# Java 8

## Lambda expressions

* + Functional interface
    - Predicate<T> -> **boolean** *test*(T t)
    - BiPredicate<T, U> -> **boolean** *test*(T t, U u)
    - Supplier<T> -> **T** *get()*
    - Consumer<T> -> **void** *accept*(T t)
    - BiConsumer<T, U> -> **void** *accept*(T t, U u)
    - Function<T, R> -> **R** *apply*(T t)
    - BiFunction<T, U, R> -> **R** *apply*(T t, U u)
    - UnaryOperator<T> -> **T** *apply*(T t)
    - BinaryOperator<T> -> **T** *apply*(T t1, T t2)
  + Final and Effectively Final
    - Lambda takes a snapshot of local variables. Local vars that’s used in lambda MUST NOT change (effectively final), it gives compiler error, even if it’s changed before lambda definition.
  + Method References – turns into lambdas in the background
    - Bound – bounded to some instance -> don’t have to specify which instance to call it on
    - Unbound – need to specify which instance to call it on. First parameter is used for executing the instance method
    - Static (Unbound) – calling static method in lambda
    - Constructor MR – calling constructor in lambda -> in MR just Type::new

## Streams

* Source – where the stream comes from e.g. array, collection or file
* Intermediate operations – transforms the stream into another one
  + filter()
  + distinct() – returns a stream with duplicate values removed
  + limit()
  + map() – creates a one-to-one mapping between elements in the stream and elements in the next stage of the stream. It transforms the data.
  + flatMap() – takes each element in the stream e.g. Stream<List<String>> and makes single stream Stream<String> that contains all elements
  + sorted(Comparator?.comparing(…))
* Terminal operation – required to start the whole process and produces the result
  + *reduce*() – combines a stream into a single object. It processes all elements
    - T reduce(T identity, BinaryOperator<t> accumulator)
    - Optional<T> reduce(BinaryOperator<T> accumulator)
    - <U> U reduce (U identity, BiFunction accumulator, BinaryOperator combiner)
  + *collect()* – mutable reduction because we use the same mutable object while accumulating. This makes it more efficient than regular reductions
    - Collectors
      * .joining()
      * .averagingInt()
      * .toMap()
      * .groupingBy(Predicate)
      * .partitioningBy(Predicate) – special case of grouping where there are only two possible groups *true* and *false*

## Collections and Generics

## Concurrency

* new Thread().start() -> creating new thread and method is executed in that new thread
* new Thread().run() -> doesn’t create new thread, method is executed in same thread
* new Thread().join(); -> main thread must wait here until new Thread() finishes task
* ExecutorService
  + Single thread pool executor
    - Tasks are processed sequentially
  + Cached thread pool executor
    - Creates new threads as needed and reuses threads that have become free
    - Care needed as the number of threads can become very large
  + Fixed thread pool executor
    - Creates a fixed number of threads which is specified at the start
  + A *Future<V>* object represents the result of an asynchronous computation.
    - Methods are provided to check if the computation is complete – *isDone()* and
    - To retrieve the result of that computation – *get().*
    - The result can only be retrieved using the method *V get()* when the computation has completed, blocking if necessary until it is ready.
  + *es.execute()* – execute the **Runnable task** asynchronously
  + *es.shutdown()* – shutdown the executor service, otherwise this application will never terminate. Existing tasks will be allowed to complete but no new tasks accepted
  + *es*.*submit() –* execute the **Callable task** (asynchronously) to the executor service and store the Future object
  + *future*.get(500, TimeUnit.MILLISECONDS) – get() will block for 500 msecs max
  + *es.invokeAll() –* maintained oreder
  + *es.invokeAny()*
  + *schedES.schedule(() -> {…}, 2, TimeUnit.SECONDS)*
  + *schedES.scheduleWithFixedDelay(() -> {…}, INITIAL\_DELAY, WAIT\_PERIOD\_AFTER\_PREV\_TASK\_FINISHED, TimeUnit.MILLISECONDS)*
  + *schedES.scheduleAtFixedRate() –* wait\_period accumulates
* *synchronized* – locks
* *Lock interface* – with synchronized, a thread is blocked if a previous thread has the lock, whereas with *Lock*, if we are unable to get the lock we are free to perform some other task
* *lock.lock(), lock.unlock()*
* *if (lock.tryLock()) …unlock} else {…}*

## Localization

## JDBC

## Java I/O API

# Java 11

## Local Variable Type Inference (LVTI)

- Since Java 10, we have the option, in certain scenarios, of using the keyword ***var*** instead of the type. In these situations, the compiler infers the type e.g. var x = “abc”; // x is a String

- Variables must be **local** and must be initialized on the line where they are declared

## Private Interface Methods

- Since Java 9, interfaces can now have private methods. They can be static and non-static.

- Why? 1) Reduce code duplication. 2) Improves encapsulation by hiding implementation details

## Annotations

public @interface MyAnnotation {

int myElement(); // looks like an abstract method

}

JVM translates the above element into an interface method and the annotation itself as an implementation of the interface.

## Security

## Modules

# Java 17

## Sealed Classes

- Sealed classes enable us to conrol the scope of inheritance by enabling us to specify a classes’ subtypes

- Also works with interfaces – we can define what classes implement the interface

## Records

- Records are a special type of class that help avoid boilerplate code. There are considered “data carriers”

- Records are immutable and are final by default. We cannot extend our custom record because records already (implicitly) extend from the *Record* class

- Can have both static fields and static methods, can have instance methods, but cannot have instance fields. All the instance fields are listed as “components” in the record declaration.

- Records can implement interfaces

## Pattern-matching switch statements

# Java 21

## Unnamed Classes and Instance Main Methods

The goal is to focus on the “programming in the small” by reducing ceremony/scaffolding for those learning the language.

- Unnamed classes:

* exactly like normal classes except that an unnamed class has only one constructor – the default no-args provided by the compiler.
* Must have a main() method

## Record Patterns

A record pattern does two things for us:

1. Checks if an object passes the *instanceof* test
2. Disaggregates the record instance into its components

## Pattern Matching for Switch

## Sequenced Collections

Easy to iterate both forwards and backwards

## 