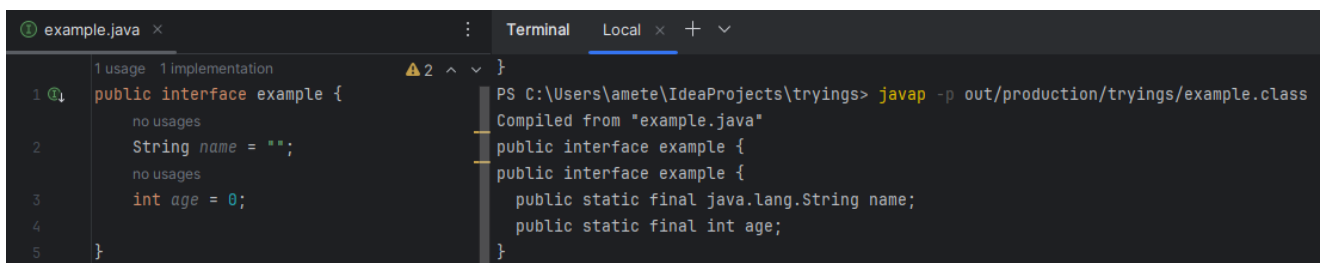


Abstract Classes

- Abstract methods are methods that must be implemented by inherited classes.
- We can turn off overriding a function with the final keyword. “public final String ...”

Interfaces

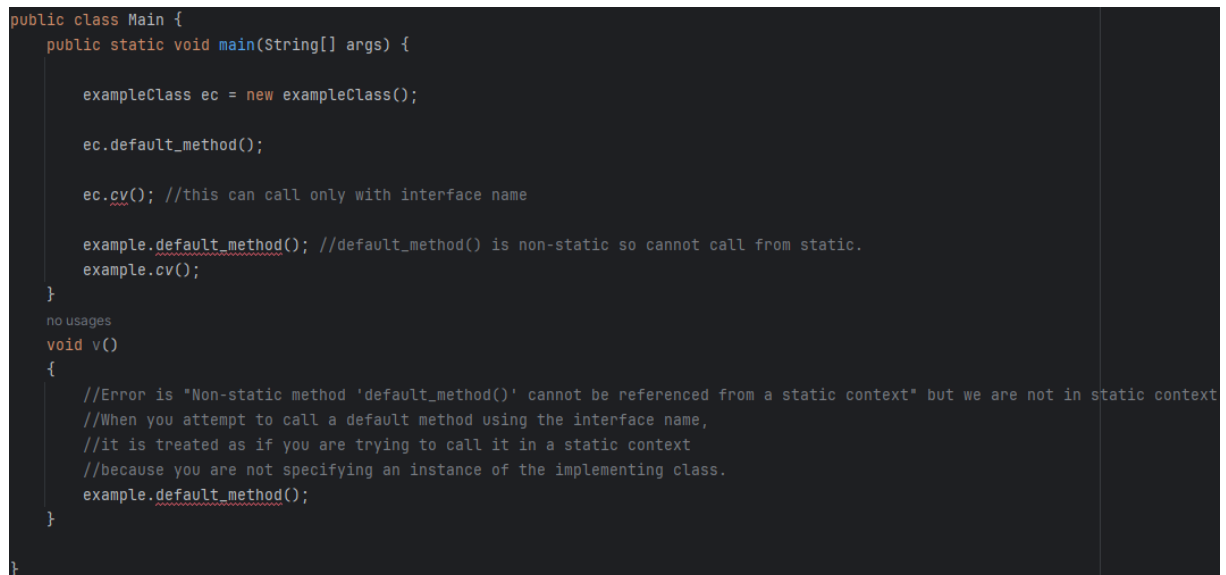
- In an interface have only method signature and specified attributes creates automatically as “public static final”. When we checked with javap -p from terminal we can see object defined as public static final while we define without these keywords.



```
example.java x
1 usage 1 implementation
1 public interface example {
  no usages
2   String name = "";
  no usages
3   int age = 0;
4 }
5 }

Terminal Local x + v
PS C:\Users\amete\IdeaProjects\tryings> javap -p out/production/tryings/example.class
Compiled from "example.java"
public interface example {
  public static final java.lang.String name;
  public static final int age;
}
```

- We can create method with body in an interface with **Default Method**. In addition, that method does not need to be override at inherited classes. But when want to override, it can be override.
- Methods with body can create with **public static** methods (they can only be static, they implicitly public) too. Only can call these methods with interface name, no other way.



```
public class Main {
  public static void main(String[] args) {

    exampleClass ec = new exampleClass();

    ec.default_method();

    ec.cv(); //this can call only with interface name

    example.default_method(); //default_method() is non-static so cannot call from static.
    example.cv();
  }
  no usages
  void v()
  {
    //Error is "Non-static method 'default_method()' cannot be referenced from a static context" but we are not in static context
    //When you attempt to call a default method using the interface name,
    //it is treated as if you are trying to call it in a static context
    //because you are not specifying an instance of the implementing class.
    example.default_method();
  }
}
```

- **concrete private, concrete private static** or **concrete static** methods can create with body in interface. These methods can call from **default method** and each other but cannot call or seen from inherited class.

```
public interface example {
    3 usages
    default void default_method()
    {
        psv();
        pv();
        cv();
    }

    3 usages
    private static void psv()
    {
        default_method(); //non-static method cannot call from static method
        pv(); //non-static method cannot call from static method
        cv();
    }

    3 usages
    private void pv()
    {
        default_method();
        psv();
        cv();
    }

    3 usages
    public static void cv()
    {
        default_method(); //non-static method cannot call from static method
        psv();
        pv(); //non-static method cannot call from static method
    }
}
```

Interfaces vs Abstract

	Abstract Class	Interface
An instance can be created from it	No	No
Has a constructor	Yes	No
Implemented as part of the Class Hierarchy. Uses Inheritance	Yes (in extends clause)	No (in implements clause)
records and enums can extend or implement?	No	Yes
Inherits from java.lang.Object	Yes	No
Can have both abstract methods and concrete methods	Yes	Yes (as of JDK 8)
Abstract methods must include abstract modifier	Yes	No (Implicit)
Supports default modifier for it's methods	No	Yes (as of JDK 8)
Can have instance fields (non-static instance fields)	Yes	No
Can have static fields (class fields)	Yes	Yes - (implicitly public static final)

Generics

- When defining a generic class, you can limit the classes that the generic class will accept with “**public class** className<T **extends** otherClassName>”extends.
- A similar limitation can be made as follows;

```
public class GenericExample<T> {  
  
    1 usage  
    private T value;  
  
    no usages  
    public void setValue(T value) {  
        if(value instanceof Integer || value instanceof String) {  
            this.value = value;  
        } else {  
            throw new IllegalArgumentException("Only Integer or String types acceptable");  
        }  
    }  
}
```

- If a method take a parameter as generic type, upper and lower bounds of this variable can define.

Argument	Example	Description
unbounded	List<?>	A List of any type can be passed or assigned to a List using this wildcard.
upper bound	List<? extends Student>	A list containing any type that is a Student or a sub type of Student can be assigned or passed to an argument specifying this wildcard.
lower bound	List<? super LPASStudent>	A list containing any type that is an <u>LPASStudent</u> or a super type of <u>LPASStudent</u> , so in our case, that would be Student AND Object.

```
public static void printList(List<? super LPASStudent> students)
```

- When use generic types if do not use diamond operator (raw using), variable will accept every type. For example "List list = new ArrayList<>();" list.add(1); list.add("hi"); both of these will accept. We can close this with <?> expression. Some of things cannot perform on Objects which defined <?> such as add.

```
public static void main(String[] args) {

    List rawList = new ArrayList<>();
    rawList.add("1");
    rawList.add(1);

    List<?> unboundedList = new ArrayList<>(rawList);
    for(Object o : unboundedList)
        System.out.println(o + "-" + o.getClass());

    unboundedList.add("2"); //error
    unboundedList.add(2);   //error
}
```

- When generic types used, these generic types changes to upper or lower bound of generic type or changes with Object while converting to bytecode or .class file in compile-time. This process called as **type erasure**.
 - For example these method signatures are same;

```
public static void printList(List<Integer> students)
public static void printList(List<String> students)
```

This is how they are handled at compile-time.

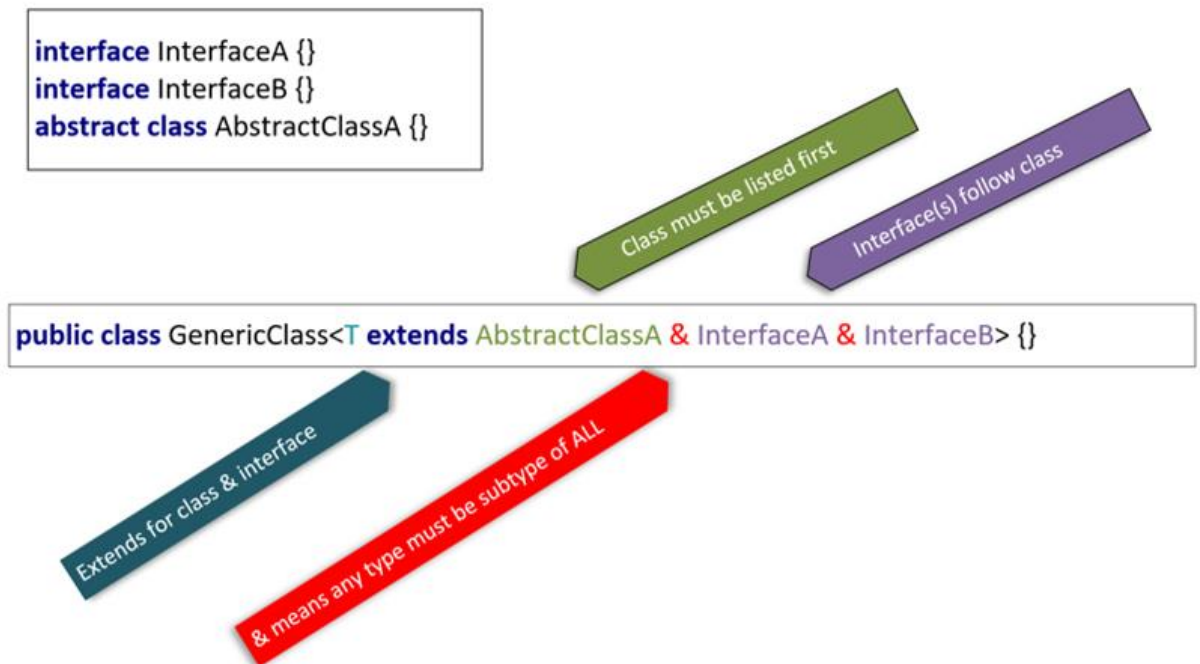
```
public static void printList(List<Object> students)
```

- So, if generic types changes in compile-time why doesn't a code like "String str = list.get(0)" give an error? Reason of this, does implicitly casting in runtime by JVM. JVM checks object can castable or not, if it possible, JVM does it.

```
public static void main(String[] args) {

    List<String> stringList = new ArrayList<>();
    stringList.add("Hello");
    String firstItem = /* implicitly */ /* (String) */ stringList.get(0);
}
```

- More than one upper bound can define as follows.



All these types must be implemented by T.

Inner Class

- If you want to instantiate an instance from inner class, you must first have an instance from enclosing class.

```

var genericEmployee = new StoreEmployee();
var comparator = genericEmployee.new StoreComparator<>();
storeEmployees.sort(comparator);

```

or

```

var comparator = new StoreEmployee().new StoreComparator<>();
storeEmployees.sort(comparator);

```

- They can reach to enclosed area. *Local classes take copies of values of variables which exist in enclosed area, and it uses these copies.* That's why when change a variable in local class, this change does not effect to variables in enclosed area. Therefore, this situation may cause problems. That's why java can give permission only **final** or **effectively final** variables in local classes.

Local Class

- They are defined inside a method, loop, or if statement. Second rule of above (Inner Class) also applies here.

Anonymous Class

- A local class which does not have name. Anonymous classes provide make shorter form your code. You can implement a class and instantiate an instance from it at the same time. They like local classes so this can reach to enclosed class and reached variables must be final or effectively final. So, rules of local classes also applies here. But it has no name.
- An anonymous class instance which inherit MyRunnable interface/class and override its run() method.

```
Thread myRunnableThread = new Thread(new MyRunnable() {  
    @Override  
    public void run() {  
        System.out.println("From the anonymous class implementation");  
    }  
});
```

Lambda Expressions

The generated Lambda Expression	Comparator's Abstract Method
(o1, o2) -> o1.lastName().compareTo(o2.lastName())	int compare(T o1, T o2)

- To use lambda expressions, you need an interface that have **FunctionalInterface** annotation.
- Abstract method of Functional Interface implements as lambda expression.

```
@FunctionalInterface  
public interface Lambda {  
  
    //it can have only one abstract method  
    //this method implements with lambda  
    //(var1, var2, ...) -> operations,  
    //another usage  
    //(var1, var2, ...) -> {  
    // operations with multiple line then return  
    // }  
    //operations must return a result as abstract method's return type  
    1 usage  
    boolean target(int num);  
  
    //it can have default method  
    1 usage  
    default void concreteDefault()  
    {  
        System.out.println("Concrete Default Method");  
    }  
  
    //it can have static method  
    no usages  
    static void concreteStatic()  
    {  
        System.out.println("Concrete Static Method");  
    }  
}
```

```
public class Test {  
  
    //this applies given lambda  
    1 usage  
    public static void tst(Lambda a)  
    {  
        int num = 1;  
        System.out.println(a.target(num));  
        a.concreteDefault();  
    }  
}
```

```

public static void main(String[] args) {

    //usage lambda expression
    Test.tst((num) -> num%2 == 0);

    //we can obtain a Lambda instance with anonymous class initialization
    //,and we can use this as instance
    Lambda lambda = (num) -> num%2 == 0;

    System.out.println(lambda.target( num: 5));

    lambda.concreteDefault();

    Lambda.concreteStatic();

    Test.tst(lambda);
}

```

- As with inner classes, variables in the enclosing scope can be accessed in lambda expressions, but they must be **final** or **effectively final**.

- | Lambda Expression | Description |
|--|---|
| element -> System.out.println(element); | A single parameter without a type can omit the parentheses. |
| (element) -> System.out.println(element); | Parentheses are optional. |
| (String element) -> System.out.println(element); | Parentheses required if a reference type is specified. |
| (var element) -> System.out.println(element); | A reference type can be var. |

- | Lambda Expression | Description |
|---|--|
| (element) -> System.out.println(element); | <p>An expression can be a single statement.</p> <p>Like a switch expression, that does not require yield for a single statement result, the use of return is not needed and would result in a compiler error.</p> |
| (var element) -> {
char first = element.charAt(0);
System.out.println(element + " means " + first);
}; | <p>An expression can be a code block.</p> <p>Like a switch expression, that requires yield, a lambda that returns a value, would require a final return statement.</p> <p>All statements in the block must end with semi-colons.</p> |

- There is a package named **java.util.function** that have **Functional Interfaces**.

Some interfaces it has and their usages;

Interface Category	Basic Method Signature	Purpose
Consumer	void accept(T t)	execute code without returning data
Function	R apply(T t)	return a result of an operation or function
Predicate	boolean test(T t)	test if a condition is true or false
Supplier	T get()	return an instance of something

Category of Interface	Convenience method example	Notes
Function	function1. andThen (function2)	Not implemented on IntFunction , DoubleFunction , LongFunction
Function	function2. compose (function1)	Only implemented on Function & UnaryOperator
Consumer	consumer1. andThen (consumer2)	
Predicate	predicate1. and (predicate2)	
Predicate	predicate1. or (predicate2)	
Predicate	predicate1. negate ()	

- Another types of Function interface. Value that will be return is always write in the end of definition. So, the **R** ones below.

Interface Name	Method Signature	Interface Name	Method Signature
Function<T,R>	R apply(T t)	UnaryOperator<T>	T apply(T t)
BiFunction<T,U,R>	R apply(T t, U u)	BinaryOperator<T>	T apply(T t1, T t2)

Method References

- It a form of lambda expressions more compact and easier.
- If you are using a lambda expression for just perform a method. You can use method reference.

For example;

```
System.out.println(square((num) -> Math.sqrt(num), val: 5.0));
System.out.println(square(Math::sqrt, val: 5.0));
```

Method reference types

Type	Syntax	Method Reference Example	Corresponding Lambda Expression
static method	ClassName::staticMethodName(p1, p2, ... pn)	Integer::sum	(p1, p2) -> p1 + p2
instance method of a particular (Bounded) object	ContainingObject::instanceMethodName(p1, p2,...pn)	System.out::println	p1 -> System.out.println(p1)
instance method of an arbitrary (Unbounded) object (as determined by p1)	ContainingType[=p1] ::instanceMethodName(p2, ... pn)	String::concat	(p1, p2) -> p1.concat(p2).
constructor	ClassName:: new	LPASStudent:: new	() -> new LPASStudent()

Examples

	No Args	One Argument		
Types of Method References	Supplier	Predicate	Consumer	Function <u>UnaryOperator</u> et. al.
Reference Type (Static)				
Reference Type (Constructor)	Employee:: new		n/a	Employee:: new
Bounded Retriever (Instance)			System.out::println	
Unbounded Retriever (Instance)	n/a	String::isEmpty	List::clear	String::length

	Two Arguments		
Types of Method References	<u>BiPredicate</u>	<u>BiConsumer</u>	<u>BiFunction</u> <u>BinaryOperator</u> et. al.
Reference Type (Static)			Integer::sum
Reference Type (Constructor)			Employee:: new
Bounded Retriever (Instance)		System.out::printf	new Random()::nextInt
Unbounded Retriever (Instance)	String::equals	List::add	String::concat String::split

Some comparator's additional helper methods

- These methods can using as method reference too

Type of Method	Method Signature
static	Comparator comparing (Function <u>keyExtractor</u>)
static	Comparator <u>naturalOrder</u> ()
static	Comparator <u>reverseOrder</u> ()
default	Comparator thenComparing (Comparator other)
default	Comparator thenComparing (Function <u>keyExtractor</u>)
default	Comparator reversed ()

Collections

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Set

HashSet

- It can contain only one null element.
- There are no duplicates. Comparable must implement for duplicates control.
- It is not ordered.

TreeSet

- It is a red-black tree.
- It is ordered.

Some TreeSet methods;

Element passed as the argument	Result From Methods			
	floor(E) (<=)	lower(E) (<)	ceiling(E) (>=)	higher(E) (>)
In Set	Matched Element	Next Element < Element or null if none found	Matched Element	Next Element > Element or null if none found
Not in Set	Next Element < Element	Next Element < Element or null if none found	Next Element > Element	Next Element > Element or null if none found

sub set methods	inclusive	description
<u>headSet</u> (E <u>toElement</u>) <u>headSet</u> (E <u>toElement</u> , boolean inclusive)	<u>toElement</u> is exclusive if not specified	returns all elements less than the passed <u>toElement</u> (unless inclusive is specifically included).
<u>tailSet</u> (E <u>fromElement</u>) <u>tailSet</u> (E <u>toElement</u> , boolean inclusive)	<u>fromElement</u> is inclusive if not specified	returns all elements greater than or equal to the <u>fromElement</u> (unless inclusive is specifically included).
<u>subSet</u> (E <u>fromElement</u> , E <u>toElement</u>) <u>subSet</u> (E <u>fromElement</u> , boolean <u>fromInclusive</u> , E <u>toElement</u> , boolean <u>toInclusive</u>)	<u>fromElement</u> is inclusive if not specified, <u>toElement</u> is exclusive if not specified	returns elements greater than or equal to <u>fromElement</u> and less than <u>toElement</u> .

When would you use a TreeSet?

The TreeSet does offer many advantages, in terms of built-in functionality over the other two Set implementations, but it does come at a higher cost.

If your number of elements is not large, or you want a collection that's sorted, and continuously re-sorted as you add and remove elements, and that shouldn't contain duplicate elements, the TreeSet is a good alternative to the ArrayList.

Map

- Just map ...

View

- It shows a specific view of existing data without any copying or creating a new data structure.
- This improves performance and makes data easier to access, but changes to the view affect the original data. However, some view types only allow read operations.

Examples:

SubList View: A sub list that obtained with using ***subList()*** method in List Collections. It represents a view of the original list.

MapView: Views that obtained with using ***keySet()***, ***values()*** or ***entrySet()*** methods.

Read-Only View: Some collection types provide only read purposed views.

EnumSet and EnumMap

- These using with only enum types.
- They designed for keep set data that contain enum constants in a high performance and optimized way.
- They are represented by bit vectors consisting of 0s and 1s. Since the operations are performed with bit mathematics, they are very fast and consume very little memory.
- They are very efficient.

Immutable

- *Immutable* object is a type of object in which the state of the data it contains cannot be changed once created.
- For an object to be Immutable it must have the following attributes;
 - **Immutable States:** Attributes can instantiate once then they cannot be changed later. So, object states remain constant after initialization.
 - **Attributes Must Be Private:** Attributes must not be accessible directly. Instead, they must initialize when object initialization or another specific way.
 - **Immutable methods:** There must no methods that can change the states of the object.
- Immutable objects prefer in Java for some reasons:
 - **Thread Safety:** Immutable objects can be accessed safely from multiple threads. Because their internal states are not changeable that's why there is no need synchronization.
 - **Cache and Performance Optimization:** Immutable objects can be optimized memory usage by sharing variables with the same values rather than initializing a new one.
 - **Reliability:** Immutable objects prevent to change it's content mistakenly or by malicious people.
-

```
String xArgument = "This is all I've got to say about Section ";
StringBuilder zArgument = new StringBuilder("Only saying this: Section ");
doXYZ(xArgument, y: 16, zArgument);
}
1 usage
private static void doXYZ(String x, int y, final StringBuilder z) {

    final String c = x + y;
    System.out.println("c = " + c);
    x = c;
}
```

At the first glance, it thinks that xArgument must change after the doXYZ method call. But it won't happen. Since strings are immutable, xArgument will not change.

Final Modifier

- Final instance methods cannot be overridden.
- Final static methods are not visible to subclasses.
- It cannot assign again to final attributes.
- They can initialize in constructor, defined scope, instance or static initializer block or global scope.
- Final classes cannot be inherited, but final classes can inherit from another class.
- Final method parameter cannot change inside of method.
-

```
private static StringBuilder doXYZ(final StringBuilder z) {  
    z.append("15");  
    return z;  
}
```

Let's z be "20". What is the output? "20" or "2015"? Output will be "2015" because **final keyword specified only reference immutability**. Content can changeable but reference not. For example if we have another StringBuilder named x and when we do z = x, this will throw an error.

Immutable Class

- Make instance fields **private final**.
- Don't define setter methods.
- Make **defensive copies** on getters.

Method Hiding

- It expresses that subclasses re-define static methods defined in superclasses with the same signature. In this situation, the new method in the subclasses hides the method in the superclass. However, this rule cannot be valid for instance methods because instance methods be subject to overriding. Overriding cannot perform on static methods because instance methods and static methods have different behaviors.

difference between instance and static methods;

Binding Mechanism: Dynamic binding occurs for instance methods. Early binding occurs for static methods.

Suppose we have definition as "BaseClass obj = new ChildClass();", and we also have two methods with the same signature except one is static and one is an instance. Overriding operations is perform on dynamic binding. When code working, instance method will call from ChildClass because of obj is an instance of ChildClass. However, if method is static it will call from BaseClass because of early binding occurs for static methods. This feature distinguishes method hiding and overriding from each other.

Defensive Copies

- If send a mutable type to immutable object defensive copy must be performed.
- It done by calling the constructor with the new keyword.
- Purpose is to protect the object, not entire content of the object.
- We can use on getters and constructors.

Suppose an array called kids that keeps kids of a person, and getKids() method returns that array. If we coded as "return kids;", kids can be intervened from outside because we are returning a reference. That's why we must be coded like "return new ArrayList<>(kids)".

Passed parameters to constructors must use like "this.start = new Date(start.getTime());" for prevent changes to the object later with passed parameters.

Unmodifiable Collections

- They do not support modification methods such as add, remove, clear, replace, sort, etc.
- These collections use generally for read-only view of another collection such as wrappers.
- You cannot modify of unmodifiable collections directly. However, original collection still can modifiable. This differs from immutable in that it is modifiable.

```
public static void main(String[] args) {  
  
    List<String> list = new ArrayList<>();  
    list.add("mete");  
    list.add("ahmet");  
    list.add("yakar");  
  
    List<String> list2 = new ArrayList<>();  
    list2.add("mete2");  
    list2.add("ahmet2");  
    list2.add("yakar2");  
  
    List<String> list3 = new ArrayList<>();  
    list3.add("mete3");  
    list3.add("ahmet3");  
    list3.add("yakar3");  
  
    List<List<String>> lists = new ArrayList<>();  
    lists.add(list);  
    lists.add(list2);  
  
    //List.copyOf() is doing a defensive copy  
    //So, when change object that we sent as parameter, list will not change,  
    //but if we change the content of element of that parameter, list will be changed  
    List<List<String>> unmodifiableList = List.copyOf(lists);  
  
    System.out.println("output1 -> " + unmodifiableList);  
  
    list.remove(index: 0);  
    lists.add(list3);  
    System.out.println("output2 -> " + unmodifiableList);  
    System.out.println("output3 -> " + lists);  
}
```

```
output1 -> [[mete, ahmet, yakar], [mete2, ahmet2, yakar2]]  
output2 -> [[ahmet, yakar], [mete2, ahmet2, yakar2]]  
output3 -> [[ahmet, yakar], [mete2, ahmet2, yakar2], [mete3, ahmet3, yakar3]]
```

- Mutator methods throws an exception called “***UnsupportedOperationException***”

-

	Unmodifiable Copy of Collection	Unmodifiable View of Collection
List	<u>List.copyOf</u> <u>List.of</u>	<u>Collections.unmodifiableList</u>
Set	<u>Set.copyOf</u> <u>Set.of</u>	<u>Collections.unmodifiableSet</u> <u>Collections.unmodifiableNavigableSet</u> <u>Collections.unmodifiableSortedSet</u>
Map	<u>Map.copyOf</u> <u>Map.entry(K k, V v)</u> <u>Map.of</u> <u>Map.ofEntries</u>	<u>Collections.unmodifiableMap</u> <u>Collections.unmodifiableNavigableMap</u> <u>Collections.unmodifiableSortableMap</u>

Instance Initializer

- It is the code block that run before the constructor when object initialization. It doesn't matter how many instance initializers are in the class. They run from top to bottom, the constructor runs after all instance initializers have run.
- They can use for initializing final variables.

Static Initializer

- It is the code block that run when class initializing (loading bytecode to memory by JVM). It has working principle same as instance initializer. Firstly, all static initializers run, secondly instance initializers, finally the constructor run.
- It can use for data or log information.
- If you need to open a database connection or another source connection, you can use static initializer. That's way when static method or class attributes use these sources, sources be ready.
- If you need to do some calculations for usage of static attributes or static methods, you can use static initializer.

Initializers can use attributes that's why entire attributes definitions must be done before initializers

Record Constructor

- **Canonical Constructor:** It must have all attributes. Other constructors can be implemented with using this constructor. It is called as **Long Constructor** too.
- **Compact Constructor:** It can only use on records. It is the easier way to use *Canonical*. Generally, in this constructor, operations such as control and editing are performed on the parameters given to the *Canonical* constructor. This constructor then automatically calls the *Canonical* constructor it created.

Enum Constructor

- When an enum class initialized, every enum constant calls a constructor. Because these constant values are static final values, and they create once when class initialization.
- If there is no defined constructor, java creates one. However, If there is a constructor with parameter, it is necessary to write a no parameter constructor as well. We can overcome this problem by giving parameters to enum constants.
- In addition, we can write toString() or other additional methods.

```
no usages
GEN_Z( startYear: 2001, endYear: 2025) {
    @Override
    public String toString() { return "Generation Z"; }
},
no usages
MILLENNIAL( startYear: 1981, endYear: 2000),
1 usage
```

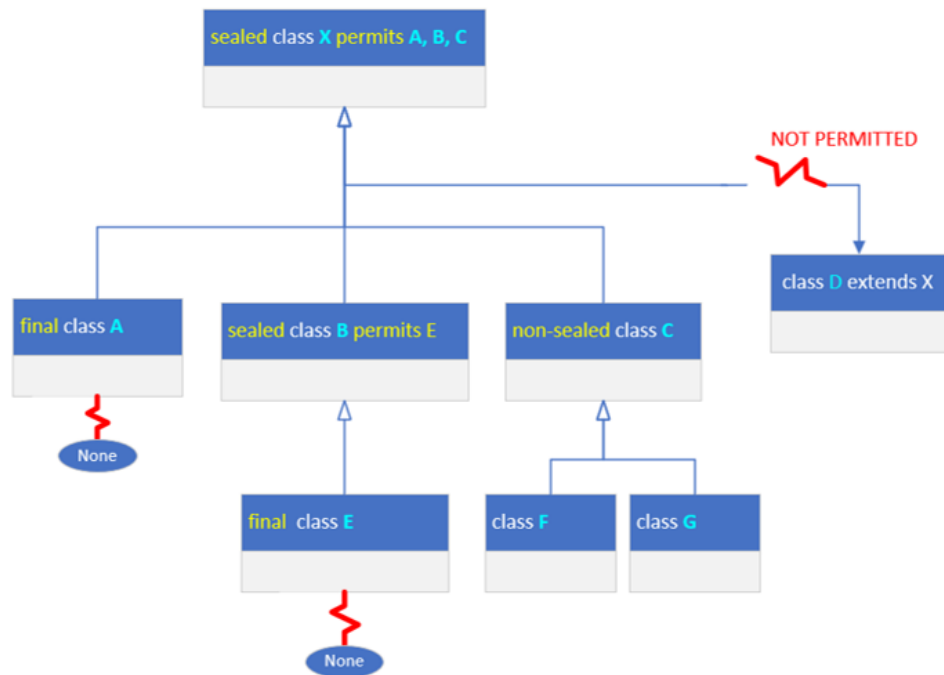
Final Classes

- These are classes that cannot be inherited.
- Enums and records are final classes.

Operations	final class	abstract class	private constructors only	protected constructors only
Instantiate a new instance	yes	no	no	yes, but only subclasses, and classes in same package
A subclass can be declared successfully	no	yes	no	yes

Sealed Classes

- This feature aims to control the inheritance of classes and which subclasses can be used.
- It uses after the access modifiers.
- With the permits keyword, you can specify which classes can be inherited from this class.



```
public sealed abstract class SpecialAbstractClass permits FinalKid, NonSealedKid,
    SealedKid, SpecialAbstractClass.Kid {

    1 usage
    final class Kid extends SpecialAbstractClass {

    }

}
```

Stream

- These take inputs and they do some process then provide results without any changes on original data.
- They support **Functional Programming** approach.
- They work as a **lazy process**. So, when terminal operation request occurs then stream pipeline run and return result.
- **Parallel Stream** process provides performance advantage when working with big data.
- Stream pipelines start with stream like “`bingoPool.stream(). ...`” and ends with terminal operation. process between stream and terminal operations called as **intermediate operations**.
- Results are stable at every situation, but processes cannot be stable. For example, process order can be changed at intermediate operations for optimization. Some processes can merge or skip. This is done automatically. Although this is a good thing, if there are **side effects** of intermediate operations problems can occur. That’s why things that may have side effects should be avoided.
- Streams have two type **finite** and **infinite**. These terms specify stream’s element number limited or unlimited.
 - **Finite Stream:** It is a stream that has a certain number of elements and retrieves elements from a limited data set. A stream that represents list or an array is a finite stream because it’s number of elements is specified.
 - **Infinite Stream:** It is a stream that element numbers unlimited and if necessary, it can create element. For example, a stream created with methods such as `IntStream.iterate()` or `Stream.generate()` is an infinite stream. Its element number is unlimited because of elements are generate according to a certain condition.
- Stream Sources;

Method	Finite	Infinite
<code>Collection.stream()</code>	X	
<code>Arrays.stream(T[])</code>	X	
<code>Stream.of(T...)</code>	X	
<code>Stream.iterate(T seed, UnaryOperator<T> f)</code>	X	X
<code>Stream.iterate(T seed, Predicate<? super T> p, UnaryOperator<T> f)</code>	X	
<code>Stream.generate(Supplier<? extends T> s)</code>		X
<code>**IntStream.range(int startInclusive, int endExclusive)</code>	X	
<code>**IntStream.rangeClosed(int startInclusive, int endExclusive)</code>	X	

•

Return Type	Operation	Description
<code>Stream<T></code>	<code>distinct()</code>	Removes duplicate values from the Stream.
<code>Stream<T></code>	<code>filter(Predicate<? super T> predicate)</code> <code>takeWhile(Predicate<? super T> predicate)</code> <code>dropWhile(Predicate<? super T> predicate)</code>	These methods allow you to reduce the elements in the output stream. Elements that match the filter's Predicate are kept in the outgoing stream, for the filter and takeWhile operations. Elements will be dropped until or while the <u>dropWhile's</u> predicate is not true.
<code>Stream<T></code>	<code>limit(long maxSize)</code>	This reduces your stream to the size specified in the argument.
<code>Stream<T></code>	<code>skip(long n)</code>	This method skips elements, meaning they won't be part of the resulting stream.

- **`dropWhile()`** method drops element from stream while predicate is true. When first false value occurs, drop process ends.
- **`takeWhile()`** method takes elements to stream while predicate is true. When first false value occurs, take process ends.

•

Return Type	Operation	Description
<code>Stream<R></code>	<code>map(Function<? super T,? extends R> mapper)</code>	This is a function applied to every element in the stream. Because it's a function, the return type can be different, which has the effect of transforming the stream to a different stream of different types.
<code>Stream<T></code>	<code>peek(Consumer<? super T> action)</code>	This function doesn't change the stream, but allows you to perform some interim consumer function while the pipeline is processing.
<code>Stream<T></code>	<code>sorted()</code> <code>sorted(Comparator<? super T> comparator)</code>	<p>There are two versions of sorted.</p> <p>The first uses the <u>naturalOrder</u> sort, which means elements in the stream must implement Comparable.</p> <p>If your elements don't use Comparable, you'll want to use sorted and pass a Comparator.</p>

- Purpose of peek method is no side effect to any stream element.
- **Reduction Operation:** It is a kind of terminal operation. These types of operations are used to merge elements of a collection or data to turn it into a simpler result. Generally, they express mathematical operations such as sum, subtract, multiply. However, they can use for different data types and operations too. Some examples for reduction are collect, reduce, toArray, toList.

- **Collectors:** It is a helper class that part of Java Stream API, and It is use for adding, grouping or converting of stream elements. It provides some methods for processing of stream results. You can convert Stream elements to another collection types, maps or special results with this way.

•

Return Type	Terminal Operations
R	<code>collect(Collector<? super T,A,R> collector)</code>
R	<code>collect(Supplier<R> supplier,BiConsumer<R,? super T> accumulator,BiConsumer<R,R> combiner)</code>
<code>Optional<T></code>	<code>reduce(BinaryOperator<T> accumulator)</code>
T	<code>reduce(T identity,BinaryOperator<T> accumulator)</code>
<code><U> U</code>	<code>reduce(U identity,BiFunction<U,? super T,U> accumulator,BinaryOperator<U> combiner)</code>
<code>Object[]</code>	<code>toArray()</code>
<code>A[]</code>	<code>toArray(IntFunction<A[]> generator)</code>
<code>List<T></code>	<code>toList()</code>

Collect(): This method provides a functional way to aggregate data from the Stream into a specific collection type or produce another result. Generally, when this method is producing a result collection, it handles every element and adds the result. This is useful various scenarios such as grouping, filtering, converting.

Reduce(): This method is used to combine elements in the stream to produce a single result. This is suitable for processes such as summing, multiplying, or combining. It takes two parameters, one is starting value, other is a BinaryOperator. BinaryOperator by applying this function on each element, it combines it with the next element.

Optional Class

- A class introduced in Java 8, used specifically for handling null value problems. This class is used to represent whether a value exists or to indicate that a method may return a value, but that value may also be null.
- A method that returns **optional**, it must not return null in any situation. Instead of that, it must return **Optional.empty()**.
- To place objects to **optional** is consume memory and slows down program, it decreases the readability and increases the complexity. That's why **it needs to be used in the right place, not everywhere**.
- They are not **Serializable**.
- Not recommended for use as fields or method parameters.

•

Factory Method	When to Use	Best Practice Notes
<code>Optional<T> empty()</code>	Use this method to create an Optional that you know has no value.	Never return null from a method that has Optional as a return type.
<code>Optional<T> of(T value)</code>	Use this method to create an Optional that you know has a value.	Passing null to this method raises a <code>NullPointerException</code> . Use <code>ofNullable</code> instead, if a possible value might be null.
<code>Optional<T> ofNullable(T value)</code>	Use this method to create an Optional when you are uncertain if the value is null or not.	

- Methods that return an *optional* in streams;

Return Type	Terminal Operations
<code>OptionalDouble</code>	<code>average()</code>
<code>Optional<T></code>	<code>findAny()</code>
<code>Optional<T></code>	<code>findFirst()</code>
<code>Optional<T></code>	<code>max(Comparator<? super T> comparator)</code>
<code>Optional<T></code>	<code>min(Comparator<? super T> comparator)</code>
<code>Optional<T></code>	<code>reduce(BinaryOperator<T> accumulator)</code>

Static import

- This feature allows us to use static members of a class directly, without having to constantly type the class name when using them.
- It is done as in normal import but, **static** keyword is added after the import keyword.

```
import static java.util.stream.Collectors.groupingBy;
// another example for import static
import static java.util.stream.Collectors.*;
```

Java.Math

- **Math.incrementExact()** is used to perform controlled increments in cases where overflow may occur. It throws an exception when there is overflow. There are also different exact functions.
- **Math.abs()** function cannot return the positive at the min value, because it does not exist. That's why you need to be careful when using it. If there is a possibility of getting the integer min value, it is necessary to use long as a parameter.
- Float supports up to 6-9 digits, so it is more accurate to use double for accuracy in comparison operations such as min and max.

Random

- `IntStream` can be returned with the `ints` method. Origin and bound can be determined by entering parameters.

```
r.ints()
    .limit( maxSize: 10)
    .forEach(System.out::println);
```

Instead of using `limit`, it could be done as `Random.ints(limit, origin, bound)` with three parameters.

- `Seed` can be used when creating a variable. Algorithms produce the same numbers with the same `seed` value. This is useful for repeating the scenario where an error occurs in situations such as debugging, testing, debugging, etc. Additionally, different seeds can be given to obtain different numbers.

Big Decimal

- This class is designed to handle fractional numbers and operation results with higher precision. It is often used in financial calculations, tax calculations, currency conversions, etc.
- The value is kept in two Integer fields instead of binary base.
- It allows us to control how numbers will be rounded without losing precision in calculations.
- You should not use double values in the constructor. Including the `valueOf` method.

-

Examples	Unscaled Value	Scale	Precision
15.456	15456	3	5
8	8	0	1
100000.000001	100000000001	6	11
.123	123	3	3

- When **`setScale()`** is used, it must be specified how rounding will be done, it must be sent as a parameter with the `RoundingMode` enum.

```
bd = bd.setScale( newScale: 2, RoundingMode.CEILING);
```

Since it is an immutable class, we assigned the object returned from `setScale` to `bd`.

- Suffixes such as `_F`, `_L`, etc. used in wrappers such as `Float` and `Double` cannot be sent when `String` is given as a parameter in the `BigDecimal` constructor.
- In it, frequently used numbers are available as static `BigDecimal` objects. One, Zero, Ten.

Java.Time

- `OffsetDateTime` and `ZonedDateTime` classes are used when it is necessary to measure and store date and time according to universal standards.
- The **`java.time.temporal`** package contains some important java interfaces;
 - `Temporal` and `TemporalAccessor` define a uniform way to read from or write to a datetime object.
 - `TemporalAdjuster`, `TemporalAmount`, `TemporalField`, `TemporalUnit` are often used as method parameters to specify what kind of information you want about a datetime object.
- `TemporalAdjusters` is a helper class that returns `TemporalAdjuster` with its methods that can give useful dates such as the first day of the year, the last day, next month, next year.

```

/* we can change value of LocalDate object with "with" methods
 * these methods returns new instantiate, so it is not changes original object */
System.out.println(May5.withYear(1));
System.out.println(May5.withMonth(8));
System.out.println(May5.withDayOfYear(1));

System.out.println(May5); //May5 was not change because LocalDate is immutable.

```

*May5 is a **LocalDate** object.*

- Methods such as plus and minus are also like *with* methods. We can add or subtract from Date.

```

may5.datesUntil(may5.plusDays( daysToAdd: 7))
    .forEach(System.out::println);

```

datesUntil returns a stream.

- Seconds and nanoseconds are kept in the **`Instant`** class. It is designed solely to represent a point in time or a timestamp.
- There is a difference of 0.9 seconds per day between GMT and UTC.

- DateFormat, SimpleDateFormat, Date, TimeZone, GregorianCalendar classes were used before JDK8. Immutable thread-safe classes (java.time) are now used and recommended.
- ZoneId is used to adjust and convert date and time values according to different geographical regions.

ResourceBundle

- is a class and mechanism used to manage an application's text and other resources in different languages.
- The ResourceBundle class stores texts and other data as key-value pairs. Each key represents text or resource in a language. Separate ResourceBundle files are created for different languages. These files usually have a ".properties" suffix and contain key-value pairs.
- They can be controlled more easily with the Resource Bundle plugin.

Regular Expressions

- It is a powerful pattern matching tool used for text processing and searching. Regular expressions are used to define, find, or replace specific patterns within text.
- They can be used in situations such as pattern matching, text validation, text editing, data extraction.
- There are regex classes built into the java.util.regex package. These classes are Pattern and Matcher classes.
- for more -> [Pattern \(Java SE 17 & JDK 17\) \(oracle.com\)](#)

Type	Examples
Character Classes	. [abc] [a-g] [A-Z] [0-9] [^abc] \d \s \w
Quantifiers	* + ?
Boundary matchers (or anchors)	^ \$ \b
Groups	()

Quantifier	Meaning	Pattern Example	Match Examples
*	pattern appears zero or more times	b*	empty string, b, bb, <u>bbb</u>
+	pattern appears one or more times	b+	b, bb, bbb
?	pattern appears zero or one time	<u>colou?r</u>	color, <u>colour</u>
{ n }	pattern must appear exactly n times	b{3}	<u>bbb</u>
{ n, }	pattern must appear at least n times	b{2,}	bb, bbb, bbbb
{ n, m }	pattern must appear at least n but not more than m times	b{3, 4}	<u>bbb</u> , <u>bbbb</u>

metacharacter	Meaning	Pattern String	Match Notes
^	matches to start of text	"^."	Matches first character in a string
\$	matches to end of text	".\$"	Matches last character in a string
\b	matches to word	"\\b"	Matches first word in a string.

- There are **Greedy** and **Reluctant** regular expressions.
 - **Greedy:** tries to match as many characters as possible. For example, `'.*'` tries to match as many characters as possible.
 - **Reluctant:** it tries to match as few characters as possible.
- Regular expressions are *greedy* by default. However, they can be made reluctant with the `'?'` quantifier modifier.

For example

"I like B.M.W. motorcycles." For the sentence

`[A-Z].*[.]` With this regex, we say start with a capital letter, followed by any number of characters, and end with a period. This regex is *Greedy*.

Result: "I like B.M.W. motorcycles."

If it is `[A-Z].*?[.]`, we say that it starts with a capital letter, followed by any number of characters, but ends at the first point. This regex is *Reluctant*.

Result: "I like B."

Pattern and Matcher Classes

- **Pattern** class offers many advantages for processing regular expressions and finding matches on text:
 - **Performance Improvement:** Creating a Pattern object by compiling regular expressions makes it faster to use the same regular expression more than once. The compilation process converts the regular expression into an intermediate format that is processed more efficiently.
 - **Expression Readability:** A compiled Pattern object makes your code more readable. A single Pattern object makes it clear that the regular expression is compiled at a particular location and what it is used for.
- **Matcher** class is included in Java's regular expression handler and can be used with Pattern objects, allowing you to find matches on text. The use of the Matcher class offers a number of advantages in text processing:
 - **Text Processing and Match Finding:** Matcher class helps you find matches within text based on a specific regular expression. This is useful in many scenarios such as text analysis, data extraction, or pattern searching.
 - **Multiple Match Operations:** The Matcher object provides the ability to find multiple matches within text. This way you can detect, and process all matches in the text.
 - **Match Information:** The Matcher class provides the starting and ending positions, lengths, etc. of matching text segments. It allows you to obtain details such as.
 - **Flexibility and Customization:** The Matcher object allows you to specify different settings and options when matching text. This way you can customize the matching behavior.

- Matcher is **not Thread-Safe**.
- **Matcher.find()** method continues where it last left off. Therefore, if we want to start over, we need to reset it with **Matcher.reset()**.

-

Method	Characteristics
matches()	Matches entire string. Reluctant expressions may be greedy when used with this method.
<u>lookingAt()</u>	Matches starting at the first character of string. It doesn't have to match the entire string. Honors reluctant expressions.
find()	Starts at the first character not previously matched. Requires reset if you want to start at beginning of string.
find(int start)	Executes a reset, and starts at index passed to the method.

When these methods are used, the groups array in the Matcher class is filled with start and end indexes. The value that found with the group method can be returned as String. After the **reset()** method, groups are reset.

- The real strength of the group method is that we can group within regex.

For example:

Let's have a regex like "**<[hH]\\d>.*</[hH]\\d>**". This regex is used to search for an HTML header.

When one of the above matcher methods runs, the value found can be obtained with **group(0)**

We can also specify the fields we want in Regex with parentheses.

Let's have our regex like this: "**<[hH](\\d)>(.*?)</[hH]\\d>**"

And let's be our string like this: "**<H1>My Heading</H1>**"

After calling any matcher method, group method returns:

group(0) = <H1> My Heading</H1>

group(1) = 1

group(2) = My Heading

This process called as **Capturing**.

- We can named **Capturings** with "**?<name>**".

Let our regex be "**<[hH](?<Num>\\d)>(?!<Content>.*</[hH]\\d>**".

Capturings can be accessed by giving their names to the group method

group(Num) = 1

group(Content) = My Heading

- You can test the regex by hovering over a regex in IntelliJ, clicking on the light bulb on the left, and using "Check RegExp".



-

```
htmlMatcher.reset();
String updatedSnippet = htmlMatcher.replaceFirst( replacement: "<em>" + htmlMatcher.group(2) + "</em>");
String updatedSnippet2 = htmlMatcher.replaceFirst((mr) -> "<em>" + mr.group(2) + "</em>");
```

Although both replaceFirst methods seem to do the same job, the first line of code gives an error. Because the groups in htmlMatcher are empty because the reset function is called. When trying to give group as a parameter, an error is returned.

This problem can be avoided with lambda expression. In the first line of code, the group method is called without calling any match method, which is why it gives an error. But in the second line of code

Since it is sent as a lambda expression, it does not give an error and works because it fills the find() groups because it is called after the match method find() is called in the replaceFirst method in this lambda expression.

This process can also be done with **back reference**.

Back Reference

- It is a way to repeat a group of capturing.

Let's our regex be "<([hH] \d)>(.*?)</[hH] \d>"

The "\\1" part in the expression "<([hH] \d)>(.*?)</[hH] \\1>" indicates the 1st capturing.

- Repeating patterns make it easier to write and read.
- In Java, replacement String (parameter name in method is *replacement*) starts with the "\$" sign. The number after the "\$" sign acts as if the group function is called as a group(number) within the function, and the value returned from the group function is written to that place. (The group function is not called within the function, only the result they give is the same.)

```
htmlMatcher.reset();
System.out.println("-----");
System.out.println("Using Back Reference: \n" +
    htmlMatcher.replaceAll( replacement: "<em>$2</em>")); //back reference
```

Naming Convention

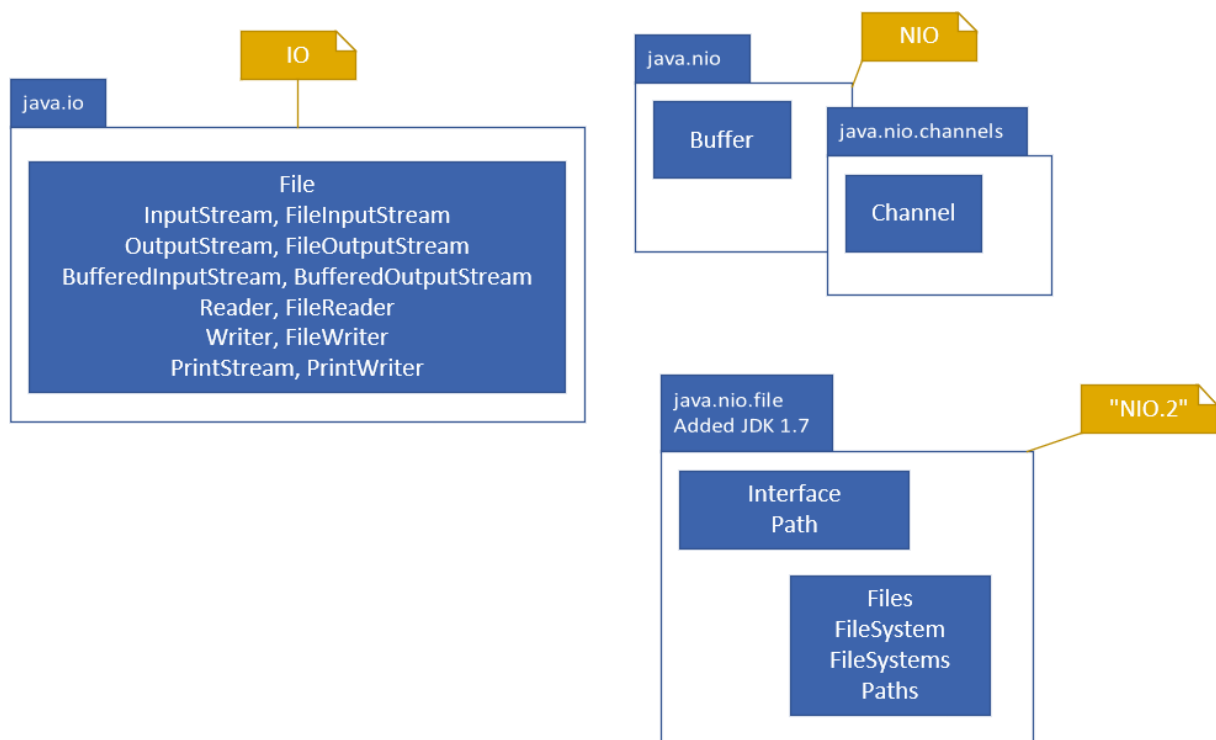
- Packages
 - lowercase.
 - Must be unique.
 - Use reversed internet domain name like “com.meteahmetiyakar.github”.
 - if there is ‘-’ in domain, use ‘_’.
 - if there is number or java keyword, add prefix ‘_’.

1world.com	→	com._1world
Switch.supplier.com	→	com.supplier._switch
Experts-exchange.com	→	com.experts_exchange
- Class
 - CamelCase
 - Must be a name.
- Interface
 - CamelCase
 - Name the Interface according to what they are or what they can do.
as List, Comparable, Serializable.
- Method
 - mixedCase
 - Generally, a verb.
 - It specifies what it can do or what it returns.
- Constants
 - UPPERCASE
 - Words separate with ‘_’
MAX_INT, SEVERITY_ERROR
- Variable and Fields
 - mixedCase
- Type Parameters
 - It is single and upper case.
 - E (Element), K (Key), T (Type), V (Value), S, U, V etc – 2nd, 3rd, 4th types

Java IO, NIO, NIO.2

- **Java IO (Input/Output):** Java IO is a classic I/O API used to perform basic input and output operations. It is available under the `java.io` package and can work with character-based and byte-based data. This API includes classes such as `InputStream` and `OutputStream` and is used to manage file read/write, communicate with sockets, and data stream operations.
- **Java NIO (New I/O):** Java NIO is an I/O API introduced with Java version 1.4. This API is available under the `java.nio` package. Java NIO is designed for advanced I/O operations and includes new concepts such as channels and selectors. In this way, it is possible to perform more efficient and faster I/O operations. Java NIO is particularly suitable for network programming and big data processing applications.
- **Java NIO.2 (New I/O 2.0):** Java NIO.2 is an update introduced with the Java 7 release. This API is available under the `java.nio.file` package and is used for file system operations. Java NIO.2 offers richer file manipulation capabilities than previous versions of Java. It makes operations such as creating, deleting, moving, and querying file properties easier. It also includes features such as symbolic links and file system monitoring (watch service).

Which API to use depends on your project's needs and performance expectations. While Java IO may be sufficient for simple file operations, Java NIO or NIO.2 may be preferred for more complex or performance-oriented operations.



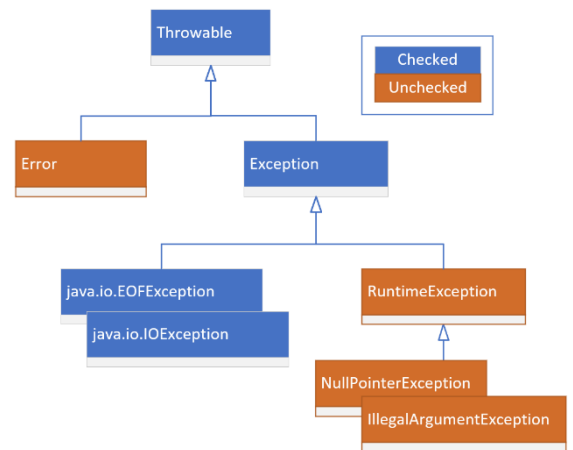
Exceptions

Throwable

- It is the basis of the error and exception mechanism in Java and is used as a superclass for all error and exception classes.
- There are two main subclasses of *Throwable* class:

Error: This subclass generally represents critical errors related to the system and refers to situations in which the program falls into a state that cannot normally be recovered. For example, errors like "OutOfMemoryError" or "StackOverflowError" fall into this category.

Exception: This subclass represents exceptions that occur at program runtime and can be caught by the program. The Exception class is the base class for the previously defined custom exception classes and user-created exception classes. You can derive from the Exception class to define a custom exception type.



- There are two exceptions kind called as **Checked Exception** and **Unchecked Exception**.

Checked Exception

- Represents an expected or common problem that may occur. For example, the file to be opened cannot be found.
- When defining a function or constructor, it is created by writing **throws "ExceptionName"** in its signature. When using this function or constructor, the exception must be handled. Otherwise, it will give an error while compiling.
- For example, we are writing a game and we will read data from the file, we tried to open the file in the read function and could not open it. Here, it is more logical to throw this exception instead of catching with try-catch, because if the data is not read, the game will not run anyway, so there is not much to do in the catch when the file cannot be read.

Unchecked Exception

- Such exceptions are checked at runtime. These are errors that occur while the program is running. For example, going beyond the boundaries of an array or dividing a number by zero. Instead of handling such exceptions with try-catch, it is necessary to take error prevention measures with if. Like null checking with if.

Suppressed Exception

- Exceptions that are thrown but then somehow ignored are called suppressed exceptions. For example, let's say we use try-with-resources, and an exception is thrown in the try block, and then an exception is thrown when the try block is exited with an exception and the given resources are closed. In this case, the first exception suppresses the second exception and the exception thrown from the try block is caught in the catch block, and the exception in the closing process is ignored.
- The `getSuppressed()` method can be used to handle suppressed exceptions.
- With `addSuppressed()`, another exception can be added to an exception to be suppressed.

```
try {
    //codes
} catch (IOException e) {
    try {
        //codes
    } catch (IOException e2) {
        e.addSuppressed(e2);
    }
}
```

Try-Catch-Finally

- Multiple exceptions can catch with | symbol.

```
} catch(ArithmeticException | NoSuchElementException e) {
```

Finally Block

- Finally block is executed under all conditions, that is, it cannot be bypassed with keywords such as return, break, continue, throw. For example, if you use try-catch in a function and return a value with return. Finally block is executed even after return.
- It is especially used in cases where resources (file operations, database connections, network connections, etc.) need to be released or closed. This prevents leaks from leaking.

Try-With-Resources

- This feature makes it easier to open and automatically close resources (file operations, database connections, network connections, etc.). It also provides that resources are safely be closed and prevent leaks.
- Resources opened within the try block are defined in the parentheses of the try-with-resources statement. These resources classes must be implemented the **AutoCloseable** interface.
- When the try block completes or in case of error, resources are automatically closed by calling the close() method. This prevents leaking.
- When opening sources, you can open more than one source at the same time. This can make your code more readable.

try-with-resources examples	traditional try clause
<pre>// First Example: try (FileReader reader = new FileReader(filename)) { // do Something } // Second Example: try (FileReader reader = new FileReader(filename); FileWriter writer = new FileWriter("New" + filename)) // do Something }</pre>	<pre>FileReader reader = null; try { reader = new FileReader(filename); } finally { if (reader != null) { try { reader.close(); } catch (IOException e) { // do Something } } }</pre>

- It is very important to use try-with-resources, especially if any method that returns a stream is used in the Files class.

LBYL and EAFP

- **LBYL (Look Before You Leap)**: recommends checking the conditions before performing an operation and making sure the operation is safe.
- **EAFP (Easier to Ask for Forgiveness than Permission)**: first tries to perform an operation and if an error occurs, it processes this error. In Java, this type of approach can be implemented with try-catch blocks.
- Which one to use is chosen depending on the situation. For example, if an input containing only letters is to be received from the user, if LBYL is performed, it is necessary to check all letters, but when an error is received with EAFP, it can be understood that the wrong input was received. Thus, a more performant code is written. Or it is more logical to use LBYL to check whether a key is in the map.

- If security is important and errors can have serious consequences, LBYL may be preferred.
- If performance is a critical factor or errors are unexpected, EAFP may be more appropriate.

Feature	LBYL	EAFP
Approach	Check for errors before performing an operation.	Assume that the operation will succeed and handle any errors that occur.
Advantages	Can be more efficient if errors are rare.	Can be more concise and easier to read.
Disadvantages	Can be more verbose if errors are common.	Can be more difficult to debug if errors are unexpected.

Working With Files

File Handle

- It is a reference that points to a file and is used by the operating system. It is not the file itself, but an abstract representation of the file that allows it to be tracked by the operating system. The file handler points to the location and state of the file, but the file itself does not contain the data.

File Resource

- It refers to the actual data stored by a file itself. This data is stored on disk and can be accessed by the operating system and applications. The file source represents the content of the file.

In summary, File Handle is an abstract representation of the file and is used by the operating system to monitor and manage the file, while File Resource refers to the actual data stored in the file itself. File Handle is used as a tool to access the file resource. This distinction is important in matters of file handling and management because it describes the difference between the file itself and the processing and use of the file.

File Class

- The `java.io.File` class is used to represent files and directories and perform file operations.
- It is used for basic file operations, that is, it can perform file or directory creation, deletion, renaming, checking, getting file path and similar operations.

Files Class

- The `java.nio.file.Files` class is part of the NIO (New I/O) suite introduced in Java 7 and later and provides modern file handling functionality.
- The `Files` class allows you to perform more advanced operations on files or directories than `File`. For example, it can be used for operations such as copying and moving files, creating, and opening symbolic links, and reading and writing file content.
- The `Files` class works with `Path` objects and allows you to perform more operations in a simpler and more powerful way.
- Asynchronous file I/O operations.
- Instead of locking the entire file, you may lock certain parts of it.
- Supports multiple threads.
- Uses memory more effectively and manages it more efficiently by reading and writing files directly from memory to buffers with `FileChannel`
- Many properties of the file can be accessed with the ***getAttribute()*** or ***readAttributes()*** methods.

String Literal to pass to the <i>Files.getAttribute</i> method	Type:	Alternate method
"lastModifiedTime"	FileTime	<i>Files.getLastModifiedTime</i>
"lastAccessTime"	FileTime	
"creationTime"	FileTime	
"size"	Long	<code>Files.size</code>
"isRegularFile"	Boolean	<i>Files.isRegularFile</i>
"isDirectory"	Boolean	<i>Files.isDirectory</i>
"isSymbolicLink"	Boolean	<i>Files.isSymbolicLink</i>
"isOther"	Boolean	

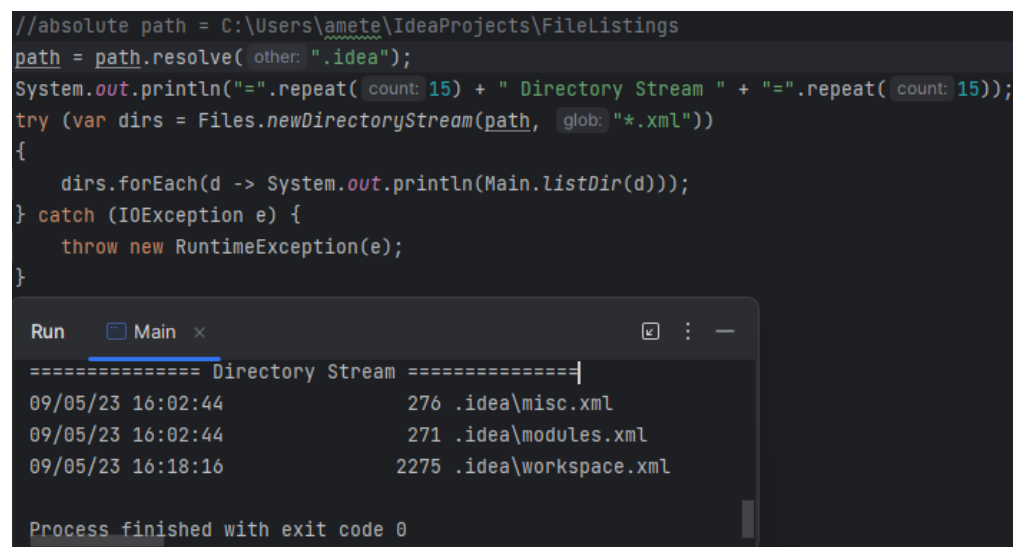
- ***Files.list()***: We can get the list of files and directories in the path with this method. But we cannot access the files and directories inside the files and directories.
- ***Files.walk()***: it does the same thing as the ***list()*** method, but the files and directories within the files and directories can also be accessed with the `maxDepth` parameter.
- ***Files.find()***: You can search within the path using this method. Depth can also be specified.

- **Files.newDirectoryStream():** With this method, files and directories can be listed and returned as a DirectoryStream object. It allows you to select specific types of files or directories using various filters. For example, you can use this method to list only files with a certain extension or files with a certain characteristics.

We can filter which routes we want in this method by using a string called **glob**.

glob is a “limited pattern language” like regular expressions but with a simpler syntax.

[glob patterns](#)



```
//absolute path = C:\Users\amete\IdeaProjects\FileListings
path = path.resolve( other: ".idea");
System.out.println("=".repeat( count: 15) + " Directory Stream " + "=".repeat( count: 15));
try (var dirs = Files.newDirectoryStream(path, glob: "*.xml"))
{
    dirs.forEach(d -> System.out.println(Main.listDir(d)));
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

```
Run Main x
===== Directory Stream =====
09/05/23 16:02:44      276 .idea\misc.xml
09/05/23 16:02:44      271 .idea\modules.xml
09/05/23 16:18:16     2275 .idea\workspace.xml

Process finished with exit code 0
```

More specific filtering can also be done by using lambda expression instead of glob.

- **Files.walkFileTree():** it is useful for navigating and performing operations on file or directory trees.
 - Getting a list of files or directories by scanning a specific directory and its subdirectories.
 - Performing a specific operation on each file or directory (for example, reading or deleting files).
 - Filtering files or directories based on specific criteria (for example, finding files with a specific extension).
 - With this method, existing folders can be navigated. For example, you have a path like “abc/def/ghj”, you want to create a folder if there is no one while browsing this path with walkFileTree. (If there is no abc, create it and enter it.) Such a thing cannot be done.

Files.walkFileTree() method is implemented using a **FileVisitor** object. **FileVisitor** is an interface for manipulating files and directories and includes the following methods:

FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs):

Called before visiting a directory.

FileVisitResult visitFile(Path file, BasicFileAttributes attrs):

Called when you visit a file.

FileVisitResult visitFileFailed(Path file, IOException exc):

Called when an error occurs while visiting a file.

FileVisitResult postVisitDirectory(Path dir, IOException exc):

Called after visiting a directory.

*These methods are automatically called by the **Files.walkFileTree()** method and can be applied to perform user-specified operations. These methods can be **overridden** and shaped as desired by inheriting the **SimpleFileVisitor** class or implementing it as an Anonymous class.*

- **Files.move():** File moving or renaming operations can be done with this method. If the file paths are the same, only the last file name is different, renaming is done; if the file paths are different, moving is done.

Input Streams

- It is used to read byte-based data and represents the operations of reading this data as an abstract interface. This class is especially used for file reading, data retrieval over the network, data compression and similar operations.
- The InputStream abstract class is implemented by subclasses, which support reading data from various sources. Here are some common InputStream subclasses:
- They return byte stream or character stream.

FileInputStream: Used to read data from a file. The Read method is very inefficient as it constantly reads Disk.

ByteArrayInputStream: Used to read data from a byte array.

BufferedInputStream: Makes data reading more efficient by buffering another InputStream.

DataStream: Used to read primitive types (int, float, double, etc.).

ObjectInputStream: Used to serialize and deserialize objects.

PipedInputStream: Used to pass data between threads or processes.

Readers

- It is an abstract class used to perform character-based data reading operations. Reader is specifically used to read text files or character-based data and operates based on the Unicode character codes of characters.
- Reader class provides an abstract interface for reading data and is implemented by subclasses. Some of the most used Reader subclasses are:

FileReader:

- It is a class used to perform file reading operations in Java. This class is used to read text-based files and is a subclass of the Reader class.
- With the read() method, the content of the file is read character by character. The read() method returns the Unicode value of a character, or -1 if it has reached the end of the file. By giving a buffer, that buffer can be filled each time. Thus, as Disk Read decreases, the cost also decreases.
- The FileReader object is closed with the close() method. It is important to release resources when the file reading is complete.
- File types such as txt, csv, xml, json can be read. Additionally, configuration files such as .properties can also be read.

BufferedReader: Makes character reading more efficient by buffering another Reader. It has a larger buffer size than FileReader. Buffer size can be specified in the constructor, but the default value is sufficient for most cases. There are also methods that make it easier to read lines of text.

StringReader: Used to read data from a character array (String).

InputStreamReader: Used to convert bytes from InputStream into characters.

Reader.transferTo()

- Data is transferred from one reader to another reader.

```
//transfer from api to console
String urlString = "https://api.census.gov/data/2019/pep/charagegroups?get=NAME,POP&for=state:*";
URI uri = URI.create(urlString);
try (var urlInputStream = uri.toURL().openStream()) {
    urlInputStream.transferTo(System.out);
} catch (IOException e) {
    throw new RuntimeException(e);
}

//transfer from api to txt file
Path jsonPath = Path.of( first "USPopulationByState.txt");
try (var reader = new InputStreamReader(uri.toURL().openStream());
     var writer = Files.newBufferedWriter(jsonPath)) {
    reader.transferTo(writer);
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

Writers

Files Class

- Writing to the file can be done using the **Files.write()** and **Files.writeString()** methods. Since these methods perform the file opening, writing, and closing operations themselves, opening and closing controls are not required. These methods use Default Open Options if OpenOption is not given. For this reason, when called without OpenOption, they overwrite the file.

Option	Description
CREATE	This creates a new file if it does not exist.
TRUNCATE_EXISTING	If the file already exists, and it's opened for WRITE access, then its length is truncated to 0.
WRITE	The file is opened for write access.

BufferedWriter, FileWriter and PrintWriter

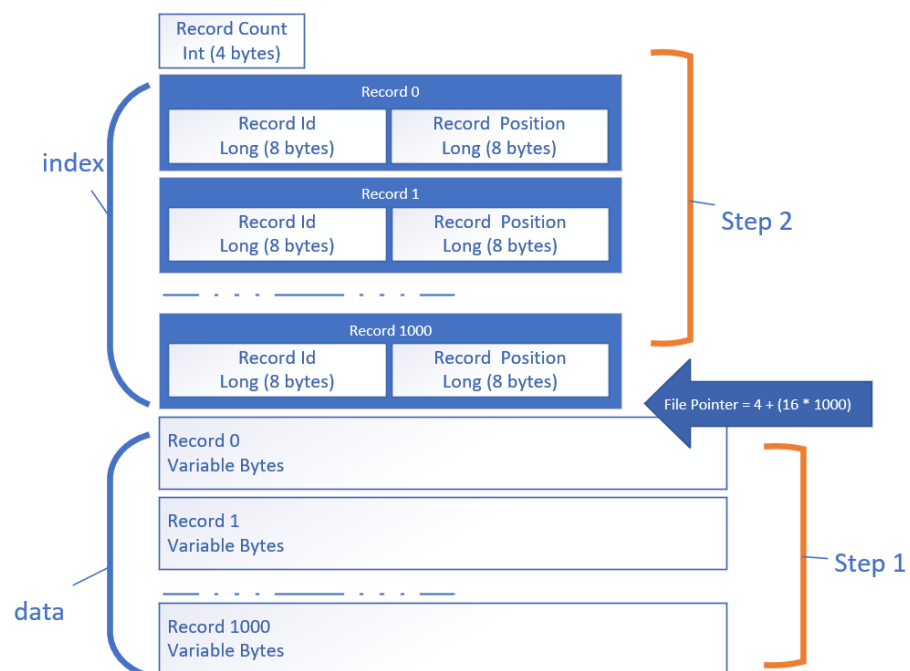
	Buffering	Data Format	Features	Use Case
BufferedWriter	Yes	Characters	Supports line breaks with newline method.	Writing large amounts of text to a file
FileWriter	Yes but much smaller buffer than BufferedWriter,	Characters	No separate method for line separators, would need to write manually	Writing small amounts of text to a file
PrintWriter	No, but often used with a BufferedWriter.	Characters, numbers and objects	Familiar methods, that have same behavior as <u>System.out</u> methods	Writing text to a file, formatting output, and outputting objects

- Classes do the flush themselves, but we can make it happen by calling the function whenever we want. It should be noted that data that is not flushed in Threads cannot be seen by other Threads.

RandomAccessFile Class

- This class provides direct access (random access) to any point in the file. It doesn't do serial or buffered input/output operations like other input/output classes, instead it allows you to go to a specific location of the file and perform read or write operations there.
- It acts as a very large byte array in the file system.
- It has a type of cursor called **file pointer**, which is used for access.
- Read and write operations can be performed.
- Binary Data can be read and written using special methods.

- Operation is carried out with the pointer as if you traverse in the ram. For example, if there is an array holding Student objects and each Student object is 50 bytes, a method such as `currentPtr + 50bytes` should be used when traversing the array. The same applies here.
- What if not all data is the same size? In this case, we keep a different file in which the starting point of each data is written. Or, both filepointers and data are kept in a single file.



Index	Record Id	File Pointer	Variable Size
0	100000	0	First Record (50)
1	1	50	Second Record (250)
2	543210	300	Third Record (150)
3	777	450	Fourth Record (500)

File pointer part must be saved for traverse in file.

- We can assign the file pointer to the desired point with the ***RandomAccessFile.seek()*** method.

DataStreams

- It is specifically used to convert (*Serialization*) data into more complex data types (for example, Java objects) or to return such data from serialized data (*Deserialization*).

DataOutputStream

- It allows writing Primitive and String Java data types to an output stream in a portable way.

DataInputStream

- It is used to read data from data streams. It can read Primitive and String data types.

ObjectOutputStream

- It allows objects to be written to an output stream in a portable way.

ObjectInputStream

- It is used to read data from data streams. Can read object.

Serialization

- It is the process of converting a data structure or object into a format that can be stored in a file. Only **Serializable** classes that implement the **Serializable interface** can be serialized.
- The default serialization mechanism writes the object's class, class signature, and values of non-static fields. Thanks to these, the object and its states can be recreated in the read operation. This process is also called **Deserialization**.
- In some cases, Incompatible Change occurs in *serialization* and *deserialization* operations, java does this by checking the **serialVersionUID** field. The reasons for incompatibilities are as follows:
 - Changing the type of a primitive field. For example, converting an int field to a long may not seem wrong at first glance and it makes sense that a long can contain an int, but since one is 4 bytes and the other is 8 bytes, an error occurs when reading from the file after the change.
 - Deleting a field.
 - Making a non-static field static or a non-transient field transient.
 - There are other, more complex replacement incompatibilities, such as moving a class within its hierarchy, replacing the writeObject() and readObject() methods after previously using them for serialization.
 - For all incompatibilities:
[Java Object Serialization Specification: 5 - Versioning of Serializable Objects \(oracle.com\)](#)
- Compatible Change is also available in *serialization* and *deserialization* operations. These:
 - Adding a field.
 - Adding writeObject() and readObject() methods.
 - Access modifier change.
 - Making a static variable non-static or making a transient field non-transient.
- Deserialization process does not call constructor etc. It assigns directly to the variable with the readObject() method. Therefore, it is important not to change the field type and name, otherwise compatible problems may occur.
- We can overcome compatibility problems by writing our own readObject() and writeObject() methods.

Transient

- It is the keyword used in areas that will be ignored in the *serialization* process.
- It can be used in areas containing sensitive information or for constantly changing variables that do not need to be permanent.

serialVersionUID

- It is a runtime field, if not explicitly created, it is implicitly created by the compiler for serializable classes. It contains details such as field number, types, and definitions.
- When reading an object from a stream, the serialVersionUID stored in the object is checked. In this control process, the serialVersionUID in the object stored in the file (stream) is compared with the object in the compiled class file. If they do not match, there is a compatibility issue and an ***InvalidClassException*** is thrown.
 - Additionally, this error may occur in different compilers. Different compilers may produce different serialVersionUIDs. For example, this error may occur when trying to read a file written in an old Java version in a different version.
 - To prevent this, it is **highly recommended to add** the serialVersionUID you created into the class.

```
private final static long serialVersionUID = 1L;
```

Concurrency/Threads

- Every thread created by a process can access the heap of this process. In other words, threads share the process' memory and files. For this reason, if you are not careful, major problems may arise.
- Each thread also has its own heap called **thread stack**, only the owner thread can access this.
- While we are waiting for something to come from a line of code or a method, we may want to do different things too. For example, we will pull data from a website and a database. If we do not use threads, we will have to wait for data to come from one and then wait for data to come from the other. Instead, we can do this with threads and perform both operations at the same time with different threads.
- You can execute parallel operations using threads in Java. However, when these threads will run and in what order they will run are determined by the JVM and the operating system. The JVM creates, starts, and terminates threads, but the operating system controls which process and on which processor core these threads physically run. It can be difficult to precisely control the order or timing of threads and sometimes you are subject to limitations in this regard. Therefore, when working with threads, it is important to manage synchronization, thread priorities and similar issues between threads.

Thread

- We can write our own thread by overriding the run method in classes from which we inherit the **Thread class**.
- Instead of calling the run() method, which we have overridden, to start the thread, we use the start() method, which calls the run() method. The reasons for this are:
 - The start() method creates a new thread and requests the operating system to run this thread. It also enables Java's thread management infrastructures.
 - Using the start() method allows different threads to run simultaneously or simultaneously. The run() method, on the other hand, is run just like an ordinary method within the current thread, and it does not create a new thread or start parallel execution.
- We can call the start method of a Thread object only once. For this reason, it may make sense to create it as an Anonymous class.

```
new Thread() {  
    public void run() {  
        System.out.println("From anonymous class thread");  
    }  
}.start();  
  
//or with lambdas  
  
new Thread(() -> System.out.println("From anonymous class thread")).start();
```

- The execution order of threads depends on the JVM and operating system and this order may change in every different calls.

```

3 public class Main {
4     public static void main(String[] args)
5     {
6
7         System.out.println("From main thread");
8
9         Thread anotherThread = new AnotherThread();
10        anotherThread.start(); //this function calls run m
11
12        System.out.println("Again from the main thread");
13
14    }
15
16

```

Output:

```

C:\Users\amete\.jdk\corretto-17.0.8\bin\java.exe "-javaagent:C:\P
From main thread
again from the main thread
From AnotherThread thread

```

- **Thread.setName():** A thread can be named with that.
- **Thread.sleep():** We can put a thread to sleep with this method. It can wake up if a different thread interrupts it. The time to wait is given in milliseconds and nanoseconds. If the operating system does not support the desired level of detail, it may not be the exact time required. For example, it may not support nanoseconds.

Creating Threads with Runnable

- **Runnable** is a **FunctionalInterface**. We are expected to implement the `run()` method. We can obtain a Thread by sending an object of the class in which we implement this interface to the Thread constructor.

Interrupt

- **Thread.interrupt():** It is a method used to stop or interrupt the execution of a thread. When a thread wants to be interrupted by another thread or if a thread is running for an unexpectedly long time, it may be possible to terminate the execution of this thread by using interrupt.
- **Interrupt Flag:** Calling the `Thread.interrupt()` method sets the **interrupt flag**. Thread can stop or continue its execution by checking this flag.
- Some operations and methods, such as **Thread.sleep()**, **Object.wait()** send an interrupt request (such as interrupting normal operation, waiting, or sleeping) to a thread. If the thread is interrupted during the interrupt request (`Thread.interrupt()`), an exception called **InterruptedException** is thrown.

- **java.nio.channels.InterruptibleChannel** interface provides the classes that implement it with the ability to process this signal when they receive an interrupt signal during I/O operations.
- If the thread does not respond to the **Thread.interrupt()** request, the flag is simply set, and the thread can continue to sleep or wait.
- By checking the interrupt flag, the thread can optionally stop its execution or ignore the interrupt flag. It is the responsibility of the thread code to respond to the interrupt flag.
- An important point to consider when using **Thread.interrupt()** is that how the thread code will handle the interrupt flag should be carefully considered. It is important that the thread safely terminates or releases its resources when an interrupt request arrives.

Join

- Let's have two threads. Let the first thread receive data from the network, and the second thread makes changes to this data. In this case, we want the second thread not to start before the first thread finishes. We achieve this with **Thread.join()**.
- What if the first thread never finishes due to an error? In this case the program will appear to be asleep or frozen and will not do anything. We can prevent this with **Thread.join(long millis)**. If the first thread finishes or millis time passes, the second thread continues.
- It is best to see this process as a suggestion to the JVM and the operating system rather than a command. Our suggestion may or may not be implemented. The operation of threads depends entirely on the JVM and operating system.

Multithreading

- It is the operation of more than one thread at the same time.
- When multiple threads are running at the same time, in some cases they may want to try to access or modify the same memory address. This can lead to erroneous behavior. This situation is called **Race Conditions**.

- **Thread interference** refers to a scenario in multithreading programming that can be caused by multiple threads simultaneously accessing and modifying shared data. It occurs in two basic situations:
 - **Race Condition:** More than one thread tries to access the same data at the same time and performs reading or writing operations on this data. If proper synchronization mechanisms are not used, this can lead to a race condition and result in data being updated inconsistently or incorrectly.
 - **Concurrent Modification:** While a thread is modifying a data structure (such as a list or map), if other threads try to access the same data structure and modify it, a concurrent modification problem may arise. In this case, the data structure may be corrupted or unexpected results may be obtained.
- If a class or method is **Thread Safe**, it means that all critical parts of it are synchronized and data integrity and consistency will be maintained during multithreaded processing.
- With the synchronized keyword, a method or critical area can be used by only one thread at a time.

```
public synchronized boolean isSynchronized() {
    return true;
}
```

```
synchronized (object) {
    // critical area
}
```

Synchronized

- Marking a method or code block as "synchronized" in Java creates an object lock between threads that want to run this method or code block.
- If a synchronized method or code block is applied to an object of a class, a lock mechanism is created on this object. This is called Object Locking.
- Object locks are created on an object basis, as the name suggests.
- An object lock is created for synchronized fields, and when a thread tries to access these synchronized fields, it is checked whether it has the object lock. If the thread has the object lock, it can enter that area, otherwise it queues and waits until the lock is passed to it. The object lock mechanism is only valid for synchronized areas, so if a thread accesses a synchronized area and obtains the object lock, and a second thread tries to access a non-synchronized area at the same time, both of them can gain access and errors may still occur.

- If a static field is synchronized. The lock is class based, not object based.
- It is best to synchronize as few code blocks as possible. Otherwise, threads will be suspended unnecessarily.
- There are functions that can only be called within synchronized fields. These; **wait**, **notify**, **notify all**. Their purpose is to prevent **Deadlock** situation.
- Synchronized block also has its disadvantages.
 - Blocked threads waiting for object lock cannot be interrupted.
 - Synchronized block must be in the same method. Even if method A is synchronized and calls method B, the object lock cannot be released in method B.
 - It is not possible to check whether an object lock is available or not.
 - If more than one thread is waiting for an object lock, there is no such thing as first one gets the object lock, that is, they are not kept in a queue. This can lead to long waits or problems.

For these reasons, the `java.util.concurrent` package was added to Java in version 1.5. Synchronization process can be done with class.

Deadlock

- It is a synchronization problem encountered in multithreading programming. Threads wait on each other and never obtain object locks. For example, let's say there are Threads A and B, and thread A has the lock of object x and thread B has the lock of object y. If thread A needs the lock on object y to continue and thread B needs the lock on object x, neither can move.
- Deadlock generally occurs when the following two conditions occur simultaneously:
 - **Hold and Wait:** When a thread locks a resource and needs another resource, this thread holds the locked resource while waiting for the other resource.
 - **Circular Wait:** A loop occurs between threads, that is, each thread waits for the resource owned by another thread. For example, if thread A locks on resource X, while thread B locks on resource Y, and thread A waits to access resource Y, a loop occurs.
- Deadlocks occur when threads try to obtain object locks. When syncing code, it's important to be alert to where this might occur.

- To prevent deadlock from occurring, object locks can be attempted to be obtained in the same order. For example, if thread1 wants the locks in the order of lock2 and lock1, and if the other written threads want the locks in the order of lock2 and lock1, a deadlock cannot occur.
- **wait():** Usually used with a condition and specifies that the thread holding the lock on the object should wait until it releases the lock and is woken up by another thread or JVM.
 - Since the thread may wake up for a different reason than notify(), for example by the operating system, it is necessary to control it with while. If it needs to continue sleeping, since it cannot exit the while loop, it returns to the beginning of the loop and the wait() function is called again.

```
while (!empty) {  
    try {  
        wait();  
    } catch (InterruptedException e) {  
        throw new RuntimeException(e);  
    }  
}
```

- **notify():** It is used to wake up one of the threads waiting on an object with the wait() method.
 - Which thread to wake up is entirely between the JVM and the operating system. It is unpredictable.
 - The thread using notify() does not release the object lock. It continues to hold the lock, but wakes up a waiting thread with wait(). The waking thread waits for the thread that using the notify() method to release the lock.
- **notifyAll():** It is exactly the same as the notify() method. The only difference is that the notify() method wakes up a random thread, while the notifyAll() method wakes up all threads waiting with wait. These threads compete for the object lock. When the thread on which the notifyAll() method is called releases the object lock, other threads start to receive it.
 - Unless we are working with a large number of threads, the traditional usage is to use notifyAll(). However, if we are working with a large number of threads, this can cause a big performance problem.

Atomic Operations

- The thread cannot be suspended by the operating system while performing atomic operations.
- Transactions accepted as atomic;
 - Reading reference variables (object1 = object2), reading or writing primitive types (except long and double because they are 64 bit and extra operations can be performed on them because they are 64 bit. For example, two separate operations are required to be used in a 32 bit architecture.)
- Reading and writing operations of variables marked with the **volatile** keyword are also considered atomic.
 - For example, let's say we have a process called counter++. The step-by-step procedures for this process are as follows.
 - 1-) Read the counter value from main memory.
 - 2-) Add 1 to the value.
 - 3-) write the value to main memory.

If the variable is not *volatile*, it depends on the JVM when to write data back to main memory. Problems may occur when trying to access from a different thread (such as reading the previous value). We prevent this with the *volatile* keyword. When *volatile*, all transactions are started, completed, and completed within the same period of time. There is no suspension.

 - For the same instance, if the variable is *volatile* and two threads try to increment it at the same time, problems may still arise. For example, thread1 reads the value from memory and let it be 1, at the same time thread2 reads the value from memory and again 1, then thread1 increases the value, thread2 also increases the value. In this case, they both have the value 2 and write it to main memory. As a result of the process, our value becomes 2, but it should have been 3. Here again, **synchronization mechanisms must be used**.
- Some atomic classes, such as ***java.util.concurrent.atomic***, provide classes that start with the Atomic header.
- **Atomic classes** are classes used in multi-threaded programming to enable shared variables to be safely modified and updated by atomic operations. These classes provide the ability to operate on shared variables without using synchronization mechanisms, thus making it easier to write thread-safe code.

Java.util.concurrent package

ReentrantLock Class

- With this class, we can do what the synchronized block does in a more flexible way.
- **ReentrantLock.lock():** We can take lock the object with this method. If the lock cannot be acquired, the thread queues to acquire the lock and is suspended. An object lock can be obtained multiple times with lock(), for example, let's call the lock() method twice and unlock() method once within a method. The object lock is not released when the method ends. This should be taken into consideration especially at points where lock() is used within the loop.
- **ReentrantLock.unlock():** With this method we can release the lock of the object.

*Lock acquisition and release operations are entirely in the hands of the software developer in this class, **nothing is done automatically.***

- *It is a good way to perform closing operations with try-catch-finally.*
 - *Readability increases and the lock-unlock section becomes more compact.*
 - *If an exception is thrown from the critical area and we cannot catch this exception with catch, the locks must be released.*
 - *Since ReentrantLock is **not AutoClosable**, we cannot use try-with-resource.*
- **ReentrantLock.tryLock():** We can try to get an object key with this method. Returns true if it can be retrieved, false if not. Operations can be performed when received or not received by placing it in an if block. There is also an overloaded version that takes a long and TimeUnit as parameters. When trying to obtain a lock, it waits a certain amount of time to obtain it, if not, it returns false.

```
if(bufferLock.tryLock( timeout: 5, TimeUnit.SECONDS))
```

- If the constructor is given the value true, such as **ReentrantLock(true)**. This indicates that an attempt is made to ensure fair locking. In other words, it ensures that the thread that has been waiting the longest for the object lock gets the object lock.

Thread Pool

- It is a design pattern used in multithreading programming. It is used to facilitate thread management.
- It pre-creates threads that execute a specific task and stores them in the pool.
- Since threads are created in advance and kept in the pool, the cost of starting and terminating them is reduced. The burden of creating threads for each operation is reduced and the application runs more efficiently.
- Thread Pool automatically distributes work to threads and balances the workload among threads. This makes better use of system resources and makes operations faster.
- The number of threads that will run simultaneously can be limited. This prevents the capacity of the system or resource from being exceeded during the thread creation process. At the same time, the number of active and blocked threads can be limited.
- Thread Pools can be created under the ***ExecutorService*** class with factory methods such as ***newFixedThreadPool()***, ***newCachedThreadPool()***, ***newSingleThreadExecutor()***.
- Since the number of threads is limited with the Thread Pool, when we want to run a task, it may not run immediately. For example, let's limit the Thread Pool to a maximum of 5 and send a task. If all 5 threads are active when we send the task, the task we send will wait in the service queue until one of these 5 threads actually terminates.

ExecutorService

- It is an Interface provided to facilitate multithreading.
- It allows you to perform tasks such as managing operations on Threads and creating a Thread Pool more easily. It allows us to focus on the code we want to run without having to deal with managing threads and their lifecycles.

- It also optimizes the cost of creating threads.
- After use, the Thread Pool must be closed with **ExecutorService.shutdown()**.
 - *ExecutorService.shutdown()* method waits for all threads to close and then closes the pool. If we want everything to shut down suddenly, the **ExecutorService.shutdownNow()** method is used.
 - *ExecutorService.shutdownNow()* method tries to stop all tasks in the pool and discards all tasks in the queue. However, there is no guarantee that you can do these. That's why it's best to use the *ExecutorService.shutdown()* method.
- Tasks can be sent to the pool using the **ExecutorService.execute()** or **ExecutorService.submit()** methods.
- **ExecutorService.execute()**
 - It doesn't return anything; it takes a process that implements the **Runnable** interface and runs it.
 - If an exception occurs during the process, it is issued directly and the thread in the pool is terminated.
 - It is generally used when you do not need to get the transaction result or are not interested in the transaction result.
- **ExecutorService.submit()**
 - It takes a process that implements the **Callable interface** and executes it and sends it as a **Future** object. The Future object allows receiving the result of the operation and even waiting until the operation is completed.
 - If an exception occurs during the process, this exception is stored in the Future object and allows the exception to be handled with the *get()* method while the process is waiting. In this way, exceptions do not terminate the thread completely.
 - It is useful if you want to work with the result of the operation.
 - There is also an overloaded version that takes a process that implements the **Runnable** interface, but it is not recommended to use it, it is better to use *execute()* instead.

ArrayBlockingQueue

- Its capacity is given by the developer in the constructor.
- It is Thread Safe.
- It allows the thread to wait while a thread is taken from the queue (*take()* method) or a thread is put into the queue (*put()* method).
 - **put()** if the queue is full, this method puts the calling thread on hold and adds the item when the queue is empty.
 - **take()** If the queue is empty, this method puts the calling thread on hold and waits until an item is added to the queue.
 - **offer()** tries to add items to the queue and returns false if not.
 - **poll()** tries to retrieve items from the queue, but returns null if the queue is empty.

```
@Override
public void run() {
    while (true) {
        synchronized (buffer) // private ArrayBlockingQueue<String> buffer;
        {
            try {
                if (buffer.isEmpty()) {
                    continue;
                }

                // --> In the non-synchronized scenario <--
                //At first glance everything seems fine, we check buffer is Empty if it isn't we peek it
                //So problem can be where place?
                //Problem exactly here, on this line.
                //buffer cannot be empty until this line, but here thread can be suspended then remove element from it any reason
                //after all of that when we call peek() method, boom!
                //because thread was suspended and now buffer is empty, but we called peek method.
                //peek method returns null because of buffer is empty then we call equal method with null
                //this will throw an exception

                //So what that all means is that we may still have to add synchronization code
                //when using thread safe classes like ArrayBlockingQueue.

                if (buffer.peek().equals("EOF")) {
                    System.out.println(color + "Exiting");
                    break;
                } else {
                    System.out.println(color + "Removed " + buffer.take());
                }
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

Thread Starvation

- In multithreading programming, it is the situation where a thread continues to wait or cannot access resources. This is usually caused by synchronization, race condition, or resource sharing issues.
- Thread Starvation can be seen in the following scenarios:
 - **Priority Issues:** When the priorities of threads are set incorrectly or ordered incorrectly, one thread may constantly gain priority over others and other threads may continue to wait for access to resources.
 - **Slow Threads:** If some threads are running slower than others, other threads may have to wait for these slow threads to finish.
- Thread Starvation can be prevented with **fair locking**.

```
private static ReentrantLock fairlock = new ReentrantLock(fair: true);
```

- If the application uses two or three threads, or if the tasks running on the thread compete very quickly, or if you are aware that it will be very rare for more than one thread to block, it is a better decision to use synchronized blocks. Because **using fair lock will affect performance**.
- A scenario where threads constantly respond to each other but do not complete their work is called **Live Lock**. For example, two people facing each other constantly take the same steps to give way and neither of them can move forward and they constantly go left and right. Even if they are in working order, they cannot progress their work.

```
21 public synchronized void work(SharedResource sharedResource, Worker otherWorker) {
22     while (active)
23     {
24         if(sharedResource.getOwner() != this) {
25             try {
26                 wait(timeoutMillis: 10);
27             } catch (InterruptedException e) {
28             }
29             continue;
30         }
31     }
32     if(otherWorker.isActive()) {
33         System.out.println(getName() + " : give the resource to the worker " + otherWorker.getName());
34         sharedResource.setOwner(otherWorker);
35         continue;
36     }
37
38     //in our example code never come to this line
39     //they regularly give the lock each other
40     //when otherWorker is active, lock gives to otherWorker and return to start of while loop
41     //but until come to this line from start of loop, otherWorker gives back lock to us
42     //so, code keeps loops between 24-36 in both workers,
43     //and it never comes to here we called this problem as LiveLock
44
45     System.out.println(getName() + " : working on the common resource");
46     active = false;
47     sharedResource.setOwner(otherWorker);
48 }
49 }
```

JUnit

- Each test method should be independent from each other.
- Value can be checked with **assertEquals()**.

```
@org.junit.jupiter.api.Test
void deposit() {
    BankAccount account = new BankAccount( firstName: "Mete ahmet", lastName: "Yakar", balance: 1000);
    double balance = account.deposit( amount: 200.00, branch: true);
    assertEquals( expected: 1200.00, account.getBalance(), delta: 0);
}
```

- Do not declare variables in the test class. Variable definitions should be included in test methods. However, if there is constant code repetition, we can prevent this by writing a method with the **@BeforeEach** annotation.
 - Methods written with the **@BeforeEach** annotation are called before each test method call. It can be used to initialize a database connection or specific objects.

```
@BeforeEach
public void setup()
{
    account = new BankAccount( firstName: "Mete Ahmet", lastName: "Yakar", balance: 1000.00, BankAccount.CHECKING);
    System.out.println("Running a test...");
}
```

- We said that the **@BeforeEach** annotation runs before every method call, but what do we do if we want a code that we want to run once? For this, **@BeforeAll** annotation is used. A static method is created that will be called only once. Operations such as starting a database connection can be performed here.
- If we want to write a function that does the opposite of **@BeforeAll**, that is, to be run once after all the checks are completed, the **@AfterAll** annotation is used. Opened links and files can be closed.
- The **@AfterAll**, **@AfterEach** annotations are the opposite of the **@BeforeEach** and **@BeforeAll** annotations.

- An exception can also be checked with the **assertThrows()** method.

```
@org.junit.jupiter.api.Test
public void withdraw_notBranch() throws Exception {

    assertThrows(IllegalArgumentException.class, () -> {
        double balance = account.withdraw(amount: 600.00, branch: false);
    });
}
```

- When we want to test with different values, we can write **Parameterized Test**.

```
@ParameterizedTest
@CsvSource({
    "100.00, true, 1100.00",
    "200.00, true, 1200.00",
    "325.14, true, 1325.14",
    "489.33, true, 1489.33",
    "1000.00, true, 2000.00"
})
public void deposit(double amount, boolean branch, double expected)
{
    account.deposit(amount, branch);
    assertEquals(expected, account.getBalance(), delta: .01);
}
```

Database

SQLite

- Although we specify types for columns in SQLite, any type of data can actually be placed in any column.
- Backups can be made with “**.backup**”.
- It is good behavior to define primary key Id fields as “_id”. Some java classes require an id column called “_id”.
- It is faster to use *index* instead of *columnName* in `ResultSet.get()` methods.

View

- A virtual table that is created based on one or more tables and mimics a table but does not have a physical data store. View is used to query the table or shape it in a specific way and is useful for simplifying more complex queries or presenting a specific point of view.
- **Virtual Table:** View does not represent a physical table. That is, it does not store real data. It is simply a "virtual" table that represents the result of a query.
- **Table Independence:** View hides how data is stored or organized and therefore allows applications to query only the view. This can reduce the need to make changes to applications when the data model changes.
- **Data Security:** Views can be used to allow certain users to see only certain columns or rows. It is very important for data security.
- **Query Simplification:** Views can be used to simplify complex queries involving multiple tables. By defining complex queries in views, you can keep application code clean and understandable.
- **Data Editing (Updateable View):** Some database systems make views that meet certain conditions editable. In this way, it is possible to add, update or delete data via the view. **While Oracle Database and Microsoft SQL Server allow updateable views, SQLite does not.**

SQL Injection

- It is an attack performed by accessing the database and manipulating the database by writing an SQL query from a place (console, textbox, etc.) where user input is taken and data etc. are retrieved from the database.
- SQL Injection is automatically blocked in the SQLite JDBC driver because the `Statement.execute()` and `Statement.executeQuery()` methods only execute one SQL statement. For example, they do not execute a second statement such as `“;DROP TABLE tablename”`.
- One other way to prevent it is to use the **PreparedStatement** class available in java.

PreparedStatement

- It precompiles the SQL queries used for database operations and allows you to substitute values when sending these queries to the database.
- It is part of JDBC, used to access relational databases.
- Main advantages:
 - **Precompilation:** PreparedStatement objects are precompiled by the database before sending SQL queries to the database. This can help the query run faster because each query is not recompiled each time.
 - **Security:** PreparedStatement is more resistant to SQL Injection attacks because query parameters are passed in and automatically sanitized. This helps in securely processing the data entered by the user.
 - **Performance:** PreparedStatement objects provide performance benefits when running the same query multiple times with different values. You can use the same query repeatedly by simply changing the parameter values.
 - **Readability:** PreparedStatement allows you to write code in a more readable and simple way when placing parameters in the query.

```
final String connectionString = "jdbc:sqlite:C:\\Users\\amete\\IdeaProjects\\TestDB\\music.db";
try {
    Connection conn = DriverManager.getConnection(connectionString);

    String sql = "INSERT INTO students (name, age) VALUES (?, ?)";
    PreparedStatement preparedStatement = conn.prepareStatement(sql);

    preparedStatement.setString( parameterIndex: 1, x: "Mete Ahmet Yakar");
    preparedStatement.setInt( parameterIndex: 2, x: 25);

    int rowCount = preparedStatement.executeUpdate();
    System.out.println(rowCount + " row added.");
} catch (SQLException e) {
    throw new RuntimeException(e);
}
```

Transactions

- It refers to the ability to group together one or more SQL queries in the database as a transaction and either perform this transaction completely or not at all.
- Transactions must be **ACID**:
 - **Atomicity**: All SQL queries within a transaction run together and either all complete successfully or none of them run. If an error occurs at any stage, the entire transaction is rolled back and the database returns to its initial state.
 - **Consistency**: When the transaction completes successfully, the database should be in a consistent state. That is, the database maintains its own internal rules and integrity criteria.
 - **Isolation**: When multiple processes run simultaneously, one process is independent of the others. That is, a transaction is not affected by the order and results of other transactions.
 - **Durability**: Once the transaction is completed, database changes are permanent, and data will not be lost even if the system is shut down.

Networking Programming

- In Java, there is **Socket** class for client socket and there is **ServerSocket** class for server socket.
- A number between 0 – 65535 can be selected as the port. If no error is returned from the “**ServerSocket serverSocket = new ServerSocket(port);**”, it means the port has been opened successfully. If an error returns, it means that the port is unusable.

URI

- A URI is a unique identifier of a resource. URI is used to express the name or location of the resource, but that location does not need to be physically accessible. URI is divided into two basic subcategories: URL and URN (Uniform Resource Name).

URL

- A URL specifies the location of a resource and expresses that location in a directly accessible way. A URL contains a complete address that specifies where a resource can be found. For example, “https://www.example.com” is a URL, and applications such as web browsers can directly access the resource using this address.

A URI is an identifier that may not provide sufficient information to access the resource it identifies. A URL, on the other hand, is an identifier that contains information about how to access the resource it describes.

The recommended usage in Java is to use it as a URI first and then convert it to a URL when necessary.

URLConnection

- *Data can be read from a URL with `URL.openStream()`. However, it is more correct to do this with **URLConnection**, which provides more features.*

JPMS (Java Platform Module System)

- It is a structure that brings together one or more packages and expresses these packages explicitly or implicitly to other modules.
- Java modules simplify code organization and maintenance by modularizing larger, complex applications. Additionally, module systems can help better control the classes needed by client applications and prevent the use of unnecessary classes.
- Each module comes with its own module descriptor file (*module-info.java*).
- Modules are divided into two: *normal modules* and *open modules*.
 - **Normal Modules:** Provides access at compile time and runtime only to types in explicitly exported packages.
 - **Open Modules:** Allows access only to types in explicitly exported packages at compile time. Additionally, it allows runtime access to types in all packages as if all packages were exported.
- Each module is identified using the specific module keyword.
- It is good practice to name modules the same as packages.

module-info.java

- **requires:** this keyword expresses the dependency of one module on another module. It is used when a module wants to use classes or interfaces provided by one or more other modules.
- **exports:** This keyword makes packages within a module available to other modules. In other words, exports are used to allow other modules to use a package.
 - Used with the "**exports ... to**" statement to export the package only to a specific module or group of modules.

exports packageName **to** otherModule1, otherModule2, ...;

- **opens:** Access with *Reflection* is determined with the **opens** keyword, that is, modules other than the specified module cannot be accessed with *Reflection*.

opens packageName **to** otherModule1, otherModule2, ...;

- **transitive:** With the **transitive** keyword, the dependency of a module on other modules can be transferred.

If module A uses module B and module C, and module B uses module C. Module A can go to module C by accessing module B.