Appendix A: Selected Terms Introduced in this Book

* **Manual Thread Dump Profiling / MTDP**: MTPD is discussed in chapter 11. This is the ‘t’ chapter in the **P.A.t.h. Checklist**. MTDP is the process of using a small handful **thread dumps** (captured using the JDK’s jstack) to determine which processes in a headless application under load are the slowest. The more frequently a process (aka **stack trace**) shows up in the thread dump, the more likely it is a candidate for optimization.
* **Code First**: Found in the introduction, this tuning approach prioritizes fixing performance defects in the code, which is the main responsibility of the development team, and the main deliverable which is migrated into production. As shown in chapter 2 in the **Modest-size Tuning Environment** and again in chapter 5 on **Invalid Load Tests**, tests can easily be designed to de-prioritize performance of the features below.   
  + the network (firewalls, content switches, routers, etc...)
  + load balancer configurations
  + some container configuration
  + multi-node clustered configurations
  + infrequently used business processes  
      
    Once the code performs well-enough in the **Modest-size Tuning Environment**, performance testing the list below can be asserted in an integration environment.  
    Code First is important because much time is wasted trying to tune things that lie outside of the development team’s main responsibility – the code.
* **Three Threads of Load and Zero Think Time / 3t0tt**: Pronounced “three tott”, this simple load plan is an approach to load testing, where all components written by the team that are bigger than “a certain size” should be examined under a small amount of load before being checked into source code control. The amount of load is small enough that it will not overwhelm a standard development workstation, but large enough to reproduce multi-threaded performance defects.   
    
  The **MTDP** technique should be used to find/fix any obvious performance issues. Lesser experienced Java develops not ready for **MTDP** should just run the test and capture the thread dumps which can later be reviewed during a code review.  
    
  What is the “certain size” mentioned above? Components roughly as big or bigger than a SOA service should be testing using 3t0tt. With this size exception, I am trying to avoid the complexities of microbenchmarking. Towards this end, 3t0tt should only include components that have at least one I/O – like one or more database hits, or one or more other network requests.
* **Write Once Run Anywhere Performance Defects**: WORA stands for “Write Once Run Anywhere,” and it was coined ages ago to brag that the exact same Java code can run on any OS platform with a JDK. This was/is important to avoid all the time spent/wasted on platform specific code of the C and C++ world. I have coopted the WORA term to also describe a great number of the performance defects seen in the enterprise over my career. Described in chapter 8, the code examples distributed with this book demonstrate that these performance defects can be easily demonstrated and detected on any platform.  
  Why is this important? Because performance defects can be found and fixed in just about any environment. We mustn’t wait around for a large computing environment to locate performance defects. That wait stymies performance progress and leads to real problems. We should get started right now load testing in a **Modest-sized Tuning Environment**.
* **Main Performance Anti-Patterns**: When assessing a performance problem try to see the that problem falls into any of these common anti-patterns. This is important because it can help you understand what’s gone wrong when looking at unfamiliar code.
  1. **Unnecessary Initialization**: A great deal of slowdowns occur in initialization code, before any of the big processing really happens. Be on the lookout for any small processing challenges that slow down processing before or after the main processing event of a request.
  2. **Inefficient Algorithm/Strategy**: A misconfigured or poorly chosen algorithm or coding strategy is causing performance problems. A strategy is a technique used throughout a code base and an algorithm is a plan used to implement a single component.
  3. **Overprocessing**: The system is doing unnecessary work. Removing that work provides measurable performance benefit. One example is retrieving too much data, where most of the data is discarded. Another example is including in your load test the wrong resource-hungry use case – one that is seldom used in production.
  4. **Large Processing Challenge**: Attempting to process and conquer a massive amount of data. Very few applications legitimately have such a need, but these do exist. Querying 4 billion rows of data. Transferring 10mb of data repeatedly over a slow network, etc…
* **Modest-sized Tuning Environment**: You can get started now finding performance defects be creating a “Modest-sized Tuning Environment.” Generally, most of the performance defects in production can be reproduced in smaller environments, and the code examples (chapter 8) demonstrate that. As detailed in chapter 2, you can get started by “stubbing out” your backend systems and run a load test right on your desktop.
* **Plug-it-in-now**: a type of performance metric that is very easy to capture, because when good metrics aren’t readily available, performance problems often go unfixed. plug-it-in-now monitoring/visibility tools provide metrics on any running JVM without requiring extra JVM configuration and without requiring a JVM restart. Even the act of having to post-process a data file (perhaps to render a graph) makes a metric less of a plug-it-in-now metric. JVM Verbose GC data are the quintessential example of metrics that are not plug-it-in-now, because of these time-wasting obstacles:
  + a dizzying array of JVM -D parameters are required to collect the data
  + a JVM restart is required to apply the -D parameters
  + most knowledge gleaned from this data requires post-processing of a data file into a graph.
* **Dark Environments**: Chapter 2 discusses dark environments, a loose term used to describe a computing environment that has very little performance monitoring. In fact, dark environments have so little monitoring that performance defects escape detection because the lack of facilities to pinpoint the root cause of the slowdown or other performance issue. Actually, dark environments don’t really exist – they are just a misperception, because all environments come with the JDK’s **plug-it-in-now** tools like jstack, jcmd and jstat. That said, the lack of Java database response time and throughput metrics are troublesome gap.
* **P.A.t.h. Checklist**: Chapters 8-12 detail the tools and techniques used the review this checklist that provides an easy-to-remember list of the main things to check for performance problems with Java server-side software. The four elements are
  + **Persistence**: Don’t forget that a chunky data access strategy outperformance a chattey one.
  + **Alien Systems**: When your JVM makes a network call over a “slows wire”, make sure the request and response data is compressed
  + **threads**: **MTDP** is a great way to detect performance problems in “**Dark Environments**”
  + **heap**: Don’t forget to use the plug-it-in-now “red-yellow-green” assessment scale to figure out whether GC changes are needed.
* **Entrypoint:** see **entrypoint-trigger-current**
* **Trigger:** see **entrypoint-trigger-current**
* **Current:** see **entrypoint-trigger-current**
* **Entrypoint-trigger-current**: These three terms coined in chapter 11 highlight the important landmarks in a stack trace used for performance troubleshooting, especially in **MTDP.**
  + The entrypoint landmark is the class/method how the thread was launched, which is an important part in determining whether a given stacktrace in a thread dump is under load or not.
  + The trigger landmark is where code in your package space calls into a 3rd party API, like a JDBC driver or other java.lang API. The trigger often highlights the point where your code made a decision on how to invoke the 3rd party API, which may or may not be an idiom that performs well.
  + The current landmark point to the Java class/method that was executing when the stack trace was captured.
* **Invalid Load Test**: -- a load test whose results should be discarded because the test does not closely enough approximate production conditions. This is the main topic of Chapter 5, which details a number of reasons why tests can be invalid.
* **First Priority Load Script**: The “First Priority” script helps you get up and going quickly to fix some of the most obvious architecture-wide performance defects quickly and to curtail performance angst. Create a basic script to test a few business processes. Be sure to check out chapter 4 for a list of “must have” load script enhancements, the ones required to convince you and your team of the validity of your load test.
* **Second Priority Load Script**: Once you have shaken out some basic performance defects using a “**First Priority**” load script, it is time to add a few more business processes and make sure the proportions of the requests roughly model what’s executed in production. Check out chapter 4 for a list of other things you’ll need to enhance the load script.
* **Scalability Yardstick**: Introduced in chapter 6, this is an easy-to-run single-node load test used to assess application scalability. This is important because this test can fill significant gap: so few systems achieve scalability these days because scalability tests are run so infrequently. The Scalability Yardstick is an incremental load test with four equal steps of load, the first of which pushes the application server CPU to about 25%. If four squared, chiseled steps of throughput are generated, then the application is likely to scale in a larger environment. See chapter 6 for examples of tests that do and do not scale.