**Core Java**

**Java 7/ 8 /11 features**

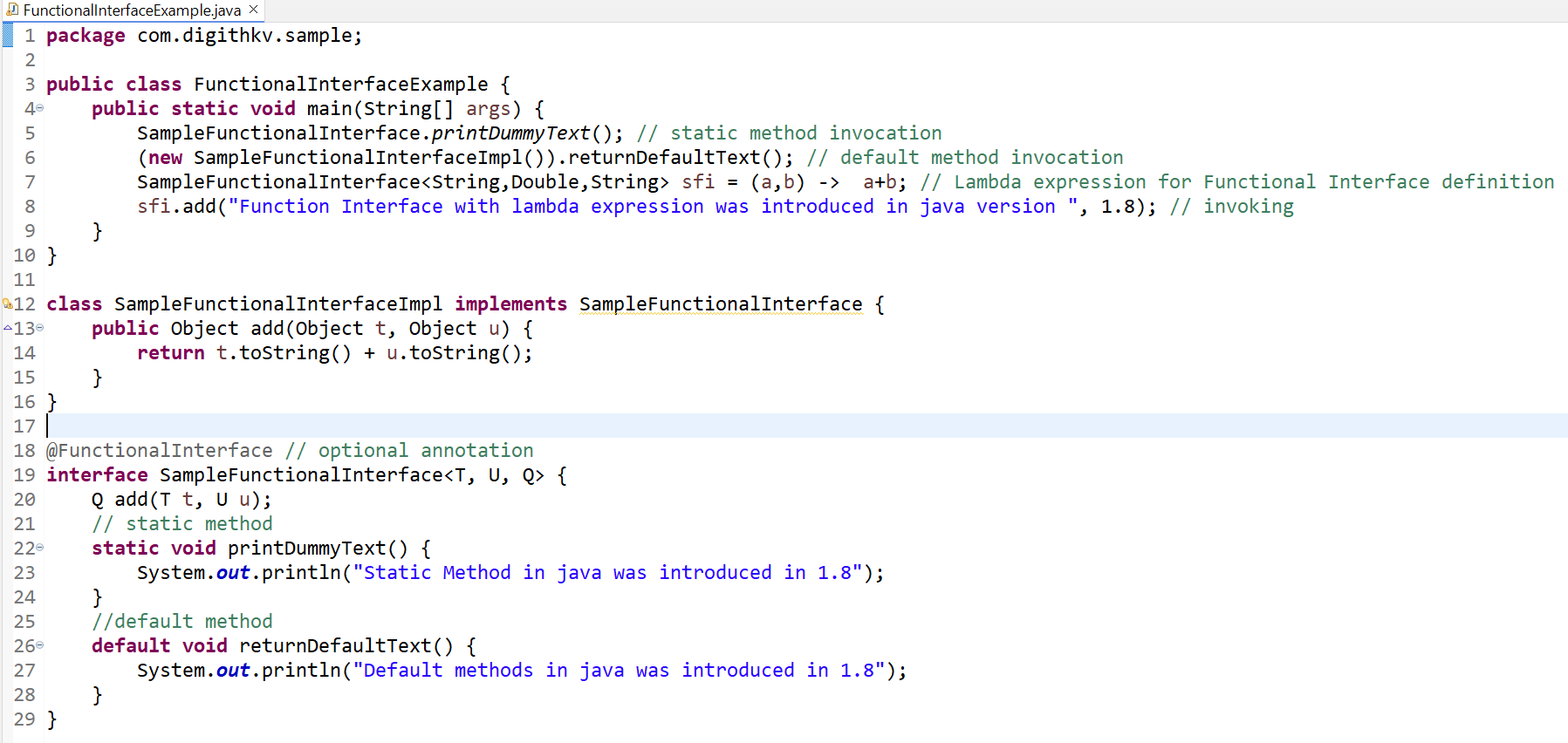
Java 8 features:

* Functional (SAM) Interface, static and default methods, Lambda expression. Interfaces that came with java 8 - Function, Predicate, Consumer, Supplier
* Streams & Parallel Streams
* Optional
* CompletableFuture
* Local Date Time API
* Nashorn Engine

Functional Interface (SAM):

Interfaces with **S**ingle **A**bstract **M**ethod - annotated with @FunctionalInterface (optional). static and default methods in interface were also introduced. Below given code contains:

* SAM Interface (SampleFunctionalInterface) with abstract method add(T t, U u) method.
* Lambda expressions are used to define functional interface (line 7)
* static methods -> avoid utility methods defined in classes (can be called directly without an instance)
* default methods -> avoid repetition in implementing class

eg:

before java 8 :

Runnable(with run()), Callable(with call()), Comparable(with compareTo(T o))

in java 8:

Function (with apply(T t)), Consumer(with accept(T t)), Supplier (with get()), Predicate(with test())

**Lambda Expression** -> to define functional interface and make code concise

Streams & Parallel Stream:

To process collection of elements in a functional style of programming we have **interface** Stream. Intermediate operations over streams **are executed only when terminal operator is there**. Parallel Stream allows you to operate on multiple cores of your machine. Advantages of Streams include:

* No Storage
* Pipeline of Functions
* Laziness
* Can be infinite
* Can be parallelized

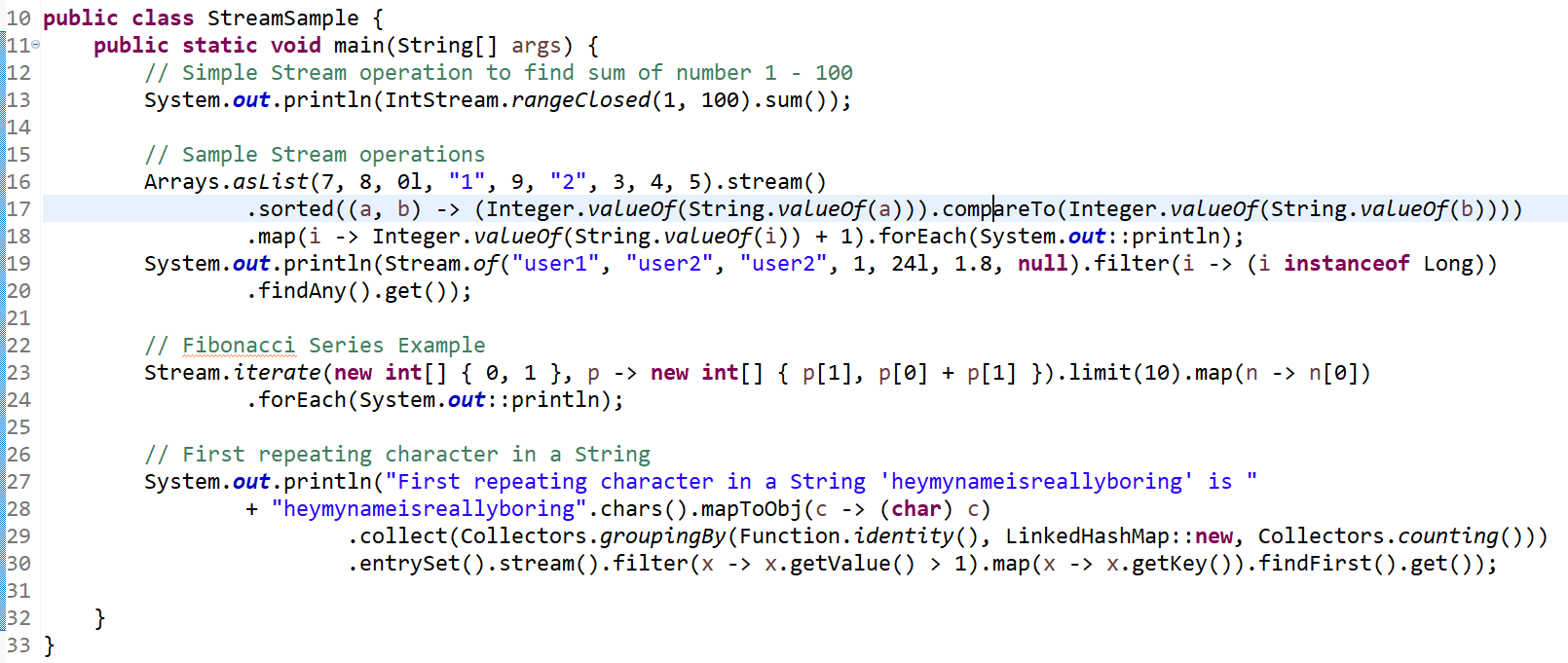
How to create a stream:

1. <Collections>.stream() / parallelStream()
2. Stream.of(T... values) // varags
3. Insteam.range(inclusive, exclusive) / Instream.rangeClosed(inclusive, inclusive)
4. Stream.iterate(initial, UnaryOperator) -> used in Fibonacci series in java 8
5. Arrays.stream()

There are two kinds of operations in Streams:

1. Intermediate (operates and gives another stream)-> map(), flatMap(), filter(Predicate), sorted(), distinct(), mapToLong/Int/Double/Object(), flatMapToLong/Int/Double/Object(), sorted(Comparator comparator), limit(long number), skip(long number)
2. Terminal Operations (operates and give non stream as output) -> collect(Collector collector), reduce(BinaryOperator accumulator), findAny(),findFirst(), toList(), toArray(), distinct(), count(), forEach(Consumer consumer)

Once stream is operated / closed if attempted another stream operation you will get “*stream has already been operated upon or closed*” instead recommendation is to reinitiate stream.



Optional:

provide null check. Created by using Optional.of(reference) (throws NullpointerException) Optional.ofNullable(reference).

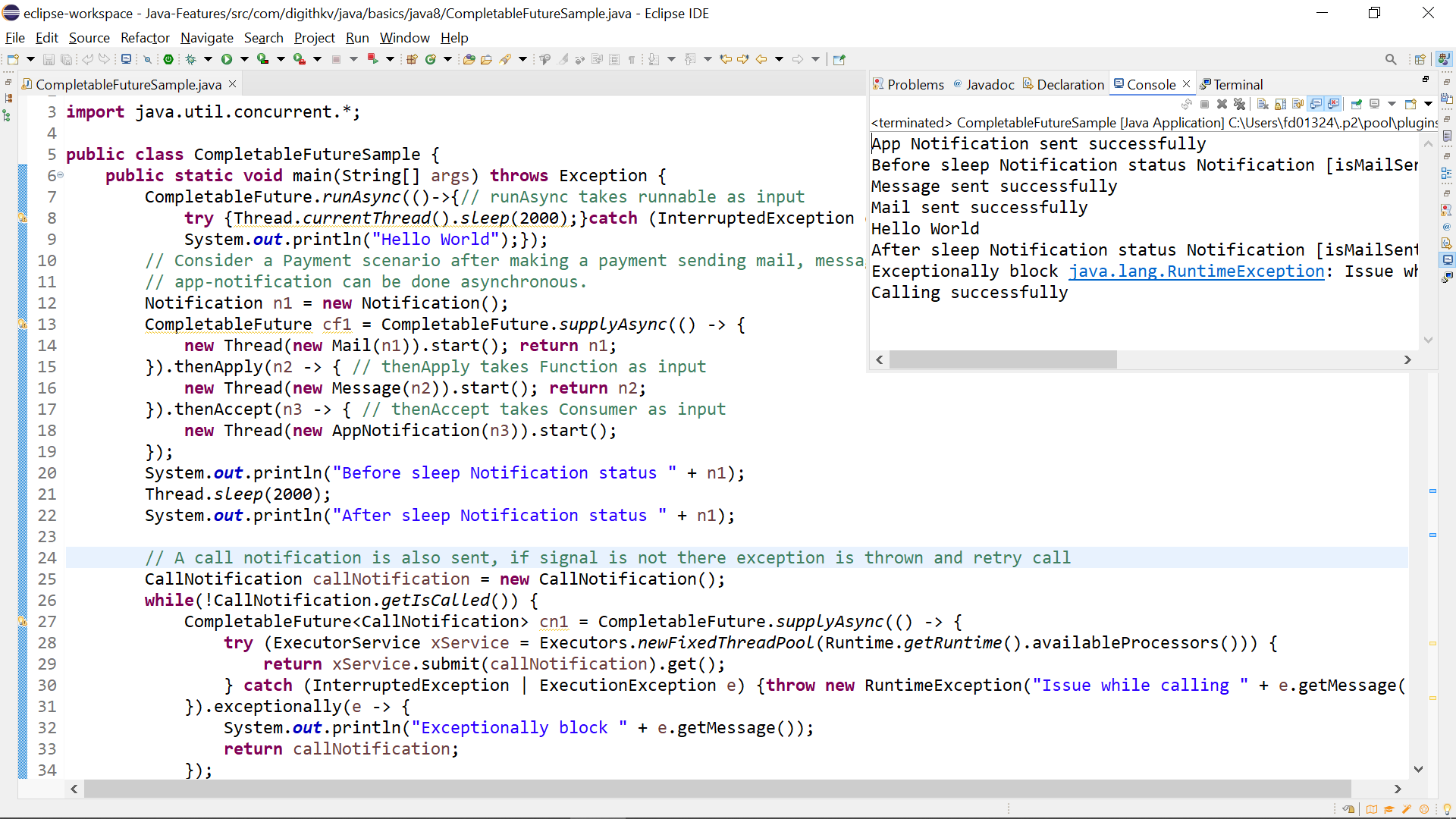
Methods:

isPresent(), isEmpty(), Optional.of(Object),Optional.ofNulllable(Object), filter(Predicate predicate), get(), orElseGet(), orElseThrow(), orElse(Object other), or(Supplier supplier), orElseThrow(CustomException exception)

CompletableFuture:

Provides a reference to asynchronous results and a way to write non-blocking code. When compared to Future provides feature such as:

* Chaining – methods like thenApply(), thenAccept(), thenCompose()
* Exception handling – method like exceptionally()
* Combining – methods like anyOf(), allOf()
* Manual Completion – methods like complete()

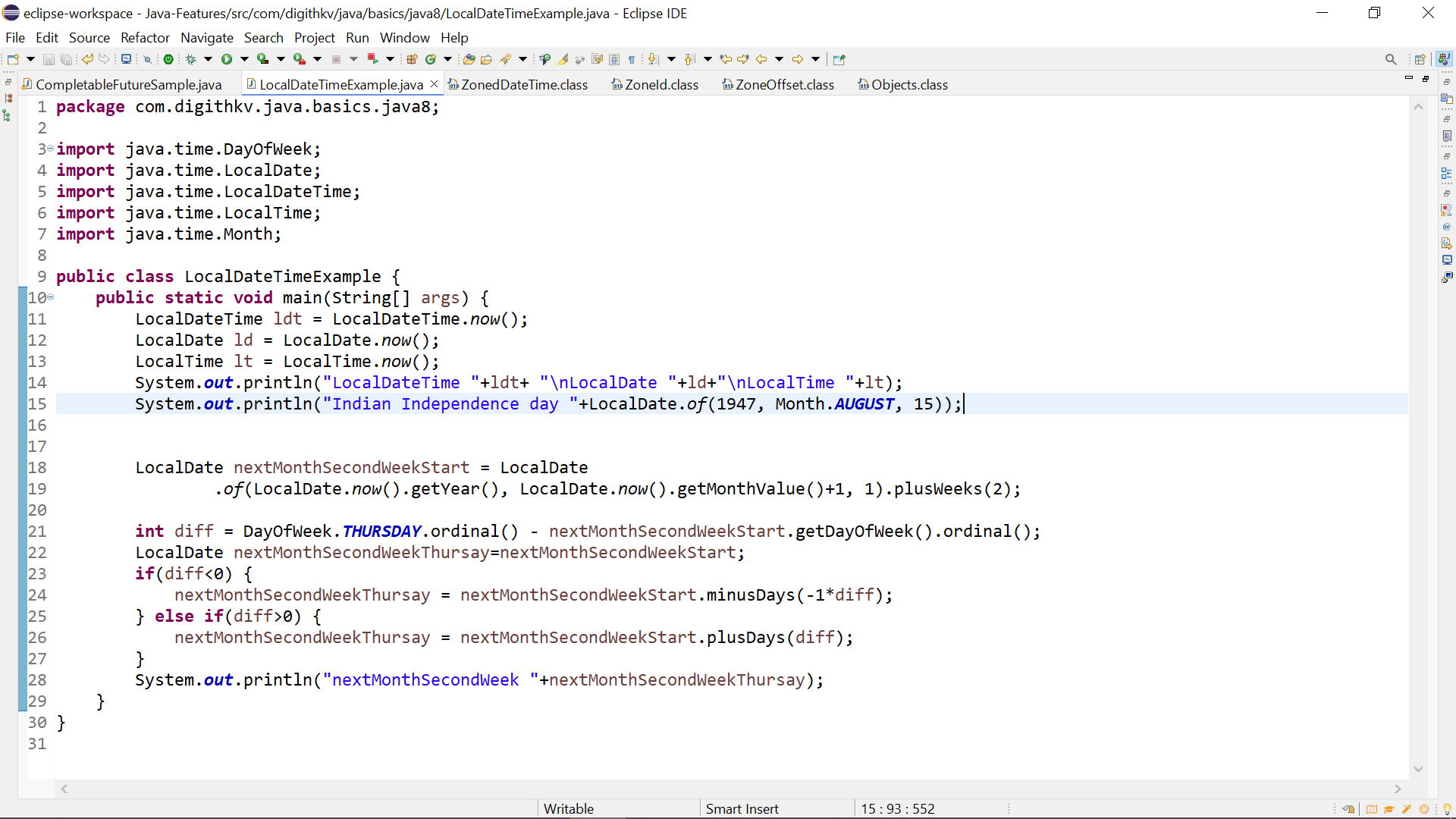


**Stages:**

Creation, Completion, Chaining, Combining Future, Error Handling

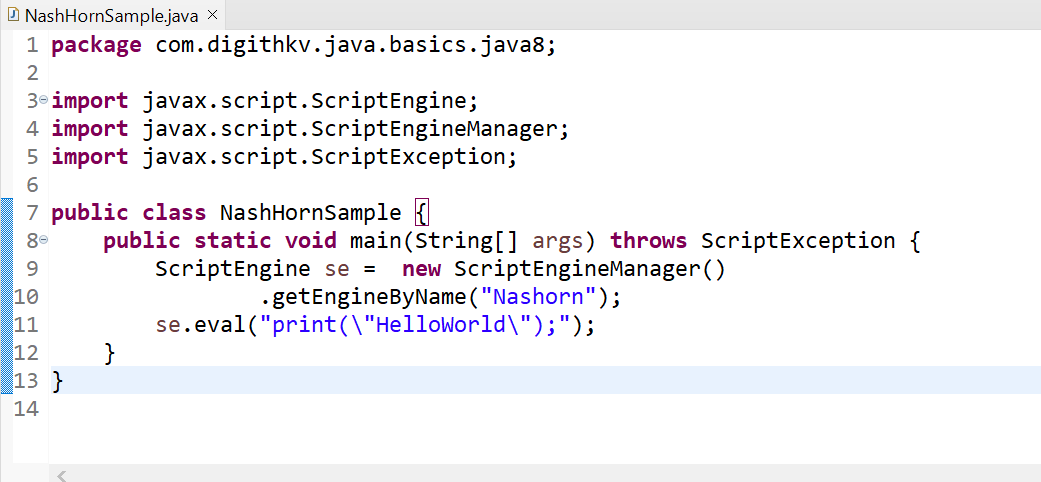
**LocalDateTime API**

LocalDateTime class is an immutable date-time object that represents a date-time. It was a replacement for Date API in java (java.util.Date & java.sql.Date). Along with LocalDateTime API LocalDate and LocalTime was introduced. Incase Timezone also needs to be considered use ZonedDateTime.



**Nashorn Engine**

To execute Java Script code in JVM we can use Nashorn Engine. It was deprecated from java 11 and GraalVM was introduced as a replacement.

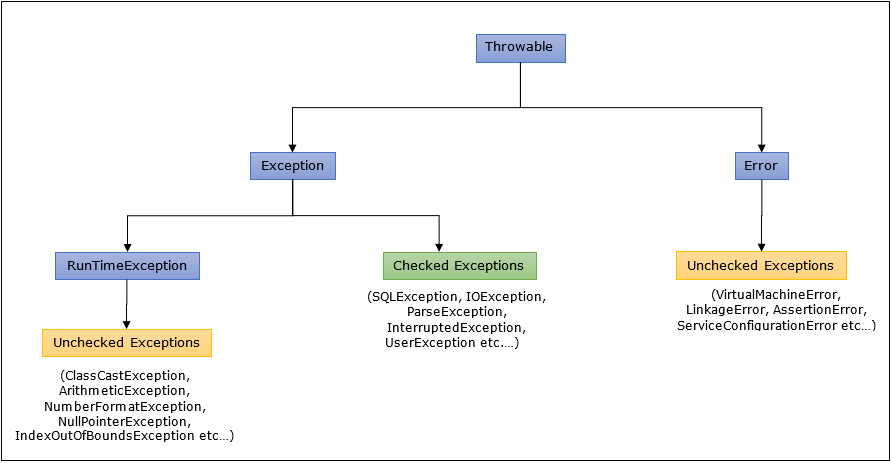


**Java 11 Features**

1. New String methods: isBlank(), strip(), stripTrailing(), stripLeading(), lines(), repeat(int)
2. var keyword: in Lambda expression and more context
3. Java Flight Recorder part of jdk
4. Optional isEmpty()
5. Z GC and Epsilon GC
6. HttpClient
7. Deprecate Thread.stop/destroy(), Nashorn javascript Engine, modules related to Java EE and CORBA, CMS GC
8. File Operations simplified using writeString(), readString()

**Exception Handling**

Exception - Unforeseen / Abnormal scenarios that can lead to unexpected ending of a program which could be handled.



Handling Exception:

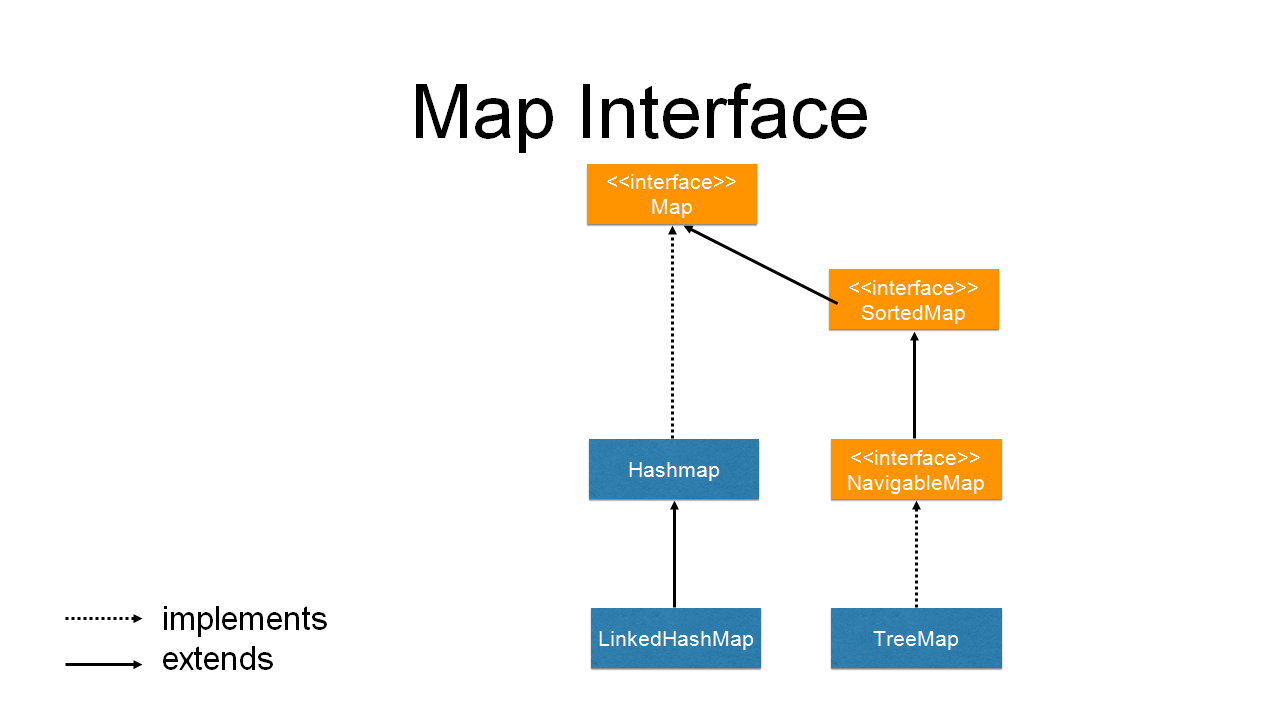
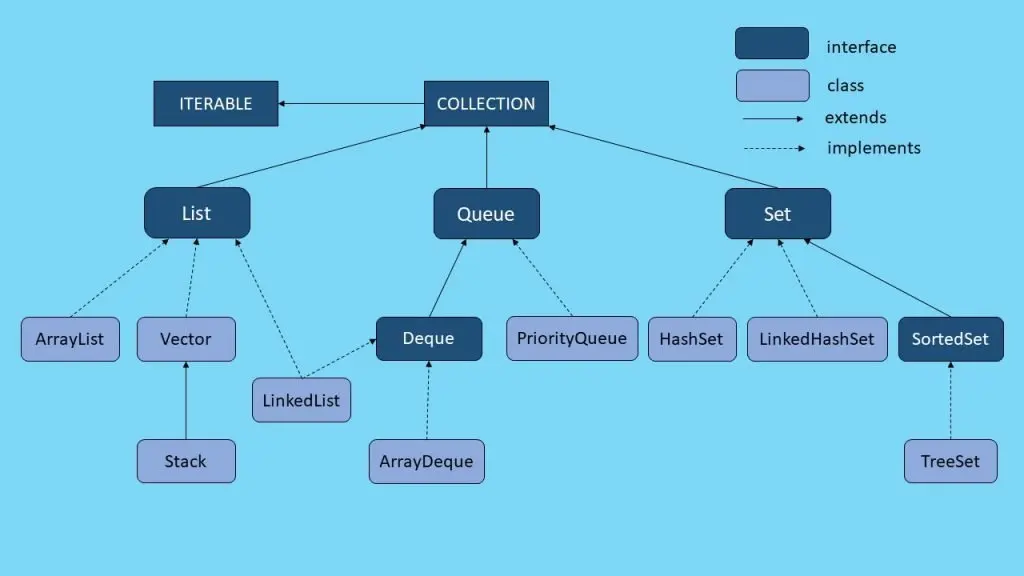
1. by surrounding try catch
2. by throws in the method declaration

Create an Exception:

1. create a new class that “extends Exception”
2. override constructor with message, cause (throwable), message & cause (Throwable)

**Note:** Checked exceptions in an overridden method thrown by a child, should be thrown by respective Parent class method also.

**Collections**

**Collection** - Root interface of collection hierarchy. Group of classes and interfaces that implements commonly reusable collection data structures.

**List** – Ordered sequence

1. ArrayList: Resizable array implementation (non-synchronized)
2. LinkedList: Doubly Linked List implementation
3. Vector: Synchronized resizable array

**Queue** –when needed certain order typically FIFO (other orders also possible)

1. LinkedList: with doubly linked list FIFO can be achieved
2. ArrayDeque: resizable deque implementation faster than LinkedList when operated as Queue
3. PriorityQueue: elements are order by natural order (minHeap) or based on the comparator
4. BlockingQueue: Concurrent application, allows blocking operation (Concurrent Collection)

**Set** –Avoid the duplication of the record

1. HashSet: Uses a hash table for storage order is not maintained
2. LinkedHashSet: Maintains insertion order
3. TreeSet: Implements red black tree for and sort elements

**Map** – Used to storekey, value pair

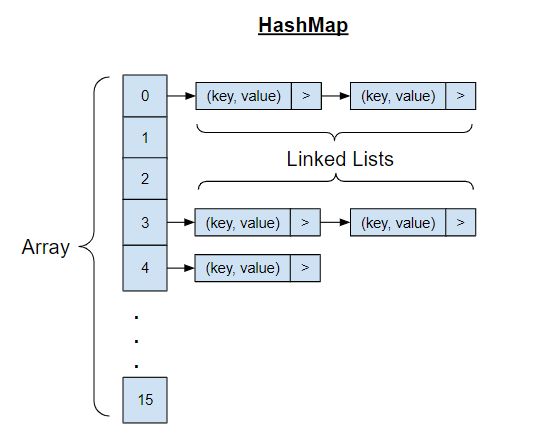
1. HashMap: (most widely used) a hash table, does not maintain order, hashing used.
2. LinkedHashMap: LinkedList as the node, preserves the insertion order
3. TreeMap: Sorted map based on natural sorting or a comparator

**Concurrency in collections**

**ConcurrentModificationException**: Arises in the scenarios were multi thread read and (or) write operation are happening simultaneously – such iterator is called **Fail Fast Iterator**

Set of collections in which above scenarios (thread safe) are handled – they are generally called Concurrent Collections, and their iterator is called **Fail Safe Iterator**. eg: ConcurrentHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet, BlockingQueue, ConcurrentSkipListMap, ConcurrentSkipListSet

**Internal Working of HashMap**



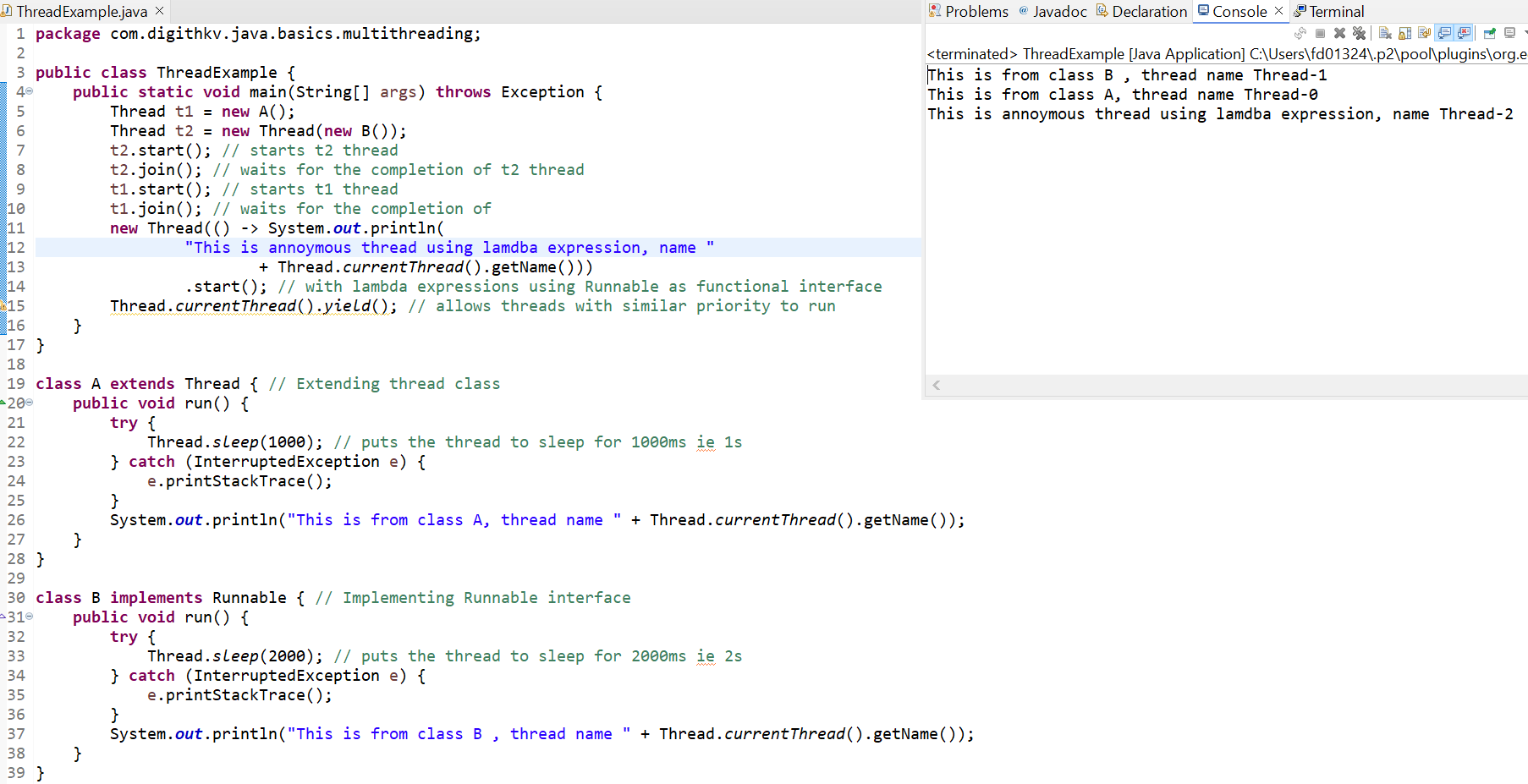
Array of Nodes, by default size 16 when size reaches the load factor (.75) ie, size of 12, resized. When a key value is to be inserted first hashing function is applied, the result is further applied by modulo (% remainder) operator. Go to the index if the Node is empty the next of node will be pointed new Node, if node is not empty then hashcode is first checked if hashcode is equal, then equals is b/w nodes are checked. If nodes are not equal, then a scenario of conflict arises which is called **hash collision**. Chaining, Open Addressing, Robin Hood Hashing are some of the popular implementations. Among these Chaining is most popular in which LinkedList of Nodes are created, in java 8 on wards after a certain length (8 by default) to reduce complexity the linkedList is converted to Binary Search Tree (Red Black Tree), after the size reduces to 6 it is converted back to linkedList.

**MultiThreading / Executors Service**

Creation of Thread:

1. Extending Thread Class
2. Implementing Runnable Interface

We can also use Executors Framework for thread creation (gives a pool of threads from which we can reuse thread).

**Methods:**

start() : Creates a thread instance and calls run() method

run() : need to be overridden and is the actual implementation of thread

join() : makes one thread wait for the completion

sleep() : pauses one thread for certain time from the execution of subsequent steps

interrupt() : Used to interrupt a thread that is in a blocked state.

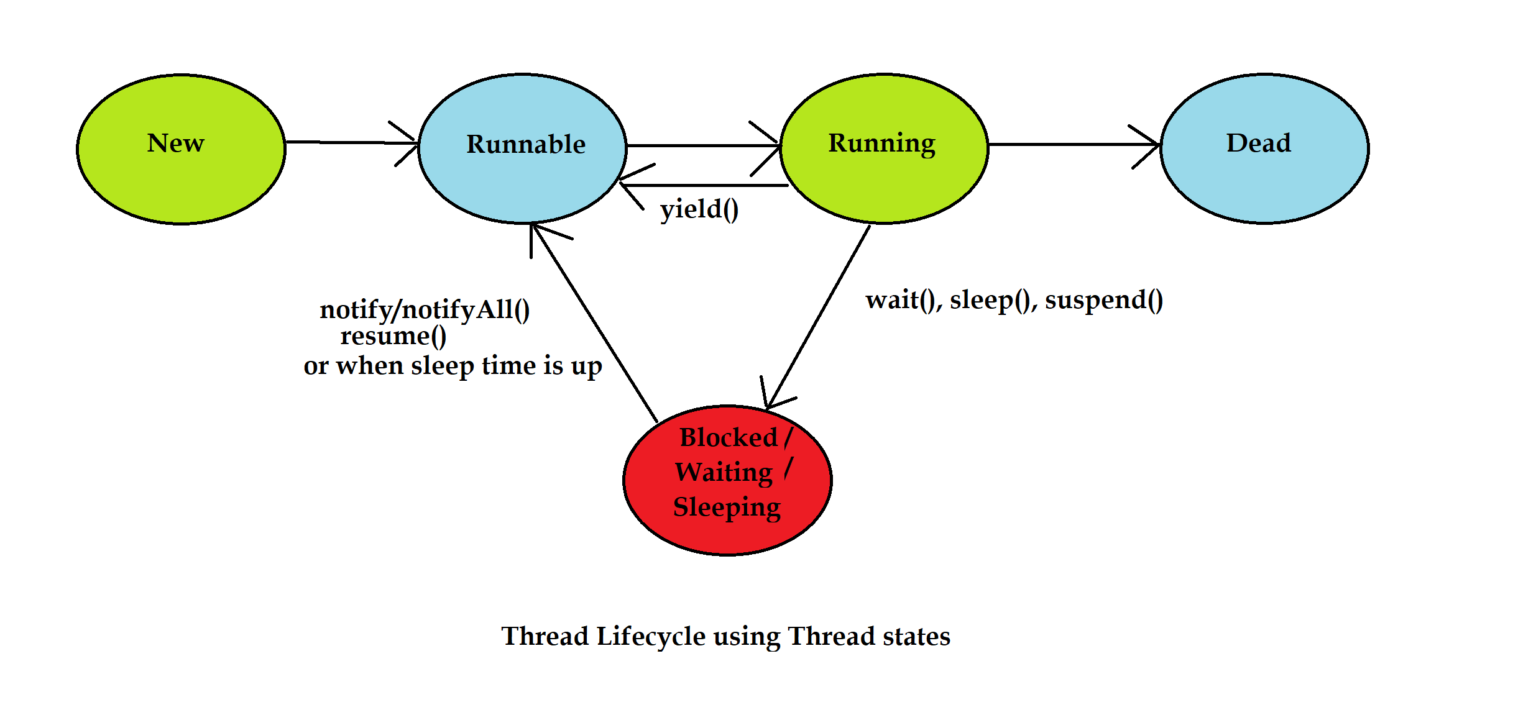
wait() : moves the thread from the execution of subsequent steps to a waiting stage until a notification from another thread is received.

notify() / notifyAll() : notifies one / all threads in waiting stage to proceed with execution of subsequent steps

getName() / getId() : Name and Id of the thread which is executing the thread.

yield(): Allows the execution of other threads with similar priority

**Thread Life Cycle:**

**New (Born state):**

Thread object is created in JVM (using new keyword)

**Runnable:**

When start() method is called runnable ie thread object is pushed to thread pool waiting for the thread scheduler to pick and start execution.

**Running:**

When thread scheduler picks and run() is invoked on a thread. A thread in running state moves back to runnable when yield is invoked

**Blocked / Waiting / Sleeping:**

When wait(), sleep(), suspend(), join() methods are called on a thread go to a waiting / blocked state. Once the thread recovers from this state it waits for the thread scheduler to pick the thread for the execution.

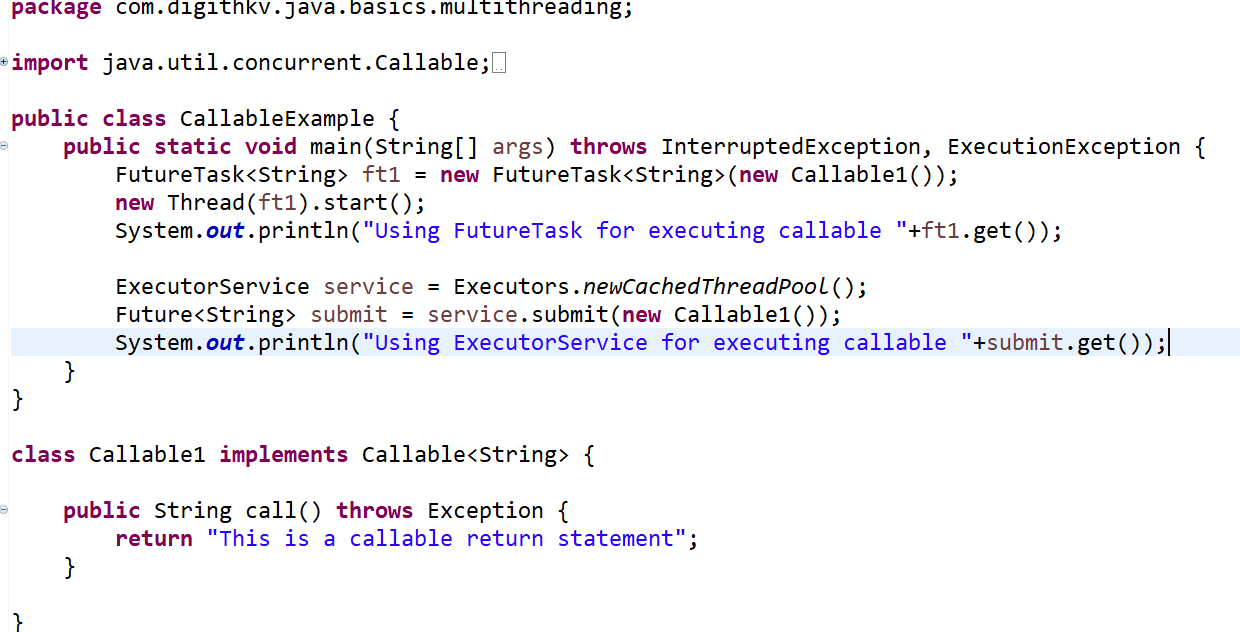
**Dead:**

Once run() method completes its execution the corresponding thread is terminated

**Callable:**

In case you are expecting some value to be returned from a thread then we use Callable Interface. Functional interface with call() method. Can be invoked using

1. Can use FutureTask
2. ExecutorsService



**Dead Lock:**

Two or more threads are unable to proceed because each is waiting for the other to release resources.

**Why Dead Lock occur:**

1. Mutual Exclusion (b/w thread): when one of the resources is held in a non-sharable mode, only thread can use the resource at a time
2. No Preemption: Resources can’t be forcibly taken by another thread, if need to use a resource must be released first.
3. Circular await: Each thread waits in a chain for the release
4. Hold & wait: when a thread is already holding a resource and is waiting to acquire a resorce that is acquired by another thread

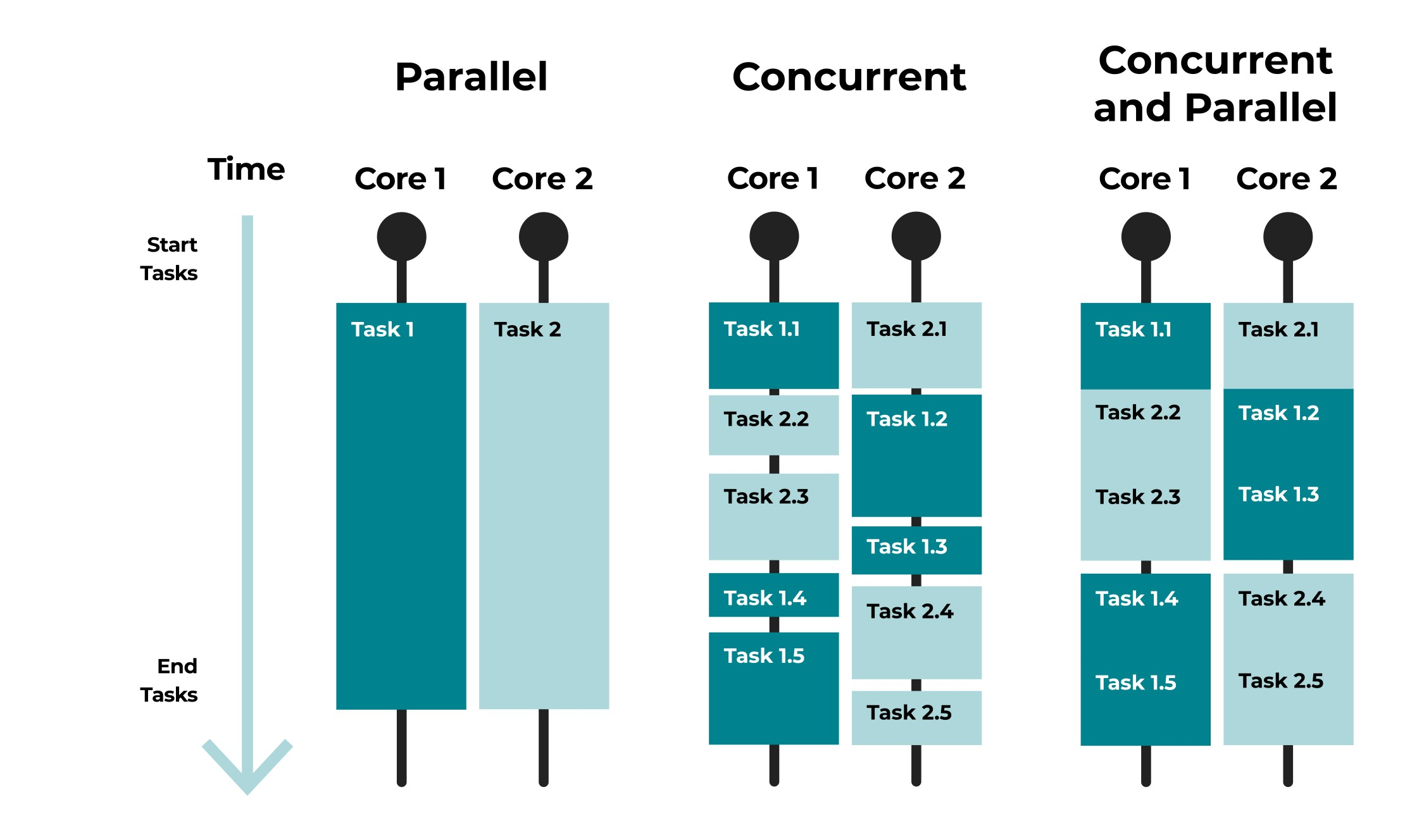
**Prevent Dead Lock:**

1. Avoid Nested Locks on resources / unneceesary locks also minimize scope of locks
2. Provide timeouts in lock ( for reentrant locks -> locks.tryLock(1000,TimeUnit.MILLISECONDS)
3. Use Immutable Objects
4. Establish a global order on accessing the resource

**Concurrency vs Parallelism**

Concurrency: Managing multiple tasks simultaneously

Parallelism: Executing tasks simultaneously

**volatile, synchronized, Lock, AtomicReference, ThreadLocal:**

**volatile**: value must be referred from main memory rather than cache, used for flags (boolean) solves visibility

**synchronized**: control the access for a block of code to a thread. Avoid race condition

**Locks**: complicated and flexible synchronization. It offers sequence guarantee, provides timedLock and lock(), unlock() can be called from different blocks of an interface. Condition based (read, write) locking is also possible.

**Atomic references**: ensures thread safety in **modification** of variable which happens through atomic operations (unlike volatile – which can handle visibility problem only) can be used in case of counters and other variables eg: AtomicInteger/Array, AtomicBoolean /Array, AtomicLong /Array, AtomicReference<T> /Array

**ThreadLocal**: ensures local copy for each thread and thereby thread safety while using a resource. Only visible to one thread and doesn’t require any synchronization.

**Executors Framework:**

Provide easiness in managing and handling large number of threads asynchronously. **Provides thread pool**, managing concurrency, scheduling task. Different methods for creating pool

*newCachedThreadPool*()

*newSingleThreadExecutor*()

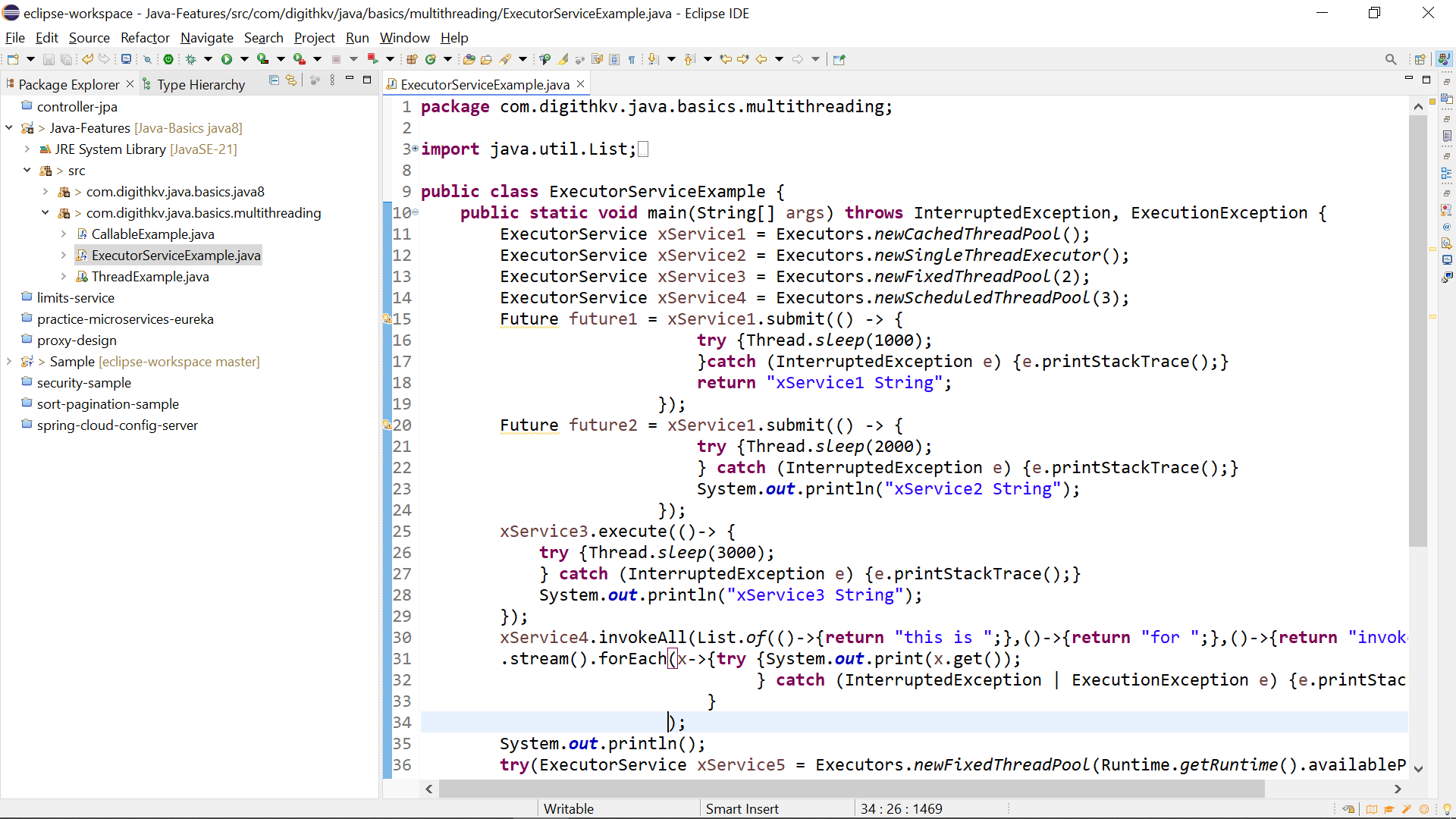
*newFixedThreadPool*(int poolsize)

*newScheduledThreadPool*(int poolsize)

ExecutorService has mainly submit(Callable /Runnable) and execute(Runnable) method and other methods include: invokeAll(Collection<Callable>), invokeAny(Collection<Callable>), isShutDown(), shutDownNow(). ExecutorService if created should be closed with shutDown() (if created directly).

**submit**(Callable /Runnable) takes Callable / Runnable and return Future reference for the result

**execute**(Runnable) can take Runnable only and doesn’t return anything



**Design Patterns**

**SOLID** Principle

**S**ingle Responsibility principle: every Java class must perform a single functionality

**O**pen Closed principle: the module should be open for extension but closed for modification

**L**iskov Substitution principle: derived classes must be completely substitutable for their base classes

**I**nterface Segregation principle: larger interfaces must be split into smaller ones

**D**ependency Inversion principle: use abstraction (abstract classes and interfaces) instead of concrete implementations

## 1.Creational Design Pattern

1. **Factory Pattern**
2. Abstract Factory Pattern
3. **Singleton Pattern**
4. **Prototype Pattern**
5. **Builder Pattern.**

## 2. Structural Design Pattern

1. **Adapter Pattern**
2. Bridge Pattern
3. Composite Pattern
4. Decorator Pattern
5. **Facade Pattern**
6. **Flyweight Pattern**
7. **Proxy Pattern**

## 3. Behavioral Design Pattern

1. **Chain Of Responsibility Pattern**
2. Command Pattern
3. Interpreter Pattern
4. Iterator Pattern
5. Mediator Pattern
6. Memento Pattern
7. **Observer Pattern**
8. State Pattern
9. Strategy Pattern
10. Template Pattern
11. Visitor Pattern

**Generics**

Generics are used to create Generic Classes and Generic methods which can work with different types (Classes).