Functional Programming in Java

Big Picture and Intuition

Imperative code: focuses on how a task is performed.
Each line of code gives a specific procedure or operation
Eg for loops and while loops

Functional code: also called declarative programming. focuses on what happens to input to produce output
Eg Lambda expressions, streams

Using Lambda expressions

- Lambdas are a programming feature that makes it very easy to implement certain kinds of interfaces in Java.
- A predicate is a function that takes a single argument and returns a boolean.
- Predicate<T> is an interface for predicate functions

Eg:
```
public final class LambdaMain {
    public static long countMatchingStrings(
    List<String> input, Predicate<String> condition) {...}

public static void main(String[] args) {
    List<String> input = List.of("hello", "I", "world",
            "!", "It", "is", "a", "good", "day", "");

long numberOfWhitespaceStrings =
        countMatchingStrings(input, new Predicate<String>() {
@Override
    public boolean test(String s) { return s.trim().isEmpty();}
        });
System.out.println(numberOfWhitespaceStrings + " whitespace strings");
    }
}
```

This can become:

- using lambda expression

long numberOfWhitespaceStrings = countMatchingStrings(input, s -> s.trim().isEmpty());

How did this transformation occur:

Replace predicate: "new Predicate<String>() { public boolean test"
with (String s) ->

↓

Now we have: (String s) -> { return s.trim().isEmpty(); };

Since lambda contains a single statement, you can remove curly brackets, hence get rid of return

↓

Now we have: (String s) -> s.trim().isEmpty();

Since java infers type to parameters, can get rid of "String" and since lambda has 1 parameter, can remove parenthesis

↓

Now we have: s -> s.trim().isEmpty();

Functional Interface

- a functional interface is an interface with exactly one abstract method, called a functional method

Eg:

```
public interface Predicate<T> {
    boolean test(T t);
    default Predicate<T> negate() { return (t) -> !test(t); }
}
```

When to use functional interface ?

- ideal for describing a single operation

Eg: BinaryOperator<Integer> add =
(Integer a, Integer b) -> { return a+b; };

- Predicate interface is a functional interface :
  - one abstract method is test()
  - test() is known as the functional method.
  - functional interface can have other default method.
    Eg negate()

- java.lang.Object methods eg toString() and equals() are not allowed in the functional method

* java.util.function :
  - contains predefined functional interface

Anonymous Subclasses

- a class that is defined "in-line" and has no name

Anonymous Class	vs	Lambda
- class generated at compile time		- Class generated at runtime
- can override equals() hashCode()		- Cannot override them; has no identity
- this refers to the anonymous class		- this refers to the enclosing class

* inside a static method there is no enclosing class

Edge Case : Capturing Variable

- Lambda can capture variable from surrounding scope
- If Lambda uses any variable from surrounding scope, then variable becomes captured variable. Variable can only be captured if they are effectively final.
- an effectively final variable is a variable whose value does not change after it is initialized

- a good way to test if a variable is effectively final is to add `final` keyword to it

Eg: `List<Runnable> runnables = new ArrayList<>(10);`
`for (int i = 0; i < 10; i++) {`
`runnables.add(() -> System.out.println(i));`
`}`

* `i` is not effectively final.

To get around this:

`List<Runnable> runnables = IntStream.range(1,10).map(i -> () -> System.out.println(i)).collect(Collectors.toList());`

Method References

- it is a short lambda expression that refers to a method that is already named

Eg: `String::valueOf` is equivalent to `:o -> String.valueOf(o);`

The Stream API

- a stream is a sequence of elements

Stream Pipeline:

Create stream → Operat → operat → ... Terminal

- Streams are single use
- lazily evaluated → no computation until the very end

Eg:
```
int getTySum ( List ( Stud ) students ) {
    return students.stream()
        .filter( objects :: nonNull )
        .mapToInt ( Stud :: getScore )
        .sum()
        .orElse (0);
}
```

Stream API: Collectors

- a collector is a terminal stream operation that accumulates stream elements into a container

- The collect() method is a terminal operation that aggregates stream of elements

Eg: Set(String) s = strList.stream().collect((Collectors.toSet()));

- Collectors can be used to perform reduction operations such as adding or counting

Eg: Map(Year, Long) graduateClassSize = studentList.stream()
    .collect(Collectors.groupingBy(Student :: getGraduationYear,
    Collectors.counting()));

- groupingBy() is used to collect elements into a Map
- Collectors.counting() counts the number of values for each key

Optional Type

- use java.util.optional class
- use Optional with streams API

- Optional is used as an alternative to using null to represent the absence of a value

- Optional can have methods invoked on it without throwing NullPointerException

Working with files and I/O

Big picture and intuition

Program memory vs Persistent Storage:

Program memory:
- variables, objects and data structures stored in the heap and stack.
  Memory access is fast
- Memory erased when program is done running

Persistent Storage:
- files, stored on disk
- web stores
- databases

File and path APIs

- files API is Java classes and methods for file operations
  It contains static methods for doing all sorts of file operations

Example of file open options:

READ → open a file for reading, fail if it doesn't
CREATE → Create a file
CREATE_NEW → same as create, but fail if file already exists
WRITE → Open a file for writing

APPEND → used to add to end of file

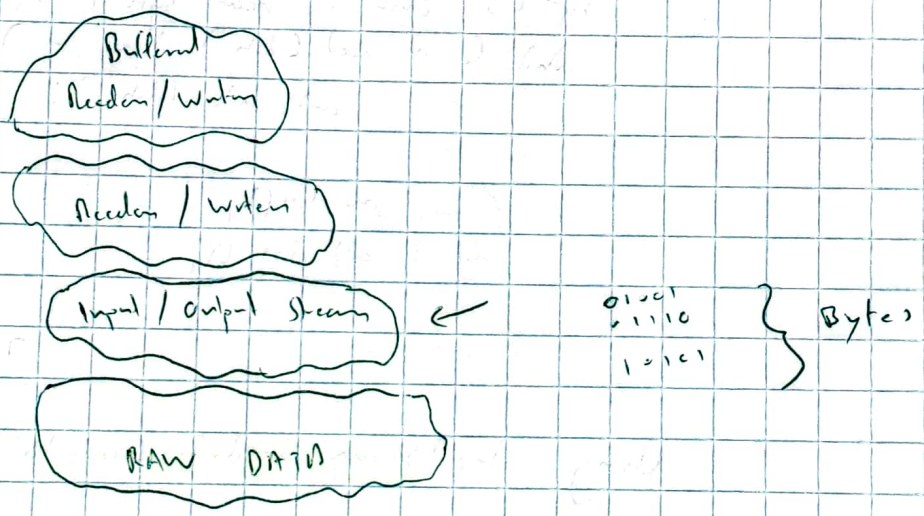
Path : Java's way to refer to a file on a file system

Eg Path p = Path.of("your/path/here");

- absolute paths start with a forward-slash (/)

Input & Output Streams

- in java utilities for reading and writing data are built on top of each other



We use input/output streams to work with bytes

InputStream Example :

```
InputStream in =
        Files.newInputStream(Path.of("file"), StandardOpenOption.READ);
        byte[] data = new byte[10];

    while (in.read(data) != -1) {
                useData(data);
                }

        in.close();
```

OutputStream Example :

```
OutputStream out = Files.newOutputStream( Path.of("hello"));
out.write("Hello, World!".getBytes());
out.close();
```

Ways to copy a file :

Reading and Writing data directly :

```
InputStream in = Files.newInputStream( Path.of(args[0]));
OutputStream out = Files.newOutputStream( Path.of(args[1]));

byte[] data = new byte[10];
while ( in.read(data) != -1 ) {
        out.write(data);
}

in.close();
out.close();
}
}
```

Using InputStream.transferTo(outputStream)

```
InputStream in = Files.newInputStream( Path.of(args[0]));
OutputStream out = Files.newOutputStream( Path.of(args[1]));
in.transferTo(out);
in.close();
out.close();
```

Using the files API

```
Files.copy( Path.of(args[0]), Path.of(args[1]));
```

Reader and Writer

Reader / Writer ← a character stream

- reader / writer are high level abstraction built on top of input and output stream

Reader Example :

```
char[] data = new char[10];
Reader reader =
    File.newBufferedReader(Path.of("test"),
                           StandardCharsets.UTF_8);
while (reader.read(data) != -1) {
    use Data(data);
}
reader.close();
```

Edge Case : Character Encodings

- character set enumerate all possible characters that can be represented by an encoding
- Unicode is the most common character set

Character encoding is a way to convert binary data and human-readable text characters in a character set.

Different Unicode Encodings :

UTF-8 : generally use UTF-8, uses at least 8 bits of data to store each character

UTF-16 : use for non-English or non-ASCII characters

Buffered Streams

- most common buffer streams are BufferReader and BufferWriter

- we use Buffered stream to reduce number of I/O operation performed by your program

Eg: BufferedReader reader =
Files.newBufferedReader(Path.of("test"), StandardCharsets.UTF_8);
String line;
while ((line = reader.readLine()) != null) {
useString(line);
}
reader.close();

Eg: BufferedWriter writer =
Files.newBufferedWriter(Path.of("test"), StandardCharsets.UTF_8);
writer.write("Hello, ");
writer.write("World!");
writer.flush(); // write content to buffer
writer.close();

- emptying out the buffer is called flushing

Preventing Resource Leaks

- use try/catch/finally to prevent resource leaks
- apply try-with resource on the closeable interface to prevent resource leaks
- prevent closing resource multiple times

Eg: try (Writer writer = Files.newBufferedWriter(Path.of("test"))) {
writer.write("Hello World");
} catch(IOException e) {
e.printStackTrace();
}

- only Closeable and AutoCloseable object can be used in the try statement.

Java Object Serialization

- Serialization is the process of converting an object into a data format that can later be deserialized back into an equal object

- to serialize java object, create an ObjectOutputStream and pass the object to the stream with Object() method.
- the ObjectOutputStream writes a sequence of bytes, and its constructor takes another stream as an argument.

Working with JSON and XML

JSON (JavaScript Object Notation)

Eg: {
"id": 17,
"name": "Geeya Wahyuni",
"email": ["geeya.allen@.com", "put@yahoo.com"]
}

XML (Extensible Markup Language)

Eg: <id> 17 </id>
<name> Geeya Wahyuni </name>
<emails>
<email> geeya@gmail.com </email>
<email> putra@gmail.com </email>
</emails>
</client>

JSON serialization / Deserialization with Jackson
- to serialize → ObjectMapper.writeValue()
- to deserialize → objectMapper.readValue()

XML Serialization / Deserialization with JAXB

- to serialize -> Marshaller. marshal

- to deserialize -> Unmarshaller. unmarshal

Design Pattern

SOLID PRINCIPLES :
- Single Responsibility Principle
- Open Closed Principle
- Liskov substitution principle
- Interface segregation principle
- Dependency Inversion principle

A design pattern is a general solution to certain kinds of common design issues that occur in software developments

There Categories : Creational, Behavioral, Structural

Creational : Pattern that deal with creating object
Behavioral : pattern that deal with how different object interact
Structural : pattern that deal with how different object fit together

Creational Pattern

Singleton Pattern

Use a singleton if :
- class has only one instance, but we dear owner
- you want that instance to be available everywhere in your code
- it instance is initialized only when it first used (lazy initialization)
- Enum types are always singletons

Note : singleton can be detrimental

Abstract factory

- a factory is anything that creates objects
- if the thing creating objects is also an object, it's known as an abstract factory

When to use abstract factory:
- you want to hide construction details from callers
- you want to encapsulate construction of several related objects into a single java interface

Builder

What is a builder?
- a mutable factory that constructs its state of a to-be-created object, property by property, then builds the object
- usually support method chaining
- often used to create immutable data objects

Behavioral Patterns
- Strategy pattern
- Template method pattern

Strategy
- you define an interface to represent a kind of task / problem
- each concrete implementation defines a different "strategy" for solving the task
- the strategies can be swapped for each other because the callers code against the interface

Lambdas can be used to implement concrete strategies when the task is a functional interface

Template Method

What is the Template Method Pattern?
- define a base class or interface for a procedure or algorithm, but leave empty placeholders for some part of the procedure
- each placeholder is a stub or default method in base class
- base class acts as a template

Structural Pattern
- involves how objects fit together to form the structure of the software

Adapter Pattern
- use adapter whenever you need to translate an API or interface into another
- typically "wraps" an existing interface to adapt it to a different interface

Decorator Pattern
- Adds new functionality to an existing object dynamically by "wrapping" it. Favors composition.

Adapter vs Decorator
- these pattern "wrap" another object, called the delegate
- an adapter return a different interface than the delegate
- a decorator return the same interface, but with added functionality or responsibility
- A proxy is similar to a decorator, but the proxy usually controls or manages access to the delegate

Dependency Injection

A dependency is anything your code needs to work eg external library, an environment variable, a remote website or a database

What is Dependency Injection?

- It is a design pattern that means the creation of dependencies is outside your code.
- you tell DI framework to create object for you and then you inject them object into your class

Using @inject Annotations

Eg: @inject private Database db;

Eg: @inject
Course Register (Database db, Class class, Registration factory)

How DI injects objects:

- DI framework will attempt to instantiate any object that is injected
- DI framework use modules to configure which classes or objects should be used when an interface is injected

Using DI to create singletons:

- when DI returns specific instance of an object whenever it is injected
- everytime it is requested by @inject, DI framework supply same instance, making it effectively a singleton

Reflection

Big picture and Intuition

what is reflection ?

- Reflection (sometimes called introspection) is the ability of a program to examine its own structure at runtime

Static vs Dynamic Code

Static :

eg Foo myObject = new Foo();

- gives benefit of static analysis
- Java compiler will return an error if you used an invalid field or method name

Dynamic :

eg : Object myObject = Class.forName("Foo").getConstructor().newInstance();

- if code is run and there is no class named "Foo", a class NotFoundException will be thrown.

Annotations

- annotations are a way to provide extra metadata about your program

What can annotations do ?

- can add extra information for the Java compiler, which helps the compiler detect additional errors at compile time
- annotation metadata can be retained at runtime, so that tools can discover them using reflection
- annotations can add compile-time information for code generation tools or other tools that plug into the Java compiler

Annotations can be used almost anywhere : classes, interfaces, fields, methods, constructors and even type declarations and type parameters

Defining Custom Annotations

Eg:
```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE) // applies to class, interface or enum
public @interface CommandInfo {
    Class<?> target() default Class.class;
    String setterPrefix() default "set";
}
```

Elements:

- annotations are allowed to have parameterised values, which are called elements
- type has to be a compile-time constant, enum, class literal or an array initializer
- Elements can also have default values. Eg:

```
Object input = ...;                    // pretend we know this is
@SuppressWarnings(value = "unchecked")
```

If an annotation only has one element, and the element is named "value", you can skip naming the "value":

Eg:
```
Object input = ...;
@SuppressWarnings("unchecked")
List<String> result = (List<String>) input;
```

Retention Policies:

SOURCE = annotations only exist in the source code

RUNTIME = annotations only exist in the .class bytecode file and is available at runtime to be used with reflection

CLASS = annotations only exists in the .class bytecode file but not exist while file is running.

Annotation target : It target types declare what part of it program can be given a particular annotation. Here are it possible target types :

ANNOTATION_TYPE : annotation type declaration

CONSTRUCTOR : constructor declaration

FIELD : field declaration, including enum constant

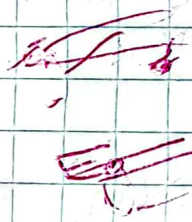
LOCAL_VARIABLE : local variable declaration

METHOD : method declaration

PACKAGE : package declaration

PARAMETER : method parameter declaration

TYPE : type declaration, such as class, interface, annotation type, or enum declaration,



Eg of Annotation :

```
@Retention ( RetentionPolicy. RUNTIME )
@Target ( ElementType. METHOD )

    Public @interface Test {
                        }
```

Reflection API

Reflection API : every class, interface and type (including primitive) has a corresponding Class object that accesses metadata about that type

- Class object are the main entry point into Java's Reflection API

Obtaining class objects

- .getClass() → on an object
- .class to create a class literal
- create classes dynamically using Class.forName()

Working with methods :

- using class API, you can also obtain Method object
- can be invoked by calling Method.invoke()

Eg : Method m = "Atticus".getClass().getMethod("toLowerCase");
System.out.println(m.invoke("FUN"));

=> Equivalent to :
System.out.println("FUN".toLowerCase());

Eg : Method m = String.class.getMethod("compareTo");
System.out.println(m.invoke("A", "B"));

=> Throws NoSuchMethodException.
The correct code :
Method m = String.class.getMethod("compareTo",
String.class);

Dynamic Proxy

- a dynamic proxy is a class that implements a list of interfaces specified at runtime
- don't need to know at compile-time what interfaces will be implemented

How do dynamic proxies work?

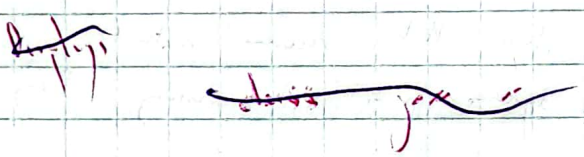
- first you create a custom InvocationHandler. This is an object that receives method invocations.
- create a dynamic proxy instance using Proxy.newProxyInstance()

Eg:

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

        Set<String> targetSet = new HashSet<>();

        Object proxy = Proxy.newProxyInstance(
            LoggingProxy.class.getClassLoader(),
            new Class[] { Set.class },
            new LoggingInvocationHandler(targetSet));

        Set<String> loggedSet = (Set<String>) proxy;
        loggedSet.add("item");

        System.out.println(targetSet.contains("item"));
    }
}

static class LoggingInvocationHandler implements InvocationHandler {

    private final Object targetObject;

    public LoggingInvocation(Object targetObject) {
        this.targetObject = targetObject;
    }
}
```

@Around

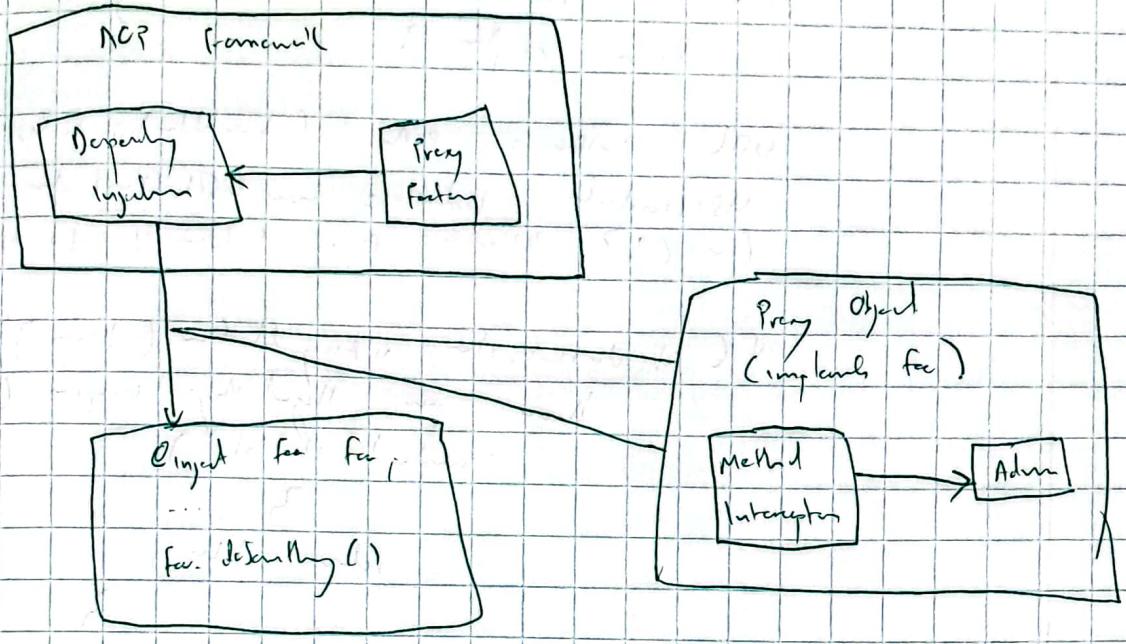
```
public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
    System.out.println(targetObject.getClass().getName() + "." +
        method.getName() + "()");
    return method.invoke(targetObject, args);
}
```



Aspect Oriented Programming

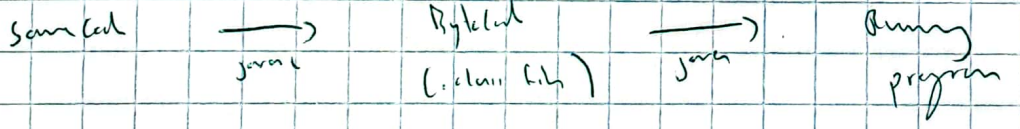
- AOP is a design pattern that organizes code into cross-cutting concerns
- A cross-cutting concern is any concern that affects many different parts of the system
- All code that handles a cross-cutting concern is organized into an aspect, which is substituted into advice
- Advice plugs into your existing code, via a method interceptor, to take care of the cross-cutting concern
- Join points are places where advice can plug into your code, usually via method interceptors
- Pointcuts are places where your code actually does use advice. An aspect is defined by advice and one or more pointcuts

How AOP with Dynamic proxies and Dependency injection



How Java Loads Classes

Java Program Lifecycle



- Java compiles to Byte code so that it can run on any computer with java installed.

Class Loaders:

- every class in Java Runtime is loaded by classLoader

Eg:

```
public static Class<?> getClass(String testFolder, String className)
        throws exception {

URL testDir = Path.of(testFolder).toUri().toURL();
URLClassLoader loader = new URLClassLoader(new URL[]{testDir});
Class<?> klass = Class.forName(className, true, loader);

if (!unitTest.class.isAssignableFrom(klass)) {
        throw new IllegalArgumentException("Class does not
        implement UnitTest");
    }
    return klass;
}
```

Intro to Concurrent Programming

Big picture and intuition

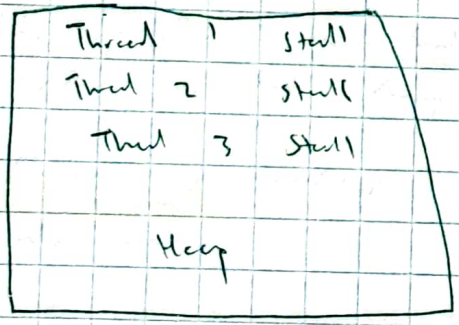
Sequential vs Concurrent vs Parallel

- Sequential program can only work on one task at a time
- Concurrent program can have multiple tasks in progress at the same time
- Parallel program can actually be working on multiple tasks at the same time.

Threads

- Threads are a way for program to execute multiple tasks in parallel at the same time

Threads and program memory :



- all threads share the same heap, this mean you can have two or more threads accessing shared state, such as an ArrayList or HashMap.

Eg of creating and running threads :

```
Thread thread = new Thread ( () -> System.out.println( "World! "),
    System.out.print ( "Hello, " );
    thread.start();
    thread.join();
```

This program print Hello, world! to terminal

- Virtual threads allow your program to create 10000 threads even if your computer CPU only support 4.

Example:

```
List<Thread> threads = new ArrayList<>(n);
for (int i = 0; i < n; i++) {
    threads.add(new Thread(new IntRunner(i)));
}

for (Thread thread : threads) {
    thread.start();
}

for (Thread thread : threads) {
    thread.join();
}

}

public static final class IntRunner implements Runnable {
    public final int value;

    IntRunner(int value) {
        this.value = value;
    }

    @Override
    public void run() {
        System.out.println("Thread id" + Thread.currentThread().
            getId() + " printed " + value);
    }
```

Edge Case : Thread Execution Order

- A race condition is a flaw in software by that happens when the execution order of parallel threads depends on a particular

Limits of parallelism

Your program can only benefit from parallelism to the extent that it calls on parallelism

Amdahl's law :

$$S = \frac{1}{1-p}$$

- p is the fraction of the program that can be parallelized
- S is how much the program could, in theory, speed up from parallelism

Limits of parallelism :

- each thread needs to have memory allocated for it call stall

Thread Pool and Executor

- A collection of threads that is used to efficiently execute and manage asynchronous work.
- reduce the cost of using threads by storing them in a worker thread pool, new work is added to a work queue, when it waits for a pooled thread to become available

Example of creating thread pools :

ExecutorService pool = Executors.newSingleThreadExecutor();

- A thread pool with only one thread

ExecutorService pool = Executors.newCachedThreadPool();

- A thread pool that reuses threads, but does not limit thread creation

ExecutorService pool: Executors.newFixedThreadPool(12);
- Thread pool that reuses fixed set of threads
of thread to 12.

Submitting Asynchronous Work:

- Thread pool has several methods that let you submit work to be executed asynchronously:

Eg:
- Submit a Runnable that no return value, and return a Future
Future<?> pool: pool.submit(() -> System.out.println("foo"));
- Submit a Runnable and return result:
pool.execute(() -> System.out.println("foo"));
- Submit a Callable, whose return value will be accessed via Future:
Future<String> futureFoo = pool.submit(() -> downloadFile());

Future:
- A Future is a reference to result of an asynchronous computation
- If asynchronous computation is done: get() will return result of computation
- If not done: get() will cause program to stop and wait for computation to finish.
- Future are parameterized. Called get() on Future<Msg> will return a Msg

Joining Asynchronous Work:
- Future.get() -> thread pauses & waiting for asynchronous work is called joining.

Example of a solution to not having a thread or future to explicitly join:

```
CountDownLatch latch = new CountDownLatch(1);
pool.execute(() -> {
    System.out.println("foo");
    latch.countDown();
});
latch.await();
```

ForkJoinPool

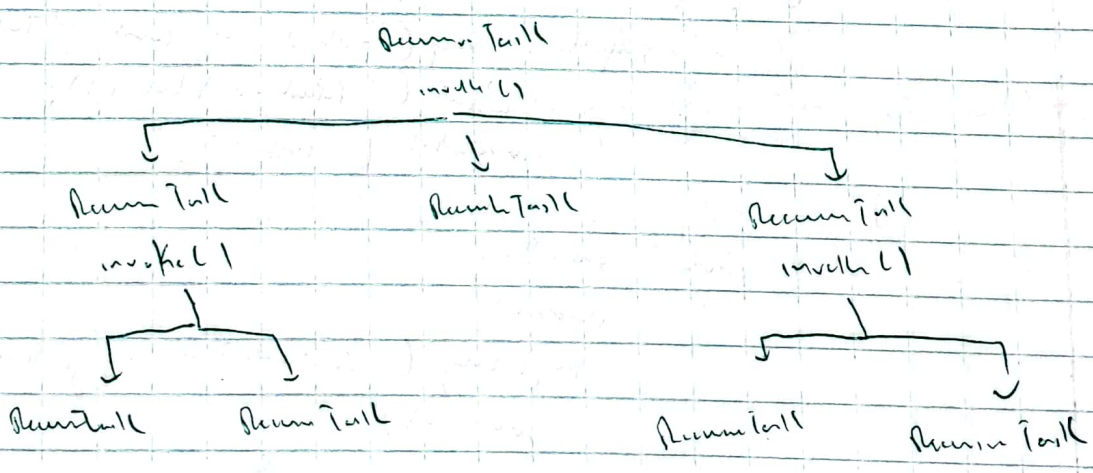
It is a specialized thread pool that has 2 key advantages:

- uses a technique called work stealing so that idle worker threads can find work to do
- its API is optimized for asynchronous work that creates more work. Called recursive work.

ForkJoinTask

- when you create work to submit to a ForkJoinPool, you usually do so by subclassing either RecursiveTask or RecursiveAction
- Use RecursiveTask for asynchronous work that returns a value and RecursiveAction for others

How to submit work:



Example of using ForkJoinPool:

```
public final class CountWordsTask extends RecursiveTask<Long> {
    private final Path path;
    private final String word;

    public CountWordsTask(Path path, String word) {
        this.path = path;
        this.word = word;
    }

    @Override
    protected Long compute() {
        if (!Files.isDirectory(path)) {
            return WordCounter.countWordsInFile(path, word);
        }

        Stream<Path> subpaths;

        try {
            subpaths = Files.list(path);
        } catch (IOException e) {
            return 0L;
        }

        List<CountWordsTask> subtasks =
            subpaths.map(path -> new CountWordsTask(path, word))
                    .collect(Collectors.toList());
        invokeAll(subtasks);

        return subtasks.
            stream().
            mapToLong(CountWordsTask::getRawResult).
            sum();
    }
```

Parallel Streams

Parallel Stream are a way to execute stream pipelines in parallel on multiple threads

Example : Non-parallel stream :

```
Map<Year, Long> graduatingClassSizes = studentList
    .stream()
    .collect(Collectors.groupingBy(
    Student::getGraduationYear, Collectors.counting());
```

Parallel version :

```
Map<Year, Long> graduatingClassSizes = studentList
    .parallelStream()
    .collect(Collectors.groupingByConcurrent(
    Student::getGraduationYear, Collectors.counting());
```

Using Parallel Stream with Thread Pool :

By default, parallel streams run threads in default ForkJoinPool.commonPool()
You can circumvent that default by wrapping the stream computation in a Callable and submitting it to a ForkJoinPool explicitly :

```
ForkJoinPool pool = new ForkJoinPool();
Future<Map<Year, Long>> graduatingClassSizes = pool.submit(() ->
    studentList.parallelStream()
    .collect(Collectors.groupingByConcurrent(
        Student::getGraduationYear, Collectors.counting());
```

Synchronization

Synchronization is the process of limiting the number of threads that can access a shared resource at the same time.

When is synchronization needed?
- when you have multiple threads accessing the same shared resource

Here are two examples of using synchronization:

- Synchronized collection wrappers:

```
Map<String, Integer> votes = Collections.synchronizedMap(new HashMap());
```

- Data structures and tools in the java.util.concurrent package that are specifically designed for concurrent access.

```
Map<String, Integer> votes = new ConcurrentHashMap<>();
```

What are atomic operations?
- is an operation that is executed as a single step and cannot be split into smaller steps.
- ConcurrentHashMap.compute() is atomic, while as, ConcurrentHashMap.get() and ConcurrentHashMap.put() is not atomic.

Immutable Object

An immutable object is an object whose value cannot change after it is created.

Mutable:
- unsafe to use as hash keys
- if used from multiple threads, must be explicitly synchronized
- harder to reason about in code

Immutable:
- can be safely used as keys in hash-based data structures
- inherently thread-safe
- easier to reason about in code

The synchronized keyword

How to use the synchronized keyword:
- the best way to synchronize is to use high-level built in tools like the synchronized collections wrapper, or data structures in the java.util.concurrent package
- when these options aren't available, you can use the synchronized keyword for low-level synchronization control.

Example:

```
public final class ValueType {
    private final My<String, Integer> value = new HashMap<>();

    public void setVal(String param) {
        synchronized (this) {
            Integer count = value.get(param);
            if (count == null) { value.put(param, 1);
            } else {
                value.put(param, count + 1);
                }
            }
        }
```

- The thing in parenthesis after the synchronized keyword is the lock object.
- When a thread enters the code block, it takes ownership of the lock.
- Only one thread at a time can own the lock at a time, so only one thread is allowed to be executing code inside the synchronized block at a given time.

Lock Objects:
- any object can be used a the lock
- or you could create a completely new object just to serve the purpose of the lock:
  Eg: private final Object lock = "SpecialLock";
- If you decide to use the this keyword, the lock object is the current instance of the class.

Advanced Synchronization

When to use the ReentrantLock class:
- the synchronized keyword only works with block of code
- sometimes, you want to take ownership of the lock in one method, and release ownership in another method.

Eg:
```
public final class VotingApp {
    private final Map<String, Integer> votes = new HashMap<>();
    private final Lock lock = new ReentrantLock();

    public void castVote(String performer) {
        lock.lock();
        try {
            votes.compute(
                performer, (k,v) -> (v == null) ? 1 : v+1);
        } finally {
            lock.unlock();
        }
    }
}
```

Semaphores:

- The synchronized keyword only allows one thread at a time with a guarded block of code. What do you do if you need to allow more than one thread?
  - The semaphore class can help with this scenario

Eg: To create a semaphore you tell it how many permits it can give out:

```
Semaphor semephor = new Semephor (10);
```

Eg: How two threads obtain and release permits from the semaphore:

```
try {
    semephor.acquire();
    // up to 10 threads can be executing here in parallel
} finally {
    semephor.release();
}
```