# Advanced Class design

## OCA concepts

* Access modifiers in method
  + protected:
    - invoke without variable reference: accessible if in same package or in subclasses
    - invoke with variable reference: accessible for reference with same or subtype for where the invocation is taking place.
* Overriding, overloading
  + @Override: annotation to tell compiler that the method will override. If it’s not, then compiler error to make you fix
* Abstract class
* Static vs final keyword
  + final instance var need to be init in constructor, at declaration and instance initializer, OR else compile error because they are not assigned to default value
* Import and package statement
* Instanceof: used to compare whether the object that the reference is pointing to is same or subtype of a class/interface.
  + Work fine with object type and type in related hierarchy
  + Compile error when instanceof class and reference type totally have no connection in inheritance tree
  + But, still compile if use instanceof with an Interface or an Interface-type reference variable instanceof a class
  + null reference using instanceof will return false

## Override equals, hashCode and toString of Object class

* public String toString(){}
* public boolean equals(Object obj){}
  + Firstly, check using instanceof
  + null as parameter will return false
  + Using null reference to call equals will cause compile error
* public int hashCode(){}
  + When return a numeric primitive instance variable, can use as it is or divide
  + When return a reference instance variable, don’t choose variable that changes time to time. Also, use hashCode method to get hash code from object reference
  + instance variable in hashCode should be equal or subset of equal method

## Enums

* Is a type of class, so if an enum and class are both public in same file→ compile error
  + After compile, can’t add more
  + Implicitly final
  + CAN’T be defined in method or constructor
  + Can only implements interface, not extends another enum or class
  + Enum Implement Comparable interface
  + When use enum as expression in switch, the case can only be the value of enum ALONE, excluding the name of enum.
* Enum design: all part are optional
  + List of enum values, with each value being instantiated according to the constructor and body to list overriding non-static, non-final methods (if you want) to the enum methods (*abstract* or non-abstract). Braces are optional for no-args constructor
  + Constructors: *private* by default, unlike class’s constructor
  + abstract or non-abstract methods: modifier are used as normal in class

|  |  |  |
| --- | --- | --- |
| **Method** | **Return** | **Note** |
| values | array of all constant of an enum |  |
| valueOf(String s) | enum value match String | Must match exactly |
| ordinal | index of enum Constant | May be diff when recompile  Zero-based |
| name | name of enum Constant |  |

## Nested classes (4 types)

* Can treat nested classes as the variable inside the method definition. Therefore, how each type of nested classes can shadow and access variable is treated similar:
  + Member (non-static) inner class can access both static and instance variable of it’s enclosing class
  + Local inner and anonymous class can access and not Shadowing local variable, can access and shadow instance variable (only if enclose in instance method) and static variable (static and non-static method) of the outer class
  + Static inner class: obviously can’t access instance variable of outer class since the nested class is attached to class

### Member (non-static) inner classes

* Can do anything normal class does plus:
  + Can use any access modifier
  + Can access outer(Containing) class private member
  + Can access outer class variable with syntax: *OuterClass.this.varName* or just the varName if there’s no conflict
  + Can extends any class or interface, including outer class
  + CAN’T have static fields and methods
* Since it’s non-static, need to instantiate an object of outer class in order to instantiate object of inner class
* Inner interfaces can be any access modifier, but other rules of outer interface members still apply here as usual (method public by default; default and static method)
* Inner class can access private member of outer class, and VICE VERSA. *private* cover scope of access of outer class, not inner class

### Local inner class

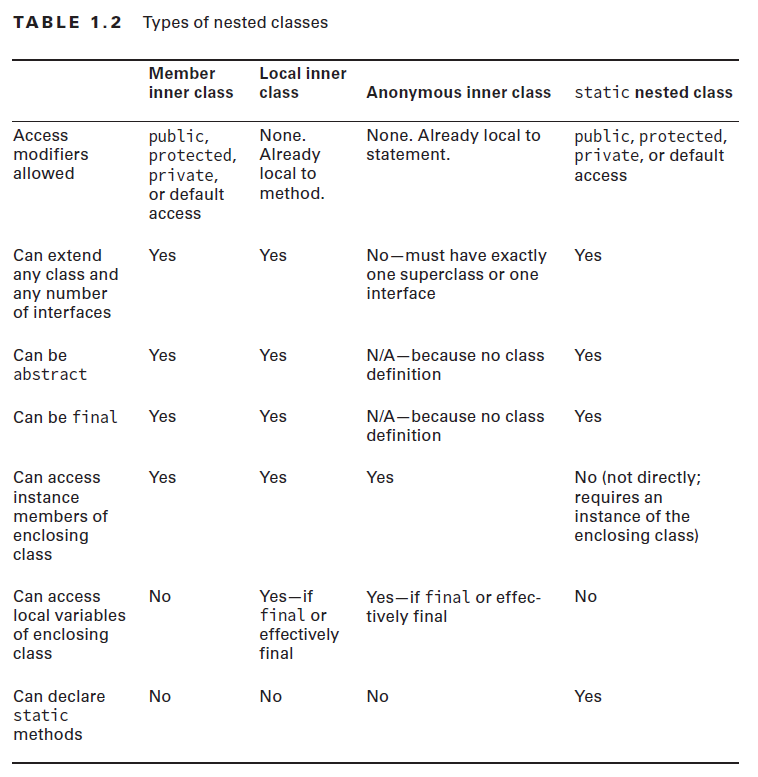
* Similar to local variable in method:
  + No access level
  + Non-static with all non-static members
  + Can access members of enclosing class
  + Access outer class variable with syntax: *OuterClass.this.varName*
  + CAN only access *final* or effectively final(ones that are assigned only ONCE, even with conditions) local variable of its enclosing method.
  + For instance and static var of outer class, instance var of local inner class, you can access without the need for *final* or effectively *final*
  + Can be used to return
* Follow the order of statement in method, can’t instantiate before local class declaration

### Anonymous inner class

* Local class with all its characteristic, with no name
  + Use *new* keyword to instantiate following the name of class or interface with braces as making new instance, and lastly the class body within curly brackets
  + Must extend class or implement interface, only one
  + Implicitly final since there’s no class name
* Bottom line: create a new subclass without name and instantiate a new object at same time, create an object that can be assign to supertype variable or use as parameter

### Static nested class

* Since it’s static, the only way to instantiate it is just calling new directly.
* The only type that can have static methods and variable, therefore those static member can be used without instantiate new object of inner class. Can be invoked through object reference, just like how outer static members behave.



# Design pattern and principles

## Interface design

* Review interface in OCA

## Functional Programming introduction

* Functional interface:
  + Interface with exactly 1 abstract method (either define in itself, override super-interface, or inherit), and other default or static methods
  + Can be marked with @FunctionalInterface for compiler & developer to check
  + 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 abstract method count since any implementation of the interface will have an implementation from java.lang.Object or elsewhere.

## Polymorphism

* OCA review
* Subtype reference cast to supertype does not required explicit casting
* Supertype reference cast to subtype reference does required explicit casting
  + Compile error if both are not related to inheritance hierarchy
  + Runtime ClassCastException if the actual object IS-NOT-A type of reference

## Design principles

* Benefits:
  + More logical
  + Understandability
  + Reusability
  + Adaptability and maintainability
* Encapsulation:
  + Instance variable *private*
  + *public* getter and setter with business logic inside those methods to modify instance variable
* JavaBeans rules for encapsulation:

|  |  |
| --- | --- |
| **Rules** | **Example** |
| private properties (instance variable) |  |
| Getter for non-boolean properties begins with get |  |
| Getter for boolean properties begins with is |  |
| Setter begins with set |  |
| Setter/getter begins those defined above and the properties name with its first letter capitalized |  |

* IS-A relationship: single-inheritance class in Java
* HAS-A relationship: object composition in which a class HAVING objects of other class and then delegate those class methods (through method invocation of object). Solve problems that single-inheritance class in Java can’t do and ensure more logical, make-sense data model of real world.

## Design patterns

* As *final* instance variable and *final*  static variable have no default value, they must initialize at:
  + Static variable: at declaration or static initializer (initializer accepted even placed before static variable declaration)
  + Instance variable: at declaration, instance initializer (initializer accepted even placed before static variable declaration), OR in constructor
* **Singleton patterns:** creating only 1 object of the class and have it shared and modified by many other classes by:
  + Make all constructors private
  + Declare a *private static* variable of name *instance* (idiom) with type it’s in. This object is called Singleton object
  + Instantiate Singleton object (should be *final* to ensure only 1 object exist) at:
    - Singleton declaration
    - Static initializer: *static{}*
  + Have a *public static method Class getInstance* (idiom) that return Singleton object
  + Lazy initialization:
    - Don’t initialize as above
    - Instantiate Singleton object in *public static synchronized Class getInstance* method if and only if Singleton object hasn’t created (reference of Singleton object == null)
    - So, Singleton object can’t be *final* in this case as *final* variable have no default value and can’t be reassigned
* **Immutable Objects pattern:**
  + Use a constructor to set all properties (instance variable) of object. However, for all mutable object reference passed in constructor, we have to create a new object (copy) that is same to it and assigned the instance variable to the new (copied) object
  + All instance variable: *private final*
  + No setter methods
  + Don’t allow referenced mutable object be modified or accessed directly (constructor above is applied with similar rule)
  + Prevent methods from overridden: by marking class or methods *final*, OR apply factory pattern
* **Builder pattern:** there’s a immutable class called target and a mutable class called its Builder
  + Design a mutable class that have same properties as its target immutable class, non-final for setter
  + Each public setter that set each properties and return the current instance reference for method chaining in using builder
  + *build* (idiom) method that return the immutable object reference by instantiating that immutable object with builder’s properties.
* **Factory Pattern:**
  + Create a factory method that return instances/objects based on a set input param

# Generics and collections

## OCA reviews

* Array and ArrayList:
  + Array convert to List: List will be linked to array, therefore, List is unresizable. Change in either of them will be reflected to other
* *Collections.binarySearch* and *Arrays.binarySearch* only work on sorted ones.
* Wrapper classes and autoboxing
* Diamond operator: on declaration side, fully written is required if used while it’s ok to use empty diamond in initialization

## Generics

* Diamond operator placed after the class/interface name in its declaration or generic method so that the type can be used inside which list type parameters that can be used within the class.
  + Must not be empty.
  + Can contain 1-many type param
  + Convention is single uppercase letter for each type param

### Places that generic can be defined

* Interface and class can use generics. However, when interface get implemented or/and a class get extended by others, generics of super interface/class in the subtype (subclass or subinterface) declaration MUST either:
  + Be defined to specific type

*class MySub extends GenericClass<String> implements GenericFace1<Integer>*

* + Must follow the type param defined in diamond operator of its sub

*class MySub<T> extends GenericClass<T> implements GenericFace2<T, Integer>*

* + Diamond operator removed, which implicitly means acceptable type is Object, which is everything

*class MySub<T> extends GenericClass2 implements GenericFace2<T, Integer>*

* Generic method:
  + Diamond before the return type
  + Return type can be method generics
  + Will shadow generic from class, interface if have same name
  + When invoke method, use ref.*<type>methodName()* if need to specify type

### Bound wildcards

* Bound wildcards are used in variable declaration and generic method
* When there’s a bound wildcard in the variable declaration, think that that variable can store reference to object with type that follow the rule of bound wildcards.
  + Have 3 types: Reread
  + Unbounded wildcard: make list “immutable”,which can’t be added more element. Therefore, usually use as method parameter to iterate
  + Upper-bounded wildcards: make list “immutable”,which can’t be added more element. Therefore, usually use as method parameter to iterate
  + Lower-bounded wildcards: Unlike previous 2, list with this generic can be added new element.
* Can only be used in variable declaration, parameter of method

### Generic spec and rules

* Generic in variable declaration:
  + Can have no diamond
  + Diamond can be specific type or use bound wildcards
  + Diamond can’t be empty if diamond is used
* Generic in variable initialization:

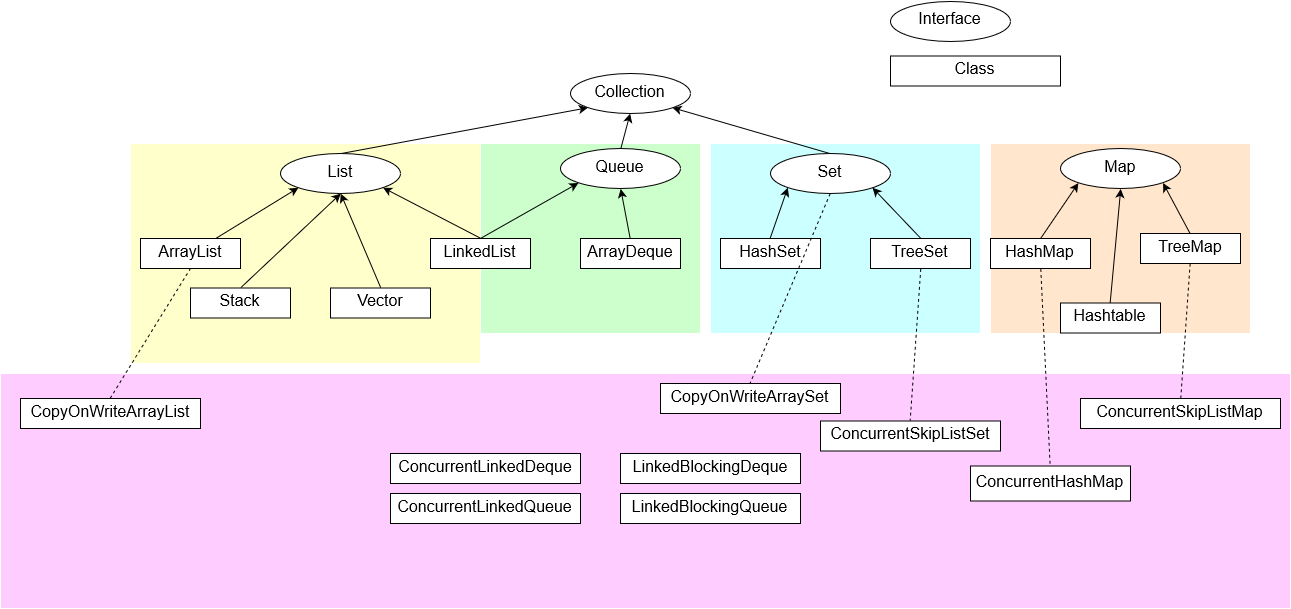
*List<? super Integer> list = new ArrayList<>();*

* + Can have no diamond -> warning when compiling and behave as empty diamond
  + Diamond have to be specific type,can’t use bound wildcards
  + Diamond can be empty if diamond is used -> object type that can be passed into method invocation through that variable will be limit to the how diamond in the variable declaration specified.
* Generic in generic method declaration and generic class/interface declaration
  + For generic method: return type must be specific in either following the type param or explicitly written (e.g: int).
  + Use normal bound (replace ? with the type identifier) in generic method and generic class/interface. Those are only 2 place in the declaration that we can use normal bound. T is the type parameter

*public class MySub<T extends Number>{}*

*public <T extends Number> T work(List<?> list){}*

## Lists, Sets, Maps and Queues



|  |  |  |
| --- | --- | --- |
| **Return** | ***Collection* interface methods** |  |
| boolean | add(E element) |  |
| boolean | remove(Object object) |  |
| boolean | contains(Object object) |  |
| boolean | isEmpty() |  |
| int | size() |  |
| void | clear() |  |
| void | forEach(Consumer<? super T> action) | Inherit from *Iterable* interface  Lambda or method reference is fine  Iterate through list and do something |

* Below data structures allows nulls except:
  + Hashtable: no null keys or values
  + TreeMap: no null keys
  + TreeSet: no null element
  + ArrayDeque: no null element
  + ConcurrentHashMap: neither null key or value

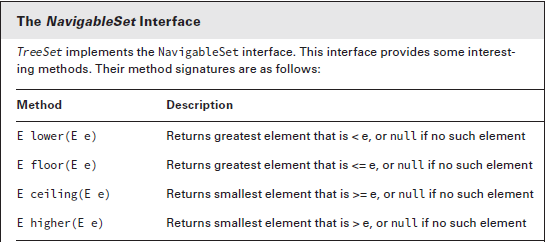
### List interface

* Ordered collection that can contain duplicate entries,. Implementation:
  + LinkedList: also implements Queue
  + ArrayList: resizable array, used 0-based index for ordering elements based on the order it is added to the collection
  + Vector and Stack: Old way
* Includes all methods inherited from Collections interface PLUS:

|  |  |  |
| --- | --- | --- |
| **Return** | ***List* interface methods** |  |
| methods from *Collections* interface | | |
| void | add(int index, E element) | overloaded  Add element at the index |
| E | get(int index) | IndexOutOfBoundException |
| int | indexOf(Object o) | -1 for not found |
| int | lastIndexOf(Object o) | -1 for not found |
| E | set(int index, E element) |  |
| E | remove(int index) | IndexOutOfBoundException |
| boolean | removeIf(Predicate<? super E> filter) | Java 8 bonus  Take lambda, not method reference  Element that make filter return true will be removed |
| void | replaceAll(UnaryOperator<E> o) | Java 8 bonus  Take lambda  Replace each element with new one based on the lambda body |

### Set Interface

* Collections with no duplicates
  + HashSet: instead of using index based-0, use hashcode of the element to order elements
  + TreeSet: implements *NavigableSet* interface



* Includes all methods inherited from Collections interface PLUS:

|  |  |  |
| --- | --- | --- |
| **Return** | ***Set* interface methods** |  |
| methods from *Collections* interface | | |

### Queue Interface

* Queue implementation:
  + ArrayDeque: double-ended queue which store elements in resizable array. Therefore, can be used as stack as Last-in-first-out OR as Queue as First-in-first-out
* Includes all methods inherited from Collections interface PLUS:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Return** | ***ArrayDeque* class methods** |  | **For**  **Queue** | **For**  **Stack** |
| methods from *Collections* interface | | | | |
| boolean | add(E element) | Add element to back  Return true OR exception | Y | N |
| boolean | offer(E e) | Add element to back  Return true OR false | Y | N |
| void | push(E e) | Add element to front | Y | Y |
| E | element() | Return next element OR exception if empty queue | Y | N |
| E | peek() | Return next element OR null if empty queue | Y | Y |
| E | remove() | Return next element OR exception if empty queue  Remove that element | Y | N |
| E | poll() | Return next element OR null if empty queue  Remove that element | Y | N |
| E | pop() | Return next element OR exception if empty queue  Remove that element | N | Y |

For deque interface, there’re

* offerFirst(E e), offerLast(E e) method:
  + Return type is boolean
  + In BlockingDeque, also have overloaded version that have timeout long and time unit, as mention in chapter below
* removeLast(E element) and removeFirst(E element):
  + Return element type
  + throw exception if deque is empty
* addFirst(E element) and addLast(E element):
  + similar to offer version

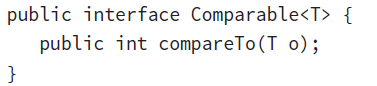
### Map interface

* Key-value pair kind of collections. Implementation:
  + HashMap: Use hashCode of element to order the elements of collection
  + TreeMap:
* Does not extends from Collections

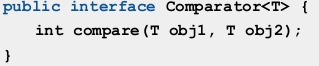
|  |  |  |
| --- | --- | --- |
| **Return** | ***Map* interface methods** |  |
| void | clear() |  |
| boolean | isEmpty() |  |
| int | size() |  |
| V | get(Object key) | Return null if none found |
| V | put(K key, V value) | Add if no key exist, else replace  Return null if none found or previous value |
| V | remove(Object key) | Return null if none found |
| boolean | containsKey(Object key) |  |
| boolean | containsValue(Object value) |  |
| Set<K> | keySet() |  |
| Collection<V> | values() |  |
| void | foreach(Biconsumer<K,V> b) |  |
|  | merge |  |

## Comparator vs Comparable

* Sorting: spaces < numbers < uppercase < lowercase



* *Comparable* Interface:
  + implement this interface, inputting generic in diamond as the implemented class name
  + If generic of interface is not specified, must explicitly cast in *compareTo* method
  + override *compareTo* method:
    - Sort by a chosen String instance variable: reuse *compareTo* method of String
    - Sort by a chosen int instance variable: either :
      * *return (int instance var - instance var of param)* for ascending
      * *return -(int instance var - instance var of param)* for descending
      * reuse compareTo method of wrapper Integer
  + Class implemented this interface can be sorted when stored in collection with static method *Collections.sort(collectionObject)*

**

* *Comparator* interface:
  + 2 ways to use
    - Make an anonymous inner class that implement compare method
    - Lambda expression
  + Use *Collections.sort(collectionObject, <Lambda or object from anonymous class>)*

## Searching and Sorting

|  |  |  |
| --- | --- | --- |
| **Return** | ***Collections.sort* methods** |  |
| void | sort(List<T> list) |  |
| void | sort(List<T> list, Comparator<? super T> c) | c can be lambda  OR a object created from an anonymous inner class that implements *Comparator* |

* For TreeMap and TreeSet, object of a type can be stored in these if either:
  + type implement *Comparable* interface
  + type does not need to implement *Comparable* interface, but inputting into object initialization a lambda of *Comparator* or object of an anonymous class that implement *Comparator* interface

## Java 8 bonus

* Method reference use functional interface to input method as the body of lambda and auto match the parameter list of lambda without explicitly written any parameter. The result is method reference, not return. There are 4 formats:
  + Static method:

*Consumer<List<Integer>> method Ref1 = Collections::sort;*

* + Instance methods of an instance

*str = “abc”;*

*Predicate<String> method Ref 2 = str::startsWith;*

* + Instance methods of an instance to be determined at runtime. Similar to static method reference but compiler can figure out whether the method is static or instance

*Predicate<String> methodRef3 = String::isEmpty;*

*BiPredicate<String, Integer> = String::startsWith;*

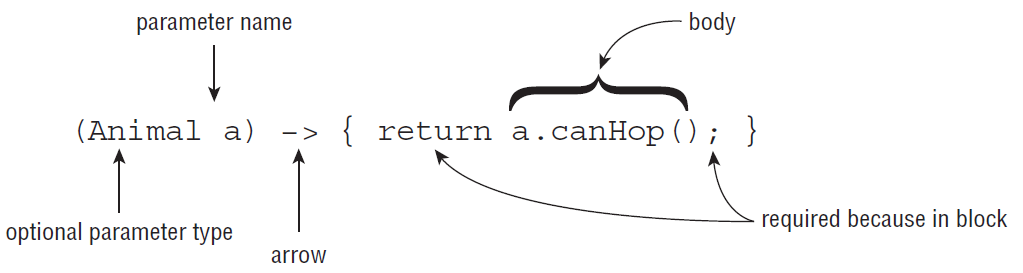
* + Constructors

*Supplier<ArrayList> methodRef4 = ArrayList::new;*

* Method reference can only replace lambda expression IF and ONLY IF the method in the lambda body take exactly same amount and type of parameter list of lambda

# Functional programming

## Lambdas structure



* Parameters lists: identifier(name) of param follow same rule as local variable. Don’t match name of other declared local variable. For static variable and instance variable, can access and shadow. For local variable, can only access final or effectively final (Damn same rule to local inner class)
  + 0 parameter: empty braces are required
  + 1 parameter: braces and type are optional. If list type of param then braces are required here
  + >1: braces are required. Either list out type of each param or list out none. The order and type of the param list here should match the method or else error
* Arrow to separate param list and lambda body
* Lambda body: can access instance var, static var, and effectively final local method param and local var.
  + void: *{}* are required. *return;* statement is optional. If there’s a single-line statement, {} and semicolon are not required
  + non-void:
    - Function body is single-line: if that’s return statement: braces, return keyword and semicolon can be removed for shortcut.
    - Multi-line function body: braces, semicolon are all required where it’s needed

## Built-in Functional Interface

* For those functional interface that takes 2 params, 2 param can be different or same type unless the generic identifiers the same

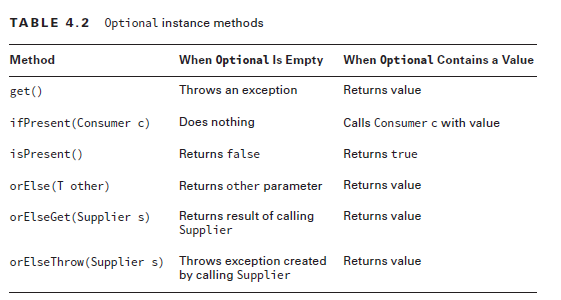
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Return** | **Functional Interface** | **Method** | **No. Param** | **Note** |
| T | Supplier<T> | get |  | Generate an object of type T out of thin air |
| void | Consumer<T> | accept | T t | Do something with the input param and return nothing |
| void | BiConsumer<T, U> | accept | T u, U u |
| boolean | Predicate<T> | test | T t | Usually for filtering. |
| boolean | BiPredicate<T, U> | test | T t, U u |
| R | Function<T, R> | apply | T t | Input param(s) will be transform in to the last type |
| R | BiFunction<T, U, R> | apply | T t, U u |
| T | UnaryOperator<T> | apply | T t | Extend from Function.  Take param(s) of same type and return same type |
| T | BinaryOperator<T> | apply | T t T t |
| Runnable and Callable are mentioned in Concurrency | | | | |

## Returning an Optional

* Optional is a container that may contain null or a value
  + Empty Optional will allow the code to run while Optional that holds null will not be able to run due to *NullPointerException* exception

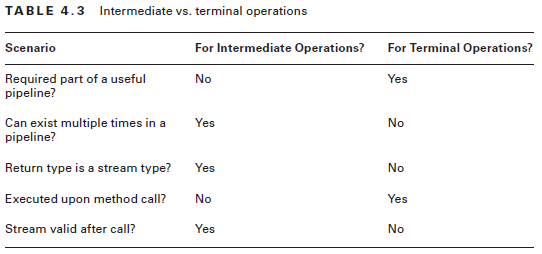
|  |  |  |
| --- | --- | --- |
| **Return** | **Static method** | **Note** |
| Optional<T> | empty() | empty Optional holding no value |
| Optional<T> | of(T value) | If value is null then get *NullPointerException* exception |
| Optional<T> | ofNullable(T value) | If value is null, then return an empty Optional |

* get() throw Runtime NoSuchElementException
  + Those “or” method will do something if empty, else just return value as normal



## Stream

* Stream is similar to an assembly line in factory
* Stream does not implement *Iterable* interface -> can’t use for loop with it
* Includes 3 part: source → Intermediate Operation → Terminal Operation
  + If intermediate operation on Stream occurs, you should assign the return Stream to a variable for further use. Once a Stream is “operate”, it can no longer be used again.
  + Terminal Operation does not return Stream, that’s where Stream ends.
* Stream intermediate operation only evaluate(run) when terminal operation is invoked.
* Intermediate operation return new Stream, not the old one, therefore:
  + Throw runtime exception if a stream is used by intermediate operation without reassigned to a variable OR Stream reach terminal operation



### Stream source

* Multiple way to create a stream source. Outputting type Stream (duh!)
* Stream<Integer> myStream =

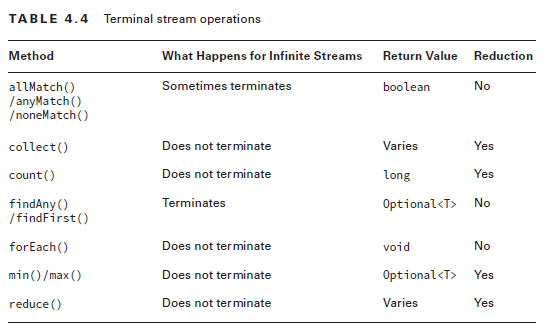
|  |  |
| --- | --- |
| **How** | **Example** |
| Empty stream | myStream = Stream.empty() |
| Stream with element | myStream = Stream.of(T … values) |
| Stream created from a collection implementation class | List<Integer> list = new Arrays.asList(“a”);  myStream = list.stream() |
| Infinite Stream  generate(Supplier<? super T> supplier) | Stream.generate( ()->”Hello” ); |
| Infinite Stream  iterate(T seed, UnaryOperator<T> f) | Stream.iterate(1, x->x+1); |

### Intermediate operations

|  |  |  |  |
| --- | --- | --- | --- |
| **Method**  **Generic** | **Return** | **Method** | **Note** |
|  | Stream<T> | filter(Predicate<? super T> predicate) | Keep element if true |
|  | Stream<T> | distinct() | Remove duplicate |
|  | Stream<T> | limit(int maxSize) |  |
|  | Stream<T> | skip(int n) |  |
| <R> | Stream<R> | map(Function<? super T, ? extends R> mapper | Transform element of Stream to other type |
| <R> | Stream<R> | flatMap(Function<? super T,  ? extends Stream<? extends R>> mapper) | Turn each element of stream into new stream, then return those streams so flatMap can combine into a single Stream |
|  | Stream<T> | sorted() |  |
|  | Stream<T> | sorted(Comparator <? super T> comparator) |  |
|  | Stream<T> | peek(Consumer<? super T> action) |  |

### Terminal operation

* T is type param from class declaration



|  |  |  |  |
| --- | --- | --- | --- |
| **Method generic** | **Return** | **Method** | **Note** |
|  | long | count() |  |
|  | Optional<T> | min(<? super T> comparator) |  |
|  | Optional<T> | max(<? super T> comparator) |  |
|  | Optional<T> | findAny() |  |
|  | Optional<T> | findFirst() |  |
|  | boolean | anyMatch(Predicate <? super T> predicate)  allMatch(Predicate <? super T> predicate)  noneMatch(Predicate <? super T> predicate) |  |
|  | void | forEach(Consumer<? super T> action) | Do something with each element. Can’t use for loop with Stream |
|  | Optional<T> | reduce(BinaryOperator<T> accumulator) | Combine all element in Stream according to BinaryOperator.  Return empty Optional if Stream is empty |
|  | T | reduce(T identity,  BinaryOperator accumulator) | Combine identify+ all element in Stream according to BinaryOperator. |
| <U> | U | reduce(U identity,  BiFunction<U,? super T,U> accumulator,  BinaryOperator<U> combiner) | Combine identify all element in Stream according to BiFunction.  Then combine with parallel Stream |
| <R> | R | collect(Supplier<R> supplier,  BiConsumer<R, ? super T> accumulator,  BiConsumer<R, R> combiner) | Mutable reduce where a mutable object is created and each element will be passed in to that object through method |
| <R,A> | R | collect(Collector<? super T, A,R> collector) |

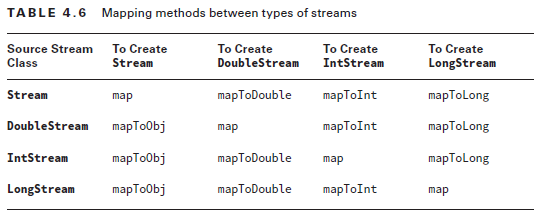
## Working with primitives

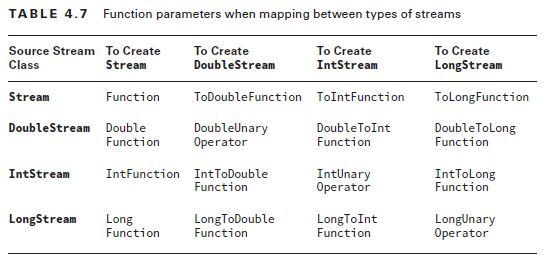
### Primitive Stream

* Don’t have generic here, obviously
* 3 Primitive Stream:
  + IntStream: for int, short, byte, char
  + LongStream
  + DoubleStream
* Since it’s primitive, the numeric literal can be implicitly upcast or explicitly down-cast following rules in OCA
* Source: similar to Stream in above plus:

|  |  |
| --- | --- |
| **How** | **Example** |
| Same method as Stream with generic replaced with appropriate the primitive type, built-in functional interface replaced with specific functional interface. | |
| Inclusive range | *IntStream iStream = IntStream.range(1,6);* |
| Exclusive range | *IntStream iStream = IntStream.rangeClosed(1,5);* |

* Intermediate operation method:

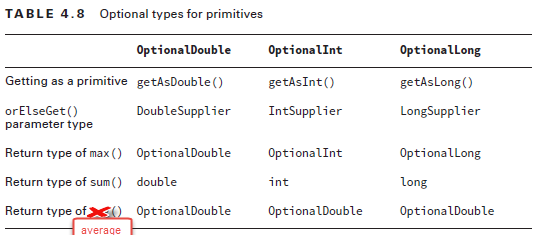




* 2 table with method map and it’s parameter
* flatMap is similar in the method name.
* Terminal operation:
  + Method of Stream plus a method *summaryStatistics* that return *PppSummaryStatistic* object that contains min, max, sum, average of the primitive Stream

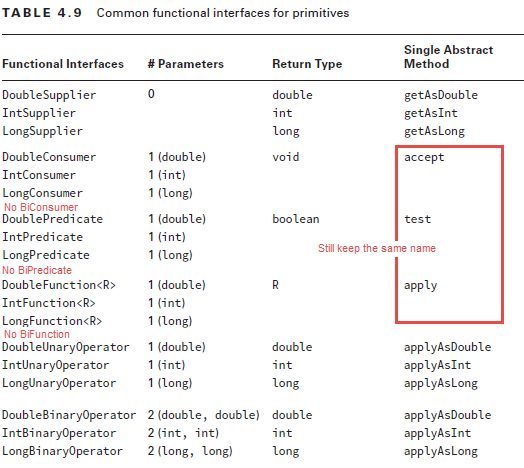
### Primitive Optional

* Optional types of primitives are Optional subclasses that can hold primitive:
  + As expected from option. *orElse\_\_* method do something else if Optional is empty.
  + Since sum return primitive, primitive Stream return 0 for empty one
  + Throw exception with orElseThrow when the Optional return from method below has no value in it.

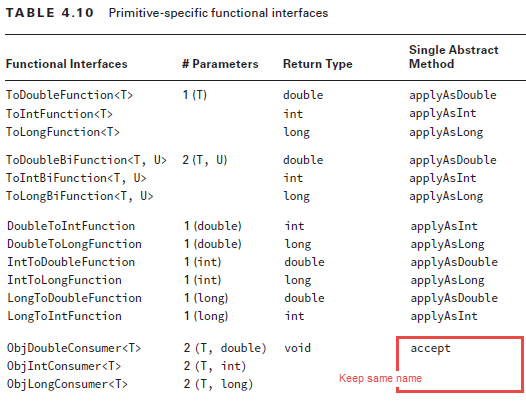


### Primitive Functional Interface

* Interface name:
  + Add Double, Long and Int prefix
  + There’s no *PppComsumer*, *PppPredicate* and *PppBiFunction*
* Parameter number and type:
  + Same number of param
  + Type become specific primitive
* Return type:
  + void and boolean keep the same
  + PppFunction return another type
* Abstract method name:
  + If return type is primitive, add postfix *AsPpp* to the method name

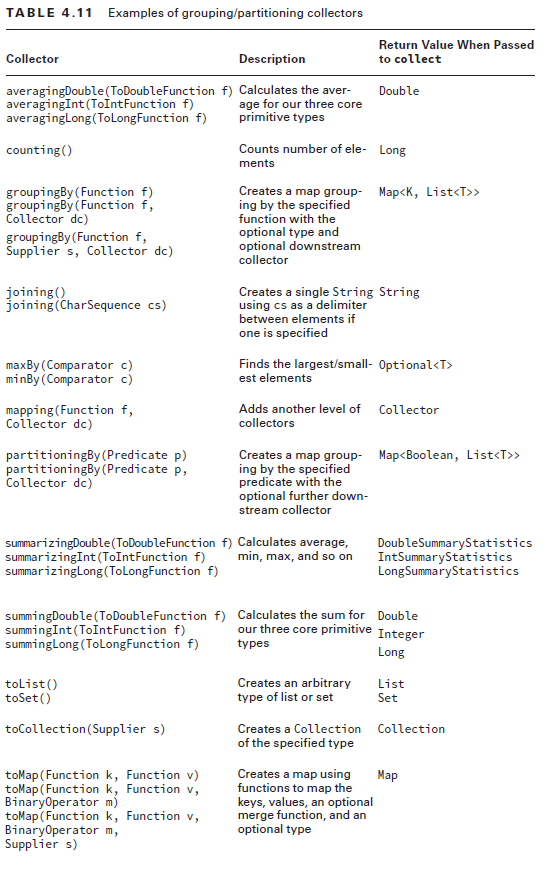


* Functional interface name: 2 type
  + *\_1\_To\_2\_Function*: (1) can be either *Double*, *Int*, *Long*, or empty; (2) contains either *Double*, *Int*, or *Long*
    - If (1) is empty, get object type param & return (2) primitive
    - If (1) is not empty, get primitive param & return (2) primitive
    - Since return are all primitive, add postfix *AsPpp* to the method name
  + *Obj\_1\_Consumer*:
    - (1) can be *Double*, *Int* or *Long* , get an object type param and an primitive param. Return void like Consumer do.



## Advanced Stream Pipeline concept

* Reminder: Create List from array will linked both
* Create Stream from List will linked both. Up until a terminal operation, the stream will include all the element get added to the list even the element is added after any intermediate operation statement.
* Following methods in Optional class are similar to Stream intermediate operation methods, except that they return Optional instead of Stream object:
  + map
  + flatMap
  + filter
* Collector static methods:
  + *joining*: required element to become String



# Dates, String and Localization

## Date and Times

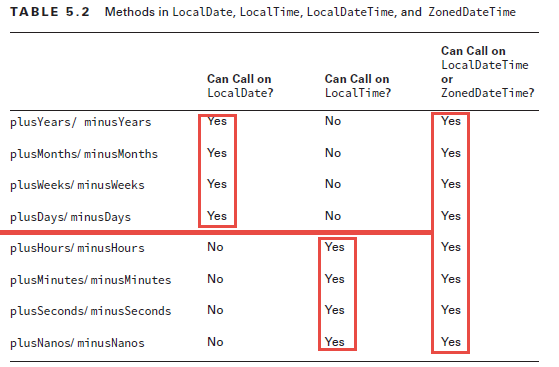
* Includes: (all below are immutable, thread-safe and use factory’s static method to instantiate)
  + LocalTime
  + LocalDate
  + LocalDateTime
  + ZonedDateTime
  + Duration
  + Period
  + Instant
* Instantiate above objects by factory’s static method, don’t use *new* keyword. Therefore, if we chain these factory’s static method, it will just create an object from the rightmost of\_\_ method.
* Calculate date time with timezone: by converting to time zone zero of the timezone standard (UTC or GMT), get the time and subtract it by time zone number (+2:00, GMT+2 or UTC+2)
  + 2015–06–20T07:50 GMT-04:00 -> GMT 2015–06–20T (07:50-(-4:00))

### Date and time instantiation with simple manipulation

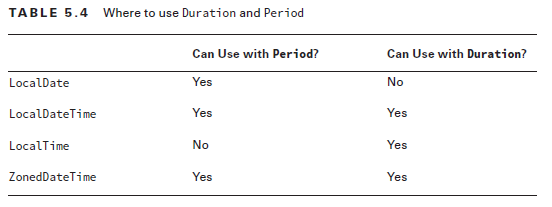
* *LocalDate*: all are required, month can be either enum Month or number (1-based as human)
* *LocalTime*: hour and minute are required. Second and nanos are optional
* *LocalDateTime*: rules from LocalDate and Time all applied here. Plus can just pass both LocalDate and LocalTime as params
* *ZonedDateTime*: all detailed param, *LocalDate*+*LocalTime*+*ZoneId*, OR *LocalDateTime*+*ZoneId*
* There’s no overloaded version of *of* in LocalDateTime and ZonedDateTime that passed in either LocalDate or LocalTime and then the rest of detailed information.
* When invalid information (e.g: 35 as dayOfMonth), java throws Runtime exception

|  |  |
| --- | --- |
|  | *of* is public static |
| LocalDate | of(int year, int month, int dayOfMonth)  of(int year, Month month, int dayOfMonth) |
| LocalTime | of(int hour, int minute)  of(int hour, int minute, int second)  of(int hour, int minute, int second, int nanos) |
| LocalDateTime | of(int year, int month, int dayOfMonth, int hour, int minute)  of(int year, int month, int dayOfMonth, int hour, int minute, int second)  of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanos)  of(int year, Month month, int dayOfMonth, int hour, int minute)  of(int year, Month month, int dayOfMonth, int hour, int minute, int second)  of(int year, Month month, int dayOfMonth, int hour, int minute, int second, int nanos)  of(LocalDate date, LocalTime time) |
| ZonedDateTime | of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanos, ZoneId zone)  of(LocalDate date, LocalTime time, ZoneId zone)  of(LocalDateTime dateTime, ZoneId zone) |

* Manipulate with common methods:



### Date and time manipulation with Period and Duration



* Throws runtime exception for using Duration or Period to manipulate in wrong object type
* Instantiate Period and Duration:
  + Duration:
    - only year, month and day are stored, so week is converted to days
    - There’s *of* method that combine those stored unit
  + Period:
    - only hour, minute and second are stored, so day is converted to hour, millis and nanos converted to fraction of second
    - No *of\_\_* method that combine those different time unit

|  |  |  |
| --- | --- | --- |
| **Factory’s static method** | **println Output:** | **Note** |
| Period | | |
| ofYears(1) | P1Y |  |
| ofMonths(3) | P3M |  |
| ofWeeks(3) | P21D | Converted to days |
| ofDays(2) | P2D |  |
| of(1, 0, 7) | P1Y7D | zero part is left out |
| Duration | | |
| ofDays(1) | PT24H | Converted to hours |
| ofHours(1) | PT1H |  |
| ofMinutes(1) | PT1M |  |
| ofSeconds(10) | PT10S |  |
| ofMillis(1) | PT0.001S |  |
| ofNanos(1) | PT0.000000001S |  |
| of(1, ChronoUnit.DAYS) | PT12H | a *long* amount with *ChronoUnit* enum:   * CENTURIES * DECADES * YEARS * WEEKS * MONTHS * DAYS * HALF\_DAYS * HOURS * MINUTES * SECONDS * MILLIES * MICROS * NANOS |

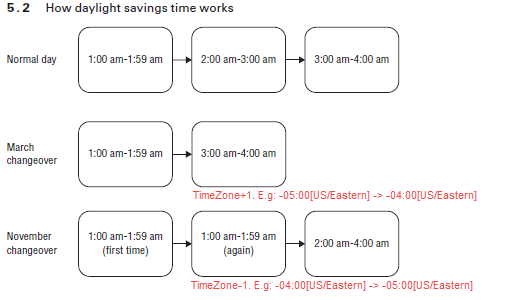
* Manipulate:
  + Used *plus* and *minus* method
  + Will wrap around or move to next bigger unit if exceed the limit of unit
  + Throws runtime exception for using Duration or Period to manipulate in wrong object type

### Instant

* Instant class represent time in GMT timezone. Differ from ZonedDateTime in that it calculate into GMT zero zone
* For instant, you can only manipulate with equivalent of no more than 1 day (<=1 days equivalent time unit)
* Instantiate by method:

|  |  |
| --- | --- |
|  | println Output |
| Instant now = Instant.now() | 2015–05–25T15:55:00Z |
| Instant instant = Instant.ofEpochSecond(epochSeconds) | 2015–05–25T15:55:00Z |
| LocalDate date = LocalDate.of(2015, 5, 25);  LocalTime time = LocalTime.of(11, 55, 00);  ZoneId zone = ZoneId.of("US/Eastern");  ZonedDateTime zonedDateTime = ZonedDateTime.of(date, time, zone);  Instant instant = zonedDateTime.toInstant(); | 2015–05–25T15:55:00Z |

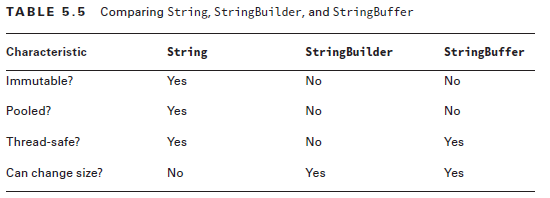
### Daylight Saving



* On March changeover, Java can figure out that between 02:00 and 02:59 does not exist and therefore automatically switch to the right GMT hour

## String

* String is immutable, final
* String concat via + and concat
* String pool
* String common methods
  + substring: if start > end index, runtime StringIndexOutOfBoundException. if start == end index, empty String.



## Internationalization and localization

* i18n and l11n
* Locale: en\_US
  + language is lowercase and required
  + underscore and country are optional, country is uppercase
* Locale instantiate:

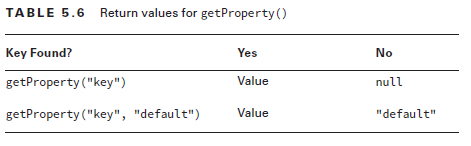
|  |  |
| --- | --- |
|  | **Note** |
| Locale.getDefault() | get computer’s locale info |
| new Locale.Builder()  .setLanguage(“en”)  .setRegion(“US”)  .build() | Use builder class  Mess up with case of String is ok here |
| new Locale(“fr”) | Constructor  Mess up with case of String is ok here |
| new Locale(“hi”, “IN”) |
| Locale.GERMAN | static field. Only language |
| Locale.GERMANY | static field. Both language and country |

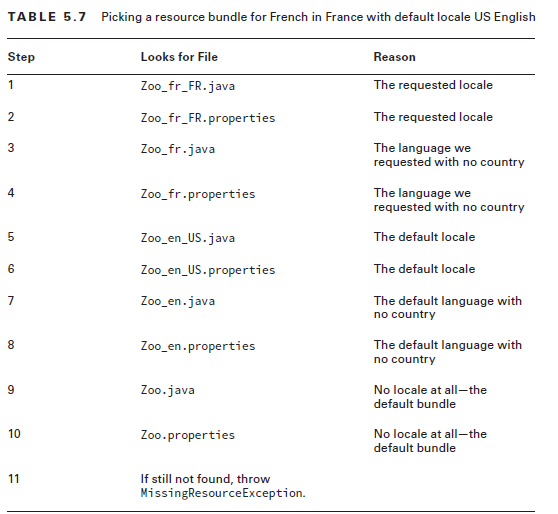
* Locale.setDefault(myLocale) to set the Locale of the program (not computer)

### ResourceBundle and Java Class Resource Bundle

* ResourceBundle instance methods:
  + getBundle(baseName) : use default locale
  + getBundle(baseName, myLocale)
  + getString(“key here”) : will not work if value is not String object
  + getObject(“key here”)
* Resource Bundle file which contain all translated string of a locale:
  + baseName\_en.properties
  + 3 ways to represents key-value:
    - hello = xin chao
    - hello:xinchao
    - hello xinchao
  + Comment: begins line with ! or #
  + Spaces: around key-value separator and beginning of line are ignored. Space at the end of line is counted
  + \ for break line as well as be able to use \t \n
* Resource Bundle class that extends ListResourceBundle:
  + Class name: baseName\_en\_US
  + Override this method from ListResourceBundle:
    - *protected abstract Object[][] getContents()*
  + One special note is getContents will only be called once, therefore, any value it created will be used over and over again through reference. Also, value part of the Object can access to instance variable
* Properties: similar to Map
  + Properties props = new Properties();

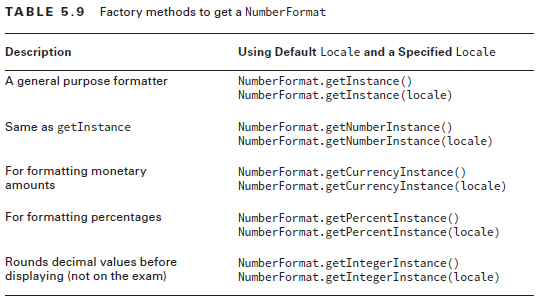
rb.keySet().stream().forEach(k -> props.put(k, rb.getString(k)));





* The key can com from an the matching bundle resource up until its parents

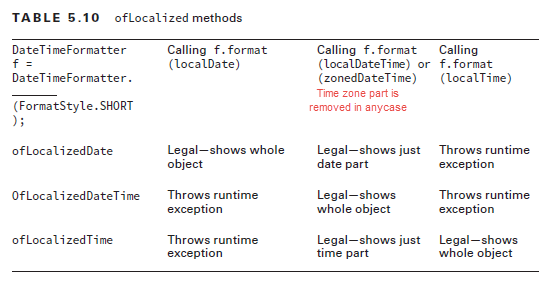
### Formatting Number



* Instance method *format* to format number to specific number format
* Instance method *parse* turn *String* into number, with specific character depends on locale
  + If see parse in method, remember to check for checked exception handling or declaring in method. If in snippet, it’s ok
  + If the String start with number or negative sign, it will parse until the character that can’t be parsed(letter, 2nd decimal separator in that locale, thousand separators in decimal part) (thousand separators in that locale is acceptable anywhere in the whole number part)
  + If the String start with character that can’t be parsed, throw *checked ParseException*

### Formatting Date and Time

* Instantiate and legal usage of DateTimeFormatter:
  + *ofPattern(“MMMM dd, yyyy, hh:mm”)*
  + *DateTimeFormatter* will format and output only parts that it can. If it can’t find, runtime *UnsupportedTemporalTypeException*

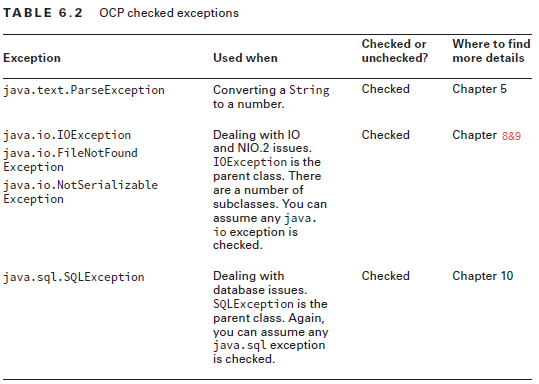


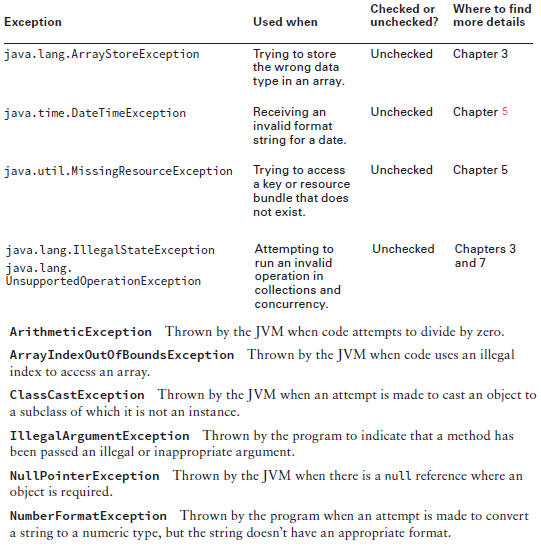
* *format* instance method is implemented in all date, time classes and DateTimeFormatter too

# Exception

## Exception review

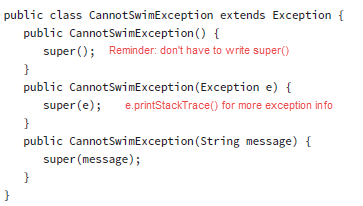
* Exception OCA
* Illegal for compiling:
  + Multiple *catch* phrase where a subclass of exception is sorted after its superclass
  + Checked exception in *catch* block MUST be potentially to happen in *try* phrase.
* throw vs throws





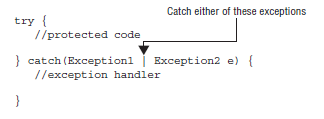
## Custom exception

* There are 3 constructors in *Exception* (checked) class:



## Multi-catch

* This is not meant by multiple catch phrase, but a single catch phrase that catch multiple kind of exception



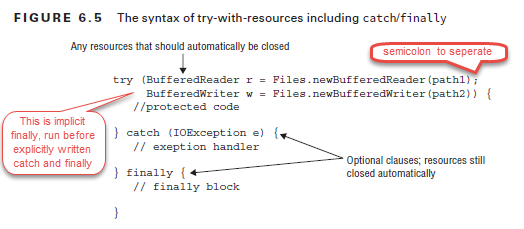
* Regarding assigning variable from catch phrase:
  + Normal catch phrase: can
  + Multi-catch phrase: can not due to type is not specified
* Similar rule of unreachable exception caught applied if mixed normal and multi-catch phrase:
  + Can’t catch checked exception that never happen in *try* phrase
  + Sub and super exception can’t be in same multi-catch phrase since 1 is never reachable
  + Multiple *catch* phrase where a subclass of exception is sorted after its superclass

## Try-With-Resources

* Only resources class that implements *AutoCloseable* interface can be used in try-with-resources:
  + Override this method from *AutoCloseable* interface:

*public void close() throws Exception;*

* + Resource declare in try-with-resource are implicitly final
  + Good practice but not required when overriding method *close* in subclass (this is called idempotent):
    - More specific than *Exception*
    - Don't change state (variable)
    - Undesirable behaviors
  + Old class similar to AutoCloseable is Closeable, which differ by:
    - Same method signature to implement except *Closeable*’s *throws IOException*
    - Required to be Idempotent
* Close all resources opened with implicit finally phrase before other phrase
  + Therefore, it’s legal to have no written catch and finally phrase as it has implicit final phrase



* Rules for exception that are throw in try-with-resources:
  + Order of exceptions might be throw:
    - First exception within try phrase → Last resource close() →

Next to last resource close() → … → First resource close()

* + - The only the first exception, called main, threw in the above order will be used to match with catch phrase, the rest will be suppressed to the main one.
    - use *getSuppressed()* method on the main exception to get array *Throwable[]* containing all suppressed exception

## Rethrowing Resource

* For catch phrase, we can use a superclass exception to catch itself and all its subclass. There is no need to write all specific checked exception to catch phrases, can use a superclass to cover multi case.
* Applying this same rule for rethrow exception in a method that declares multiple exception:
  + Most common option: *catch(Exception e)* as this cover all exception

## Assertions

* 2 forms of statement: wrong syntax here will make compile error even assertion is disabled. Parenthesis around boolean expression is optional.
  + *assert boolean\_expression;*
    - *assert (boolean\_expression);*
  + *assert boolean\_expression: error\_message;* Anything but void expression after evaluation
* Enable assertions: (assertions disabled by default)
  + *java -enableassertions com.program.ClassName*
  + *java -ea com.program.ClassName*
  + *java -ea:com.wiley.demos... com.program.ClassName*
  + *java -enableassertions :com.wiley.demos.TestColors com.program.ClassName*
  + *java -ea:com.wiley.demos... -da:com.wiley.demos.TestColors com.program.ClassName*
  + *java -ea:com.wiley.demos... -disableassertions:com.wiley.demos.TestColors com.program.ClassName*
* Not good practice for assert statement to alter state(variable), checking for input param of public method, but doable in exam.
* Good practice: check input param of private method or certain facts

# Concurrency (java.util.concurrent package)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Return** | **Functional Interface** | **Method** | **No. Param** | **Note** |
| void | Runnable | Run | 0 |  |
| V | Callable<V> | Call | 0 | method *throws Exception*  This is the first functional interface with method that allow checked exception so far. |

## Threads

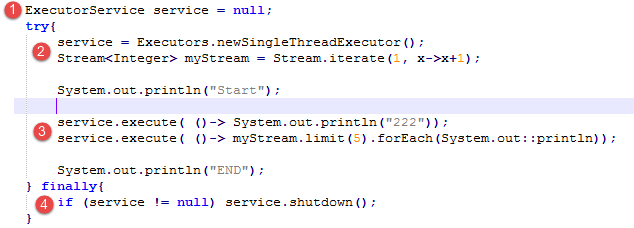
* Task → thread → process
* Thread types:
  + System thread (e.g: GC thread): throw Error if fail
  + User-defined thread

|  |  |  |
| --- | --- | --- |
| **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 myThread()) | new myThread() |
| Do tasks on seperate thread | (new Thread(new myThread())).start() | (new myThread()).start() |
| Do tasks on main thread | (new Thread(new myThread())).run() | (new myThread()).run() |

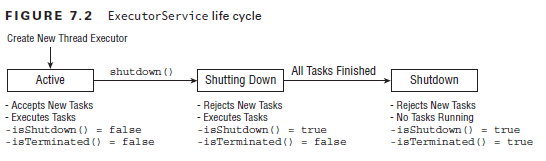
* Since Runnable interface is Functional interface, can used constructor *Thread(Runnable object)* and input lambda as param.
* start vs. run
  + With using run, the task in thread object will be executed on main thread and have order.
  + With using start, the task in thread object will be executed on seperate threads and have no order to guarantee
* *Thread.sleep(int millis)* static method to wait in the thread it is in. Need to declare or handle checked *InterruptedException*

*public static void sleep(long millis) throws InterruptedException*

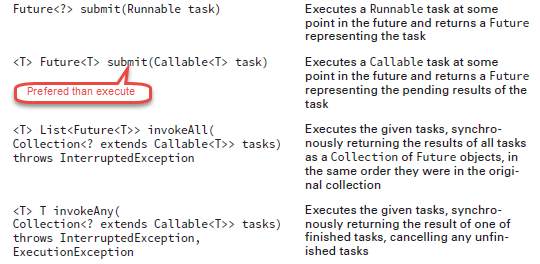
## Creating Threads with ExecutorService



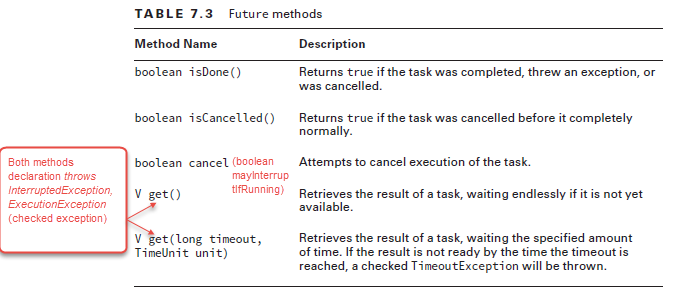
* (1) + (2): Create Single Thread Executor:
  + Since *ExecutorService* is interface, use *Executors* factory to create implementation class object
  + Since this has a single thread, the order tasks that the *SingleThreadExecutor* is done synchronously (one by one in order they are assigned)
* (4): Service have to be shutted down when no longer used. When it get notified to shut down, the thread executor reject new task with runtime *RejectedExecutionException* and try to finish undone tasks. If not shutdown then the code will hang there. 2 methods:
  + *service.shutdown()*
  + *service.shutdownNow()*: attempt(not guarantee) to close even undone tasks



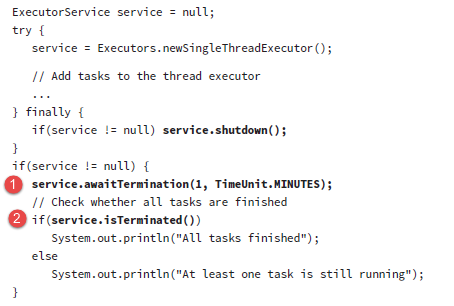
* (3): there’s multiple ways to submit(~assign task) to Thread Executor:
  + *execute(Runnable r)*: void return
  + *submit(Runnable task)*: if use get method on *Future* return, it will be *null* since *Runnable* return type is void
  + *invokeAll* : wait indefinitely until all task are completed successfully or throw exception
  + *invokeAny*: wait indefinitely until 1 task are completed successfully and return



* Future class:
  + TimeUnit enum:
    - TimeUnit.NANOSECONDS
    - TimeUnit.MICROSECONDS
    - TimeUnit.MILLISECONDS
    - TimeUnit.SECONDS
    - TimeUnit.MINUTES
    - TimeUnit.HOURS
    - TimeUnit.DAYS

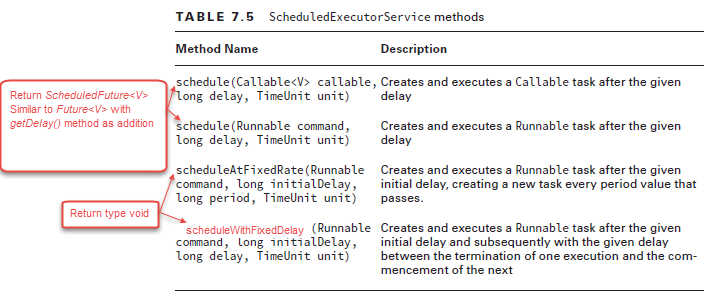


* There’s 2 way to wait for task to finish:
  + Need result from task way: use *get* method of return *Future* object
  + Don’t need result way: after try-finally to shutdown thread executor service:

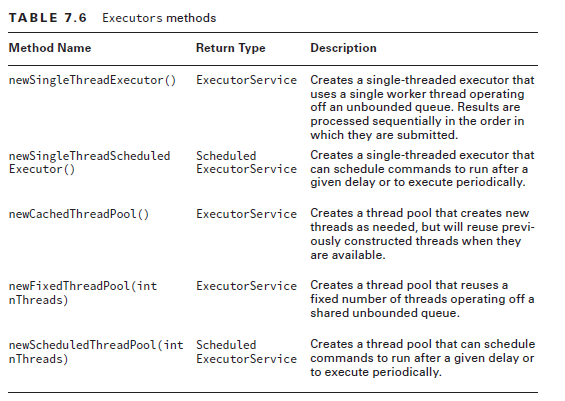


* Scheduling tasks:
  + Use ScheduledExecutorService which is subinterface of ExecutorService
  + These methods don’t throw any checked exception
  + Use *Executors* factory implementation to create object

ScheduledExecutorService service = Executors.newSingleThreadScheduledExecutor();



* The last 3 Executor with ThreadPool
  + Has the same methods as its single Thread counterpart.
  + Each tasks will be assigned to a thread until the thread pool is run out (except in case of CachedThreadPool as it creates new Thread as needed)

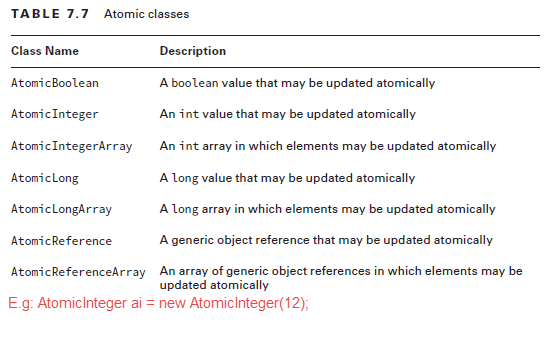


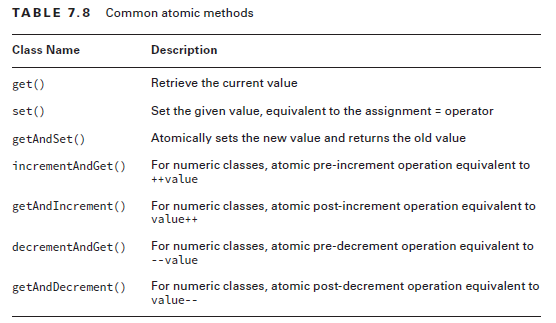
## Synchronizing data access

* Race condition, not-safe thread execution
* There’s 2 ways to avoid this race condition in multi-thread execution

### Atomic classes

* Atomic classes: container object that hold primitive or other object in it (similar to Optional in Stream API or Future in thread execution return)
  + Avoid race condition by combine read and write on the variable into a single operation
  + However, the order is not guarantee, unlike lock(or monitor).





Also, have method *addAndGet(int i)* and *getAndAdd(int i)*

Also, have method *compareAndSet(expectedValue, newValue)*

### Synchronized keyword

* Use an object as a monitor(lock) and given to the first thread that come to it while the late arrived thread have to wait until the lock(monitor) is released
* Synchronized keyword can be used as:
  + *synchronized* code block: can use *this* keyword to use itself as monitor in instance method

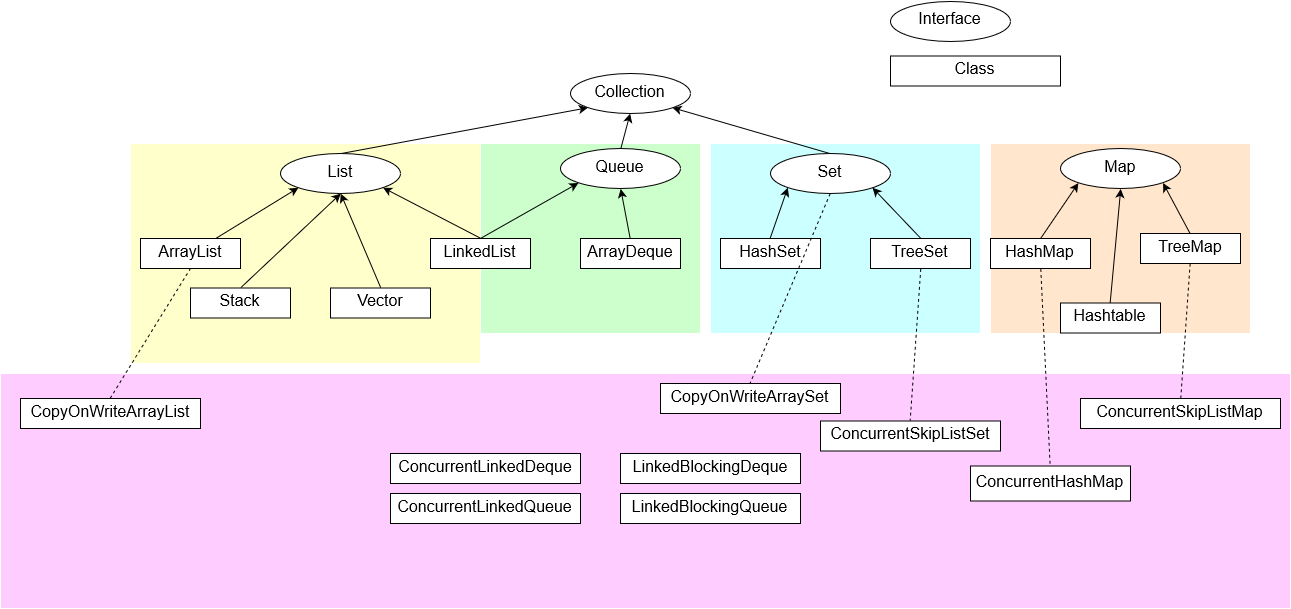
*synchronized(lockObjectt){*

*//code statements here*

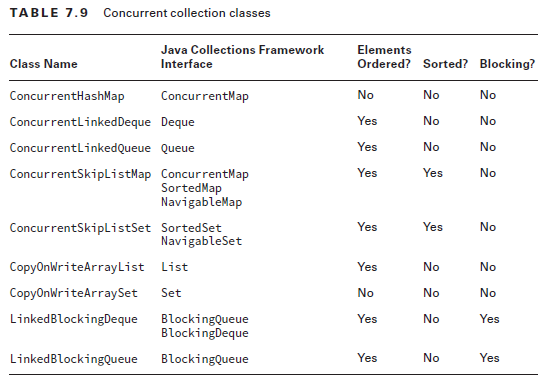
*}*

* + *synchronized* methods: worked on both *static* non-abstract and non-abstract non-static(instance) methods

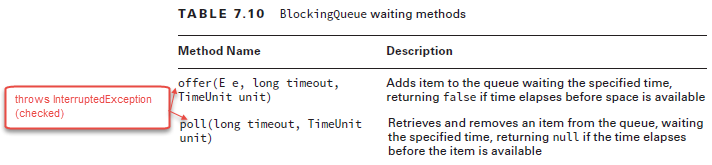
## Concurrent Collections

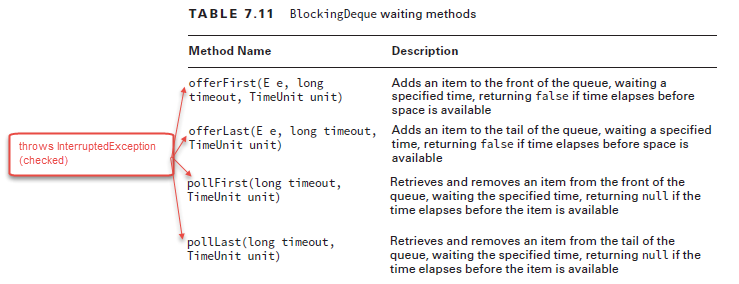


* Memory consistency error
* While iterating through collections, *ConcurrentModificationException* might be threw if we modify the collections itself. → concurrent collections to solve issues when adding and remove element of collections while iterating it

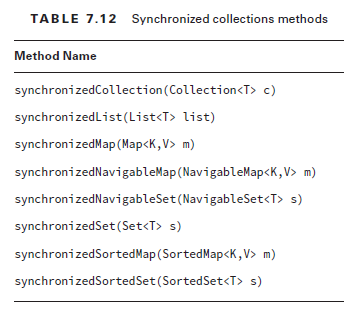


* *ConcurrentHashMap*, *ConcurrentLinkedDeque* and *ConcurrentLinkedQueue*: similar methods signature to their non-concurrent counterpart
* Blocking queues and block deque (double-ended-queue): provide methods that wait for operation completion





* *ConcurrentSkipListSet* and *ConcurrentSkipListMap* : counterpart of *TreeSet* and *TreeMap*
* *CopyOnWriteArrayList* and *CopyOnWriteArraySet* : copy all data to an underlying structure when element is added, modified or removed. Used on iteration since it guarantee that the iterator can see 1 consistent data there.
* Synchronized collections: only synchronized access to data (get and set), not access on iterator. These are *static synchronized* method *Collections* that return a reference that match the underlying structure



## Parallel Streams

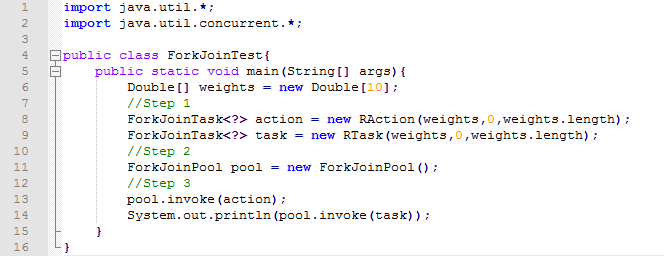
* 2 ways to create:
  + *parallel()* intermediate operation: serial Stream → parallel Stream
  + *parallelStream()* method of List: List → parallel Stream
* Avoid stateful expression in lambda of stream operation, which means avoid modifying data (state) of other object
* flatMap() always produce serial Stream
* Stream operations based on order will output the same result in both serial and parallel Stream. Including:
  + *findFirst()*
  + *limit()*
  + *skip()*
* To ensure *reduce()* terminal operation has the same result in serial and parallel stream:
  + The identity must be defined such that for all elements in the stream u , *combiner.apply(identity, u)* is equal to u . (“” in case of String, 0 in case of Integer)
  + The accumulator operator op must be associative (change order of operands with affecting the result) and stateless such that (a op b) op c is equal to a op (b op c) . (String concat or + operator for now)
  + The combiner operator must also be associative and stateless and compatible with the identity, such that for all u and combiner.apply(u,accumulator.apply(identity,t)) is equal to accumulator.apply(u,t). Changing the order of operands of combiner probably reverse the result
* To ensure *collect()* terminal operation has the same result in serial and parallel stream:
  + 3-args version: same as in reduce(), use concurrent collection as supplier to avoid *ConcurrentModificationException*
  + 1-argument (take Collector as argument): bottomline: only *Collectors.toConcurrentMap()* and *Collectors.groupingByConcurrent()* which take argument and result the same as their non-concurrent counterparts

## Manage Concurrent processes

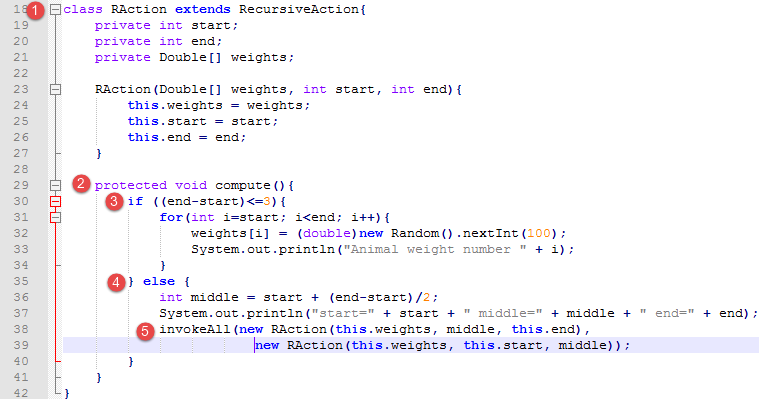
* CyclicBarrier:
  + Constructors:
    - *CyclicBarrier(int limit)*
    - *CyclicBarrier(int limit, Runnable barrierAction)*
  + use *await() throws InterruptedException, BrokenBarrierException* instance method to wait until pending thread reach the limit and then release, resetting the awaiting back to zero
  + If limit > than thread pool size, hang forever because the barrier would never reach its limit to break
* Fork/Join Framework steps:
  + Create ForkJoinTask : Write a class that implement either abstract class *RecursiveAction* or *RecursiveTask* (both are implemented ForkJoinTask interfaces)
  + Create ForkJoinPool
  + Start ForkJoinTask

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Return** | **Abstract class implemented ForkJoinTask** | **Method** | **No. Param** | **Note** |
| void | RecursiveAction | compute | 0 | *protected* access  Override with a recursive method that include a base case and a recursive case that will eventually broken down to base case |
| V | RecursiveTask<V> | compute | 0 |

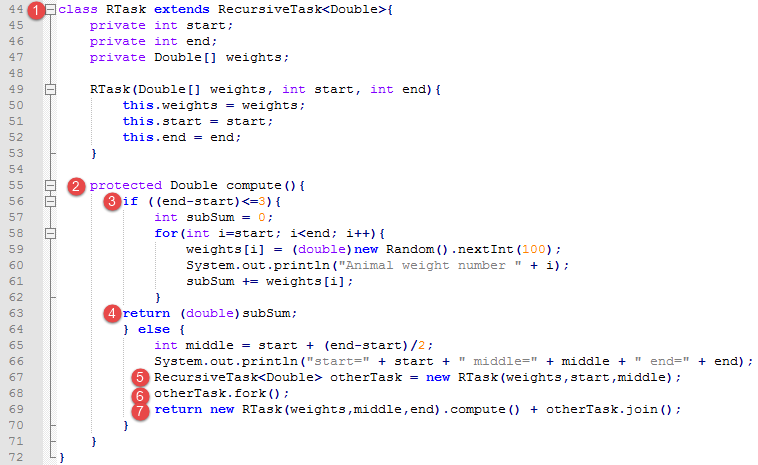
* Steps in Fork/Join framework
  + pool.invoke method has return type



* RecursiveAction extend
  + (1): extends class
  + (2): override method
  + (3): base case, which the recursive case will reach eventually
  + (4): recursive case, calling the method itself
  + (5): use inherited *invokeAll* method to do recursive task



* RecursiveTask extend
  + (1) define generic type of RecursiveTask
  + (2) method now has return type same as generic and return value in both base and recursive case
  + (5) + (6) recursive call 1 part and work out on seperate thread
  + (7) recursive call the other part and join result with (5) + (6)



## Threading problems

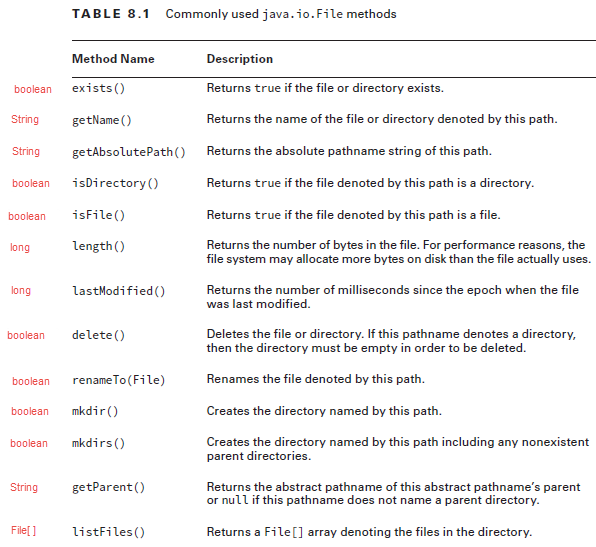
* Deadlock: hang, happens when threads try to acquire lock that are not released from other
* Starvation: thread wait indefinitely for the lock to release
* Livestock: actively running in infinite circle
* Race conditions

# IO (java.io package)

From now on, a **file** refer to file record (includes regular file that can have content or directory.)

## Files and directories (java.io.File class)

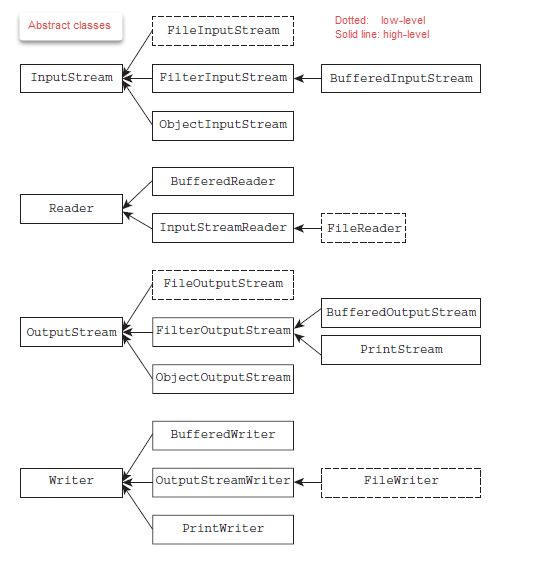
* *java.io.File* class:
  + Read information about files and directories
  + List contents of directory, which are files that it contains
  + Create/delete files and directories
  + Regular file can have no extension, therefore, it can look like directory
  + Either double backslash \\ or forward slash / to separate path
* Instantiate with constructors:
  + *File(String path)* : can be absolute path or relative path
  + *File(File parent, String path)*: if parent is null, then it is skipped
* Common methods of *File* class:
  + All of these method does not *throws IOException*. Therefore, appropriate values (e.g: false, null) are returned if the methods does not work out



## Data Streams (not Stream API)

|  |  |  |
| --- | --- | --- |
| **Keyword in class** | **Meaning** | **Classification** |
| *InputStream*  *OutputStream* | byte or binary Input data type  byte or binary Output data type | Input/Output data type |
| *Reader*  *Writers* | character or string Input data type  character or string Output data type |
| *Buffered* | Read date (byte or char) in group |  |





* High-level io class can wrap around low-level and high level (even the same high-level class object)
* Common operations:
  + *close()*: can be used in finally block or with try-with-resource
  + *flush()*: for OutPutStream, write data from memory to files immediately
  + *mark(int i)* and *reset()*: for *InputStream* and *Reader*, use *markSupported()* to check → mark with limit of data storage, which store the amount of data stream read into memory → reset
  + skip() : for *InputStream* and *Reader*, skip certain number of bytes

## Working with stream classes

* Initialize object of these class are grouped in these:
  + Low-level class (start with File): have constructors that take path String or File object
  + Enhanced version (start with Buffered) and Object serialization class: take all object of class that implement its IO abstract class.
    - BufferedWriter(Writer w)
    - ObjectOutputStream(OutputStream)
  + Print-based classes: take objects of OutputStream, objects of Writer, File object OR path String
* Remember to close the resource or use with try-with-resource for data stream

### FileInputStream & FileOutputStream | BufferedInputStream & BufferedOutputStream

1. All method in this table *throws IOException* (checked)

|  |  |  |
| --- | --- | --- |
| **Return** | **Method** | **Note** |
| Use these methods with normal version | | |
| int | read() | Either a byte or -1 for end of file |
| void | write(int) | Write byte to file (int casting to byte) |
| Use these method with buffered version | | |
| int | read(byte[ ] b) | Length of byte read (=array length)  Either a byte or -1 for end of file |
| void | write(byte[ ] b, int offset, int len) | offset usually is 0  len usually is length of array |

### FileReader & FileWriter | BufferedReader & BufferedWriter

* All method in this table *throws IOException* (checked)

|  |  |  |
| --- | --- | --- |
| **Return** | **Method** | **Note** |
| Use these methods with normal version | | |
| int | read() | Either char or -1 |
| void | write(int) | Write char to file (int casting to char) |
| boolean | ready() | check whether bytes left to read. Another way to iterate through content beside *read()* method |
| Use these method with buffered version | | |
| String | readLine() | Either String or null |
| void | write(String) | Write String into file |
| void | newLine() | Write line separator |

### Serialization (not in objective, but should know)

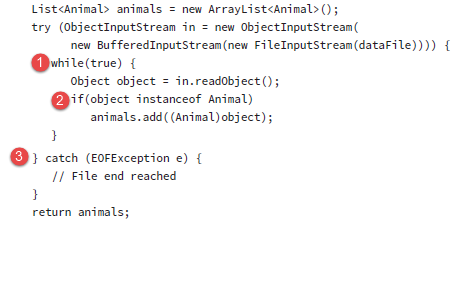
* In-memory object → stored data (in file) serialization
* Stored data (in file) → In-memory object deserialization
* Object of classes with these requirement can be serialized:
  + Primitive can be serialized
  + Implements *Serializable* interface, which has absolutely no methods in it
  + All instance variables belong to classes that are *Serializable*. This includes all the object inside object instance variable and further to be *Serializable*
* Most built-in classes from OCP are Serializable (String …)
* *transient* (modifier) instance variable are ignored in serialized an object and always get appropriate default value when serialized or deserialized
* *static* variable are ignored in serialized an object
* When deserializing, no-args constructor of the first non-serialized parent (call through super() , usually is Object class) will be called. All the constructors and initialization blocks are skipped.
* Special case: this static variable is used to marked the version of the object that are serialized
  + *private static final long serialVersionUID = 1L;*

### ObjectInputStream & ObjectOutputStream

* readObject methods in table *throws* *IOException, ClassNotFoundException*
* writeObject methods in table *throws* *IOException*

|  |  |  |
| --- | --- | --- |
| **Return** | **Method** | **Note** |
| Object | readObject() | Since return Object type, instanceof and explicitly casting is required to use the object properly |
| void | writeObject(Object o) |  |

* Serialize object and stream to file: simple
* Deserialize from file to object: complex unlike reading String or byte from files
  + (1): to loop through and read all object until reach end of file and exception is threw (3) since *readObject* return Object, not int.
  + (2): Check if object belong to class to explicitly cast, avoiding runtime *ClassCastException*
  + Constructors and default initialization that set value instance variable are all ignored.



### PrintStream & PrintWriter

* These classes write date (byte or char) to text-based output stream (console, log file) with format
* print-based method classes (This table list PrintWriter methods, but PrintStream have similar method too)
  + These method don’t throws IOException, unlike other data stream’s method

|  |  |  |
| --- | --- | --- |
| **Return** | **Method** | **Note** |
| PrintWriter | append | can take String, char or CharSequence and print to the output stream |
| void | print | Lots of overloaded version that take primitive, String and Object  Call *String.valueOf(input)* |
| void | println | Similar to print, with ending String with line break  Even has no-arg version to print line break only |
| PrintWriter | format(String format, Object args. . .) |  |
| PrintWriter | printf(String format, Object args. . .) |  |

## Interacting with users (java.io.Console)

* java.io.Console
* Singleton object
* Might return null depends on the OS
* *Console* methods

|  |  |  |
| --- | --- | --- |
| **Return** | **Method** | **Note** |
| Reader | reader() |  |
| PrintWriter | writer() |  |
| Console | format(String fmt, Object... args) | Can use PrintWriter object from writer() method or just these methods directly on the Console object |
| Console | printf(String fmt, Object... args) |
| void | flush() |  |
| String | readLine() | Get a user input line and return String |
| char[] | readPassword() | Use array instead of String because: String might be stored in String pool, and array will get deleted after it’s used |

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

From now on, a **file** refer to file record (includes regular file that can have contains or directory.)

**Summary for OCP:**

* java.nio.file.Path:
  + Instantiate: Paths.get factory class’s method OR through FileSystem
  + Methods: Mostly contain methods that modify the path → *not throw IOException* except for toRealPath() which check for file existence
* java.nio.file.FileSystem, another way to create Path object
  + Instantiate: FileSystems.getDefault() factory class’s method
  + Methods: getPath(String s) to return Path object
* java.nio.file.Files, Path’s helper class with multiple purpose:
  + Instantiate: no need since it use static method that take Path objects and other type to work on
  + Method purposes:
    - Operate on file: copy, move, delete, check existence, read and write
    - Get single attribute and set single attribute
    - Return Stream (not I/O stream) to work on file system level
* java.nio.file.attribute.BasicFileAttributes
* java.nio.file.attribute.BasicFileAttributeView

## Introduction (java.nio.file.Path interface)

* Use factory class *java.io.file.Paths* to instantiate:
  + *Paths.get(String s, String … args)* 
    - Passes in a path (abs or relative)
    - Passes in list of Strings and then the method will join all with path separator
  + *Paths.get(URI uri) throws URISyntaxException* (checked): which use *java.net.URI*:
    - Must be absolute path
    - Start with schema file://, http:// ,etc..

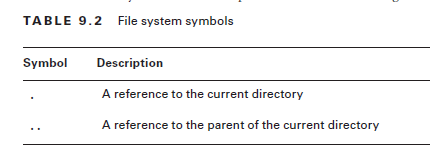
*Path path2 = Paths.get(new URI("file:///c:/zoo-info/November/employees.txt"));*

* *java.nio.file.FileSystem* use factory class *java.nio.file.FileSystems* to create object:
  + Path path1 = FileSystems.getDefault().getPath("pandas/cuddly.png")
    - This is the same as *Paths.get(String s, String … args)*
    - Use this to access local file system
  + Path path2 = FileSystems.getFileSystem(new URI("http://www.selikoff.net")).getPath(“duck.txt”)
    - Use this to access remote file system
* Converting:
  + Path → File pathObj.toFile()
  + File → Path fileObj.toPath()
  + Path → URI pathObj.toUri()

## Path and files

* *Path* object methods (belong to classes that implements java.nio.file.Path):
  + On *Path* object that use relative path, methods will not transverse (~track) outside of what is provided. Basically there would not be root info in relative path
  + Except toRealPath(), these methods will work on String aspect of Path object(like calling toString on Path objects)

|  |  |  |
| --- | --- | --- |
| **Return** | **Method** | **Note** |
| String | toString() |  |
| boolean | equals(Object) | Call toString on 2 path and compare the String |
| int | getNameCount() | Return number of element of path  Count all except root. Work on same on both relative and absolute path |
| Path | getName(int i) | Return element of path as Path object  Zero-based index and root is not counted. |
| Path | getFileName() | Get Path part of rightmost file element |
| Path | getParent() | Work fine on absolute path  If relative path does not contain parent or root info, then return null |
| Path | getRoot() |
| boolean | isAbsolute() | Different result on different OS |
| Path | toAbsolutePath() | Transform a Path to absolute path Path version |
| Path | subpath(int inclusive,  int exclusive) | Zero-based index and root is not counted.  *IllegalArgumentException* if inclusive=exclusive or out of path |
| Path | relativize(Path p) | Calculate how a file relativize (move) to another file  If 2 file in same directory, then include ../ at start  Work only both absolute path OR relative path. If mixed→ *IllegalArgumentException* |
| Path | resolve(Path p) | Absolute path is invoked with relative path param → combine  Relative path is invoked with relative path param → combine  Absolute path is invoked with absolute path param → copy of param  Relative path is invoked with absolute path param → copy of param |
| Path | resolve(String p) |  |
| Path | normalize() | Remove all file system symbol and write the short correct version of path |
| Path | toRealPath(Path p) | *Throws IOException*  Return absolute path normalized of file that exists OR throw exception  Will track down to where symbolic link point to |



* *java.nio.file.Files* is a helper class of *java.nio.file.Path*, NOT java.io.File .This table includes static helper method of Files
  + Most *throws IOException* since it check of file existance. These are throws for the operation (create, delete) not complete properly due to file not exist or could simply can’t be operated

|  |  |  |
| --- | --- | --- |
| **Return** | **Method** | **Note** |
| boolean | exists() | does not throws IOException here |
| boolean | isSameFile(Path p1,  Path p2) | *throws IOException* for any Path does not exists  First check for with equal method, then check whether 2 files records are in the same location of file system  Work with any mix of relative and absolute path |
| void | createDirectory(Path p) | *throws IOException* for any parent dir does not exist or file already exists  Only create the rightmost directory element |
| void | createDirectories(Path p) | *throws IOException* file already exists  Create rightmost element and any necessary non-existances parent dir |
| Path | copy(Path source,  Path target,  CopyOption … opt) | *throws IOException* for source not exists or target exists  For directory, it’s shallow copy only |
| long | copy(InputStream source,  Path target,  CopyOption... opt) | *throws IOException* for source not exists or target exists  long represent how many byte copy |
| long | copy(Path source,  OutputStream ot) | *throws IOException* for source not exists or target exists  Does not contains CopyOption since the OutputStream might be out of system |
| Path | move(Path source,  Path target,  CopyOption... opt) | *throws IOException* for source, target already exists  Will also move metadata of file, unlike copy |
| void | delete(Path p) | *throws IOException* for not empty dir or file not exists  If symbolic link, only delete link |
| boolean | deleteIfExists(Path p) | throw exception (not IOException) for not empty dir  For file not exists, false return instead of IOException |
| BufferedReader | newBufferedReader(Path p) | *throws IOException*  For Charset, Charset.forName("US-ASCII") OR Charset.defaultCharset() is ok  For OpenOption, can just skip them  For writing, will overwriting file if already exists |
| newBufferedReader(Path p,  Charset cs) |
| BufferedWriter | newBufferedWriter(Path p,  OpenOption… opt) |
| newBufferedWriter(Path p,  Charset cs,  OpenOption… opt) |
| List<String> | readAllLines(Path p) | *throws IOException* |
| List<String> | readAllLines(Path p,  Charset cs) |

## File attribute

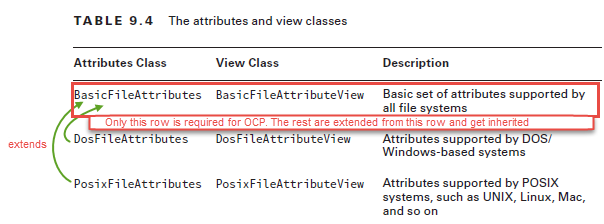
* File attribute ~ file metadata
* Continues of *Files* methods that related to file attribute:
  + Some *throws IOException* and some not

|  |  |  |
| --- | --- | --- |
| **Return** | **Method** | **Note** |
| boolean | isDirectory(Path p) | If symbolic link is passed, then check the pointed file  false if file no exists or match the rule |
| boolean | isRegularFile(Path p) |
| boolean | isSymbolicLink(Path p) | false if file no exists or match the rule |
| boolean | isHidden(Path p) | *throws IOException*  Hidden or not depend on OS |
| boolean | isReadable() | false if file no exists or match the rule |
| boolean | isExecutable() |
| long | size() | *throws IOException*  Size in byte.  return 0 for directory |
| FileTime | getLastModifiedTime(Path p) | *throws IOException*  *FileTime* has 2 methods:   * *toMillis()*: return epoch time * *fromMillis(long v)* : return FileTime object for epoch seconds |
| Path | setLastModifiedTime(Path p,  FileTime f) |
| UserPrincipal | getOwner(Path p) | *throws IOException*  *UserPrincipal* (\*)has following methods:   * *getName()* : return String name of owner |
| Path | setOwner(Path p,  UserPrincipal up) |

(\*) To get UserPrincipal, there are 2 ways:

* UserPrincipal owner = FileSystems.getDefault().getUserPrincipalLookupService().lookupPrincipalByName("jane");
* UserPrincipal owner = Paths.get(“file.txt”).getFileSystem().getUserPrincipalLookupService().lookupPrincipalByName("jane");

## File attributes views (java.nio.file.attribute package)



* Use attribute views to get more than 1 metadata about the file, unlike methods in previous section which only get 1 single attribute at a time
* *BasicFileAttributes* methods, which obviously does not modify file attributes
  + Instantiate: *BasicFileAttributes data = Files.readAttributes(path,BasicFileAttributes.class);*

|  |  |  |
| --- | --- | --- |
| **Return** | **Method** | **Note** |
| boolean | isDirectory(Path p) | Work like methods in Files class |
| boolean | isRegularFile(Path p) |
| boolean | isSymbolicLink(Path p) |
| long | sized() |
| boolean | isOther() | If it’s not regular file, dir or even symbolic link |
| FileTime | creationTime |  |
| FileTime | lastModifiedTime() | Work same like *getLastModifiedTime* method of Files but have no *get* in its name |
| FileTime | lastAccessTime() |  |
| Object | fielKey() | Return unique id if file system support or null if not support |

* BasicFileAttributeView methods, which can get BasicFileAttributes, and can only modify date/time related-attribute (make sense anyway, can’t just change the metadata of size)
  + Instantiate:

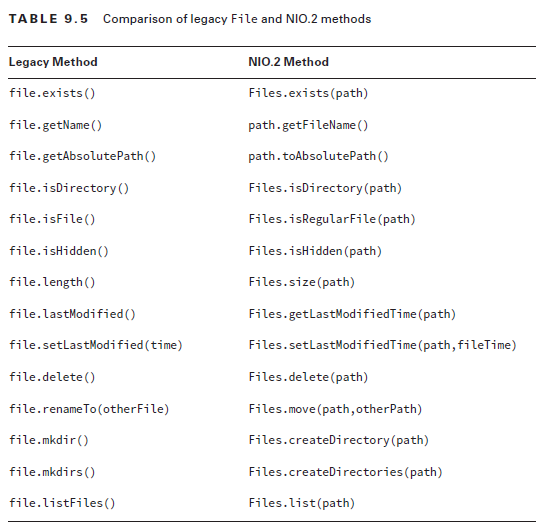
BasicFileAttributeView view =Files.getFileAttributeView(path,BasicFileAttributeView.class);

|  |  |  |
| --- | --- | --- |
| **Return** | **Method** | **Note** |
| BasicFileAttributes | readAttributes() | Use any methods in the previous table on this return BasicFileAttributes |
| void | setTimes(FileTime lastModifiedTime,  FileTime lastAccessTime,  FileTime createTime) | If any attribute you don’t want to change, just put null there. |

## New stream methods

* Stream API in using with Path will perform depth-first searching (search from the deepest up to root) with default max depth of *Integer.MAX\_VALUE* due to most file system would not pass that depth limit.
  + If the depth is set to 0, it means that only work on the path itself, not even what contains in the path
* Files helper class provide several method that return *Stream<Path>* which use depth-first searching and lazily transverse as used

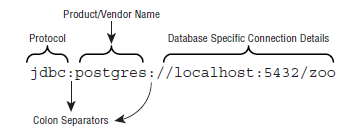
|  |  |  |
| --- | --- | --- |
| **Return** | **Method** | **Note** |
| Stream<Path> | walk(Path start,  FileVisitOption... options) | *throws IOException*  Can set max depth of search or just use default  By default, will not transverse where symbolic link point to, but can use FOLLOW\_LINK option (recommended with maxDepth to avoid infinite circle) |
| Stream<Path> | walk(Path start,  int maxDepth,  FileVisitOption... options) |
| Stream<Path> | find(Path start, int maxDepth, BiPredicate<Path,BasicFileAttributes> matcher, FileVisitOption... options) | *throws IOException* |
| Stream<Path> | list(Path p) | *throws IOException*  Only work on directory path  Unlike walk in that it only search in current dir for all file (regular files, sub dir) |
| Stream<String> | lines(Path p) | *throws IOException*  Unlike *readAllLines()* method of *Files* which will hardly worked on large file, this is lazy |
| Stream<String> | lines(Path p, Charset cs) |



# JDBC (java.sql package)

All of the below methods throws SQLException (checked)

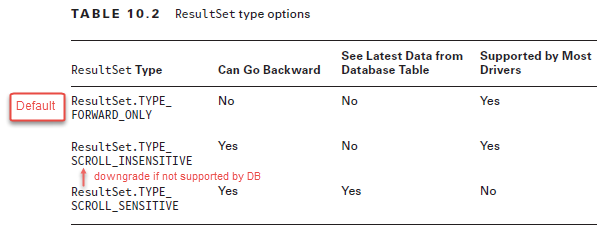
## Connecting database

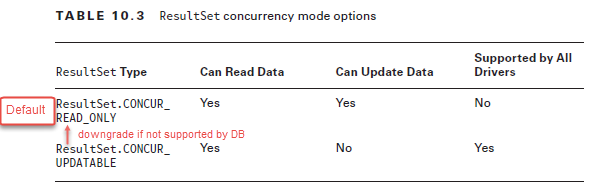


* JDBC URL:
  + Protocol: always the same
  + Product/Vendor: depends on which db used
  + DB details: can be as //localhost, //domain.com or @12.23.128.123 with port as optional
* For OCP, use DriverManager factory to get connection:
  + *Connection conn = DriverManager.getConnection("jdbc:derby:zoo")*
  + *Connection conn = DriverManager.getConnection("jdbc:postgresql://localhost:5432/ocp-book", "username", "password");*
* In order to connect to a DB, need to pass driver location in file system to command line: *java -cp "<java\_home>/db/lib/derby.jar:." ClassName*
  + Replace *<java\_home>* with the real location to driver JAR file

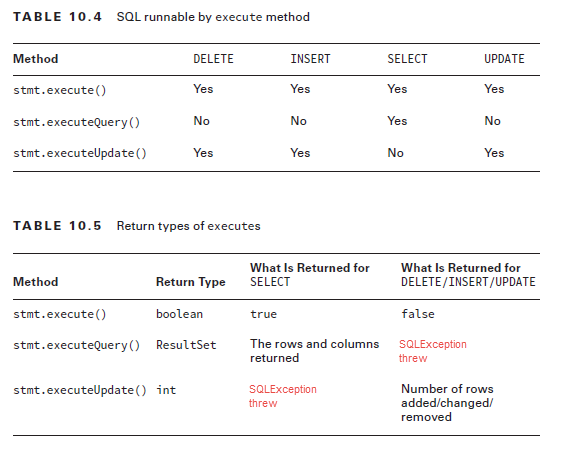
## Obtaining statement

* Creating statement from a Connection object: *Connection conn = …*
  + *conn.createStatement() : use default settype and setconcurrency*
  + *conn.createStatement(int resultSetType, int resultSetConcurrency)*
  + TYPE\_SCROLL\_SENSITIVE and CONCUR\_UPDATABLE are not commonly supported, therefore it can be downgrade if not supported



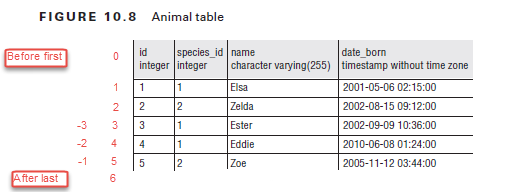


## Executing a Statement



## Getting data return from ResultType

* Let: *rs = stmt.executeQuery(String myQuery);*



* ResultSet method in moving “cursor”:
  + Cursor start at 0
  + When move cursor out beyond before first row or after last row, it will be set to before first row/after last row, NOT throwing SQLException
  + For empty ResultSet, all the movement to cursor will return false if return type is boolean

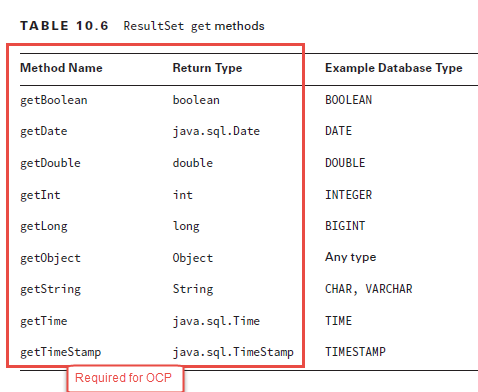
|  |  |  |  |
| --- | --- | --- | --- |
| **Return** | **Method** | **ResultSet**  **Type** | **Note** |
| boolean | next() | Forward  Scrollable |  |
| boolean | previous() | Scrollable |  |
| boolean | absolute(int rowNum) | Scrollable | Positive int for ascending order  Negative for descending order with -1 as the last row |
| boolean | relative(int moveNum) | Scrollable | Positive to move next n times  Negative to move previous -(n) times |
| void | beforeFirst() | Scrollable |  |
| boolean | first() | Scrollable |  |
| boolean | last | Scrollable |  |
| void | afterLast() | Scrollable |  |

* ResultSet has different method that retrieve info of a row in different type:
  + Each of these method has 2 overload version: 1 that take column index (1-based index) and 1 that take String columnLabel
  + *java.sql.Date* have *toLocalDate* method to convert to *LocalDate*
  + *java.sql.Time*  have *toLocalTime* method to convert to *LocalTime*
  + *java.sql.TimeStamp* have *toLocalDateTime* method to convert to *LocalDateTime*

*java.sql.TimeStamp sqlDt= rs.getDate(1);*

*LocalDatetime dt = sqlDt.toLocalDateTime();*

* + *getObject* return *Object* type, therefore, there’s need to have *instanceof* boolean check to cast *Object* to different type

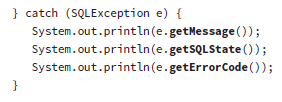


## Closing database resources

* Good practice: Using try-with-resource to open resource in order Connection → Statement → Execute Statement so that the resource can be close in the right (reverse) order
* Bad practice: just need to close Connection object and all Statement, ResultSet created from it will be closed
  + Due to closing a resource will automatically close all resource that one created too
* ResultSet can be closed if the same statement is used to execute another query

## Exception Dealing

* getMessage() : human readable message
* getSQLState(): code that can be googled what went wrong in DB
* getErrorCode() database-specific code



# OCP tips

* Can chain instantiate object using new keyword with method: E.g: new Object().toString()
* If symlink point to a dir, then that symlink can use to point to subdir of where it pointed
* Compiler warning:
  + Generic specified type and legacy (no type specified in diamond)

|  |  |  |
| --- | --- | --- |
| **Type** | **Name** | **Package** |
| I | Comparable | java.lang |
| C | String |
| C | StringBuilder |
| I | Runnable |
| Collection | | |
| I | Collections | java.util |
| I | List |
| C | ArrayList |
| C | LinkedList |
| I | Queue |
| C | ArrayDeque |
| C | HashSet |
| C | TreeSet |
| I | Map |
| C | HashMap |
| C | Hashtable |
| C | TreeMap |
| Stream & Optional | | |
| C | Optional | java.util |
| C | Stream | java.util.stream |
| C | IntStream |
| C | DoubleStream |
| C | LongStream |
| C | Collectors | java.util.stream |
| Date time and i18n, i11n | | |
| C | LocalDate | java.time |
| C | LocalTime |
| C | LocalDateTime |
| C | ZonedDateTime |
| enum | ChronoUnit | java.time.temporal |
| C | Locale | java.util |
| C | ResourceBundle | java.util |
| C | DateTimeFormatter | java.time.format |
| C | NumberFormat | java.text |
| I | ExecutorService | java.util.concurrent |
| C | Executor |
| C | Future |
| C | atomic classes for safe thread operation | java.util.concurrent.atomic |
| C | File | java.io |

|  |  |
| --- | --- |
| **System.out.println()** | **Result** |
| exception | Only name and message. Not stack trace |
| exception.printStackTrace() | Name, message, not stack trace |
| Object in general | If not override toString of Object -> ClassName@hashcode  If override toString method of Object -> depends overriding method return.  Class that implements toString for OCA:   * *java.lang.String* * *java.util.StringBuilder* |
| array | [ for each dimension follow+a letter + element type and hash code |
| StringBuilder | The String |
| ArrayList | [ element, element, element ] |
| Period | Starting with *P* and print non-zero element following by 1 appropriate uppercase letter.  Except: Week will be converted to *Days* in *Period* |

# More on exam

**Difference between immutable and Singleton vs Synchronized vs ThreadLocal**

1. 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 HashMaps for faster retrieval
   4. Their state doesn’t change they can be used in concurrent programming without any need for synchronization .
   5. Eg. String , wrapper class like Interger , Long
2. Mutable ex StringBuffer any modificaitn is done on objecxt reflects