* **Threads**
  + Process - Program, can have many threads
  + Thread - Path of execution within a process
  + Every thread has its own call stacks, but shares memory with other threads
  + When you start a Java program, three threads are created
    - Main
    - Garbage Collector
    - Thread Scheduler
      * Daemon Thread - We don’t interact with or control these. They’re low-priority background processes
  + How to create a thread
    - Extend Thread

|  |
| --- |
| public class MyThread extends Thread { public void run() { //Does something } } |

* + - * Start by calling .start() which is inherited from Thread
      * A Thread is already a Runnable
    - Implement Runnable interface

|  |
| --- |
| public class MyRunnable implements Runnable {  public void run() {  //Does something  } } |

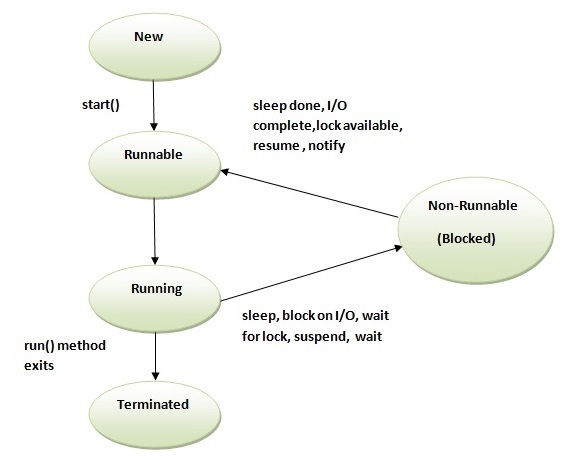
* Start this thread by
  + Creating a Thread
  + Pass the Runnable to it
  + Call start()
* Think of Runnable as a job

|  |
| --- |
| Thread t = new Thread(new MyRunnable()); t.start(); |

* Could subclass something other than Thread using Runnable, since each class can only extend one other class. Also separating the class from the executing Thread running the class.
* On the other hand, anything that extends Thread is already considered a Thread, so it is also convenient to use at times.
  + Threads can be garbage collected?
    - Yes, and as long as it’s active, it won’t get garbage collected
  + Synchronized - keyword that ensures that only one thread can access a resource at a given time. (Methods, variables (objects, too), code blocks)
    - By synchronizing, you gain consistency, but you lose speed
    - To allow multiple threads to read (but not write) and synchronize writing, create a normal reading method and a synchronized writing method.
    - When a Collection is thread-safe (Synchronized), its members are protected from being accessed by more than one thread at a time.
  + Thread-safe - one thread at a time can access
    - If there’s an ArrayList of synchronized elements
  + Thread Methods
    - Document with list of interview questions on Thread methods will be sent out
    - You can have multiple threads with the same priority

|  |
| --- |
| start() //Calls run() getPriority() //Values from 1 (lowest) through 10 (highest) setPriority() //Values from 1 (lowest) through 10 (highest) isAlive() //Checks whether the thread is running wait() //Specify the amount of time to wait. It can be notified. sleep() //Will pause for the specified amount of time. Can't be  //notified notify() //Wake up sleeping threads notifyAll() //notify all sleeping threads join() //Wait for another thread to stop executing. Catches up //with another thread |

* + Thread States
    - NEW
      * Has not started yet
    - RUNNABLE
      * Currently executing
    - BLOCKED
      * Waiting for a lock to be released
    - WAITING
      * Thread is waiting indefinitely for another thread to perform a particular action
      * wait() with no timeout, join() with no timeout
    - TIMED\_WAITING
      * Thread is waiting for another thread to perform a particular action for a specified waiting time
      * sleep(), wait() with timeout, join() with timeout
    - TERMINATED
      * Thread completed execution, terminated



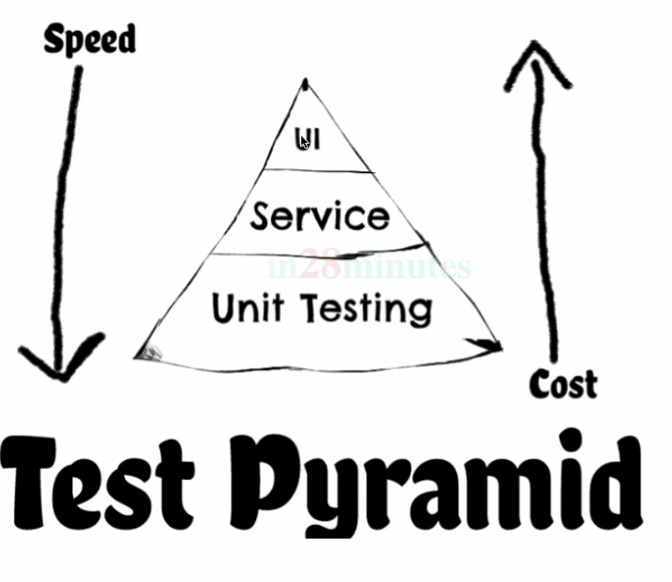
* + Deadlock
    - When multiple threads are blocking each other, trying to access the same resource.
  + Producer/Consumer Problem
    - Two threads interacting with the same resource. How do you synchronize these threads?
    - Producer: adding data
    - Consumer: removing data
* **Design Patterns**
  + Singleton
    - Ensure a class has only one instance and provide a global point of access to it. Makes use of the lazy loading design pattern, which creates an object only when it is needed, which improves performance since creating objects takes time (see Wikipedia for more info on lazy loading).

|  |
| --- |
| public class Singleton {  private static Singleton instance;  private Singleton()  {  ...  }  public static synchronized Singleton getInstance()  {  if (instance == null)  instance = new Singleton();  return instance;  }  ...  public void doSomething()  {  ...  } } |

* + - Important points
      * Implementation
        + A private constructor.
        + A Static private property of the self class.
        + A Static method to get the unique instance, where it creates the instance if it has not been created.
    - Real World example: https://dzone.com/articles/design-patterns-singleton
    - Real World uses
      * Singletons are used a lot where you need to provide a registry, or something like a thread pool, multithreading. Logging is also another popular use of Singletons, providing one single access point to an applications log file.
    - Useful Sites
      * Examples of Singleton use: https://www.javaworld.com/article/2073352/core-java/simply-singleton.html
  + Factory
    - Create an object without exposing the creation logic to the client.
    - Use a Factory method to return the appropriate object type from a list of classes that implement a common interface (for example, by passing in a String with that describes what object type you want as a parameter)
* **Maven (Apache)**
  + Java Build Tool (Like: Ant, Cradle)
  + We can use it to manage our dependencies (i.e. .jar (Java Archive) files). How are we leveraging it initially?
  + We’ll use it to build projects
  + .war (War Archive) files, which we’ll use when we start on servlets at the end of week 3
  + **Default build lifecycle - Very Cool Tennis Players Vape InDoors**
    - **Validate** - checking if the project is correct, and everything needed to compile is there
    - **Compile**
    - **Test** - will run testing framework for you if you ask it to (and if you have a testing framework set up.
    - **Package** - taking the compiled code and putting it into the desired format (like a .jar or .war file)
    - **Verify** - checking any additional metrics you set up are satisfied
    - **Install** - putting it in a local repository
    - **Deploy** - deploying your code somewhere. Putting a .war file into a web container
    - We’ll mostly be using compile, test, package and deploy
  + POM -> pom.xml
    - Project Object Model
  + Maven Repository
    - Where all of your dependencies are living
    - Two versions of this
      * Remote Maven Repository - dependencies anywhere. When you go to use a new one, if it’s known, Maven will download the dependencies into the local version
      * Local Maven repository. You can find this in your ~/.m2 folder.
    - Jarmageddon - properly include all of your .jar files and dependencies or bad things will happen. On Apache maintenance documentation as something Maven prevents by using their repository.
    - How do we use it for logging and unit testing?
    - Group Id: namespace so “com.revature”
    - Artifact Id: “MavenDemo”
    - If your Maven project is breaking, there’s likely a problem with your pom.xml file.
    - To update pom file, save then right click the Maven project -> Maven -> Update Project... (or just hit Alt + F5)
    - To update to Java 1.8, open the pom.xml file and include the following:

|  |
| --- |
| <properties>  <maven.compiler.source>1.8</maven.compiler.source>  <maven.compiler.target>1.8</maven.compiler.target>  </properties> |

* **Types of Testing**
  + Automated Testing
    - UI
    - Service
    - Unit Testing
  + UI and Service
    - * Depend on the entire app and external dependencies being available
      * Creating all possible tests scenarios is difficult/ near impossible
        + Depends on databases, external stuff
      * Tough to debug in case of failure
    - Unit Testing
      * Easy to debug
      * Runs very fast
      * Coverage is easy
        + Scenarios limited



* What is Unit Testing?
  + Smallest scale testing.
  + Usually methods are the unit i.e. 1 method gets 1 test
    - Can be group of methods if it fits the test case
  + Done by the programmer, not a team of testers, because it requires detailed knowledge of the source code itself.
  + Two types of coverage:
    - Line Coverage - make sure every line executes at least once
    - Branch Coverage - make sure every possible branch (true/false, switch case, etc) executes at least once. Superset of line coverage.
* **JUnit**
  + Java Unit Testing Framework
    - Write tests in src/test in a parallel structure to your existing code.
    - Absence of failure is success!
    - Many annotations used (heavily annotation-based)
      * @Test - method which forms a JUnit test
      * @BeforeAll - STATIC method that runs once before all test methods
      * @AfterAll - STATIC method that runs once after all test methods
      * @After - runs after a specific test
      * @BeforeEach- used on method to have it run before each test method
      * @AfterEach- used on method to have it run after each test method
        + Delete mock data from database
      * Use assert methods to determine success or failure (very large suite of these. Provided by org.junit.Assert).
        + AssertTrue, AssertFalse, AssertNull, AssertNotNull, AssertEquals
      * Junit 5 -<https://junit.org/junit5/>
        + @BeforeClass

Replaced @Before from JUnit4

defining something that runs before all the methods in a given test suite.

* + - * + @AfterClass

Replaced @Before from JUnit4

defining something that runs after all the methods in a given test suite.

* + - * + @RepeatedTest(some number)

Add to Junit5

Allows to repeat test x number of times

Allows to check for methods using random values, multiple threads, or asynchronus calls, etc

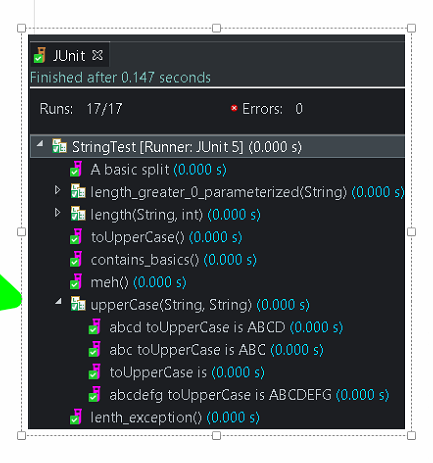
* + - * + TestInfo is a class in Junit that has methods we can use!

@AfterEach

void afterEach(TestInfo info) {

System.out.println("Clean up Test Data for " + info.getDisplayName());

}

* @ParameterizedTest
  + Allows you to pass in parameters instead of rewriting method for each test
  + Can add name to make each test more readable
    - @ParameterizedTest(name = "{0} toUpperCase is {1}")
    - This will show up in the Junit output
  + @ValueSource(strings = {"ABCD", "ABC", "DE"})
    - Pass in values using an annotation
    - Works if only need 1 parameter
  + @CsvSource(value= {"abcd, ABCD", "abc , ABC", " ``, ``", "abcdefg , ABCDEFG"} )
    - Pass in multiple parameters
    - Can be different variable types

@ParameterizedTest

@CsvSource(value= {"abcd, 4", "abc , 3", "'', 0", "abcdefg , 7"} )

void length(String word, int expectedLength) {

assertEquals(expectedLength, word.length());

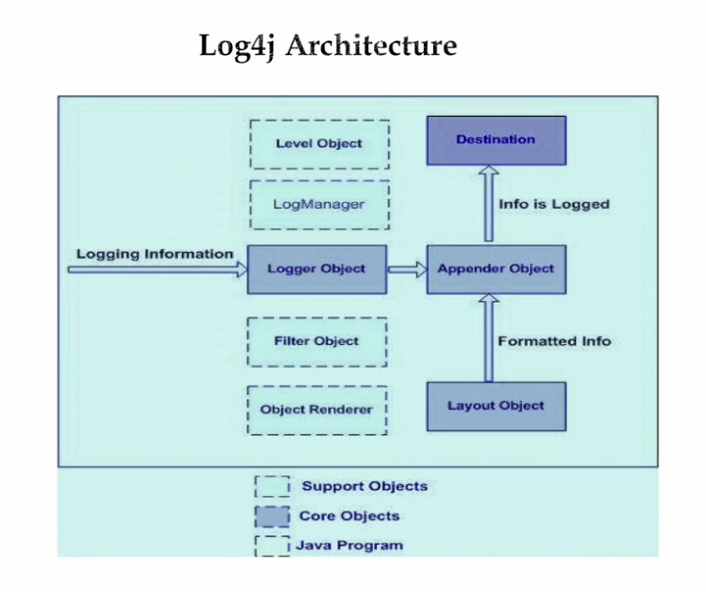
}

* Grouping/Nesting Tests
  + @Nested on a class

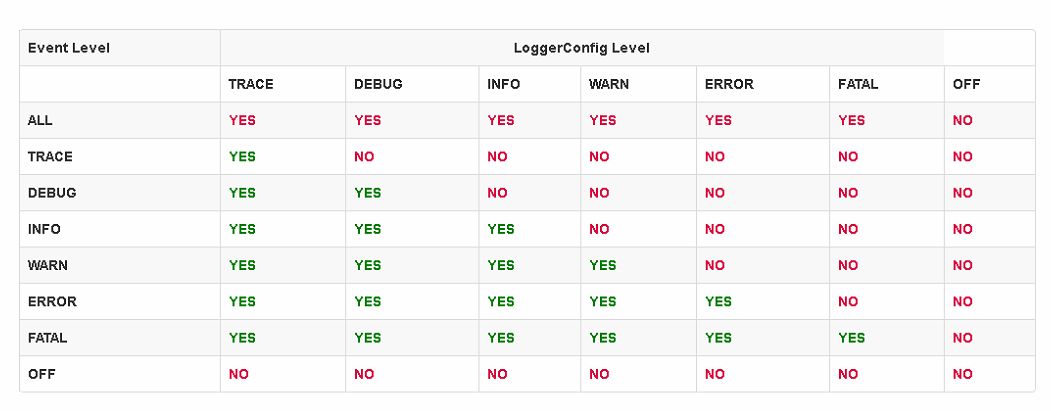
* Performance Testing
  + Can use assertTimeout(duration.ofSeconds(5)…
    - How long should/does it take a method to run
* Disable a unit test or test class
  + @Disabled
    - New to Junit5. @Ignored for Junit4

* **Log4J**
  + It’s a logging tool that helps us keep track of what’s going on inside of code
    - Highly Configurable and flexible
    - Can send info to 1+ various output
      * Database
      * File
      * Console
      * UniSys Log

* Better than using sysout all the time
  + Sysout is a synchrozined method that is very costly in terms of processing
  + Sysout is not a debugging tool!
* One of the popular logging frameworks. Others include Logback, etc.
* A good logging framework should let you decide a couple of things:
  + Levels - what to log (going from worst to least bad)
    - Fatal
    - Error
    - Warn
    - Info
    - Debug
    - Trace
  + Formatting/Layout - how to log
    - Define layout
    - Date/time style
    - Can output in HTML format
  + Appenders - where to log
    - Log files - in a file for reference later
    - Console - lots of console output
* We keep our properties (logging configuration) in a file called log4j.properties or an XML file called log4j2.xml
* Log4J Features
  + - Thread safe
    - High speed optimization
    - Multiple appenders support
    - Customizable
      * .xml/.properites file config
    - Various logging levels
    - Layout class
      * Customized formatting of the log messages
    - Advantages
      * Quick debuggin
      * Easy maintenance
      * Structured storage
    - Disadvantages
      * Slows down app
      * Scrolling blindness due to config file being to verbose



* Log4J Components
  + Core Objects
    - Loggers
      * Register the class to the logger
      * Gathers the logging info and are captured in the namespace heiracrchy
    - Appenders
      * Publishes the logging info to the respective destination(s)
      * Lower level object
    - Layouts
      * Styling the logging information for readability
      * i.e timestamps etc
  + Support Core Objects
    - Level
      * Defines granularity and priority of logging info
    - Filter
      * Analyze and decide whether to keep the logged info
      * Appenders can use multiple filters
      * Tell the appender whether or not to print the logs to the respective destination
    - Object Renderer
      * Provides the string representation of the logging information
    - Log Manager
      * Reads the config parameters



* Configure Log4J
  + log4j.properties or log4j2.xml

