1z0-809 Preparation

Java Class Design

Implement encapsulation

- The four principles of object oriented programming are
    - Encapsulation ➜ Hiding class data from outside the class and operate on them through method
        - Allows code reuse
        - The instance variables are declared as private
        - Methods are defined to access the variables
    - Inheritance ➜ Mechanism allowing a subtype to acquire fields and methods from its supertype. It implements "IS A" relationship

Visibility modifiers:

Modifier	*Class*	*Package*	*Subclass*	*All*
public	x	x	x	x
protected	x	x	x	
default	x	x		
private	x			

-
    - Composition ➜ Mechanism which implement "HAS A" relationship. A class would have objects from other types as its properties.
        - Allows code reuse
    - Polymorphism ➜ Mechanism allowing a method to behave differently based on the object it is invoked on
        - Overriding a method⚠
    - Abstraction

Override hashCode, equals, and toString methods from Object class

- **hashCode()**: signature is **public int hashCode()**
    - In class **Object**, it is the value of the object's memory address in hexadecimal
    - It is used as a key to reference a bucket in HashMap, HashSet and HashTable
    - If two objects are equals as returned by the **equals()** method, their hash code must be equal ➜ when overriding **equals()**, **hashCode()** must also be overridden
    - ⚠ **int** is the return type of **hashcode** ➜ compilation error
- **equals()**: signature **public boolean equals(Object o)**
    - In class **Object**, equals uses the == operator, therefore returns true if and only if the object is exactly the same
    - Must be overridden to compare objects in a meaningful way based on their states
    - ⚠ The parameter of **equals** is of type **Object** ➜ Overloading

- **toString()**: signature **public String toString()**
    - In class object returns: <class name>@<hashCodeHexa>
- ⚠ When overridden these method must be **public** ➜ compilation error

Create and use singleton classes

- Singleton common properties
    - Constructor is private to avoid instantiation done from outside
    - A public static method returns the only instance of the class
- Common patterns used:
    - Lazy initialisation
        - The instance is created on demand the first time it is requested
        - The get instance method must be synchronized
    - Initialization On Demand Holder Idiom (most commonly used)
        - A **private static** nested class SingletonHolder will hold the instance of the singleton as a **private static final** property
        - The getInstance method will get the property of the SingletonHolder
    - Eager initialization
        - The instance is a **private static final** property of the class and thus created when the class is loaded:
          private final static Singleton INSTANCE = new Singleton()
        - The instance could also be a **public static final** field that wouldn't need a getter.
    - Using enums (the best method)
- Immutable classes
    - Characteristics
        - All fields are **private final**
        - No method that would change state would be accessible (no setters)
        - Prevent methods from being overridden by subclasses
        - Do not expose directly properties that would be mutable objects

Develop code that uses static keyword on initialize blocks, variables, methods, and classes

- Static initialisation blocks are called once when the class is loaded for the first time
- Static classes are nested in others classes, they have access to the containing class

Advanced Java Class Design

Develop code that uses abstract classes and methods

- An abstract class may or may not contain abstract methods
- ⚠ Even if an abstract class cannot be instantiated, it can contain constructors that can be invoked by subclasses
- ⚠ Code involving **instanceof** does not compile when there is no way for it to evaluate true.

Develop code that uses the final keyword

- Final variables must be initialized whether they are static or not
    - **Static** final variables can be initialized on declaration **OR** in a static initializer block
    - **Instance** final variables can be initialized on declaration **OR** in an instance initializer block **OR** in all constructors of the class
    - **Local** final variables can be assigned on declaration **OR** later in their method. They can be assigned only once
    - ⚠ As opposed to non final variables, they are not assigned a default value ➔ A compilation error would occur.
- Final methods cannot be overridden
- Final classes are used to create immutable objects

Create inner classes including static inner class, local class, nested class, and anonymous inner class

- **Static inner** classes (aka: static nested classes) are static members of their enclosing class ➔ previous chapter
- **Non static inner** classes exists in the scope of an instance of their outer class
    - They require an object of their outer class to be instantiated first:
      Syntax: **Outer.NonStaticInner inner_inst = outer_inst.new Outer.NonStaticInner()**
    - ⚠ They cannot have static methods or variables except constants of **primitive** type or class **String** only
- **Local** classes are classes defined in a block of code
    - ⚠ They must be defined before being used in their enclosing block
    - The only modifiers that can be used are **abstract** and **final**
    - They have the same access level to the outer class as the enclosing block
    - They can access local variables of their block if they are effectively **final**
      Effectively final ➔ variable that never change after its initialization
    - ⚠ They cannot have static variables except constants of **primitive** type or class **String** only
- Anonymous classes are a special case of local classes
    - They must extend a class or implement an interface
    - They can't have a constructor since they have no name
    - They are instantiated using
        - The new operator
        - The name of a class or an interface
        - A list of arguments in parenthesis
        - Class body

Use enumerated types including methods, and constructors in an enum type

- Enum implicitly extends java.lang.Enum class ➜ they cannot extend any other classes
- == can be used to compare them
- An enum can have constructors methods and fields
- ⚠ The enum list declaration must be placed at the first position before fields constructor and methods➜ compilation error
- ⚠ The semicolon at the end of the enum list declaration can be omitted if there is no other statements
- Constructors have the following characteristics:
  - The only possible access modifier is **private**, the keyword is optional because it is implicit. Any other access modifier would result in a compilation error
  - The **super()** constructor cannot be invoked explicitly. It is done by the compiler.
- ⚠ All instance methods inherited from java.lang.Enum are **final** except toString➜ they cannot be overridden.
- The methods inherited from java.lang.Enum are:
  - **String name()** returns the name of the enum it is **final**.
  - **String toString()** returns the name of the enum **NOT final** and thus can be overridden.
  - **int ordinal()** returns the position of the enum in the declaration (starting with 0).
  - **static <T extends Enum> valueOf(Class<T> enumType, String name)** returns the enum object with this name within class enumType.
  - **int compareTo(T)** comparison of enums are based on the ordinal number.
  - **Class<E> getDeclaringClass()** returns the class of this enum.
- In addition to the methods inherited from java.lang.Enum, two implicit method exists
  - **public static E[] values()** returns an array containing all the values of this enum in their declaration order.
  - **public static E valueOf(String)** returns the enum object corresponding to the name
- It is also possible to add an anonymous class body to enum types.
  - ⚠ In this case, public methods in these class body will be invocable from outside of the enum only if they override methods defined in their enclosing enum.

Develop code that declares, implements and/or extends interfaces and use the @Override annotation.

- An interface can extend more than one interface
- public is the only access modifier in an interface
- Three types of methods
  - **abstract** methods: the keyword is implicit must be overridden by the implementers
  - **default** methods: the default method may or may not be overridden
  - **static** methods: they provide an implementation and **are not inherited**
    - They can be hidden by the sub interfaces: a sub interface can define a non static method with the same signature
    - A static method cannot override a non static method
- ⚠ Variables and fields are always **static final** ➜ attempting to modify them in default or static methods will lead to a compilation error
- Use the **@Override** annotation.
  - This annotation indicates that the following method overrides the implementation of a superclass or interface

- It cannot apply to hiding method

Create and use Lambda expressions

- Functional interface have
  - One and only one abstract method
  - May or may not have one or more static or default methods
  - May or may not be annotated with **@FunctionalInterface** annotation
- Lambda expression provides implementation for functional interfaces. They are composed of
  - A comma separated list of parameters enclosed with parentheses
    - The parameter type can be omitted, but if specified, it must be provided for **all** parameters
  - The arrow operator “**->**”
  - A body:
    - Single statement
      - No {}
      - No **return** statement
    - Block (**return** is necessary if the block needs to return a value)
- Parameters of a lambda expression must be exactly the same as the interface specification
  - No boxing unboxing
  - No sub types

Generics and Collections

- Examples here: http://www.jdoodle.com/a/1MqL

Create and use a generic class

- Benefits
  - Strong type checks **at compile time**
  - No castings needed
  - Allows implementation of generic algorithms
- Syntax:
  - Declaration of a class: modifier **class** ClassName**<**T1,T2,T3...Tn**>** {}
  - The diamond **<>** allows omitting the types when instantiating the object. It can be used only with the **new** operator
    Ex: List<String> = new ArrayList<>()
  - ⚠ it is not mandatory to use strong type checking thus the following syntax are allowed:
    - List l = new ArrayList()
    - List l = new ArrayList<>()
    - List l = new ArrayList<String>()
- It is possible to bound the type of parameter being used:
  - Ex: **public class** Person**<**T,S **extends** Number**>**
  - The second parameter must be a subclass of Number
- Bound types

Type	*Syntax*	*Example*
Unbounded	?	List<?> l = new ArrayList<String>() ;
Upper bound	? extends type	List<? extends Exception> l = new ArrayList<RuntimeException>() ;
Lower bound	? super type	List<? super Exception> l = new ArrayList<Object>() ;

- Generic classes are considered belongs to the same inheritance chain if
  - Their raw type are in the same inheritance chain
  - Their type arguments are the same
  - Ex: ArrayList<Integer> is a subtype of List<Integer> but is unrelated to List<Number> or ArrayList<Number>
    ⚠ By default if the type is not specified, the generic type is considered to be of type Object
- Generic methods
  - It is supported for static or instance methods to use generics
  - Declaration syntax: the <T> must be placed before the return type indicates the type to be substituted to the generic parameter type:
    - public static <T> void myMethod(T t)
    - public static <T> T myMethod(T t)
  - Invocation syntax
    - Usually invoked normally
    - Optionally the type can be placed before the method name;

- <String>myMethod(“this is a String”) ;

Create and use ArrayList, TreeSet, TreeMap, and ArrayDeque objects

- Types Matrix

Interface	*Duplicates*	*Ordered*	*Is key/value*	*add/remove ordered*
List	Yes	No	No	No
Map	Yes for values	No	Yes	No
Queue	Yes	Yes as inserted	No	Yes
Set	No	Yes	No	No

- Implementation comparison

Type	*Interface*	*Sorted?*	*Uses hashcode*	*Uses compareTo*
ArrayList	**List**	No	No	No
ArrayDeque	**Queue**	No	No	No
HashMap	**Map**	No	Yes	No
HashSet	**Set**	No	Yes	No
Hashtable	**Map**	No	Yes	No
LinkedList	**List,Queue**	No	No	No
Stack	**List**	No	No	No
TreeMap	**Map**	Yes	No	Yes
TreeSet	**Set**	Yes	No	Yes
Vector	**List**	No	No	No

- Manipulation methods

Action	***List***	***Map***	***Queue***	***Set***
Check emptiness	boolean isEmpty()	boolean isEmpty()	boolean isEmpty()	boolean isEmpty()
Check if an element is present	boolean contains(E)	boolean containsKey(K) boolean containsValue(V)	boolean contains(E)-	boolean contains(E)
Check if a collection of elements is contained	containsAll(coll)		containsAll(coll)	containsAll(coll)
Adding elements	boolean add(E) boolean addAll(coll)	void putAll(map)	boolean add(E) boolean offer(E) boolean addAll(coll)	boolean add(E) boolean addAll(coll)
Adding an element at a specific position	void add(index, E) boolean addAll(index,coll)	V put(K,V) V putIfAbsent(K,V)		-)
Removing an element	boolean remove(E) boolean removeAll(coll)	-	boolean remove(E) boolean removeAll(coll)	boolean remove(E) boolean removeAll(coll)
Removing an element at a specific position	E remove(index)	V remove(K)		
Remove conditional	boolean retainAll(coll) boolean removeIf()	boolean remove(K,V)	boolean retainAll(coll) boolean removeIf()	boolean retainAll(coll) boolean removeIf()
Remove all elements	clear()	clear()	clear()	clear()
Getting position of an element	int indexOf(E) int lastIndexOf(E)	-		-
Getting an element at a specific position	E get(int)	V get(K) V getOrDefault(K)	E peek() E element()	-
Get an element and remove it	-	-	E Poll() E remove()	--
Replacing elements	E set(index, E) void replaceAll(unaryOp)	V replace(K,V) V replace(K,V,V) void replaceAll(BiFunction<K,V,V>) V merge(K,V,BiFunction<? super V,? super V,? extends V>)		-

- Create and use **ArrayList**
  - Implemented interfaces: **Collection**, **List**
  - Can be declared with or without <>
  - Duplicates are allowed
  - The important methods are:
    - **equals**() returns true if the two lists have the same elements in the same order. Calls the method equals on each element
    - **listIterator()**
  - Specific methods not inherited from List
    - boolean removeRange(fromindex, toIndex)
  - Action on backing array
    - By default the ArrayList is created with a capacity of 10 elements. Each time it need more space, it increases the capacity by 50%
    - **ensureCapacity()** will set the capacity to the highest value between current capacity + 50% and the value given as parameter
    - **trimToSize()** reduce the capacity to the current size of the list.
- Create and use **TreeSet**
  - Implemented interface: **Collection**, **Set**
  - **TreeSet** is an implementation of the **Set** interface which is aimed to store elements based on their ordering. The elements are sorted based on their natural ordering or by a **Comparator**, given as a parameter to the constructor
    - ⚠ No duplicates
    - ⚠ If trying to insert an element not implementing Comparable a ClassCastException is thrown
  - Constructors
    - TreeSet()
    - TreeSet(SortedSet<E>)creates a TreeSet with the same elements as the SortedSet
    - TreeSet(Collection<? Extends E>)
    - TreeSet(Comparator<? Extends E>)
  - Methods coming from Set
    - iterator()
  - Specific methods
    - **E first(), last()**: returns the first and last elements of this TreeSet
    - **E ceiling(E)**, **E floor(E):** returns respectively the least element greater than **or equal**, and the greatest element less than **or equal** to the one specified
    - **E higher(E)**, **E lower(E):** returns respectively the least element **strictly** greater than, and the greatest element **strictly** less than to the one specified
    - **SortedSet<E> headSet(E)**,**tailSet(E)** : returns respectively the portion of the TreeSet that is strictly less or strictly greater than the specified element
    - **NavigableSet<E> headSet(E, boolean)**, **tailSet(E, boolean)** : returns respectively the portion of the TreeSet that is less or greater than the specified element, inclusive or not depending on the boolean parameter
    - **descendingIterator()**
    - **comparator()**

- Create and use **TreeMap** https://www.jdoodle.com/a/1FqC
  - Implemented interface: **Map**
  - TreeMap has the following characteristics
    - Elements are sorted based on the natural ordering of their key or by a comparator
    - The ordering should be consistent with the equals method
  - Constructors
    - TreeMap()
    - TreeMap(SortedMap<K, ? extends E> map)
    - TreeMap(Map<? Extends K, ? extends E> map)
    - TreeMap(Comparator<? super K>)
  - Specific methods
    - **Set<Map.Entry<K,V>> entrySet()** set view of the entries
    - **Set<K> keySet()** set view of the keys (Set no duplicates
    - **Collection<V> values()** collection of values (remember collection allows duplicate)
    - **K firstKey(), lastKey()**
    - **Map.Entry<K,V> firstEntry(), lastEntry()**
    - **Map.Entry<K,V> ceilingEntry(K)** / **floorEntry(K)** returns the first entry greater / less than or equals than ➜ floor <= key <= ceiling
    - **K ceilingKey(K)** / **floorKey(K)**
    - **K higherKey(K)** / **lowerKey(K)** returns the first entry greater / less than or equals than ➜ lower < key < higher
    - **K ceilingKey(K)** / **floorKey(K)**
    - SortedMap<K,V> subMap() returns a view of a portion of the map. It is backed by this map ➜ any change in the backing map is reflected
- Create and use **ArrayDeque** objects: https://www.jdoodle.com/a/1Ft8
  - Implemented interfaces: **Collection**, **Deque**, **Queue**
  - Constructors
    - **ArrayDeque()** creates an empty ArrayDeque with a capacity of **16 elements**
    - **ArrayDeque(int n)** creates an empty ArrayDeque with a capacity of **n elements**
    - **ArrayDeque(Collection<? Extends E>)** creates an ArrayDeque from the elements of the collection in the order returned by the collection iterator
  - Methods are all inherited from the **Deque** interface
    - **addFirst()** / **addLast()**
    - **push()** is equivalent to **addFirst**()
    - **E removeFirst()** / E **removeLast()**
      - Retrieves and remove the first/last element from the Deque
      - Both throw a NoSuchElementException if the Deque is empty
    - **E pop()** is equivalent to **removeFirst()**
    - **E getFirst()** / **E getLast()**
      - Retrieves but does not remove the first / last element
      - Throw an exception if the Deque is empty
    - **E peekFirst()** / **peekLast()**
      - retrieves without removing the first / last element
      - Returns null if the Deque is empty
      - Throw an exception if the Deque is empty
    - **E peek()** is equivalent to **E peekFirst()**
    - **E element()**:

            - Retrieve but does not remove the first element
            - Returns null when the **Deque** is empty (no exception as opposed to peek)
        - **pollFirst()** / **pollLast()** retrieves and remove the first / last element
            - Returns null if the Deque is empty
        - **offerFirst**() / **offerLast**() insert a new element at the beginning or at the end
        - iterator()
        - descendingIterator()

Use java.util.Comparator and java.lang.Comparable interfaces

Exemples here: http://www.jdoodle.com/a/1FUM

- Characteristics of the Comparable interface
    - Contains a single method: **int compareTo(T that)** which returns
        - 0 if the objects being compared are equals
        - >0 if this object is greater than **that** object
        - <0 if this object is less than **that** object
    - Used to sort collections or array
        - **Arrays.sort(T)** ➜ T must implement Comparable
- Characteristics of the Comparator interface
    - It has one abstract method: **int compare(T object1, T object2)** ➜ it's a functional interface
        - **Int compare(o1,o2)** must return
            - **Positive** int if o1 > o2
            - **Zero** if o1 = o2
            - **Negative** int if o1 < o2
    - An instance of comparator can be passed to the constructor of an array or a collection
    - The behavior should be consistent with the equals method
    - Useful when
        - the objects to be sorted does not implements Comparable
        - Different comparison types are needed
    - ⚠ Comparable must be parameterized :
        - implement Comparable will not work ➜ **compilation failure**
        - Implement Comparable<MyClass> will be ok

Collections Streams and Filters

Examples here: https://www.jdoodle.com/a/1G9w

- Some method from Stream suchAs filter uses a Predicate<T> parameter which contains
    - A single method **boolean test(T)**
- **removeIf(Predicate<T>)** from the Collection allows removing the elements from a collection based on a Predicate

Iterate using forEach methods of Stream

- The **forEach** method in **Stream**
    - Signature: void **forEach(Consumer<T>)**
    - It takes a parameter of type **Consumer<T>** interface
        - **Consumer<T>** interface has one abstract method: **public void accept(T)**

-
    - This is a terminal operation
    - ⚠ The behavior is non deterministic because the action can be executed at any time and in any thread
- The **forEach** method in **List**
    - It is inherited from the **Iterable** interface
    - It has the same signature as the one in Stream
    - ⚠ Unlike the **forEach** method in **Stream** is that the behavior **IS** deterministic: the elements are processed in their iteration order.

Describe Stream interface and Stream pipeline

Examples here: http://www.jdoodle.com/a/1Gca

- Stream is a sequence of elements that supports aggregation
- A stream can be created by a static method of the Stream interface or by a stream creation on a data source (ex: **list.stream()**) ;
- ⚠ A stream can be operated once and only once: IllegalStateException would be thrown otherwise
- ⚠ the source should not be modified while being processed. Even if no terminal operation has been used
- Two types of operations that can be combined into a stream pipeline
    - Intermediate
        - They return a stream so they can be chained together
        - They are lazy and triggered only when a terminal operation is initiated
    - Terminal
        - When performed the stream is considered as consumed
        - Produces a result or a side effect
        - They return void so there cannot be other operation after

Filter a collection by using lambda expressions

- Use method references with Streams: http://www.jdoodle.com/a/1GU4
    - Replace lambda expression if the lambda does nothing but calling a method
    - Syntax: **<Classname|ObjectName>::<methodName>**
    - 4 type can be used
        - Static method:
            - In this case the current object is passed as an argument to the method
        - Instance method of an arbitrary object of a particular type
            - In this case the method is called on the current object
        - Instance method of a particular object
        - Constructor

- **Collections.binarySearch(list, value)** is intended to be used on a list sorted in **ascending** order. Otherwise the result is undefined
- **Collections.binarySearch(list, comparator, value)** is intended to be used on a list sorted according to the **specified comparator**

Lambda Built-in Functional Interfaces

Use the built-in interfaces included in the java.util.function package such as Predicate, Consumer, Function, and Supplier

Examples here: http://www.jdoodle.com/a/1GVL

	Abstract method	Default or static methods
Predicate<T>	boolean test(T)	Predicate>T> negate() Predicate<T> and(Predicate<? Super T>) Predicate<T> or(Predicate<? Super T>) **Predicate isEqual(Object)***
Consumer<T>	void accept(T)	Consumer<T> andThen(Consumer<? Super T>)
Function<T,R>	R apply(T)	<V> Function<T,V> andThen(Function<? Super R, ? extends V> after) <V> Function<T,V> compose(Function<? Super R, ? extends V> before) **Function<T,V> identity()***
Supplier<T>	T get()	-

(*) static method

- Functional interfaces can be seen as place holders for lambda expressions
- Types in package **java.util.function** are functional interfaces
- **Predicate<T>** represents a condition to be evaluated against an argument.
- **Consumer<T>**:represents an operation on a given argument which does not return a value
- **Function<T,R>** represents a function that takes an argument and returns a function
  ⚠ Pay attention: the first argument is the input type, the second one is the result type
- **Supplier<T>** represent a supplier of object of a particular type
- **UnaryOperator<T,T>** inherits of **Function<T,T>** has the same default methods as function

Develop code that uses primitive versions of functional interfaces

Examples here: http://www.jdoodle.com/a/1GWc

Interface family	Primitive versions	Method Name	Default or Static Methods
Supplier	BooleanSupplier DoubleSupplier IntSupplier LongSupplier	boolean getAsBoolean() double getAsDouble() int getAsInt() long getAsLong()	-

Consumer	DoubleConsumer
	IntConsumer
	LongConsumer
		void accept(double)
		void accept(int)
		void accept(long)
			andThen(DoubleConsumer)
			andThen(IntConsumer)
			andThen(LongConsumer)

Predicate	DoublePredicate IntPredicate LongPredicate	boolean test(double) boolean test(int) boolean test(long)	and, or, negate and, or, negate and, or, negate
Function	DoubleFunction<R> IntFunction<R> LongFunction<R>	R apply(double) R apply(int) R apply(long)	-
	ToDoubleFunction<T> ToIntFunction<T> ToLongFunction<T>	double applyAsDouble(T) int applyAsInt(T) long applyLong(T)	-
	DoubleToIntFunction DoubleToLongFunction IntToDoubleFunction IntToLongFunction LongToDoubleFunction LongToIntFunction	int applyAsInt(double) long applyAsLong(double) double applyAsDouble(int) long applyAsLong(int) double applyAsDouble(long) int applyAsInt(long)	-
UnaryOperator	DoubleUnaryOperator IntUnaryOperator LongUnaryOperator	double applyAsDouble(double) int applyAsInt(int) long applyAsLong(long)	andThen, compose, identity*
BinaryOperator	DoubleBinaryOperator IntBinaryOperator LongBinaryOperator	double applyAsDouble(double,double) int applyAsInt(int,int) long applyAsLong(long,long)	-

(*) static method

⚠ The primitive versions don't inherit from the object versions

- Provide functional interfaces to operate on primitive types instead of reference type
  - To avoid boxing / unboxing at compile time **and** run time.
  - Supported types are **int**, **long** and **double**
  - **boolean** is supported for **Supplier** only
- Primitive versions of **Function** don't have default or static methods as opposed to the **Function<T,R>**
- Primitive versions of **Predicate** have default method but they have lost the **isEqual** static method,
- **BinaryOperator** inherits of **BiFunction<T,T,T>**

Develop code that uses binary versions of functional interfaces

Family	Binary version	Abstract method	Default methods
Predicate	BiPredicate<T,U>	boolean test(T,U)	and, or, negate
Consumer	BiConsumer<T,U>	void accept(T,U>	andThen
	ObjIntConsumer<T> ObjLongConsumer<T> ObjDoubleConsumer<T>	void accept(T,i) void accept(T,l) void accept(T,d)	-
Function	BiFunction<T,U,R>	R apply(T,U)	andThen
	ToIntBiFunction<T,U> ToLongBiFunction<T,U> ToDoubleBiFunction<T,U>	int applyAsInt(T,U) long applyAsLong(T,U) double applyAsDouble(T,U)	-
	BinaryOperator<T>	T apply(T,T)	andThen minBy* maxBy*

- Binary functional interfaces are functional interfaces with two parameters
- Except Supplier, all built in interfaces have a binary version. This is because the abstract method of Supplier does not have any parameter
- These interfaces are prefixed by "Bi"
- **Predicate** has one specialisation **BiPredicate<T,U>**
- **Consumer** has four specialisations:
- **Function** also has four specialisations
- **BinaryOperator** is a specialisation of BiFunction producing a result of the same type as the operands
    - **BinaryOperator<T>** in herits of interface (similar to **BiFunction<T,T,T>**)

Develop code that uses the UnaryOperator interface

- **UnaryOperator<T>** is a specialisation of **Function** producing a result of the same type as its parameter (similar to **Function<T,T>**)
    - Abstract method **T apply(T)**
    - It inherits the abstract and default methods of **Function**

Java Stream API

Develop code to extract data from an object using peek() and map() methods including primitive versions of the map() method

- Examples here : http://www.jdoodle.com/a/1GZl (See: peekAndMapDemo method)
- The **peek** method performs an action on each element of a stream. It's argument is a consumer.
    - ⚠ It is an intermediate action of the stream and is executed once the stream is consumed.
    - It is usually used for logging
    - Its signature is; **Stream<T> peek(Consumer<? Super T>)**
- The **map** method of Stream applies a mapper on each element of the stream
    - ⚠ as peek it is an intermediate operation and will be executed once the stream is consumed.
    - It's signature is: **Stream<R> map(Function<? Super T, ? extends R>)**
    - It has three primitive specializations
        - **mapTo[X]** where [X] stands for **Int Long** and **Double**
        - Each one takes the corresponding primitive version of the function interface
        - The signatures are:
            - IntStream mapToInt(ToIntFunction<? Super T> mapper)
            - LongStream mapToLong(ToLongFunction<? Super T> mapper)
            - DoubleStream mapToStream(ToDoubleFunction<? Super T> mapper)

Search for data by using search methods of the Stream classes including findFirst, findAny, anyMatch, allMatch, noneMatch

- In order to search data in a stream we have two types of function
    - find*() which returns an Optional object describing an element of the stream
        - No parameters
        - Two forms:
            - **Optional<T> findFirst()** selects the first element of the stream
            - **Optional<T> findAny()** non deterministic, my return any element of the stream, maximum performance in parallel operations
    - *Match(Predicate<? Super T>) returns a boolean true if a matching element is found false otherwize:
        - The parameter is a predicate
        - Three forms:
            - **boolean anyMatch(Predicate<T>)** returns true if one element matches the predicate
                - False is returned if the stream is empty
            - **boolean allMatch(Predicate<T>)** return true if
                - All elements matches the predicate
                - The stream is empty
            - **boolean noneMatch(Predicate<T>)** return true if no element matches the predicate ***or if the stream is empty***
        - The predicate is evaluated only if necessary
    - ⚠ both find* and *Match are "short circuiting" **terminal** operations

Develop code that uses the Optional class

- **Optional<T>** class is aimed to better manage null values
- No public constructor, instead static methods are used
  - **Optional.empty()** will create an optional with no value
  - **Optional.of(T)** will create an optional with value T **BUT** raises a NullPointerException if the value passed is null
  - **Optional.ofNullable(T)** will create an optional with value T if not null or an empty optional otherwize.
- The following methods returns the value if present but behaves as follows when the value is not present:
  - **T get()** throw a NoSuchElementException
  - **T orElse(T)** returns a substitution value passed as argument
  - **T orElseGet(Supplier<T>)** return the value from the supplier passed as argument.
  - **T orElseThrow((Supplier<T extends Throwable>)** throws an exception returned by the exception supplier passed as argument.
- **boolean isPresent()** method will return true if a value is present false otherwise
- **void ifPresent(Consumer<T>)** will invoke the consumer passed as argument.
- **Optional<T> filter(Predicate<T>)** will return an optional with the value if it matches the predicate passed as argument otherwise returns an empty Optional
- **Optional<R> map(Function<T,R>)** will transform this Optional<T> into an Optional<R> using the mapping function passed as argument. If no value is present or if the result is null an empty Optional is returned.
- **Optional<R> flatMap(Function<T,Optional<R>>)** will transform this Optional<T> into an Optional<R> using the mapping function passed as argument. If no value is present or if the result is null an empty Optional is returned.
  - Used instead of **map** with functions returning an Optional

Develop code that uses Stream data methods and calculation methods

- They are reduction operations
- The following are present in the reference Stream and in its primitive specializations
  - **long count()** returns the number of elements of this stream
  - **T min(Comparator<T>)** returns the minimum element of this stream
  - **T max(Comparator<T>)** returns the maximum element of this stream
  - The **reduce** functions takes the following forms:
    - **Optional<T> reduce(BinaryOperator<T> accumulator)** in this case the first element of the stream is taken as identity
    - **T reduce(T Identity, BinaryOperator<T> accumulator)**
  - ⚠ **min** and **max** does not need a comparator argument in the primitive versions because the natural ordering is used
- In addition to the previous functions the primitive versions of the stream interface have the following functions:
  - Optional<Double> **average**()
  - **sum**()

Sort a collection using Stream API

- Two methods are provided by the stream API:

- **Stream<T> sort()** elements are sorted according to their natural order, T must implement the Comparable interface
- **Stream<T> sort(Comparator<T>)** elements are sorted according to the comparator passed as parameter.

Save results to a collection using the collect method and group/partition data using the Collectors class

- Collect method can be more efficient than reduce when it comes to accumulating data into a mutable container: ex user a StringBulder to concatenate strings instead of creating a new String each time
- The collect method has two forms:
  - **<R> collect(Supplier<T>, BIConsumer<R, ? super T>, BIConsumer<R,R>)**
    - First argument ***supplier*** provide instances of the result container
    - Second argument **accumulator** is a function that incorporates element in a result container
    - Third argument **combiner** will combine the result containers
  - **<R> collect(Collector<? Super T, A, R>)**
    - The **collector** argument handle all aspects of collection. For example:
- The JDK provides a lot of implementations of Collector, through the Collectors class:
  - **<T> Collector<T,?,List<T>> toList()** returns a collector that accumulates the elements of a stream into a List
  - **<T> Collector<T,?,Set<T>> toSet()** returns a collector that accumulates the elements of a stream into a Set
  - **<T,C extends Collection<T>> Collector<T,?,C> toCollection(Supplier<C>)** accumulates in a collection in encountered order. The parameter collectionFactory is a supplier that will return a collection of the appropriate type
  - <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? Super T, ? extends K>, Function<? Super T, ? extends U>)
    - First argument is a keyMapper function
    - Second argument is a valueMapper function
  - groupingBy function has two forms:
    - One argument: <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)
    - Two arguments <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)
  - partitioningBy function has two forms:
    - <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> predicate)
    - <T,D,A> Collector<T,?,Map<Boolean,D>> partitioningBy(Predicate<? super T> predicate, Collector<? super T,A,D> downstream)

Use flatMap() methods in the Stream API

- flatMap() operates a transformation to each element of the stream into a stream
  - Different of map() which returns a single element
  - The resulting streams are then flattened into a resulting stream
  - This is a one to many relationship
- signature is

  - **<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)**
- It has three primitive specialisations
  - **IntStream flatMapToInt(Function<? Super T, ? extends IntStream>)**
  - **LongStream flatMapToLong(Function<? Super T, ? extends LongStream>)**
  - **DoubleStream flatMpToDouble(Function<? Super T, ? extends DoubleStream>)**

Exceptions and Assertions

Use try-catch and throw statements

- Already seen not developed here.

Use catch, multi-catch, and finally clauses

- ⚠ When a finally block is present, and an exception is thrown, it is put "on hold" until the finally block executes

Use Autoclose resources with a try-with-resources statement

- The resources declared in the **try()** statement, **must** implement the **AutoCloseable** interface
- Syntax: one or more resources can be declared between the parentheses of the try() statement, they must be separated by a ";". (No ";" after the last resource)
- The resources are closed in the reverse order of their creation
- ⚠ The catch and finally blocks are optional
- In a "*try with resources*" statement the catch and finally blocks are executed <u>**after**</u> the declared resources are closed
- If an exception occurs during the try block then during the closing of resources:
    - The exception from the try block is thrown
    - The exception during resource closing is suppressed
- At the opposite, if an exception occurs in a classic try block then in its finally block;
    - The exception from the try block is suppressed
    - The exception in the finally block is thrown

Create custom exceptions and AutoCloseable resources

- **Closeable** and **AutoCloseable** interfaces
    - **AutoClosable** is a superclass of **Closeable** and its abstract method is
        - void close() throws **Exception**
    - **Closeable** is a subclass of **AutoCloseable** and its abstract method is
        - void close() throws **IOException**

Test invariants by using assertions

- Assertions are made to test invariants during program execution
    - Internal invariants
    - Control flow invariants
    - Pre conditions post conditions
- They are executed at runtime only if they are enabled
    - java **-ea** ...
- They are not suitable to validate input of public methods
- There are two forms:
    - **assert <boolean expression> ;** will throw an AssertionError if <boolean expression> is false with no message

- **assert <boolean expression> : <value or message> ;** will display an additional information with the AssertionError

Use Java SE 8 Date/Time API

- Exemples here: http://jdoodle.com/a/1I9R

Create and manage date-based and time-based events including a combination of date and time into a single object using LocalDate, LocalTime, LocalDateTime, Instant, Period, and Duration

- This API conforms to ISO-8601
- Several methods are used to manipulate objects from the java.time API:

Action	***LocalDate***	***LocalTime***	***LocalDateTime***	***Instant***

Creation from date fields (all types are int except month that can also be of type Month)	of(y,m,d)
	of(y,Month,d)
		of(h,m,s,n)
		of(h,m,s)
		of(h,m)
			of(y,m,d,h,m,s,n)
			of(y,m,d,h,m,s)
			of(y,m,d,h,m)
			of(y,Month,d,h,m,s,n)
			of(y,Month,d,h,m,s)
			of(y,Month,d,h,m)
				-

Creation from other temporal types	-	-	of(LocalDate, LocalTime)	-

Creating from	of**Year**Day(int y, int d)
	of**Epoch**Day(long)
		of**Nano**OfDay(long)
		of**Second**OfDay(long)
			of**Epoch**Second(long)	ofEpochMilli(long)
				ofEpochSecond(long)
				ofEpochSecond(long, long)

Creating from a string	parse(s) **YYYY-MM-DD** parse (s, DateTimeFormatter)	parse(s) **HH:MM** parse (s, DateTimeFormatter)	parse(s) **YYYY-MM-DD THH:MM:SS** parse (s, DateTimeFormatter)	parse(s) **YYYY-MM-DD THH:MM:SSZ**

Adding time (param is of type long)	plus**Days**(d)
	plus**Months**(m)
	plus**Weeks**(w)
	plus**Years**(y)
		plus**Hours**(h)
		plus**Minutes**(m)
		plus**Seconds**(s)
		plus**Nanos**(n)
			plus**Days**(d)
			plus**Months**(m)
			plus**Weeks**(w)
			plus**Years**(y)
			plus**Hours**(h)
			plus**Minutes**(m)
			plus**Seconds**(s)
			plus**Nanos**(n)
				plusSeconds(long)
				plusMillis(long)
				plusNanos(long)

Removing	minus**Days**(long)
	minus**Weeks**(long)
	minus**Months**(long)
	minus**Years**(long)
		minus**Hours**(h)
		minus**Minutes**(m)
		minus**Seconds**(s)
		minus**Nanos**(n
			minus**Days**(d)
			minus**Months**(m)
			minus**Weeks**(w)
			minus**Years**(y)
			minus**Hours**(h)
			minus**Minutes**(m)
			minus**Seconds**(s)
			minus**Nanos**(n)
				minusSeconds(long)
				minusMillis(long)
				minusNanos(long)

Modify one field specific	with**DayOfYear**(int) with**DayOfMonth**(int) with**Month**(int) with**Year**(int)	with**Hour**(int) with**Minute**(int) with**Second**(int) with**Nano**(int)	with**DayOfYear**(int) with**DayOfMonth**(int) with**Month**(int) with**Year**(int) with**Hour**(int) with**Minute**(int) with**Second**(int) with**Nano**(int)	-

Conversion			toLocalDate()
			toLocalTime()
				atOffsetZoneOffset()
				atZone(ZoeID)

Getting info specific	Era get**Era**() int get**Year**() Month get**Month**() int getMonthValue() DayOfWeek get**DayOfWeek**() int get**DayOfMonth**() int get**DayOfYear**()	int get**Hour**() int get**Minute**() int get**Second**() int get**Nano**()	Era get**Era**() int get**Year**() Month get**Month**() int getMonthValue() DayOfWeek get**DayOfWeek**() int get**DayOfMonth**() int get**DayOfYear**() int get**Hour**() int get**Minute**() int get**Second**() int get**Nano**()	-

Modify one field generic	with(TemporalAdjuster)
	with(TemporalUnit, long)			

Adding temporal amount **Most used Period**	plus(TemporalAmount)			
Adding temporal unit	plus(long, TemporalUnit)			
Getting info generic	int get(TemporalField) long getLong(TemporalField)			

- Duration and periods

	Period	***Duration***
Range	Year ➔ Day	Hour ➔ Nano
Format	P<y>Y<m>M<d>D	PT<h>H<m>M<s>S<n.nn>N

Creating
⚠ Pay attention to the type
	of(int, int, int)
	ofYears(int)
	ofMonths(int)
	ofWeeks(int)
	ofDays(int)
		ofDays(long)
		ofHours(long)
		ofMinutes(long)
		ofSeconds(long)
		ofSeconds(long,long)
		ofMillis(long)
		ofNanos(long)

Adding	plusYears(long)
	plusMonths(long)
	plusDays(long)
		plusDays(long)
		plusHours(long)
		plusMinutes(long)
		plusSeconds(long)
		plusMillis(long)
		plusNanos(long)

Removing	minusYears(long)
	minusMonths(long)
	minusDays(long)
		minusDays(long)
		minusHours(long)
		minusMinutes(long)
		minusSeconds(long)
		minusMillis(long)
		minusNanos(long)

Modifying specific field
⚠ Pay attention to the type
	withYears(int)
	withMonths(int)
	withDays(int)
		withSeconds(long)
		withNanos(**int**)

Operations

- Remarquable classes
    - **LocalDateTime**, **LocalDate** and **LocalTime**
    - **Instant** represent an instantaneous point on the timeline not meant to represent a time unit larger than second
    - **Period**: date based amount of time from year to Day
    - **Duration**: time based amount of time goes from hours to nanos
    - ⚠ Period and duration cannot be added one to the other

Work with dates and times across time zones and manage changes resulting from daylight savings including Format date and times values

	ZonedDateTime	*OffsetDateTime*
Creating	of(y,m,d,h,m,s,n,ZoneId)	of(y,m,d,h,m,s,n,ZoneOffset)

Creating from other types	of(LocalDate,LocalTime, ZoneId)
	of(LocalDateTime, ZoneId)
		of(LocalDate,LocalTime, ZoneOffset)
		of(LocalDateTime, ZoneOffset)

- Time zone is represented by an object of **ZoneId** type:
    - Created by static method ZoneId.of(String)
        - Parameter is a string representing
            - The time zone name: ex “Europe/Paris”
            - The offset: ex “UTC+06.00”
    - **ZoneOffset** is a subclass of **ZoneId** it is created as follows:
        - ZoneOffset.of(String)
            - ZoneOffset.of(“+6”) or ZoneOffset.of(“+06:00”)
            - ⚠ the “+” is mandatory for positive offsets, if the offset is 0 the letter “z” is used instead of parameter 0
        - ZoneOffset.ofHours(int)
- The **OffsetTime** class represent a time with an offset from UTC:
    - It is created from a LocalTime and a ZoneOffset
- The **OffsetDate** and **OffsetDateTime** are on the same principle
- DateTimeFormatter

Define and create and manage date-based and time-based events using Instant, Period, Duration, and TemporalUnit

- **Instant**
- **Period**
    - The daylight saving is considered for periods ➔ a day is equivalent to 24H
    - ⚠ adding a Duration of one day and adding a period of one day will not give the same result on daylight saving.

- **Duration**
    - Duration does not handle daylight saving ➜ one day is always 24 hours
- **TemporalUnit**
    - An interface representing units of time
    - Implementation in the JDK is **ChronoUnit**
        - Provides time units from NANOS to FOREVER
        - Remarkable methods
            - addTo
            - Between
            - isDateBAsed
            - isTimeBased

Java I/O Fundamentals

Example heres: http://www.jdoodle.com/a/1IZN

Read and write data from the console

- Input output
  - **InputStream** System.in ➜ user input
  - **PrintStream** System.out ➜ standard output
  - **PrintStream** System.err ➜ standard error
- **Scanner** class is used to get input from the user
  - It is instantiated: Scanner sc = new Scanner(System.in)
  - It can parse strings or some primitive types
    - Ex: **getNextLine()**,**getNextInt()**
  - It must be closed
- **Console** class is the character based console.
  - It may or may not exist depending
    - On the underlying platform
    - On the manner the JVM is launched
      - No console if launched by a batch or an IDE
      - Console if launched from the command line
  - It is obtain using **System.console()** *(may return null)*
    - **printf** and **format** methods are supported
    - Two methods will get the user input:
      - **String readLine()** expects user input
      - **char[] readPassword()** expect user input but does not show it on the screen
    - They have overloaded versions that will display a formatted prompt before expecting user input
      - **String readLine(String, Object ...)**
      - **char[] readPassword(String, Object ...)**
- Console vs Standard stream:
  - Standard stream are byte based, Console is character based
  - Standard stream always exists, Console existence depends on the platform
  - Standard stream belongs to the platform, Console is associated to the JVM

Use BufferedReader, BufferedWriter, File, FileReader, FileWriter, FileInputStream, FileOutputStream, ObjectOutputStream, ObjectInputStream, and PrintWriter in the java.io package.

- **File** class provides an abstract representation of file or directory
  - Constructors
    - File(String) creates a File object with the given path
    - File(String, String)
    - File(File, File)
  - Common methods
    - **boolean exists()** returns true if this file or directory exists
    - **boolean createNewFile()** and **boolean mkdir()** physically creates this file or directory and returns true if the operation succeeded

            - If the file or directory already exists it will return false
        - **boolean renameTo(File)** renames this file with the new file and returns true if the operation succeed
- **FileReader** convenient class used to read a File
- **FileWriter** convenient class used to write a File
    - Constructors
        - FileWriter(File)

Java File I/O (NIO.2)

Examples here: http://jdoodle.com/a/1Kov

Use Path interface to operate on file and directory paths

- Path interface does not check the validity
- Can be absolute or relative
- Created using
  - static methods of the **Paths** class: **Paths.get(String, String...)**
  - Static methods of the **FileSystems** class: **FileSystems.getDefault().getPath()**
- It is possible to convert to and from a File object using:
  - Instance method from Path: **Path.toFile()**
  - Instance method from File: **toPath()**
- Methods
  - **Path getFileName()** returns the last name (basename)
  - **Path getParent()** return the parent path or null if there is no parent
  - **Path getRoot()** return the root element of the path or null if there is no root (ex, “/”, “C:\”)
  - **Int getNameCount()** returns the number of components in the path
  - **Path getName(int)** returns the element from the path at the specified index (root is not counted)

  - **Path subpath(int, int)** returns a relative path of this path from first index inclusive to last index exclusive
  - **boolean isAbsolute()**
  - **Path toAbsolutePath()**
  - **Path normalize()** returns a normalized path without the redundancies
  - **Path relativize()** returns the relative path from this path to the specified one.
    ⚠ Both paths must be relative or both absolute otherwize IllegalArgumentException is thrown
  - **Path resolve(Path)**
  - **Path resolveSibling(Path)**

Use Files class to check, read, delete, copy, move, manage metadata of a file or directory

- Files.exists notExists
  - **boolean exists(Path, LinkOption...)** return true if the file or directory exists
    - The link option parameter indicates the API not to follow symbolic links the only possible value is **LinkOption.NOFOLLOW_LINKS**
    - If not parameter is specified, the symlinks are not followed
  - **boolean notExists(Path, LinkOption...)** return false if the file or directory exists true otherwize
  - ⚠ Both can return **false** if a symbolic link is present in the path and **NOFOLLOW_LINKS** is specified
- Reading files
  - **List<String> Files.readAllLines(Path)** returns a list of string containing all the files from a file
    - By default, the character set used is UTF-8
    - It is possible to specified a specific character sert using the following overloaded method: **List<String> Files.readAllLines(Path, Charset)**
  - **byte[] Files.readAllBytes(Path)** returns an array of bytes containing the file

  - ⚠ Both methods can throw an **IOException**
  - ⚠ Both methods ensure that the file is closed once it is entirely read or if an exception is thrown
- Deleting files
  - **void Files.delete(Path) throws IOException**
    - Delete the file and throws a NosuchFileException if does not exists (NoSuchFileException is a descendent of IOException)
  - **boolean Files.deleteIfExists(Path) throws IOException**
    - Does not throw any exception if the file does not exists
    - Returns true if the file has been deleted
- Copying files
  - **CopyOption** interface determines how a file will be copied or moved. It has two implementations
    - **LinkOption**
      - NOFOLLOW_LINKS
    - **StandardCopyOption**
      - ATOMIC_MOVE
      - COPY_ATTRIBUTES
      - REPLACE_EXISTING
  - The **copy()** method has three overloaded forms :
    - Path **copy**(Path source, Path target, CopyOption…options) throws IOException
    - long **copy**(Pathsource , OutputStream out) throws IOException
      - No options
      - Returns the number of bytes copied
    - long **copy**(InputStream in, Path target, CopyOption … options) throws IOException
      - Returns the number of bytes copied
- Moving files
  - Achieved using the move method from Files class:
    Path move(Path source, Path target, CopyOptions...options) throws IOException
    - Supported options are **REPLACE_EXISTING** and **ATOMIC_MOVE**
    - ⚠ If ATOMIC_MOVE is set the other option is ignored
- Reading metadata from files can be done using a set of methods provided by the **Files** class
  - FileTime getLastModifiedTime(Path, LinkOption...)
  - UserPrincipal getOwner(File, LinkOption…)
  - boolean isDirectory(Path, LikOption…)
  - boolean isRegularFile(Path, LikOption…)
  - boolean isSymbolicLink(Path)
  - boolean isReadable(Path)
  - boolean isWritable(Path)
  - boolean isHidden(Path)
  - boolean isSameFile(Path, Path)
- File attribute views is another way to get file attributes
  - Most important sub interface is BasicFileAttributes
  - To get attribute from a file the **getAttribute** method can be used:
    Object getAttribute(Path, String attributeName, LinkOption….)
    - attributeName is formatted as follows [type]:attributename
      By default the type is **basic**
  - The attributes can be retrieved in a bulk operation using three methods

        - **Map<String,Object> readAttributes(Path, String, LinkOption...>)**
          The second parameter specifies the requested attributes
          Ex: basic:*
        - **<V extends FileAttributeView> getAttributeView(Path, Class, LinkOption...)**
        - **<A extends BasicFileAttributes> readAttributes(Path, String, LinkOption...)**
          Works only with BasicFileAttributes or DosFileAttributes
    - To change the value of an attribute:
      Path setAttribute(Path, String, Object, LinkOption...)

Use Stream API with NIO.2

- The **lines()** method will read all lines from a file into a Stream<String>:
    - **Stream<String> lines(Path)**
    - **Stream<String> lines(Path, Charset)**
- The **list()** method from the Files class returns a Stream of Path corresponding to the files or directories contained in the current directory:
  Stream<Path> list(Path)
- The **walk()** method will be used to walk through the whole directory tree starting from this directory, there are two overloaded forms:
    - Stream<Path> walk(Path, FileVisitOption...)
    - Stream<Path> walk(Path, int, FileVisitOption...)
    - The parameters are:
        - FileVisitOption specifies whether the symbolic links must be followed:
            - FileVisitOption.FOLLOW_LINKS
        - Int depth specifies the maximum number of directory levels to walk through
    - ⚠ Symbolic links are not followed by default
    - ⚠ At least one element is returned: the given directory
- The **find()** method will search files matching the given BiPredicate in the directory tree starting from the given Path:
    - Stream<Path> find(Path, int, BiPredicate<Path, BasicFileAttribute>, FileVisitOption...)

- Compare legacy and NIO

Legacy Method	NIO.2 Method
file.exists()	Files.exists(path)
file.getName()	path.getFileName()
file.getAbsolutePath()	path.toAbsolutePath()
file.isDirectory()	Files.isDirectory(path)
file.isFile()	Files.isRegularFile(path)
file.isHidden()	Files.isHidden(path)
file.length()	Files.size(path)
file.lastModified()	Files.getLastModifiedTime(path)
file.setLastModified(time)	Files.setLastModifiedTime(path,fileTime)
file.delete()	Files.delete(path)
file.renameTo(otherFile)	Files.move(path,otherPath)
file.mkdir()	Files.createDirectory(path)
file.mkdirs()	Files.createDirectories(path)
file.listFiles()	Files.list(path)

Java Concurrency

Examples here: http://www.jdoodle.com/a/1Ltu

Create worker threads using Runnable, Callable and use an ExecutorService to concurrently execute tasks

- **Runnable**
    - **Functional interface** designed to have its implementers running in a separate thread
    - One abstract method: **void run()**
        - Takes no argument and returns no value
- **Callable**
    - Functional interface where implementers are intended to run on separate thread
    - One abstract method: **V call() throws Exception**
        - Returns a value and can throw a checked exception
- **Future**
    - Represents the result of an asynchronous task
    - The result wrapped in the Future can be retrieved using the get method
        - **V get() throws InterruptedException, ExecutionException**
            - ⚠ The invoking thread will be suspended until the task is executed
        - **boolean cancel(boolean)**
- **Executor**
    - Decouples task execution from task submission
    - Executes **Runnable** tasks using method:
        - **void execute(Runnable)**
- **ExecutorService** is a sub interface of **Executor**
    - Provides additional methods to
        - Execute tasks:
            - <T> Future<T> submit(Callable<T> task)
            - <T> Future<T> submit(Runnable task, T result)
            - Future<?> submit(Runnable task)
        - Shutdown the **Executor**
            - **shutdown()** initiate shutdown of the executor once all the threads have terminated
            - **shutDownNow()** kills the currently running threads and shutdown
    - Must be shut down when no longer needed to avoid Thread keeping running
    - Objects of type **ExecutorService** can be created from factory method in the **Executors** class
    - Methods:
        - **V invokeAny(Collection<Callable<V>>) throws ExecutionException, InteruptedException**
            - Executes a collection of tasks
            - Returns the result from the first completed one
            - Cancels the others
        - **List<Future<V>> invokeAll(Collection<Callable<V>>) throws ExecutionException, InteruptedException**
            - Executes a collection of tasks
            - Returns a List of futures containing the results

- ⚠ The elements of the returned list are in the same order as the iterator of the input collection

Identify potential threading problems among deadlock, starvation, livelock, and race conditions

- **Deadlock** is a situation where two or more threads are blocked forever waiting for a resource to be released by each other.
- **Starvation** is a situation where one or more threads are waiting for a resource to be released by other threads that are active and keeping the resource for a long time
- **Livelock** is a situation where multiple thread affects the results from each other
- **Race condition** is a situation are reading and modifying the same resources causing unpredictable results

Use synchronized keyword and java.util.concurrent.atomic package to control the order of thread execution

- Intrinsic lock: each object has a one and only one implicit lock called the **intrinsic lock**
- **Synchronized** method
  - They are preceded by the synchronized keyword
  - They take the **intrinsic lock** and release it when terminating the method
    ⚠ As a consequence, only one **synchronized** method can be called at a time
- **Synchronized** blocks
  - They are preceded by the synchronized keyword
  - They must specify the object that will provide the intrinsic lock
- ⚠ A thread will not be blocked trying to acquire a lock it already owns
  ➔ **reentrant synchronization**
- The **java.util.concurrent.atomic** package provides wrapper classes to support atomic operations:
  - **AtomicBoolean**, **AtomicLong**, **AtomicInteger**, **AtomicReference…** and many others
  - The constructors are
    - **AtomicXXX**() in this case the underlying value will be the default value of the underlying primitive type
    - **AtomicXXX(xxxx)** in this case the value will be the value of a given primitive type
  - Methods
    - **compareAndSet(val, new)** will set the object to the new value if the current value is == to the given value as a parameter (does not use equals)
    - **void set(xxx)** set the new value
    - **xxx getAndSet(xxx)** set the new value and returns the previous one
  - **AtomicLong** and **AtomicInteger** defines ohers useful methods:
    - **xxx addAndGet(val)** adds a value and return the updated one
    - **xxx getAndAdd(val)** adds a value and return the previous one
    - **xxx incrementAndGet()** increment by one and return the updated value
    - **xxx getAndIncrement()** increment by one and return the previous value
    - **xxx decrementAndGet()** decrement by one and return the updated value
    - **xxx getAndDecrement()** decrement by one and return the previous value
  - Al the atomic types **except AtomicBoolean** have the following methods that leverages the functional interfaces
    - **X updateAndGet(UnaryOperator<X>)** update the value and returns the updated value
    - **X getAndUpdate(UnaryOperator<X>)** updates the value and returns the replaced one

        - **X accumulateAndGet(X, BinaryOperator<X>)** updates the current value from applying a given binary operator to the current value and given value and returns the new value
        - **X getAndAccumulate(x, BinaryOperator<X>)** updates the current value from applying a given binary operator to the current value and given value and returns the previous value

Use java.util.concurrent collections and classes including CyclicBarrier and CopyOnWriteArrayList

- **CyclicBarrier** class is used when we need multiple threads to wait for each other before continuing their execution.
    - Constructors:
        - **CyclicBarrier(int)** creates a cyclic barrier for n parties to be synchronized
        - **CyclicBarrier(int, Runnable)** creates a barrier for n parties and execute a Runnable action once all of them have reached the barrier
    - **await()** methods suspends the calling thread until the last arriving thread has called the method or the barrier is broken.
      It returns the arrival order of the current thread
        - **int await() throws InterruptedException, BrokenBarrierException**
        - **int await(long, TimeUnit) throws InterruptedException, BrokenBarrierException, TimeoutException**
        - A barrier is broken
            - When the current thread or one of the others waiting thread is interrupted
            - When the timeout is reached
            - When the barrier is reset
    - **int getNumberWaiting()** number of threads currently waiting the barrier
    - **int getParties()** number of parties required to call the await method
    - **boolean isBroken()**
    - **void reset()** resets this barrier that will then be reusable even if it was broken
- **CopyOnWriteArrayList<T>** class is a list where a copy of the underlying array is done on each mutation operation ➜ writing in it is expensive
    - **iterator()** returns an iterator on the current underlying array of CopyOnWriteArrayList
      ⚠ Subsequent modifications of this List will not be visible because it
    - All methods from the List interface are implemented, in addition CopyOnWriteArrayList provides two additional methods:
        - **boolean addIfAbsent(T)** adds the given element if it is not allread present and returns true if the element could be added
        - **Int addAllAbsent(Collection<T>)** adds all the elements of the given collection that are not already present and returns the number of methods added

Use parallel Fork/Join Framework

- Concepts
    - The parallel fork join framework is designed for works that can easily be divided into smaller pieces recursively.
    - It uses a work stealing algorithm ➜ when a thread is running out of work it can steal work from busy threads
- **ForkJoinPool** class is the main component of the framework
    - It's an implementation of the **ExecutorService** interface

- It can be created by
    - Static method **commonPool()** returns the common pool used by **ForkJoinTask**
        - Help reduce resources because the threads are slowly reclaimed in idle periods and restored when needed
    - Constructors give more control
        - **ForkJoinPool()** creates a pool with parallelism equal to the number of available processors to the JVM
        - **ForkJoinPool(int)** creates a pool specifying a level of parallelism
- Methods
    - **<T> invoke(ForkJoinTask<T>)** performs the work and return its result when completed

- **ForkJoinTask** is an abstract class representing tasks running in a **ForkJoinPool**
    - Methods
        - **fork** asynchronously executes this task in the pool
        - **join** retrieves the result of computation when done
        - **invoke** performs this task, awaits completion, and returns the result
        - **invokeAll** performs all the given tasks and awaits completion
    - Subclasses
        - **RecursiveAction** used for actions that do not return results
        - **RecursiveTasks** used for actions that return results
        - Both have one abstract method that must be implemented:
            - **compute()** this method should be implemented with the following algorithm:

```
If the work is small enough
    Perform it directly
Else
    Split the work into smaller pieces
    Invoke those pieces and wait for results
```

Use parallel Streams including reduction, decomposition, merging processes, pipelines and performance.

- Facilitates parallel execution by rephrasing the processing into a pipeline
- It should produce the same result regardless if it is executed sequentially or in parallel
- The benefit of parallelism may be diminished if
    - The stream contains stateful operations
    - The stream has a defined encounter order
- Two ways to create a parallel stream:
    - myCollection.**parallelStream()**
    - myCollection.**stream().parallel()**
- The **isParallel()** method checks whether a stream is parallel or not
- **reduce()**: in addition to the two forms mentioned previously the **reduce** method has an additional form:
    - reduce(T identity, BIFunction<U, ? super T,U> accum, BinaryOperator<T>, combiner)
      The third parameter is a BinaryOperator aimed to combine the results of the parallel substreams
- **collect(Supplier<R> sup, BIConsumer<R, ? super T> acc)** also has a new form
    - collect(Supplier<R> sup, BIConsumer<R, ? super T> acc , BiConsumer<R,R> comb)
- **collect(Collector)<? Super T, A, R>)** can use a concurrent collector ex:
    - groupingByConcurrent
    - toConcurrentMap

Building Database Applications with JDBC

Describe the interfaces that make up the core of the JDBC API including the Driver, Connection, Statement, and ResultSet interfaces and their relationship to provider implementations

- Driver interface
  - Implemented by all drivers
  - Loaded automatically by the DriverManager class
  - Most noticeable method is connect
    it should not be called directly
  - The way a driver object can be obtained is dependent on the vendors implementation
- Connection interface
  - Connection objects can be obtained by DriverManager::getConnection
- Statement interface
  - Objects can be created by Connection::createStatement
  - When the connection of a statement is closed, the statement itself is also closed
  - Three methods to execute a query:

Method	Return type	Value for INSERT, UPDATE, DELETE	Value for SELECT
Statement.execute()	boolean	false	true
Statement.executeQuery()	ResultSet	n/a	The result of the select query
Statement.executeUpdate()	int	The number of impacted rows	n/a

  - The execute method can be used for both writing and selecting:
    - statement.getResultSet() can be used to get the result
    - statement.getUpdateCount() can be used to get the number of impacted rows
- ResultSet interface
  - The result set is automatically closed when
    - Its statement is closed
      ⚠ it has no effect when called on an already closed result set
    - Its statement is re-executed
  - ResultSet has three types types (int constants):
    - **TYPE_FORWARD_ONLY**: (default) scrollable only in one direction
    - **TYPE_SCROLL_INSENSITIVE**: scrollable in both directions, does not see database changes
    - **TYPE_SCROLL_SENSITIVE**: scrollable in both directions, reflects the concurrent database modifications (not supported by every provider)
  - ResultSet has two concurrency modes ‘int constants)
    - **CONCUR_READ_ONLY** (default) the resultSet cannot be updated
    - **CONCUR_UPDATABLE** the resultSet can be updated
  - The types and concurrency mode can be chosen when creating a Statement:
    - Connection.createStatement(int type, int concurrencyMode)
    - Connection.prepareStatement(String sql, int type, int concurrencyMode)
    - Connection.prepareCall(String sql, int type, int concurrencyMode)
  - ⚠ The numbering of result set columns start with 1

- Methods to navigate into a ResultSet:

Method	Type	Description
next()	boolean	False when there are no more rows after
previous()	boolean	False when there is no more rows before
beforeFirst()	void	Positions the result set before the first row
afterLast()	void	Positions the result set after the last row
absolute(int pos)	boolean	Positions the cursor to a specific position: When positive the position is understood from the first row When negative the numbering is understood from the last row Returns false if the position is before or after the last row
relative(int pos)	boolean	Positions the cursor to a specific position from the current one Returns false if the position is before or after the last row
first()	boolean	Positions the cursor on the first row returns false if the cursor has no rows
last()	boolean	Positions the cursor on the last row returns false if the cursor has no rows

⚠ The row numbering starts with 1

- JDBC drivers implementations
  - Must implement all the interfaces described earlier
  - Four categories:
    - Type 1: Mapping of another data access API such as ODBC
    - Type 2: Implementation relying on native API (Oracle OCI)
    - Type 3: communication using a middleware to communicate with the DB
    - Type 4: pure java implementation of the DB network protocol

Identify the components required to connect to a database using the DriverManager class including the JDBC URL

- DriverManager class can register as many drivers as requested ➔ the relevant target will be determined by the connection URL
- The getConnection method has three overloaded forms: (all forms throws SQLException)
  - Connection getConnection(String url)
  - Connection getConnection(String url, String user, String pass)
    ⚠ if user/pass is specified both in the url and the argument, which will take precedence depends on the implementation of the DB vendor
  - Connection getConnection(String url, Properties properties)
    ⚠ the supported set of properties depends on the vendor's implementation

Submit queries and read results from the database including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections:

- PreparedStatement
  - Created by Connection prepareStatement(String)
    - The parameter is the SQL query to be executed
  - The statement is executed by
    - **executeQuery()** with no parameters
      if a parameter is specified, an exception is thrown at runtime
- Statement
  - Created by Connection createStatement()
    - No parameter
  - The statement is executed by
    - **executeQuery(String)** the parameter is the query to be executed
- ResultSet
  - **boolean next()** returns true when there are no more rows to read

Localization

Examples here; http://www.jdoodle.com/a/1Lwt

Read and set the locale by using the Locale object

- A Locale object can be built using
  - A Locale.Builder object
    ➜ **new Locale.Builder().setRegion("FR").setLanguage("fr").build()**
  - Three public constructors
    - **Locale**(String language)
    - **Locale**(String language, String country)
    - **Locale**(String language, String country, String variant)
  - Locale.forLanguageTag(String local)
    - The string must comply with IETF BCP 47 standard:
      <ISO language alpha2>-<ISO region alpha2>
      "-" not"_"
      Ex: en-US
- Static fields representing some locales are also available in the Locale class
  - If the name is a country the language is also included
    Ex: Locale.FRANCE => fr_FR
  - If the name is a language only the language is included
    EX: Locale.FRENCH => fr
- ⚠ The locale object just represent an identifier ➜ no check is done against the validity of the locale

Create and read a Properties file

- Characteristics of a property file
  - Has the extension ".properties"
  - Contains key-value pairs separated by "=", ":" or " "
    The most used is "="
    ⚠ when "=" or ":" are used, surrounding white spaces are ignored
- ResourceBundle
  - ResourceBundle is an **abstract** class the following implementations are provided
    - ListResourceBundle (also abstract)
    - PropertyResourceBundle
  - A resource bundle is a property file
  - getBundle has two forms:
  - ResourceBundle.getBundle(String basename)
    - Gets the property file with specified basename **for the default local**
  - Two methods can be used to get the keys from a **ResourceBundle**
    - Enumeration<String> getKeys()
    - Set<String> keySet()
  - The value for a given key can be retrieved
    - String getString(String key)
      ⚠ If the key does not exists ➜ MissingResourceException is thrown

Build a resource bundle for each locale and load a resource bundle in an application

- ResourceBundle is used for properties dependant from the Locale. It has 2 descendents:
  - **PropertyResourceBundle** handling string properties
  - **ListResourceBundle** handling objects
- Resource bundles belong to a family of files that have the same base name in common and have additional components that identifies their locale
- Many overloaded versions of static getBundle allows retrieving a resource bundle from name and locale:
  basename_<locale>.properties
  <locale> can be <language>_<region> or <language>
  ⚠ As opposite to locale "_" is used (not "-")
  Ex: exam_fr_FR.properties or exam_fr.properties
- If the resource bundle with specified locale does not exists:
  - Find the closest match
  - Find the bundle for default locale
  - Find the one with only the basename
- Ex: fr_CA is needed the default locale is en_US ➜ the order will be the following:
  - bundle_fr_CA
  - bundle_fr
  - bundle_en_US
  - bundle_en
  - Bundle
- **PropertyResource** bundle is backed by a properties file
  - To create a PropertyResourceBundle from a properties file **in the class path**:
    PropertyResourceBundle bdl = (PropertyResourceBundle) ResourceBundle.getBundle((basename)
  - To create a PropertyResourceBundle from a properties file **in the current directory**:
    PropertyResourceBundle bdl = (PropertyResourceBundle) ResourceBundle.getBundle((new FileInputStream("file.properties"))
    PropertyResourceBundle bdl = (PropertyResourceBundle) ResourceBundle.getBundle((new FileReader("file.properties"))
- **ListResourceBundle** is backed by a java class. It is an **abstract** class its subclasses must implement:
  - Object[][] getContents()
  - The content is backed by a two dimensional array where each inner array is a key value pair

Assume the following:

- **Missing package and import statements**: If sample code do not include package or import statements, and the question does not explicitly refer to these missing statements, then assume that all sample code is in the same package, or import statements exist to support them.
- **No file or directory path names for classes**: If a question does not state the file names or directory locations of classes, then assume one of the following, whichever will enable the code to compile and run:
  - All classes are in one file
  - Each class is contained in a separate file, and all files are in one directory
- **Unintended line breaks**: Sample code might have unintended line breaks. If you see a line of code that looks like it has wrapped, and this creates a situation where the wrapping is significant (for example, a quoted String literal has wrapped), assume that the wrapping is an extension of the same line, and the line does not contain a hard carriage return that would cause a compilation failure.
- **Code fragments**: A code fragment is a small section of source code that is presented without its context. Assume that all necessary supporting code exists and that the supporting environment fully supports the correct compilation and execution of the code shown and its omitted environment.
- **Descriptive comments**: Take descriptive comments, such as "setter and getters go here," at face value. Assume that correct code exists, compiles, and runs successfully to create the described effect.