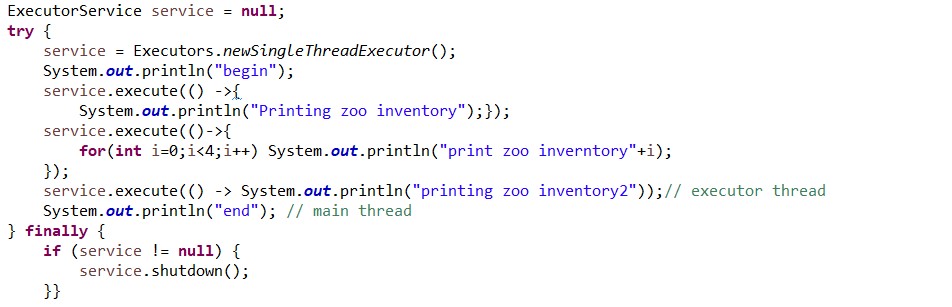
|  |  |  |
| --- | --- | --- |
| **Using class to create Thread** | **implement Runnable interface** | **extends Thread class** |
| Overriding method | public void run() | public void run() |
| Create thread object | new Thread(new myRunnable()) | new myThread() |
| Do tasks on seperate thread | (new Thread(new myRunnable())).start() | (new myThread()).start() |
| Do tasks on main thread ie new thread does not created or it work as plain run() method | (new Thread(new myRunnable())).run() | (new myThread()).run() |

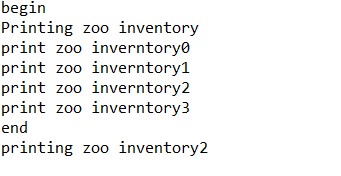


1. thread.join(Int i): **Waits** at most millis milliseconds for this thread to die. Here in this example main program thread waits 0 seconds for t1 thread to die. So here t1 thread finished then main program is run.

We can also wait for main thread by Thread.sleep(**long** millis) but it wait for that time while in join if t1 tread finishes then it automatically run the main thread.



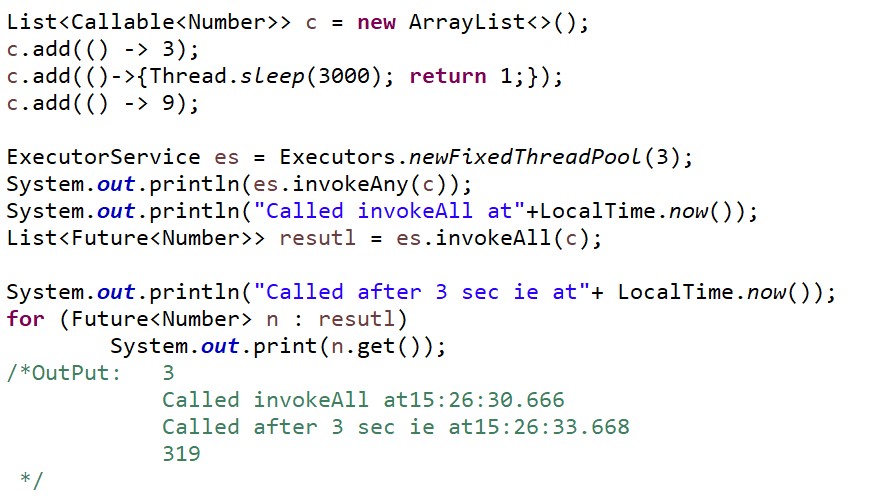
1. for converting parallel to sequential stream use stream.sequential
2. If an interface declares an abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface's **abstract method count** since any implementation of the interface will have an implementation from java.lang.Object or elsewhere.
3. java.lang.Comparable (compareTo(T t)){{ remember compareTo means we are comparing with someone}} for natural ordering and java.util.Comparator (compare(W w, X x)){{ it compare means we are comparing two things}} for my own sorting.
4. int java.lang.String.compareTo(String anotherString)
   1. Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings.
   2. Don’t confuse it with Comparable **compareTo**()
5. In computer programming, an automatic variable is **a lexically-scoped** variable which is allocated and de-allocated automatically when program flow enters and leaves the variable's scope
6. parallelStream() is used for collection to parallel and parallel() is used for stream to parallel. Calling parallel() to parallel stream is allowed.
7. Executor  has two sub interface ExecutorService, and the ThreadPoolExecutor
8. execute() can only implement runnable while submit() have both runnable and callable
   1. In submit we want result while execute we don’t want it.
9. static ExecutorService newSingleThreadExecutor(){} , below is diagram for creating threads using executor service

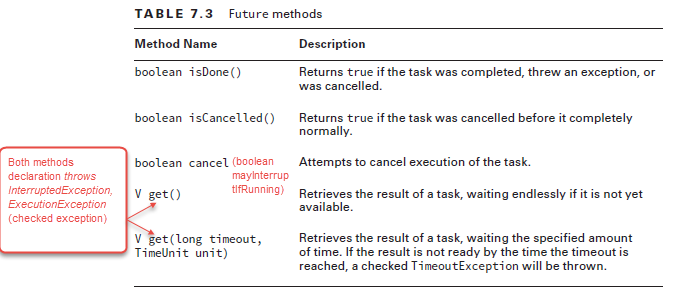
will give output like

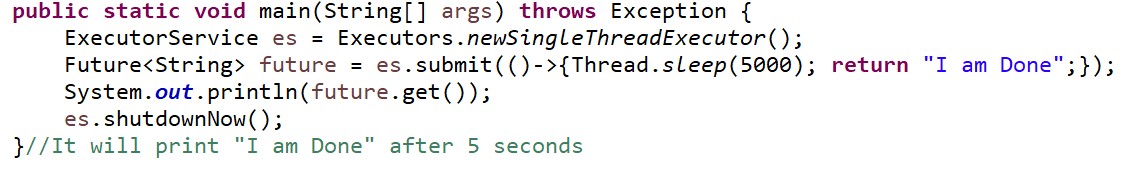
1. **invokeAll**():Executes the given tasks, synchronously returning the results of all tasks as a Collection of Future objects, in the same order they were in the original collection.

Return a list of Futures representing the tasks, in the same sequential order as produced by the iterator for the given task list, each of which has completed

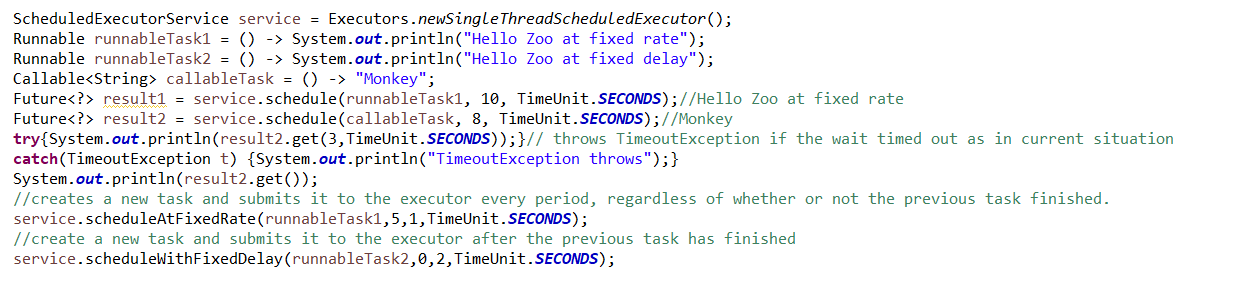
**invokeAny**():Executes the given tasks, synchronously returning the result of one of finished tasks, **cancelling any unfinished tasks**.



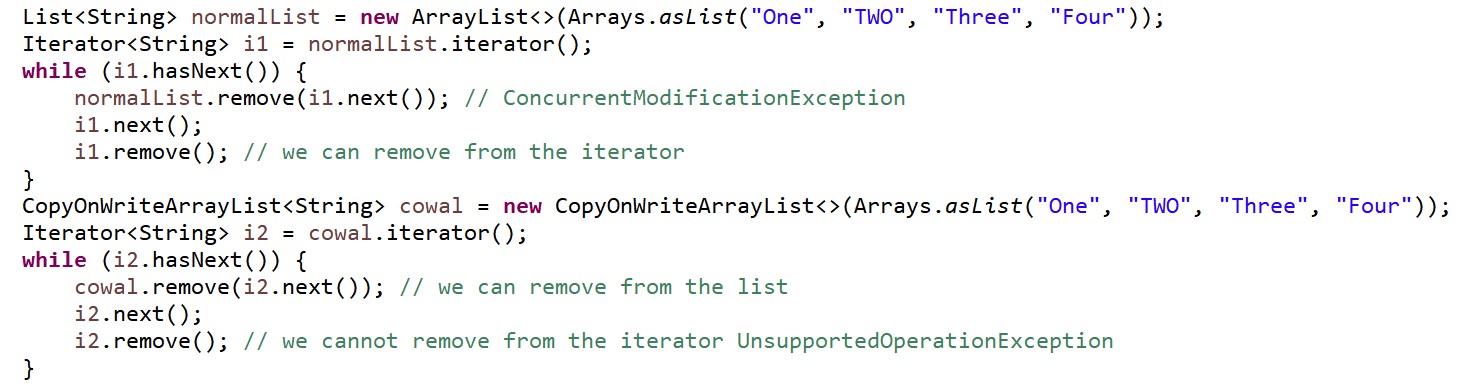
1. java.util.concurrent.Callable<V> can throw checked exception and return a java.util.concurrent.Future<V> while java.lang.Runnable does not return anything and didn't have neccessary to import as it is in lang package.
2. Future(Interface) throw Exception that must be handle or declare and can be used with Runnable lambda interface.(get method wait **endlessly** if tasked not complete).
   1. V get() **throws** InterruptedException, ExecutionException;
   2. V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException;



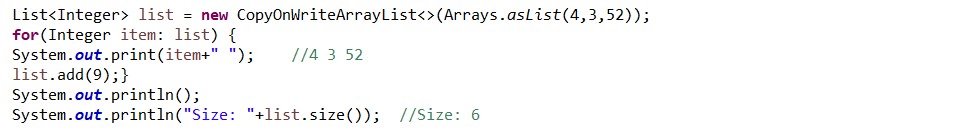
1. Scheduling Tasks:
   1. scheduleAtFixedRate and scheduleWithFixedDelay only take the runnable task
   2. Here scheduleAtFixedRate is delayed by 5 sec and give delay 1 sec to execute it task



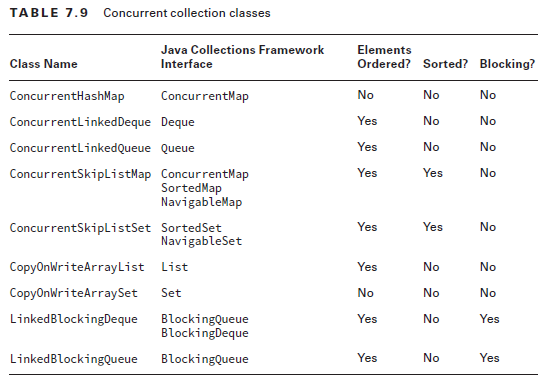
1. java.util.concurrent.TimeUnit; A TimeUnit represents time durations at a given unit of granularity and provides utility methods to convert across units, and to perform timing and delay operations in these units.
2. Executors method:
   1. newSingleThreadExecutor()
   2. newSingleThreadScheduledExecutor()
   3. newCachedThreadPool()
   4. newFixedThreadPool(int nThreads)
   5. newScheduledThreadPool(int nThreads)
3. **Working with concurrent collections:**
   1. CopyOnWriteArrayList is a thread-safe variant of ArrayList where operations which can change the ArrayList (add, update, set methods) creates a clone of the underlying array.
   2. In List when we call remove then it can throws ConcurrentModificationException, however in CopyOnWriteArrayList modification methods(remove, set, and add) are not supported. Throw: UnsupportedOperationException because it still had access to old container so give wrong impression.

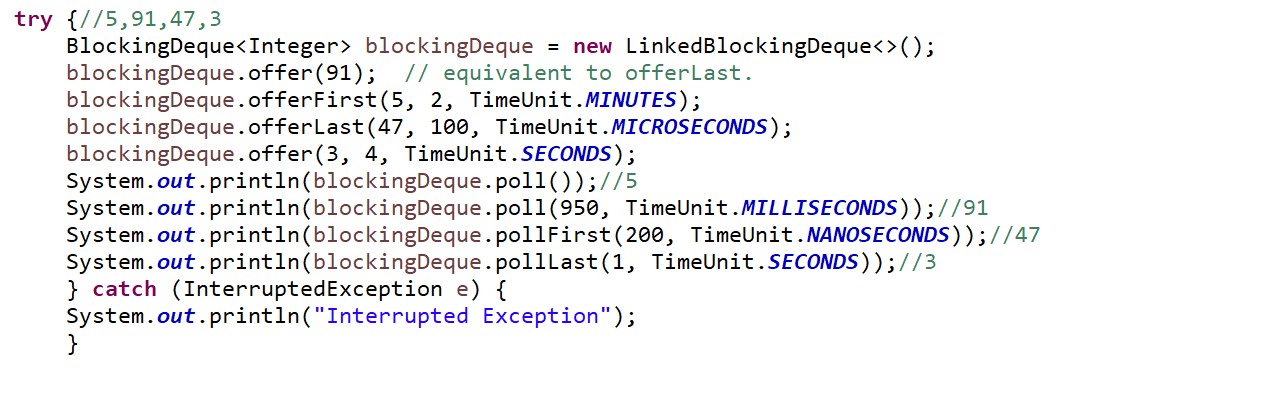


* 1. **null** can be added to the list.



1. *Synchronizing data access*

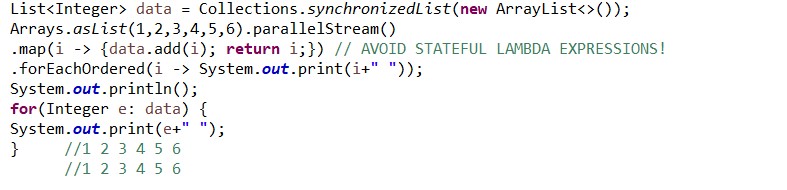
**

1. The BlockingQueue is just like a regular Queue, except that it includes methods that will **wait a specific amount of time to complete an operation**. It also throws checked InterruptedException.
2. SkipList classes, ConcurrentSkipListSet and ConcurrentSkipListMap are concurrent versions of TreeSet and TreeMap respectively.
3. Concurrent Atomic Package:

The classes such as AtomicInteger do not use a lock; rather, they internally use volatile variables and a low-level mechanism known as Compare-And-Set (CAS). For this reason, using AtomicInteger and related classes is faster than using locks using synchronized keyword.

* 1. Method used in AtomicInteger:
     1. addAndGet(int delta) : add the delta value to current value and return that value.
     2. getAndSet(int newValue)
     3. boolean compareAndSet(int expect, int update)
  2. There is no AtomicDouble class in atomic package.

1. Elements **Ordered** means that collection maintains the order of the elements on the sequence you put stuff into / remove from the collection. While **sorted** collection keep the elements in sorted criteria.
2. **Working with parallel stream**:
   1. Any stream operation that is based on order, including findFirst(), limit(), or skip() may actually perform more slowly in a parallel environment. Because parallel processing task forced to coordinate in synchronized like fashion. On the plus side the result is consistent with a serial stream ex calling **skip**(5),**limit**(2),**findFirst()** will return the same result in serial and parallel stream.
   2. **flatMap**() always produce the serial Stream by default pg. 367

****

* 1. combining result with collectors:



1. Fork/Join Framework
   1. Constructor of ForkJoinPool
      1. **public** ForkJoinPool() {}
      2. **public** ForkJoinPool(**int** parallelism) {}
      3. **public** ForkJoinPool(int parallelism, ForkJoinWorkerThreadFactory factory, UncaughtExceptionHandler handler, boolean asyncMode) {}
   2. java.util.concurrent.RecursiveAction extends ForkJoinTask<Void>{}
   3. java.util.concurrent.RecursiveTask<V> extends ForkJoinTask<V> {}
   4. void java.util.concurrent.ForkJoinTask.invokeAll(ForkJoinTask<?> t1, ForkJoinTask<?> t2)
   5. **RecursiveAction** an abstract class that requires to implement the **compute()** method which return void.
   6. **RecursiveTask** is an abstract class that requires to implement the **compute()** method which return the generic. Here Task(Work) so it must return a value.
   7. **Fork()**instructs fork/join framework to compute the task in separate thread while **join**() causes the current thread to wait for the results.
   8. A ForkJoinPool differs from other kinds of ExecutorService mainly by virtue of employing **work-stealing ( one thread created the task so other thread can used it)**.
   9. The worker threads in the ForkJoinPool extend java.lang.Thread and are created by a factory.
   10. **ForkJoinPool implements Executor and not the threads in the pool.**
2. Liveness is the ability of an application to be able to execute in a timely manner. There are three kind of the liveness **deadlock**, **livelock**, **starvation**
   1. **Deadlock :** Deadlock occurs when two or more threads are blocked forever, each waiting on the other.
   2. **Livelock :** Livelock occurs when two or more threads are conceptually blocked forever, although they are each still active and trying to complete their task. It is a **Special** case of starvation.
   3. **Starvation**: Starvation occurs when a single thread is perpetually denied access to a shared resource or lock. or **Starvation occurs when one thread cannot access the CPU because one or more other threads are monopolizing the CPU**
3. **A race condition** is an undesirable result that occurs when two tasks, which should be completed sequentially, are completed at the same time.
4. Intrinsic Locks and Synchronization <https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html>

Allowing a thread to acquire the same lock more than once enables **reentrant synchronization**.

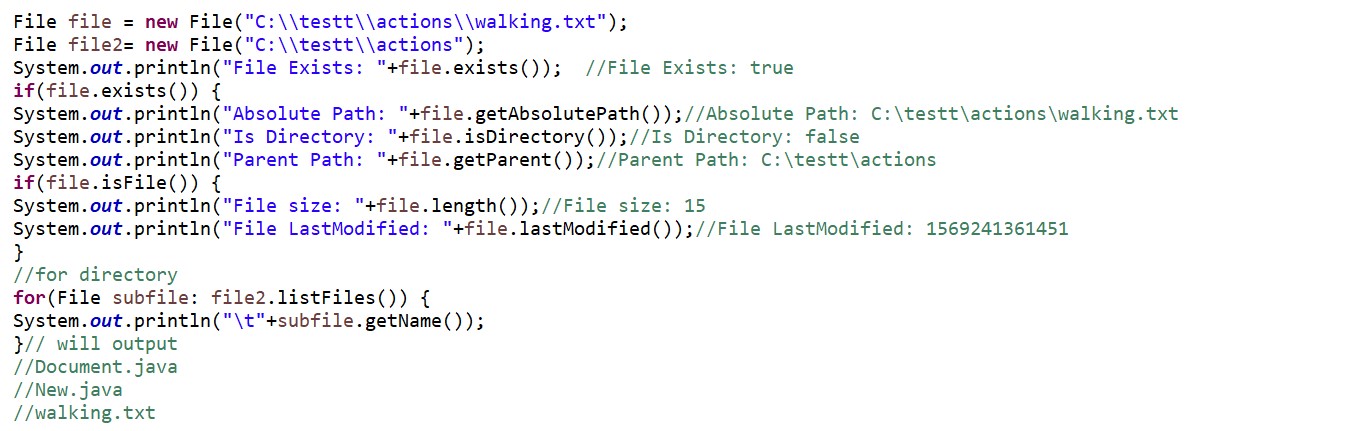
1. In some OS’s like windows CPU time is given to threads in ratio of their priority while in other OSs like Unix a lower priority thread executes only after higher priority thread ends ie different way for priority.
2. Difference between newCachedThreadPool and newFixedThreadPool?
3. ScheduledExecutorService ->ScheduledThreadPoolExecutor.
   1. The scheduleAtFixedRate and scheduleWithFixedDelay methods create and execute tasks that run periodically until cancelled.
4. **ReentrantReadWriteLock** Like ReentrantLock, the ReentrantReadWriteLock allows a thread to acquire the read lock or write lock multiple times recursively, thus the word “Reentrant”.

Using it also thread can temporarily release the locks it has when it is going to a wait state.

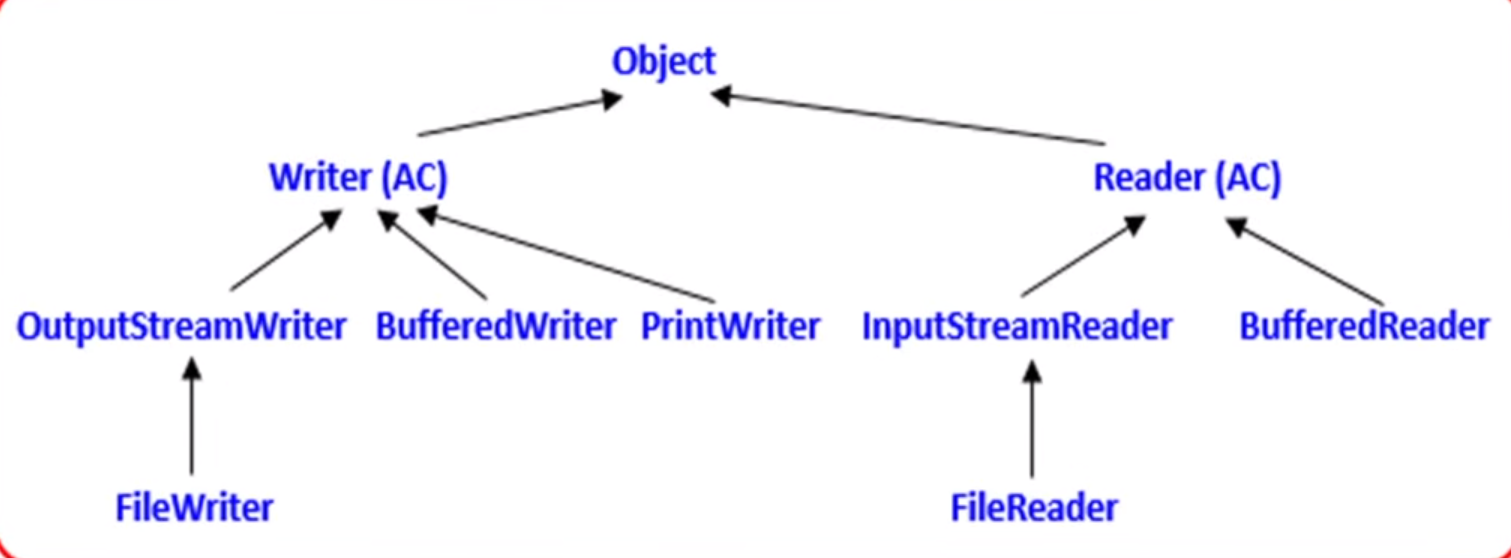
1. **Collectors Class:** The Collectors class includes two sets of methods for retrieving collectors that are both UNORDERED and CONCURRENT, Collectors.toConcurrentMap() and Collectors.groupingByConcurrent(), and therefore it can perform parallel reductions efficiently.
2. **CyclicBarrier :** Briefly, a CyclicBarrier allows multiple threads to run independently but wait at one point until all the coordinating threads arrive at that point. Once all the threads arrive at that point, all the threads can then proceed. It is like multiple cyclists taking different routes to reach a particular junction. They may arrive at different times, but they will wait there until everyone arrives. Once everyone is there, they can go on further independent of each other. <http://examples.javacodegeeks.com/core-java/util/concurrent/cyclicbarrier/java-util-concurrent-cyclicbarrier-example/>
3. **ThreadLocalRandom :**There is problem with Math.random()as it is synchronized method so it can be used by multiple thread but the cost of performace.So better to use ThreadLocalRandom.current().nextInt(1,11);
4. It is okay for a thread to acquire lock on obj again, and such an attempt will succeed

# IO (java.io package)

1. Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.
2. For path to file always include ‘//’ rather than ‘/’ because ‘/’ is used **escape character** and C:file/xyz so java treat /x as a different thing.
3. Data are of two types text data and binary data(video, audio etc.). For text data we use reader, writer and for binary data we use InputStream and OutputStream.
4. File list() method returns a list of String values of path, not File only and it is not recursive.
   1. .filter(Files::isRegularFile) it lists only files.



1. Java has three **built-in stream** System.in System.err and System.out
2. Stream class are used for input and output data in the form **of binary or byte** while reader or writer used in the form of character and string data. ex FileReader is a class that reads data from a file as character or strings and furthermore ObjectOutput**Stream** writes object data to byte stream
3. Closing a high stream cause closing a low stream. Check after try statement that they even are used causes the IOException.



**AC means Abstract Class**

1. parent of all **IO** classes
   1. abstract class **InputStream** implements Closeable{}
   2. abstract class **OutputStream** implements Closeable, Flushable{}
   3. abstract class **Reader** implements Readable, Closeable {}
      1. **public** **boolean** ready() **throws** IOException {}

Tells whether this stream is ready to be read.

* 1. abstract class **Writer** implements Appendable, Closeable, Flushable {}

1. Inputstream read():
   1. Reads the next byte of data from the input stream, or -1 for EOF.
   2. read(byte[] b) Reads some number of bytes from the input stream and stores them into the buffer array b.



1. **FileInputStream** extends InputStream {}, Low
2. **FileOutputStream** extends OutputStream {}, Low
   1. **Reads/Write file data as bytes i.e. they have stream on them**
3. BufferedInputStream extends FilterInputStream {}
4. BufferedOutputStream extends FilterOutputStream {}
   1. They are for only the stream buffer i.e. byte file reader
   2. BufferedInputStream adds functionality to another input stream-namely, the ability to buffer the input and to support the mark and reset method
   3. Be suspicious of any code samples that call the mark() or reset() method without first calling markSupported().
5. FileReader extends InputStreamReader{} , Low
   * 1. FileReader fr= new FileReader(String path);
     2. FileReader fr= new FileReader(File f);
   1. Method of FileReader
      1. int read(); // it returns the next Unicode value.
      2. int read(char[] ch);// it read the file data up to ch.length() and return the number of character copied from file into char[] // for whole file we use char[] ch= new char[(int)f.length]
      3. close();
6. FileWriter extends OutputStreamWriter {} , Low
   1. Writes file data as characters or to the file. It creates the file if file is not available.
      1. FileWriter fw= new FileWriter(String path);
      2. Filewriter fw= new FileWriter(File f);
      3. FIleWriter fw= new FileWriter(String path,Boolean append);
      4. FileWriter fw= new FileWriter(File f, Boolean append);
   2. Methods of FIleWriter:
      1. write(int ch) // to write a single character to the file writer(‘d’) and write(100) are the same
      2. write(char[] ch)// to writer an array of character to the file
      3. write(String str) // to write string of data to file
      4. flush() : for guarantee that total data is written on file
      5. close() :must close and it generally call flush() also in the case of writer only.
   3. Demerit of FileWriter is we must use new line(/n) which varies system to system as some system does not support /n as new line as seen in Durga classes.
7. BufferedReader extends Reader {} ,High
   1. The main advantage over reader is that we can read line by line rather than char.
      1. BufferedReader br new BufferedReader(Reader r);
      2. BufferedReader br=new BufferedReader(Reader r, int buffersize)
   2. Methods:
      1. int read();
      2. int read(char[] ch)
      3. void close();
      4. String readLine(); returns null if next Line not available
8. BufferedWriter extends Writer{} ,High
   1. Reads/Writes character data to an existing Reader/Writer(char way data) in a buffered manner, which improves efficiency and performance. It does not communicate directly with file. It takes writer as argument which maybe PrintWriter, BufferedWriter.
      1. BufferedWriter bw= new BufferedWriter(Writer w);
      2. BufferedWriter bw= new BUfferedWriter(Writer w, inf buffersize);
   2. BufferedWriter bw= new BufferedWriter(new BufferedWriter(new Writer(file\_path))); is valid
   3. It has newLine() extra method.
9. **ObjectInputStream** extends InputStream implements ObjectInput, ObjectStreamConstants{}, High
   1. Deserializes primitive Java data types and graphs of Java objects from an existing InputStream
10. **ObjectOutputStream** extends OutputStream implements ObjectOutput, ObjectStreamConstants {}, High
    1. Serializes primitive Java data types and graphs of Java objects to an existing OutputStream
11. **InputStreamReader** extends Reader {}, High
12. **OutputStreamWriter** extends Writer {}, High
    1. Reads/write character data from an existing InputStream/OutputStream or byte to char converter
13. **PrintWriter** extends Writer {} High
    1. The main advantage is that we can write primitive data type directly to file.
    2. It can communicate either directly to file or via some with Writer object.
       1. PrintWriter pw= new PrintWriter(String file)
       2. PrintWriter pw= new PrintWriter(File f)
       3. PrintWriter pw= new PrintWriter(Writer w)
    3. PrintWriter methods.
       1. **write**(int ch); print(char ch); println(char ch);
       2. **write**(char[] ch); print(int i); println(int i);
       3. **write**(String s); print(double d); println(double d);
       4. flush(); print(boolean b); println(boolean b);
       5. close(); print(String s); println(String s);
    4. pw.write(100) // print d
    5. pw.print(100) // print 100 it is the main difference between it with other writer.
14. **PrintStream** extends FilterOutputStream implements Appendable, Closeable, High
    1. PrintStream never throws an IOException instead it set internal flag.
    2. Writes formatted representations of Java objects to a binary stream
    3. Append()
    4. print()/println() // it print the char , double ,int with various method overloading method
    5. format()/formatf()
    6. System.**out out is example of print stream.**
15. **Difference between PrintWriter , PrintStream, FileWriter, BufferedWriter, OutputStreamWriter?**
    1. **PrintWriter does not support encoding while PrintStream support encoding**
16. flush() is only for writer and outputstream so for reader and input stream it causes compile time error.
17. skip() method is defined on InputStream and Reader, not OutputStream and Writer.
18. **Stream (not Stream Class) common operation :-**
    1. close(): can be used in finally block or with try-with-resource
       1. **public** **void** close() **throws** IOException {}
    2. flush(): for OutPutStream, write data from memory to files immediately
       1. public void flush() throws IOException {}
    3. For InputStream and Reader, use **markSupported**() to check → **mark**(int i) marking a particular point in a stream and **reset**() resets the stream to the most recent mark. These methods provide a book-marking feature that allows you to read ahead in the stream to inspect the upcoming data.
       1. **public** **synchronized** **void** mark(**int** readlimit) {}
       2. **public** **synchronized** **void** reset() **throws** IOException {}
       3. **public** **boolean** markSupported() {}
    4. skip() : for InputStream and Reader, skip certain number of bytes
       1. **public** **long** skip(**long** n) **throws** IOException {}
    5. read()
       1. **public** **abstract** **int** read() **throws** IOException;
       2. **public** **int** read(**byte** b[]) **throws** IOException {}
          1. read num of bytes did it read
       3. **public** **int** read(**byte** b[], **int** off, **int** len) **throws** IOException {}
19. **Serialization**:
    1. The process of converting an **in-memory** object **to** a stored data format(File or Network) is referred to as **serialization**. It can be achieved by ObjectOutputStream(convert object to bytes) and FileOutputStream(save byte data to file) while the reciprocal process of converting stored data into an object, which is known as deserialization.
    2. **Serialization** of an object stores only the instance variable data, not the static class data or **serializing an object throws away the static class data**.
    3. In order we serialized the object the same order we deserialized the objects. E.g. If we serialized the objects of dog , cat and rat so when we deserialized it in the same order.
    4. A class must implement Serializable to use serialization if not it throws : java.io.NotSerializableException.
    5. Serializable is a marked interface.
    6. To use **customized serialization,** we must use two function in same class.
       1. private void writeObject(**ObjectOutputStream** os) throws Exception{ os.defaultWriteObject();}
       2. private void readObject(**ObjectInputStream** is) throws Exception{ is.defaultReadObject();}
    7. If parent class is serializable than child class is **automatically** serializable.
    8. serialVersionUID is created by jvm by it .class file.
    9. If an object is being deserialized, that means the class of that object implements Serializable. Therefore, **its** constructor will never be called. However, constructor for the super class may be invoked **if** the super class does not implement serializable interface.
    10. During deserialization, the constructor of the class (or any static or instance blocks) is not executed. However, if the super class does not implement Serializable, its constructor is called.
    11. During deserialization if no args constructor was not found in super class that have not implemented Serializable then it throw runtime exception InvalidClassException.
20. **The transient and Volatile variable Example in Java**
    1. static **volatile** int sharedVar = 6;
    2. **Volatile keyword here makes sure that the changes made in one thread are immediately reflect in other thread. Transient** is a variables modifier used in [serialization](http://quiz.geeksforgeeks.org/serialization-in-java/). At the time of serialization, if we don’t want to save value of a particular variable in a file, then we use **transient** keyword. When JVM comes across **transient**keyword, it ignores original value of the variable and save default value of that variable data type.
    3. Since static variable are not part of objects so it does not participate in serialization. So ***transient static int =5;*** is useless as it does not take part in serialization. Same for **final** variable also as it is in form of value not variable at runtime.
21. Console is meant to interact with the user typically through command/shell window and keyboard. Thus, binary data doesn't make sense for the console.
    1. Whether a virtual machine has a console is dependent upon the underlying platform and also upon the manner in which the virtual machine is invoked. If the virtual machine is started from an interactive command line without **redirecting(Redirecting means to redirect ip/op to other sources like file)** the standard input and output streams then its console will exist and will typically be connected to the keyboard and display from which the virtual machine was launched. If the virtual machine is started automatically, for example by a background job scheduler, then it will typically not have a console.
    2. If System.console() is not available, it returns null.
    3. **Java.io.Console** class methods:-

Console use singleton pattern, so it had private constructor so called by System.console()

* 1. public PrintWriter **writer**() - retrieve the unique PrintWriter object associate with console e.g. console.writer().println();
  2. public Reader **reader**() - retrieve the unique Reader object associate with console
  3. public Console **format**(String fmt, Object... args) - write a formatted string to the console
  4. public Console **printf**(String format, Object... args) - write a formatted string to console using specified format string
  5. The printf() was added as a convenience method, since many other languages use printf() to accomplish the same task as format().
  6. public String **readLine**(String fmt,Object... args) - provide a formatted prompt then read a single line
  7. public String **readLine**() - read a single line from the console
  8. public char[] **readPassword**(String fmt,Object... args) - provide a prompt then read a password
  9. public char[] **readPassword**() read a password
  10. public void **flush**() flushes the console and forces any buffered output to be written immediately .
  11. Remember there is no read(), print(),println() method.

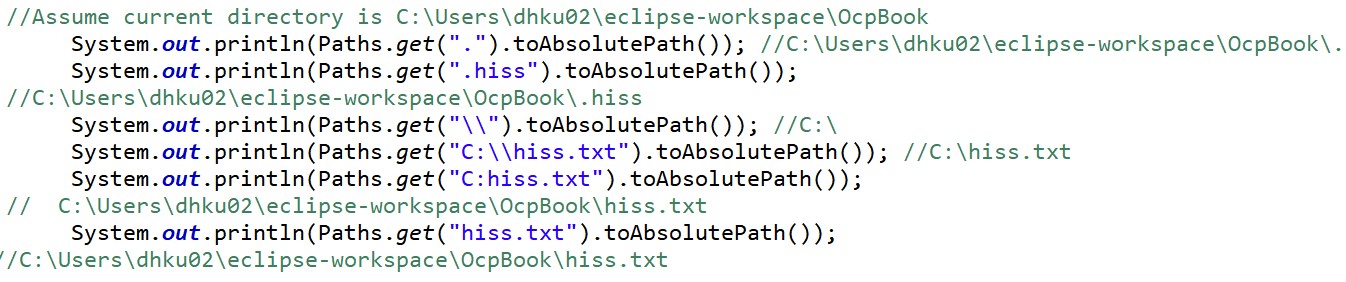
1. Without a try-with-resources statement or an equivalent finally block, any exception thrown by the write() method would cause the **resource** not to be closed in the method, possibly leading to a resource leak.
2. Valid way to iterate over all elements of a file using ObjectInputStream is to continue to callreadObject() until an EOFException is thrown.
3. Here console(Console console = System.console(); ) is Console objects while out in System.out.println() is System in built printstream objects.
4. File Basics :
   1. File f= new File(“abc.txt”); // we are creating a file reference not any physical file.
   2. File f= new File(String subdir, String name);
   3. File f= new File(File subdir, String name);
5. File Methods
   1. f.createNewFile()// will create a new file
      1. new File("/c/myfile.txt").createNewFile(); It will create file myfile.txt in c folder
   2. f.mkdir()//will **create** the directory
      1. new File("/c/myfile.txt").mkdirs(); it will create myfile.txt directory in c folder
      2. Remember in a directory a file and folder of same name cannot exists.
   3. String [] s=f.list() // return the name of all files and directory
   4. Long l= f.length(); // find the length of the file
6. Difference between java.io.FileNotFoundException, java.nio.file.NoSuchFileEception and Java.nio.file.InvalidPathException:
   1. NoSuchFileException :This is thrown when program tries to create BufferReader to read file specified by the Path object
   2. InvalidPathException: it check the argument is valid or not like c:c:test etc.
   3. FileNotFoundException: The exception is thrown by FileInputStream, FiltOutputStream and RandomAccessFile constructors when file is not existing or not **accessible**.



# NIO.2 (java.nio version 2 package)

CAUTION : root of windows is drive letter while in mac and Linux is /

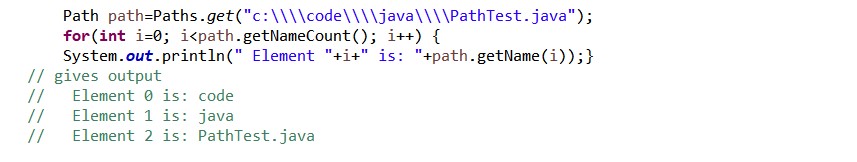
. represent current working directory while / represent current root directory.



1. There is java.nio.file.Path is interface and represent file/directory path and define some useful methods while java.nio.file.Paths is utility class.
2. java.**nio**.file.Files is introduced in java 7 as a utility class while java.**io**.File is an old class (java 4) that represent files/directories path.
3. Their definiations
   1. interface **java.nio.file.Path** extends Comparable<Path>, Iterable<Path>, Watchable {}
   2. final class **java.nio.file.Paths** { }
   3. final class **java.nio.file.Files**{}
   4. public class **java.io.File** implements Serializable, Comparable<File>{}
   5. The java.net.URI class is used to create and manage URI values.
      1. Paths.get(new URI("file:///home/zoodirectory"));
      2. Uniform Resource Identifier it is superset of **URL**.
4. Optional arguments in nio2 :-
   1. **NOFOLLOW\_LINKS** use for Test file existing, read file data, copy file, move file
   2. **FOLLOW\_LINKS** use for traversing a directory
   3. **COPY\_ATTRIBUTES** use for copy file
   4. **REPLACE\_EXISTING** use for copy and move file
   5. **ATOMIC\_MOVE** use for moving file
5. What happens to a Path supplied to a compatible NIO.2 method that includes the NOFOLLOW\_LINKS option?

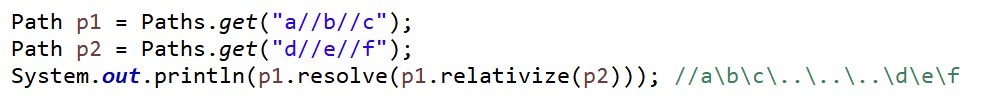
If the Path is a symbolic link, the operation will be performed on the link itself, rather than the target of the symbolic link.

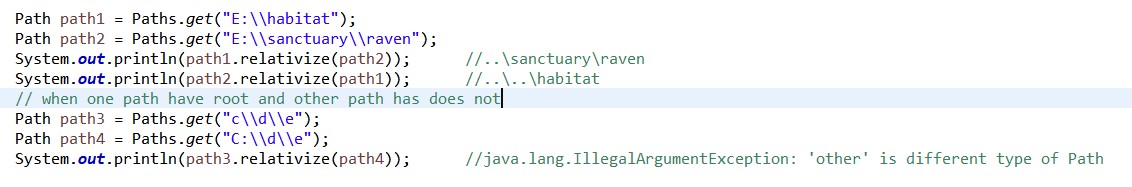
1. Path path = file.toPath(); and File file= path.toFIle();
2. Here path is just location. Some operation does not require the path to exist ex: retrieving the parent or root directory while other require path to exist like Path.toRealPath().
3. getName() reference to each element of the path



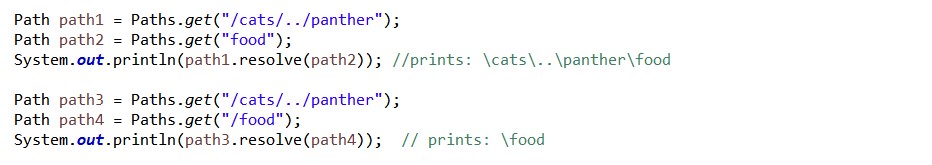
* 1. p1.getRoot() is **c:\** ((For Unix based environments, the root is usually / ).
  2. IllegalArgumentException - if index is negative, index is greater than or equal to the number of elements, or this path has zero name element

1. Relativization is the inverse of resolution.

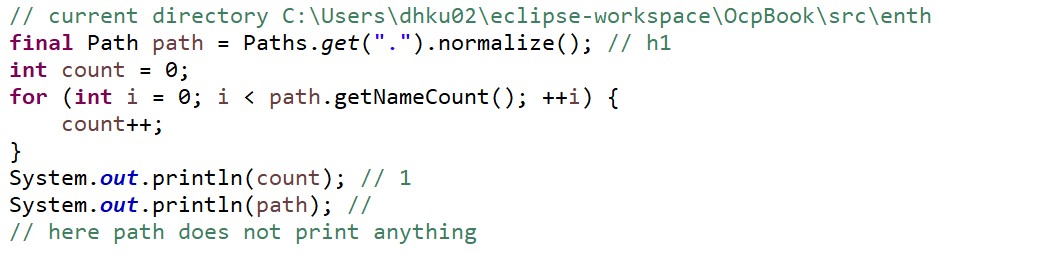




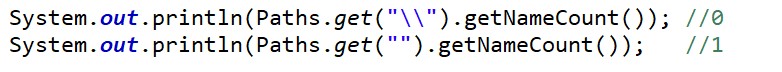
1. Path resolveSibling(Path other) : Resolves the given path against this path's parent path. This is useful where a file name needs to be replaced with another file name. For example, suppose that the name separator is "/" and a path represents "dir1/dir2/foo", then invoking this method with the Path "bar" will result in the Path "dir1/dir2/bar". If this path does not have a parent path, or other is absolute, then this method returns other. If other is an empty path then this method returns this path's parent, or where this path doesn't have a parent, the empty path.
2. resolve() it is used to join path objects. e.g.



1. normalize(Path) method to eliminate the redundancies in the path. ex:-
   * 1. E:\data\..\user\home to E:\user\home
   1. The normalize() method does not convert a relative path into an absolute path



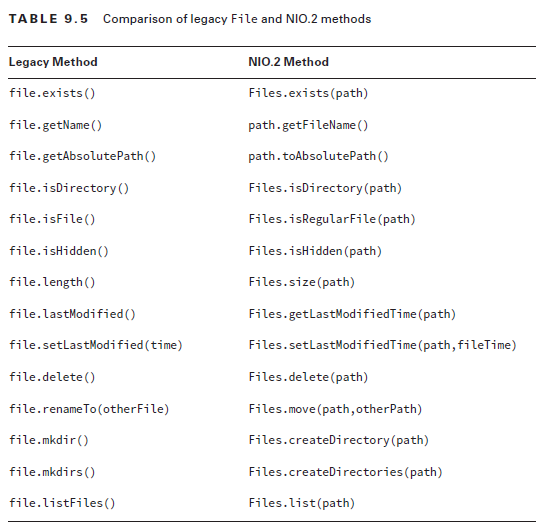
* 1. Root getNameCount is zero.



1. toRealPath() it checks file existence and is the **only** path method to support **NOFOLLOW\_LINKS**
   1. ex: /..zebra/food.source to
   2. /horse/food.txt

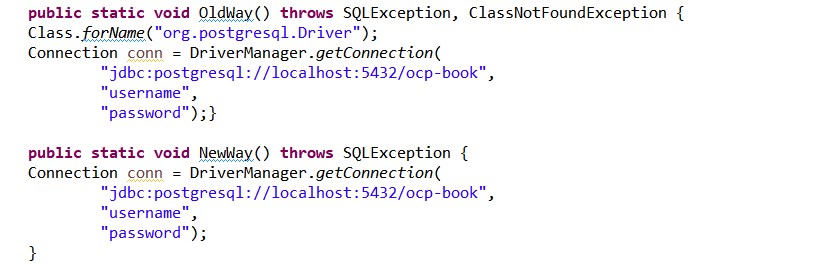
***toRealPath() vs toAbsolutePath()***:toRealPath is similar to the toAbsolutePath() method in that it can convert a relative path to an absolute path, except that it also verifies that the file referenced by the path actually exists, and thus it throws a checked IOException at runtime if the file cannot be located. It also normalize the path while toAbsolute

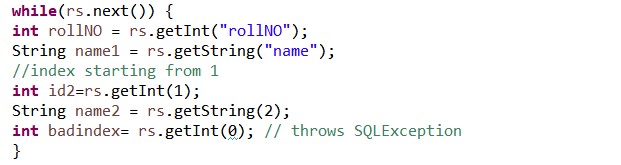
1. Files{} method:
   1. Files.getAttribute() It takes variable number of parameters: a Path object, an attribute name, and the link options.
   2. Files.copy(Path source, Path target,CopyOption … opt): By default, copying files and directories will traverse symbolic links, although it will not overwrite a file or directory if it already exists, nor will it copy file attributes.
   3. Path Files.move(Path source, Path target, CopyOption... options)
2. Some **stream** method related to the Files
   1. static Stream<Path> **walk**
      1. static Stream<Path> walk(Path start, FileVisitOption... options) throws IOException {}
      2. static Stream<Path> walk(Path start, int maxDepth, FileVisitOption... options) throws IOException{}
   2. static Stream<Path> **find**(Path start, int maxDepth, BiPredicate<Path, BasicFileAttributes> matcher, FileVisitOption... options) throws IOException {}
      1. Path and BasicFileAttributes, which are the two variable in BiPredicate Stream<Path> list(Path dir) throws IOException {}
   3. static Stream<String> **lines**
      1. static Stream<String> lines(Path path, Charset cs) throws IOException {}
      2. static Stream<String> lines(Path path) throws IOException {}
   4. static byte[] **read**(InputStream source, int initialSize) throws IOException {}
3. **Difference between Files.walk(), Files.find () and File.list()?**
   1. **Files.list(file):** It does not recursively traverse the entries.
   2. **Files.walk():** Files.walk() throws FileSystemLoopException If cycle had detected.
   3. **Files.find()** is used if we want to filter out files by **attributes or with a match conditions**.
4. Path vs File
   1. Path throws IOException more often than File and rarely return a Boolean to tell if something was done (mkdirs(), delete()).
   2. <https://www.oracle.com/technetwork/articles/javase/nio-139333.html>



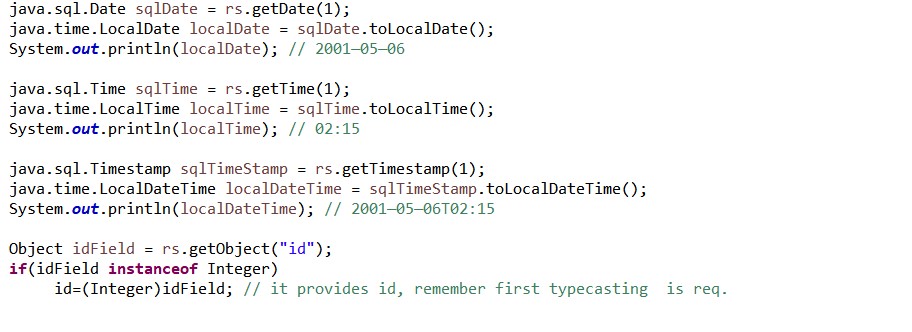
1. **isSameFile**() method first checks if the Path objects are equal in terms of equal() , and if so, it automatically returns true without checking to see if either file exists. If the Path object equals() comparison returns false , then it locates each file to which the path refers in the file system and determines if they are the same, throwing a checked IOException if either file does not exist
2. **void** java.nio.file.attribute.**BasicFileAttributeView**.setTimes(FileTime, FileTime, FileTime)(FileTime lastModifiedTime, FileTime lastAccessTime, FileTime createTime) **throws** IOException;

# JDBC (java.sql package)

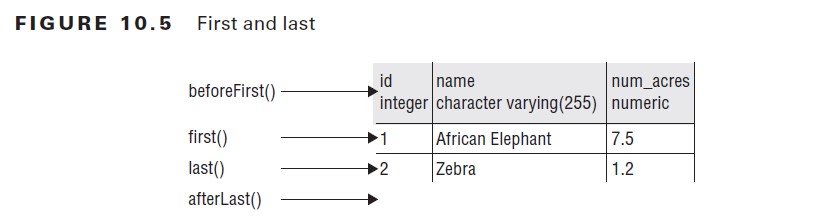
1. JDBC history:
   1. java 1.0 ->jdbc 1.0
   2. java 1.4 ->jdbc 3.0
   3. java 6.0 ->jdbc 4.0
   4. java 7.0 ->jdbc 4.1
   5. java 8.0 ->jdbc 4.2
2. JDBC API have two package
   1. **java.sql**(simple class):Interface: Driver, connection , statement, prepared statement, callablestatment result set,
      1. class: date, drivermanager,
   2. **java.sqlx**(complex class): rowset , RowSetInterface, ConnectionEventListner.
3. DriverManager looks for a jar that driver META-INF/service/java.sql.Driver
   1. for ex postgresql file
   2. postgresql-42.2.5.jar\META-INF\services\java.sql.Driver ->org.postgresql.Driver
   3. DriverManager.getConnection() throws **SQLException**{}
4. forName() is a static method that loads a class, it is find in Class class. This lets DriverManager use a Driver, even if the JAR doesn’t have a META-INF/service/java.sql.Driver file. Class.forName() throws **ClassNotFoundException**{}
5. We can use DataSource interface or DriverMangaer class to connect a JDBC app to target data source for getting **connection.**
6. With JDBC 3.0, the driver is **allowed**, to contain a java.sql.Driver file, and the code getting a Connection is required to call Class.forName(). While for JDBC 4.0 vicerversa.
7. In JDBC 4.0, the drivers are loaded automatically based on the information provided by the driver's META-INF/services/java.sql.Driver file. Therefore, no need to write **Class.forName**(“org.postgresql.Driver”) that the above files have it load automatically.
8. Connection c = DriverManager.getConnection("localhost:1234/myDB", "user", "pwd");
   1. It is used for getting connection not the driver details
9. Legacy code for getting connection v/s new code: 
10. **Properties p = new Properties();  
      p.setProperty("user", userid);  
      p.setProperty("password", pwd);  
      Connection c =  DriverManager.getConnection(dburl, p);**
11. Key JDBC interfaces
    1. Driver : knows how to get a connection to the database
       1. The **class** java.sql.DriverManager { } provides a factory method called getConnection() to get a Connection implementation
       2. **public** **interface** java.sql.Driver {}
    2. Connection: how to communicate with the database
       1. interface java.sql.Connection extends Wrapper, AutoCloseable {}
    3. Statement : how to run the SQL
       1. interface java.sql.Statement extends Wrapper, AutoCloseable {}
    4. ResultSet : What was returned by SELECT query
       1. interface java.sql.ResultSet extends Wrapper, AutoCloseable {}
12. JDBC Url
    1. ports are optional when using the default
    2. A location can be localhost or an IP address or a domain name
    3. two ways for the connection DRIVERMANAGER OR DATASOURCE.
       1. Eg jdbc:mysql://[host][:port]/[database][?property1][=value1]…
       2. "jdbc:mysql://localhost:3306/whiz?"+"user=root&amp;password=whizlabs
       3. “jdbc:mysql://localhost:3306/whiz”,"root”,”whizlabs”
       4. "jdbc:mysql://:3306/whiz?user=root&password=whizlabs"
13. Statement stmt =conn.createStatement();
    1. Statement stmt = conn.createStatement(ResultSet.TYPE\_FORWARD\_ONLY, ResultSet.CONCUR\_READ\_ONLY);
       1. by default TYPE\_FORWARD\_ONLY but also TYPE\_SCROLL\_INSENSITIVE and TYPE\_SCROLL\_SENSITIVE
       2. If not available the driver get downgrade i.e. TYPE\_SCROLL\_SENSITIVE to TYPE\_SCROLL\_INSENSITIVE
       3. SAME FOR THE ResultSet.CONCUR\_UPDATABLE -> ResultSet.CONCUR\_Read\_ONLY
    2. A Statement automatically closes the open ResultSet when another SQL statement is run.
14. Statement: It is used to execute normal SQL queries. The performance is very low.
15. PreparedStatement: It is used to execute dynamic or parameterized SQL queries. It is used when we have to run statement multiple time and is faster than **statement.**
16. CallableStatement: It is used to execute store procedure and functions. The performance is high as compared to PreparedStatement.
17. Executing the statement:
    1. int java.sql.Statement.executeUpdate(String sql) throws SQLException
       1. it gives int for no of delete, insert , update
    2. ResultSet java.sql.Statement.executeQuery(String sql) throws SQLException
       1. it is used for giving the result set only that have data
    3. boolean java.sql.Statement.execute(String sql) throws SQLException
       1. return boolean to indicate whether the statement was query(true) and not query(false).
18. Getting data from ResultSet:
    1. Column indexes begin with 1.
    2. You can use an index(Starting from **1**) as well as column name for retrieval of data from ResultSet.



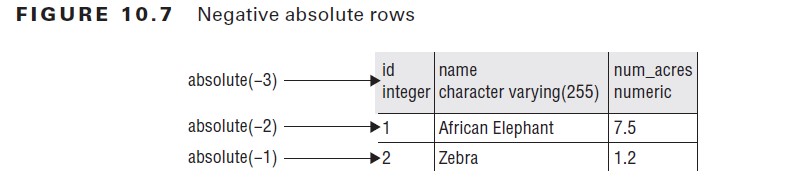
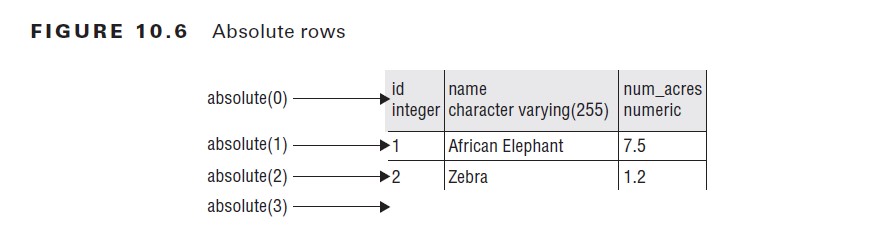
* 1. For **date** and objects.



1. Scrolling the **ResultSet**
   1. ResultSet has a cursor, which points to the current location in the data
   2. Make sure that the type is scrollable whenever you see methods other than next().
   3. boolean rs.**next**() -> move the cursor one row forward
   4. boolean rs.**previous**() -> move the cursor one row backward
   5. boolean rs.**first**() -> void rs.beforeFirst() it is before the first line i.e. starting of the data
   6. boolean rs.**last**() -> void rs.afterLast() it is after the last line

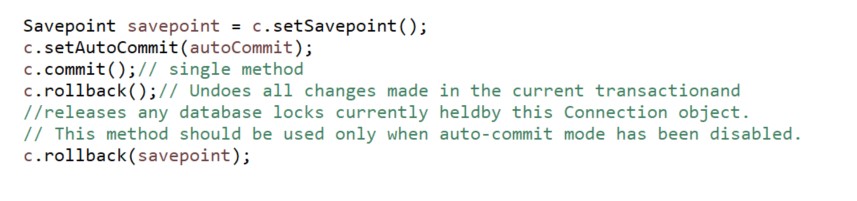


* 1. boolean rs.**absolute**(int i) -> move the cursor to the specified row number
     1. Absolute(0) gives **false** and is for title.

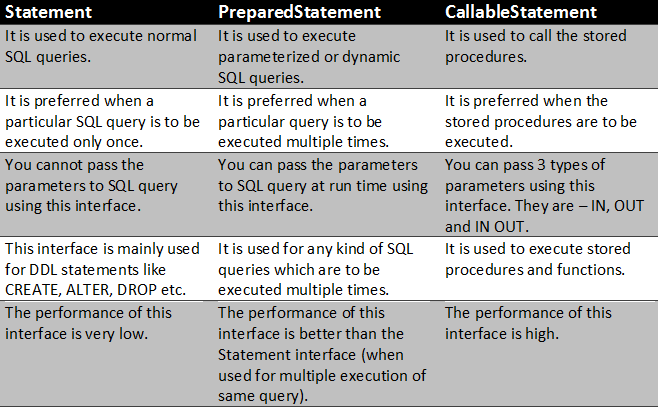


* 1. **relative**(int i) method that moves forward or backward the requested number of rows. It returns a boolean if the cursor is pointing to a row with data
  2. **beforeFirst**() and **afterLast**() method used to go outside of the ResultSet and return void.

1. Rollback, Savepoint and commit.
   1. We cannot rollback to savepoint when the transcation is already commited. Rollback is used for roll back small parts of a bigger transaction.
2. **RowSet:** All RowSet objects are JavaBeans components because they have properties and JavaBeans Notification Mechanism

****

1. Dealing with the exceptions
   1. SQLException.getMessage() // get human readable code
   2. SQLException.getSQLState() //returns a code as to what went wrong db specific
   3. SQLException.getErrorCode()
   4. For closing all the three resources, the ReslultSet must be closed first ,then Statement and then Connection as closing the later will automatically close the earlier.
2. As per Section 6.2 of JDBC 4.1 Specification:A JDBC API implementation must support Entry Level SQL92 plus the SQL command Drop Table.Entry Level SQL92 represents a "floor" for the level of SQL that a JDBC API implementation must support. Access to features based on SQL99 or SQL:2003 should be provided in a way that is compatible with the relevant part of the SQL99 or SQL:2003 specification
3. The **actual** classes for Connection, Statement, and ResultSet interfaces are provided by the JDBC driver and are **therefore driver dependent**.st4.7
4. Difference between Statement, PreparedStatement and CallableStatement?
   1. Statement interface:- Used to execute normal SQL queries. This is used to execute normal SQL query.
   2. PreparedStatement:- Used to execute dynamic or parameterized SQL queries.
   3. CallableStatement:- Used to execute the stored procedures. It extends PreparedStatement.



# More on exam

1. **Q 1. What is difference between Synchronized and Concurrent Collections in Java?**

The class object and the instances of the class are not related in terms of synchronization.

A process runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process, e.g. memory and CPU time, are allocated to it via the operating system.

Synchronization means to control the access of multiple threads to a shared resource**. Block synchronization** in Java is preferred over method synchronization in Java because by using block synchronization, you only need to lock the critical section of code instead of the whole method Read more: https://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html#ixzz5iboh4ENZ [https://javarevisited.blogspot.com/2011/04/synchronization-in-java- synchronized.html](https://javarevisited.blogspot.com/2011/04/synchronization-in-java-%20synchronized.html)

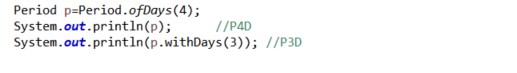


1. Difference between sort in Arrays and collection is that Arrays.sort() take **Arrays** while Collection.sort() take **List**.



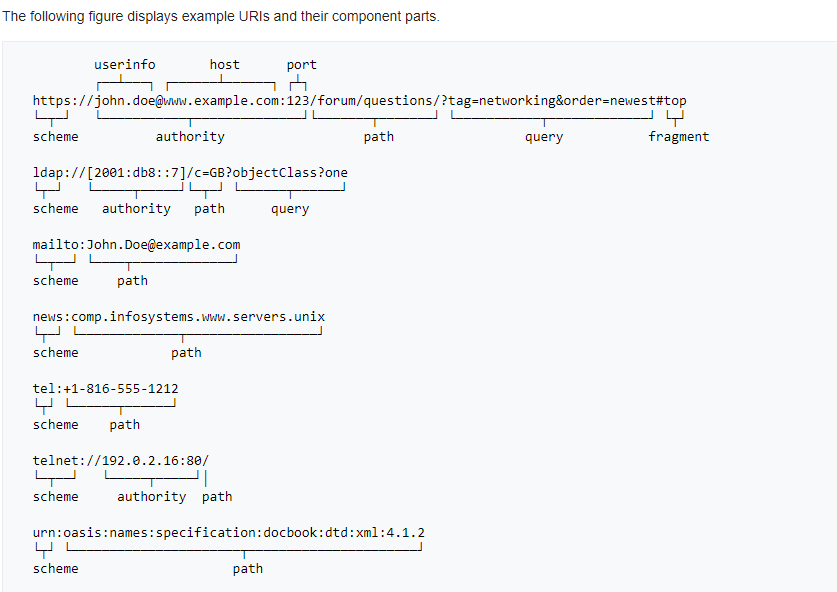
* 1. Note that since a.sort() doesn't return anything, print a.sort() will print Nothing.
  2. The main difference is that sorted() returns a new list i.e. does not modifies the old list and it is the intermediate operation of Stream while sort() modifies the old list
  3. Collections.sort(collection, new CustomComparator());
  4. collection.stream().sorted(new CustomComparator());
  5. For sort() to works pass comparable class or pass comparator with Object list.

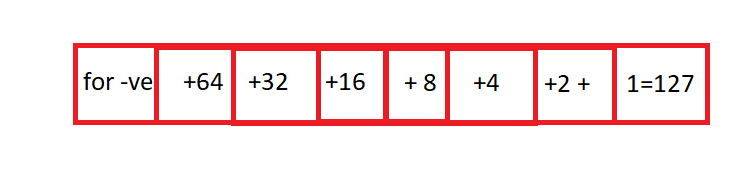
1. **Period ofDays v/s withDays()**



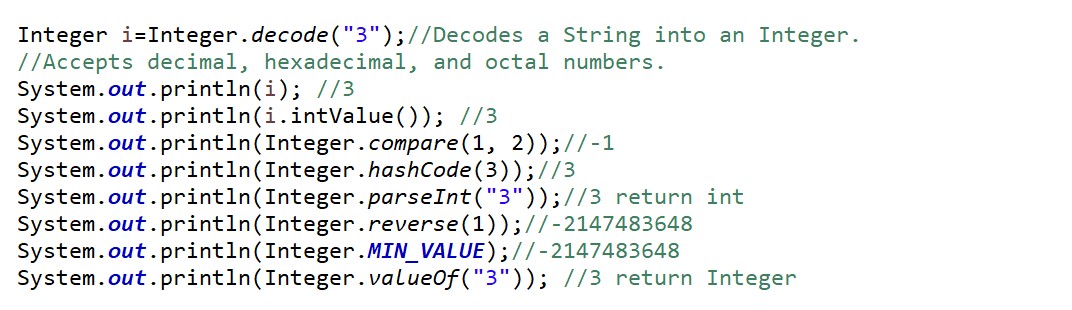
1. **Why we use List al = new ArrayList(); ?**

This is called programming to interface. This will be helpful in case if you wish to move to some other implementation of List in the future. If you want some methods in ArrayList then you would need to program to the implementation that is ArrayList a = new ArrayList().

1. **Error from the eclipse** 
   1. whizlab<> my = **new** whizlab<>();
   2. **class** whizlab<Double> {}
   3. Incorrect number of arguments for type whizlab<Double>; it cannot be parameterized with arguments <>
2. Queue vs Deque method.
   1. pollFirst()/pollLast() - poll is a Queue method. Therefore pollFirst and pollLast will remove elements from the front and from the end respectively.
   2. removeFirst()/removeLast() - These are Deque specific methods. They will remove elements from the front and from the end respectively. These methods differ from pollFirst/pollLast only in that they throw an exception if this deque is empty.
   3. offerFirst(e)/offerLast(e) - offer is a Queue method. Therefore offerFirst and offerLast will add elements to the front and to the end respectively.
   4. addFirst(e)/addLast(e) - add is a Queue method. Therefore addFirst and addLast will add elements to the front and to the end respectively.
   5. peek(), peekFirst(): return the first element from the front of the queue but does not remove it from the queue. peekLast() : returns the last element from the end of the queue but does not remove it from the queue.
   6. element(): retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque). This method differs from peek only in that it throws an exception if this deque is empty.
   7. Notice that there **is no pushFirst(e) and pushLast(e).**
   8. You may wonder why there are multiple methods for the same thing such as offer(e) and add(e). Well, they are not exactly same. add(e) throws an exception if the element cannot be added to the queue because of lack of capacity, while offer(e) does not. There are similar differences in other methods but they are not too important for the exam.
   9. Note that the [peek](https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html#peek()) method works equally well when a deque is used as a queue or a stack; in either case, elements are drawn from the beginning of the deque.
3. **System vs console class ?**
   1. Can we create an object of PrintStream and call println function with that object to print to the standard output (usually the console)? The answer is NO. Instantiating a PrintStream will allow you to write to a File or OutputStream you specify, but don't have anything to do with the console.
   2. **System.in**
   3. System.in is an InputStream which is typically connected to keyboard input of console programs. System.in is not used as often since data is commonly passed to a command line Java application via command line arguments, or configuration files. In applications with GUI the input to the application is given via the GUI. This is a separate input mechanism from Java IO.
   4. **System.out**
   5. System.out is a PrintStream. System.out normally outputs the data you write to it to the console. This is often used from console-only programs like command line tools. This is also often used to print debug statements of from a program (though it may arguably not be the best way to get debug info out of a program).
   6. **System.err**
   7. System.err is a PrintStream. System.err works like System.out except it is normally only used to output error texts. Some programs (like Eclipse) will show the output to System.err in red text, to make it more obvious that it is error text.
4. **System.out.println() v/s System.out.write()?**
   1. Writes the specified byte to this stream. If the byte is a newline and automatic flushing is enabled then the flush method will be invoked.
   2. Prints an Object and then terminate the line. This method calls at first String.valueOf(x) to get the printed object's string value, then behaves as though it invokes [print(String)](eclipse-javadoc:%E2%98%82=OcpBook/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava.io(PrintStream.class%E2%98%83PrintStream~println~Ljava.lang.Object;%E2%98%82%E2%98%82print%E2%98%82String) and then [println()](eclipse-javadoc:%E2%98%82=OcpBook/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava.io(PrintStream.class%E2%98%83PrintStream~println~Ljava.lang.Object;%E2%98%82%E2%98%82println%E2%98%82).
5. **Thread.run() vs Thread.start()?** 
   1. In Thread.run() new thread does not created while in Thread.start() new thread is created. When run() is called it is work as normal method and neither as Thread.start() we can call Thread.run() many times.
6. **Nested Classes**
   1. Nested class are divide into two categories static and not-static. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class. As a member of the OuterClass, a nested class can be declared private, public, protected, or package private. (Recall that outer classes can only be declared public or package private.)
   2. Why nested class ?
      1. To group class that is used in one place.
      2. To increase encapsulation
      3. For readability and maintain code.
   3. Two kind of inner class local classes and anonymous classes. **Local class** are class that are define in a block. They are define in a **method body** , for **loop** or an **if** clause.
7. Differences between TreeMap, HashMap and LinkedHashMap in Java ?
   1. **HashMap: HashMap offers 0(1) lookup and insertions.**
   2. public class HashMap extends AbstractMap implements Map,Cloneable, Serializable
   3. A HashMap contains values based on the key.
   4. It maintains no order.
   5. **LinkedHashMap:** Keys are ordered by their insertion order. It is implemented by doubly linked buckets.
   6. Public class LinkedHashMap extends HashMap implemets Map{}
   7. **TreeMap**: TreeMap offers **O(logN)** lookup and insertion**. Keys are ordered**, so if you need to iterate through the keys in sorted order. TreeMap is implemented by Red-Black-Tree.
   8. Public class TreeMap extends AbstractMap implements NavigableMap, Cloneable, Serializable{}
   9. **Hashtable:** public class Hashtable extends Dictionary implements Map, Cloneable, Serializable{}. Hashtable is array of list and each list is known as bucket.
8. URI: Uniform Resource Identifier:
9. **My comman mistake while solving problem?**
   1. Use curly braces while using return in lambda.
   2. When connection.setAutoCommit() is false it does not commit automactically we have to used to used commit() separately.
   3. forEachOrdered() method is only define in Stream class.
   4. (protected, public private) Illegal modifier for parameter l; only final is permitted inside a **class method** .
   5. If trywithresource don’t use closable or Autoclosable then it is compile time error with the resource type Prac32.Player does not implement java.lang.AutoCloseable
10. Difference between Iterator(1.2) and Enumeration(1.0) ?
    1. The functionality of Enumeration is duplicated by the iterator interface.Iterator adds an optional remove operation,and has shorter method names.
    2. Enumeration methods:
       1. hasMoreElements() : return true of false
       2. nextElement() : return the element or **NoSuchElementException**
    3. Iterator Methods:
       1. hasNext() : return true of false
       2. next() : return void or throws UnsupportedOperataionException(if remove operation is not supported) or IllegalStateException(if remove method is already called after the last call to next method)
       3. remove() : return the element or **NoSuchElementException**
11. As per https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html :
12. **A reduction operation** (also called a fold) takes a sequence of input elements and combines them into a single **summary result by repeated application of a combining operation**, such as finding the sum or maximum of a set of numbers, or accumulating elements into a list. The streams classes have multiple forms of general reduction operations, called reduce() and collect(), as well as multiple specialized reduction forms such as sum(), max(), or count().
    1. For example, you can combine elements of a stream to express complicated queries such as "Calculate the sum of all sales in a given list of transactions" or "Find the highest sales transaction in a list of transactions" using reduction operations. They are called as such because such queries combine all the elements in the stream repeatedly to produce a single value such as a Double. In other words, a stream is reduced to a value by applying these operations.
    2. Note that filter(Predicate<? super T> predicate) is not a reduction operation because it does not **combine** all the elements and produce one value.
13. Some Number Format:
    1. Byte
       1. **Byte Maximum value :** System.out.println(Byte.MAX\_VALUE); is 127



* 1. Integer Class
     1. **Integer i=Integer.decode("3");//**Decodes a String into an Integer.Accepts decimal, hexadecimal, and octal numbers.
     2. i.doubleValue(): //3.0 convert a given Integer/Float to double value



1. **Autoboxing and Unboxing**: It was introduced in java 1.5. Compiler uses valueOf() method to convert primitive to Object and uses intValue(), doubleValue() etc to get primitive value from Object.
2. Widening or Automatic Type Conversion:- Some rule regarding widening:
3. Widening wins over boxing and var-args
4. Boxing wins over var-args
5. Widening of reference variable depends on inheritance(so, Integer cannot be widened to Long. But, Integer widened to Number).
6. Widen and boxing is not possible
7. Boxing and widening is possible
8. var-args can be combined with either boxing or widening
9. Narrowing or Explicit Conversion: It does not occurs automatically, we have especially Explicit conversion to it.