jKool™ TNT4J API

jKool Track and Trace 4 Java (TNT4J) API. Version 0.7a
Application Diagnostic Library for Java (Confidential)
© 2014 Copyright Nastel Technologies, Inc.

Critical Elements of Application Debugging and Diagnostics

- Application Behavior
  - Logs, traces, debug messages
    - Timing, exceptions, messages
  - Relationship between various messages
    - What log entries are related to a specific request?
- Application Runtime State
  - State of variables during application execution
  - Metrics, variables, collections specific to the application
- JVM Runtime State
  - Memory, properties, GC activity
  - Stack traces, properties, environment, etc.
  - Object state, fields, memory footprint
- Must be quick, easy and automated
  - Analyzing trace logs is manual and time consuming

STANDARD LOGGING: EASY TO USE HARD TO ANALYZE

Standard Logging is Painful

- You spend a lot of time manually analyzing traces and log files
  - Often log messages contain little useful information. Trace messages are free form text.
  - Do you need to look at the code to understand what the log message actually means?
  - Is your framework's conditional logging too broad and generates too much output?
    - Do you generate too much log output making it hard or impossible to analyze?
    - Server applications running many concurrent threads, processing many requests. What goes with what?
- You cant match log entries even within the same log file
  - Matching entries emitted by concurrently running threads is painful
  - You cant make sense of the log produced by the application that you developed? (sucks)
- You cant match log entries across multiple logs and applications
  - Composite server side applications are especially tough to trace. Too many moving parts.
  - Can you match upstream with downstream activities across multiple apps?
  - Are you chasing problems across multiple log files?
- You need performance, timing, correlation recorded automatically
  - You need to correlate requests, responses and downstream activities
  - You need trace only those activities that match a specific criteria and across multiple applications. Log only request that match location XYZ*.
- Are you subtracting timestamps to figure out how long it takes to execute an activity? What about 1000s of concurrent activities?
  - Are you building timing code directly into your applications?
- You need to know application & JVM state during diagnostics
  - Log messages do not record the state of internal data structures, stack traces, variables.

What is TNT4J?

- **TNT4J Mission**
  - Dramatically **reduce time it takes to troubleshoot & debug** application behavior using **logging paradigm** by enriching event streams with **tags, correlators, performance** metrics and **application state** to improve speed of manual analysis and enable automated log analysis
  - Simple to understand and use
  - Improve quality and readability of logs to accelerate diagnostics
  - Enrich log entries for automated log analysis
  - Decrease or eliminate development of custom diagnostic code
  - Standard way to log, dump, correlate and analyze traces
- **Application Track & Trace library for Java (TNT4J)**
  - Logging, Tracing, Tracking
  - Application state: dump generation, object state, formatting
  - Conditional tracking based on application tokens, patterns
- **Uses log4j** as a default logging framework
  - Any other logging framework can be used
- **Plugin Architecture**
  - Custom loggers, trackers, event sinks, formatters
  - Custom dump handlers

Why simple logging is not enough?

- **Simple logging is not enough (like log4j). Why?**
  - How to identify, time, trace related activities within and across apps?
  - How to easily identify related log entries in the same log or across logs without knowing much about log format?
  - Can you time application activities easily without manual time stamp comparisons?
  - How to control logging behavior based on application tokens, data and across applications? Can you trace requests that match a certain pattern only?
  - How to extract and log application state during diagnostic phase? Application, JVM state dump is essential during troubleshooting. **Debug log entries are not enough**
  - How to analyze event logs programmatically without parsing message text format?
  - What is performance when messages are logged? What was CPU usage when an error occurred? What was memory usage? Etc.

Why jKool TNT4J?

- Complete framework for debugging and diagnostics
  - **Simple programming model** to facilitate fast root cause, log analysis
  - Application **state dump** framework
  - **Conditional tracking behavior** based on application defined tokens
  - Automated timing of application activities
  - **Inter-log correlation** of log entries (correlators and tags) between multiple related application
  - **Intra-log correlation** of related activities and sub-activities between multiple applications and threads
  - Event location tags such as GPS, server.
  - **Event flow direction** for composite applications that exchange messages (e.g. SOAP, JMS, and SQL etc.)
  - Automated CPU, memory logging, thread statistics per process/thread. User defined metrics.

jKool API for Java (TNT4J)

By Example

TNT4J Activity Synopsis

Tracking Activity

| App Context | Signature | Timing | Properties |

Tracking Event

| Signature | Correlators | Timing | Location | Exception |

| App Context | Signature | Timing | Properties |

| Signature | Correlators | Timing | Location | Exception |

App Context			
Name	Container	OS Info

Signature	
Parent GUID	GUID

Correlators	
Correlator	Tag

Properties		
Name	Value	Timestamp
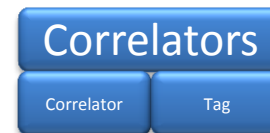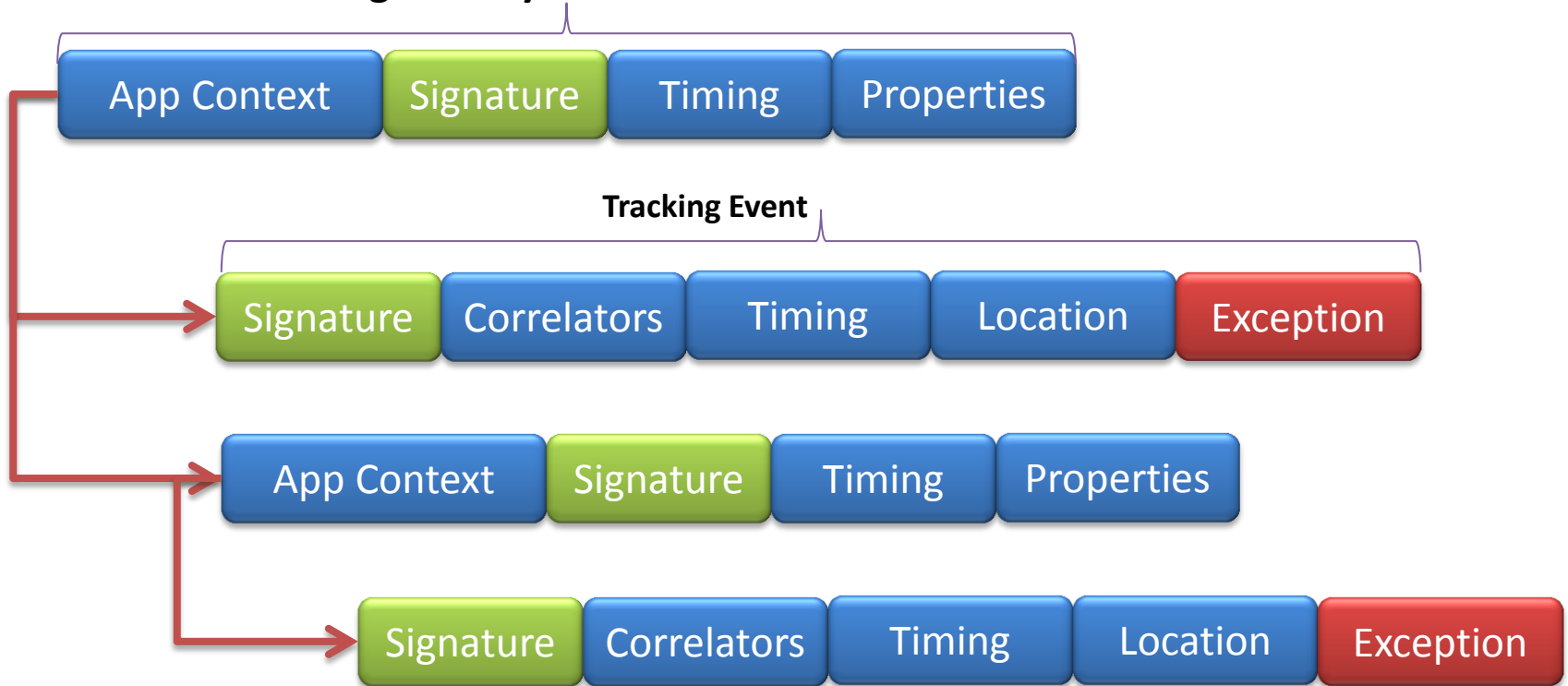
TNT4J Activity Synopsis (2)
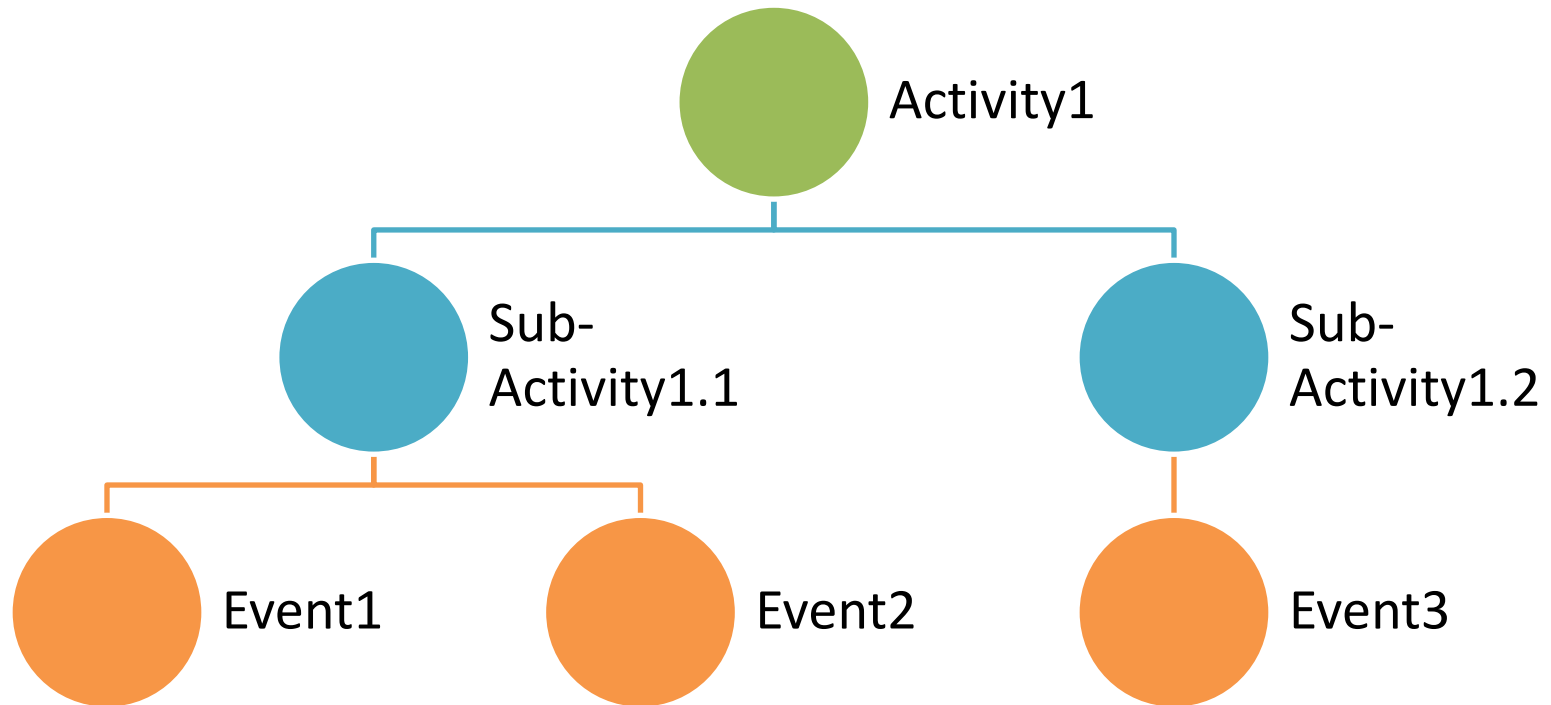
TNT4J Basic App: Error Reporting

```java
public static void reportError(String[] args) {
    if (args.length < 6) {
        System.out.println("Usage: appl server msg correlator opname location");
        System.exit(-1);
    }

    // register with jKool TNT4J framework
    TrackingLogger.register(args[0], args[1]);

    TrackingSelector selector = TrackingLogger.getTracker().getTrackingSelector();
    for (int i=0; i < 10; i++) {
        try {
            if (selector.isSet(OpLevel.DEBUG, "myappl.correlator", args[4])) {
                TrackingLogger.tnt(OpLevel.DEBUG, args[3], "Processing request", "runSampleActivity");
            }
            runSampleActivity(args[2], args[3], args[4], args[5]);
        } catch (Exception ex) {
            TrackingLogger.tnt(OpLevel.ERROR, args[3], "Failed to process request", "runSampleActivity", ex);
        }
    }
    System.exit(0);
}
```

Register current thread with the jKool framework. Once for each thread.

Report and exception as ERROR

Pass args[3] as correlator

Error messages

Operation name associated with the error

Exception

TNT4J Conditional Tracking

```java
public static void reportError(String[] args) {
    if (args.length < 6) {
        System.out.println("Usage: appl server msg correlator opname location");
        System.exit(-1);
    }
    // register with jKool TNT4J framework
    TrackingLogger.register(args[0], args[1]);

    TrackingSelector selector = TrackingLogger.getTracker().getTrackingSelector();
    for (int i=0; i < 10; i++) {
        try {
            if (selector.isSet(OpLevel.DEBUG, "myappl.correlator", args[4])) {
                TrackingLogger.tnt(OpLevel.DEBUG, args[3], "Processing request", "runSampleActivity");
            }
            runSampleActivity(args[2], args[3], args[4], args[5]);
        } catch (Exception ex) {
            TrackingLogger.tnt(OpLevel.ERROR, args[3], "Failed to process request", "runSampleActivity", ex);
        }
    }
    System.exit(0);
}
```

Report a debug message only if user defined correlator matches for DEBUG severity

Tracking and Tracing Application Activities, State Dumps

- **Application State Dump**
  - Record of internal data structures, variables
  - Dumps generated on demand, JVM shutdown
  - Very instrumental during problem diagnostics
  - Application specific **dump providers, dump listeners**
    - **Thread, properties, memory, application state dumps, etc.**
    - **Get notified on pre, post dump generation**
- **Application Activity** (TrackingActivity)
  - A collection of related events (TrackingEvent)
  - Tracking event is discreet event, message, action such as JDBC, JMS, SOAP call or anything that application deems appropriate
  - Independent timing for activity and each tracking event
  - **Examples**:
    - Processing of an a request can be an activity consisting of several tracking events:
      - HTTP-ACTIVITY {
        SQL-CALL, RMI-CALL, SOAP-CALL, JMS
        }
      - ORDER-ACTIVITY {
        ACCOUNT-LOOKUP, PAYMENT,SHIPPING,NOTIFY
        }

TNT4J Example: Activity Tracking, State Dumps

```java
public static void main(String[] args) {
    if (args.length < 6) {
        System.out.println("Usage: appl server msg correlator opname location");
        System.exit(-1);
    }
    // register with the TNT4J framework
    TrackingLogger.register(args[0], args[1]);

    // optionally register application state dump
    // by default dumps are generated on JVM shutdown
    TrackingLogger.addDumpListener(new DumpNotify());
    TrackingLogger.addDumpProvider(new MyDumpProvider("MyAppl", "JavaSystem"));

    // create and start an activity
    TrackingActivity activity = TrackingLogger.newActivity();
    activity.start();
    for (int i=0; i < 10; i++) {
        TrackingEvent event = TrackingLogger.newEvent(OpLevel.DEBUG, "Running sample=" + 1, "runSampleActivity");
        event.start(); // start timing current event
        try {
            runSampleActivity(args[2], args[3], args[4], args[5]);
        } finally {
            event.stop();
            activity.tnt(event); // associate current event with the current activity
        }
    }
    activity.stop(); // stop activity timing
    TrackingLogger.tnt(activity);    // log and report activity
    System.exit(0);
}
```

Register current thread with the jKool framework. Once for each thread.

Register a dump listener, and application dump provider

```java
class MyDumpProvider extends DefaultDumpProvider {
    public MyDumpProvider(String name, String cat) {
        super(name, cat);
    }

    @Override
    public DumpCollection getDump() {
        LinkedHashMap<Object, Object> map = new LinkedHashMap<Object, Object>(149);
        map.putAll(System.getProperties());
        return new Dump("javaProperties", map, this);
    }
}

class DumpNotify implements DumpListener {

    @Override
    public void onDumpEvent(DumpEvent event) {
        switch (event.getType()) {
        case DumpProvider.DUMP_BEFORE:
        case DumpProvider.DUMP_AFTER:
        case DumpProvider.DUMP_COMPLETE:
        case DumpProvider.DUMP_ERROR:
            System.out.println("onDump: " + event);
        }
    }
}
```

Dump Provider and Listener implementations.

Advanced: Tracing & Tracking Activities

```java
static private void runSampleActivity(String msg, String cid, String opName, String location) {
    TrackingActivity activity = TrackingLogger.newActivity();
    activity.start();
    int runs = rand.nextInt(50);
    int sev = rand.nextInt(OpLevel.values().length);
    for (int i = 0; i < runs; i++) {
        int limit = rand.nextInt(10000000);
        TrackingEvent ev4j = runTNT4JEvent(msg, opName, OpLevel.valueOf(sev), location, limit);
        ev4j.setCorrelator(cid);
        ev4j.setLocation(location);

        TrackingEvent log4j = runLog4JEvent(msg, opName, OpLevel.valueOf(sev), location, limit);
        log4j.setCorrelator(cid);
        log4j.setLocation(location);

        if (TrackingLogger.isSet(OpLevel.INFO, "tnt4j.test.location", location)){
            activity.tnt(ev4j);
            activity.tnt(log4j);
        }
    }
    activity.stop();
    TrackingLogger.tnt(activity);
}
```

Create and start timing a new activity

Generate tracking events with a correlator and locator

Stop timing and report given activity. Activities are reported to one or more destinations. Record default performance metrics such as CPU, memory, etc.

Conditionally associate tracking events with the given activity based on application severity, value, token. Logging happens only when location token matches.

```java
static private TrackingEvent runTNT4JEvent(String msg, String opName, OpLevel sev, String location, int limit) {
    TrackingEvent event = TrackingLogger.newEvent(sev, msg, opName);
    TrackingSelector selector = TrackingLogger.getTracker().getTrackingSelector();
    try {
        event.setTag(String.valueOf(TrackingLogger.getVMName()));
        event.setData(msg + ", tnt4j.run.count=" + limit);
        event.start();
        for (int i = 0; i < limit; i++) {
            selector.isSet(OpLevel.INFO, "tnt4j.test.location", location);
        }
    } finally {
        event.stop();
    }
    return event;
}
```
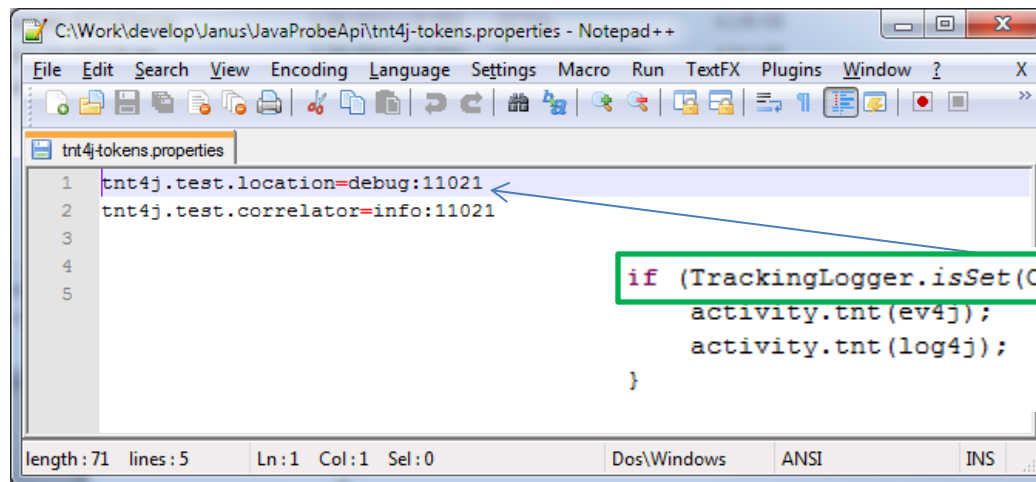
Create a new tracking event

Associate a tag, message data with this event and start timing.

Stop timing of the tracking event.

TNT4J Conditional Tracking

- TNT4J Conditional tracking is very useful when:
  - Server applications where only certain requests need to be traced. Example trace request with location matching zip code 11021.
  - Applications where tracing must be coordinated with other application downstream or upstream. Example: trace all requests matching a specific correlator/location.
  - Applications that need to dynamically set what needs to be traced:
    - **TrackingLogger.getTracker(). getTrackingSelector().set(OperationSeverity.INFO, "tnt4j.test.location", location);**
- Token selectors and token repositories
  - Token selectors allow applications to log only when certain severity/key/value pairs are matched in the token repository
  - Token repositories can be files, caches or in a data grid shared among multiple applications. Sharing of logging tokens allows central control over logging behavior at runtime across multiple applications.
  - Token selector is invoked when **TrackingLogger.isSet(..,"tnt4j.test.location",..)** is called

Dump: Application State Reporting

```java
TrackingLogger.register(args[0], args[1], args[2]);
TrackingLogger.addDumpListener(new DumpNotify());
TrackingLogger.addDumpProvider(new MyDumpProvider("MyAppl", "JavaSystem"));
TrackingLogger.dump();
```

Register a dump listener, dump provider and generate a default dump.

```java
class MyDumpProvider extends DefaultDumpProvider {
    private long startTime = 0;
    public MyDumpProvider(String name, String cat) {
        super(name, cat);
        startTime = System.currentTimeMillis();
    }

    @Override
    public DumpCollection getDump() {
        Dump dump = new Dump("runtimeMetrics", this);
        dump.add("appl.start.time", new Date(startTime));
        dump.add("appl.elapsed.ms", (System.currentTimeMillis() - startTime));
        dump.add("appl.activity.count", TNT4JTest.activityCount);
        dump.add("appl.event.count", TNT4JTest.eventCount);
        return dump;
    }
}
```

Dump provider class that actually generates application state dump as a collection of key/value pairs.

```java
class DumpNotify implements DumpListener {

    @Override
    public void onDumpEvent(DumpEvent event) {
        switch (event.getType()) {
        case DumpProvider.DUMP_BEFORE:
        case DumpProvider.DUMP_AFTER:
        case DumpProvider.DUMP_COMPLETE:
        case DumpProvider.DUMP_ERROR:
            System.out.println("onDump: " + event);
        }
    }
}
```

Dump listener is called on before, after, complete or error during dump generation for every dump provider registered with jKool

Sample Dumps

Generating Object Dumps

- Diagnose the state of important application objects
  - Use ObjectDumpProvider and register with the framework and a given user object
    - This provider holds weak reference to the supplied object
  - Dump is generated when TrackingLogger.dumpState() is called
    - Either on demand (on call) or on VM shutdown.
- Dump object fields and sizes
  - Private, protected, public fields
  - Shallow and deep object sizes (estimated memory usage)
    - Requires "-javaagent:"jkool-tnt4j-api.jar" command line option
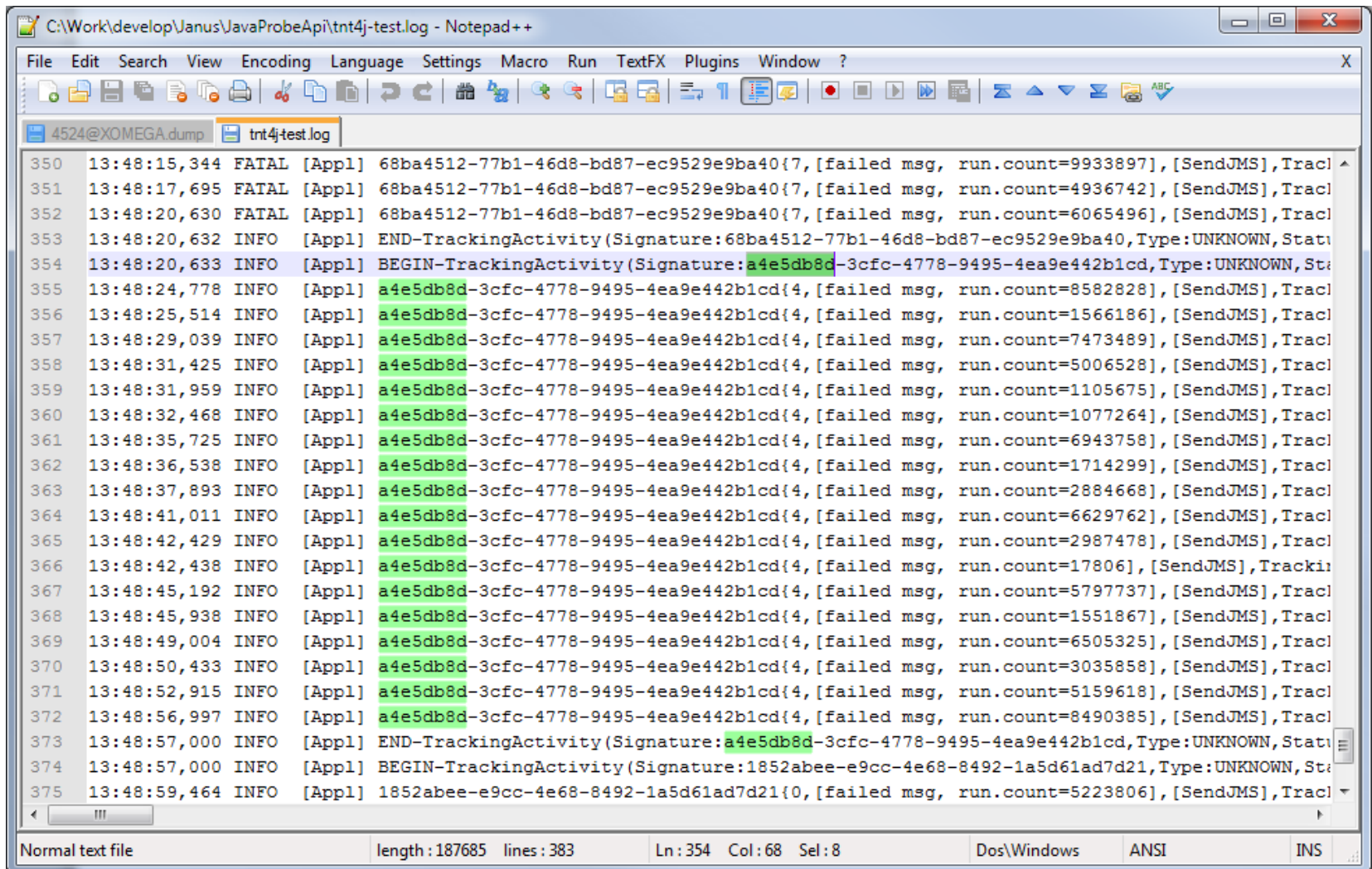
```
// optionally register application state dump
// by default dumps are generated on JVM shutdown
TrackingLogger.addDumpListener(new DumpNotify());
TrackingLogger.addDumpProvider(new ObjectDumpProvider(args[0], MyObj));
TrackingLogger.addDumpProvider(new MyDumpProvider(args[0], "ApplRuntime"));
```

Register a object dump provider to dump the contents of MyObj when dump is generated.

Example of an Object Dump

{
"dump.name": "com.nastel.jkool.tnt4j.tracker.TrackerImpl@35549f94",
"dump.provider": "com.nastel.Test",
"dump.category": "Objects",
"dump.time.string": "Wed Mar 26 11:09:47 EDT 2014",
"dump.time.stamp": 1395846587568,
"dump.collection": {
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.$sizeOf": 24,
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.$deepSizeOf": 10296,
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.$classloader": "sun.misc.Launcher$AppClassLoader@12360be0",
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.logger.$type": "com.nastel.jkool.tnt4j.sink.EventSink",
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.logger.$modifiers": "private static",
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.logger.$value": "com.nastel.jkool.tnt4j.logger.Log4jEventSink@4b0ab323",
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.logger.$sizeOf": 24,
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.logger.$deepSizeOf": 1824,
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.eventSink.$type": "com.nastel.jkool.tnt4j.sink.EventSink",
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.eventSink.$modifiers": "private",
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.eventSink.$value": "com.nastel.jkool.tnt4j.sink.SocketEventSink@732b3d53{host: localhost, port: 6408,
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.eventSink.$sizeOf": 40,
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.eventSink.$deepSizeOf": 5576,
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.tConfig.$type": "com.nastel.jkool.tnt4j.config.TrackerConfig",
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.tConfig.$modifiers": "private",
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.tConfig.$value": "com.nastel.jkool.tnt4j.config.TrackerConfigStore@5f70bea5{source: com.nastel.Test,
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.tConfig.$sizeOf": 56,
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.tConfig.$deepSizeOf": 2024,
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.selector.$type": "com.nastel.jkool.tnt4j.selector.TrackingSelector",
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.selector.$modifiers": "private",
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.selector.$value": "com.nastel.jkool.tnt4j.selector.DefaultTrackingSelector@2c96cf11",
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.selector.$sizeOf": 32,
    "com.nastel.jkool.tnt4j.tracker.TrackerImpl.selector.$deepSizeOf": 6144
}
"dump.elapsed.ms": 18
}

Sample Log: Highlighted related log entries based on a single activity

Performance: Elapsed Time of Events and Activities

Correlated Activities based on location, & correlators

Performance Metrics associated with logged activities

TNT4J High Level Architecture
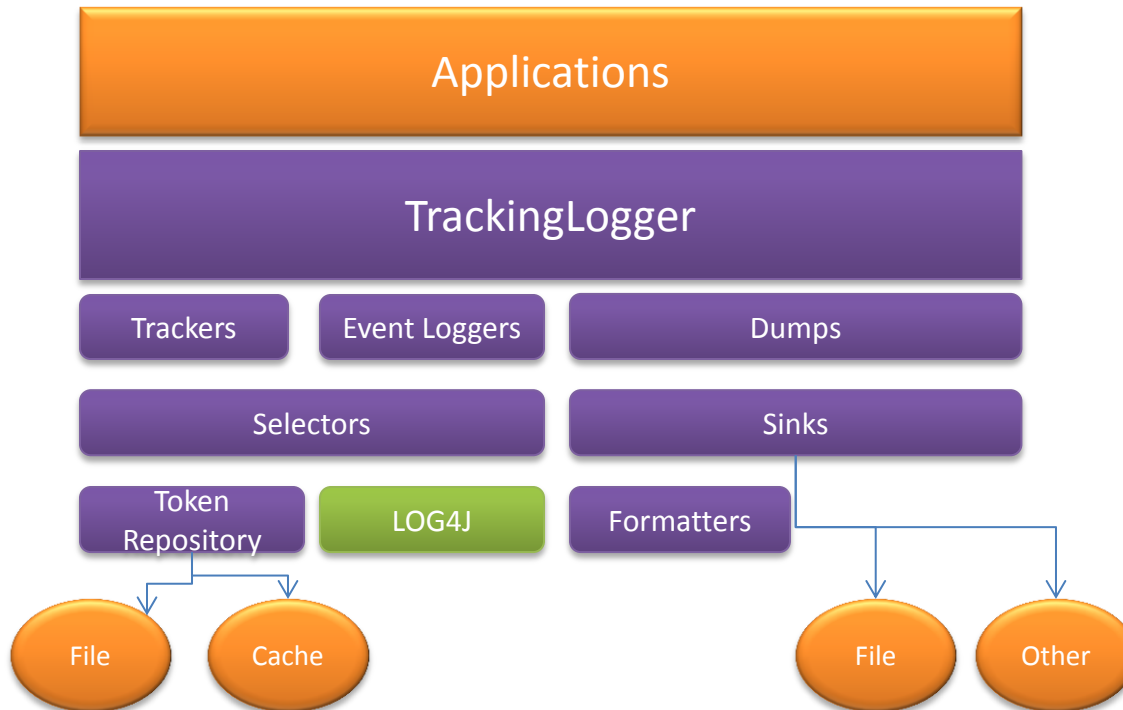
TNT4J Configuration

- ## TNT4J.PROPERTIES
  - Defines all configuration for all sources
  - For each source pattern
    - Factories, formatters, sinks and selectors
  - User defined implementations can be configured without changing code

```
// register with the TNT4J framework
TrackerConfig config = DefaultConfigFactory.getInstance().getConfig(args[0]);
config.setActivityListener(new MyActivityHandler());
TrackingLogger.register(config.build());
TrackingLogger.addSinkErrorListener(new MySinkErrorHandler());
```

Load configuration based on a given source name.

Default TNT4J.PROPERTIES

```
;Default tracking configuration for all sources (source: *),
;used only if no other stanza matches.
{
    source: *
    tracker.factory: com.nastel.jkool.tnt4j.tracker.DefaultTrackerFactory
    dump.sink.factory: com.nastel.jkool.tnt4j.dump.DefaultDumpSinkFactory
    event.sink.factory: com.nastel.jkool.tnt4j.logger.DefaultEventSinkFactory
    event.formatter: com.nastel.jkool.tnt4j.format.JSONFormatter
    tracking.selector: com.nastel.jkool.tnt4j.selector.DefaultTrackingSelector
    tracking.selector.Repository: com.nastel.jkool.tnt4j.repository.FileTokenRepository
}

; Configuration for TrackingLogger
{
    source: com.nastel.jkool.tnt4j.TrackingLogger
    tracker.factory: com.nastel.jkool.tnt4j.tracker.DefaultTrackerFactory
    dump.sink.factory: com.nastel.jkool.tnt4j.dump.DefaultDumpSinkFactory
    event.sink.factory: com.nastel.jkool.tnt4j.logger.DefaultEventSinkFactory
    event.formatter: com.nastel.jkool.tnt4j.format.JSONFormatter
    tracking.selector: com.nastel.jkool.tnt4j.selector.DefaultTrackingSelector
    tracking.selector.Repository: com.nastel.jkool.tnt4j.repository.FileTokenRepository
}

;Stanza used for sources that start with com.nastel
{
    source: com.nastel
    tracker.factory: com.nastel.jkool.tnt4j.tracker.DefaultTrackerFactory
    dump.sink.factory: com.nastel.jkool.tnt4j.dump.DefaultDumpSinkFactory
    ;event.sink.factory: com.nastel.jkool.tnt4j.logger.DefaultEventSinkFactory
    event.sink.factory: com.nastel.jkool.tnt4j.sink.SocketEventSinkFactory
    event.sink.factory.Host: localhost
    event.sink.factory.Port: 6408
    event.formatter: com.nastel.jkool.tnt4j.format.JSONFormatter
    tracking.selector: com.nastel.jkool.tnt4j.selector.DefaultTrackingSelector
    tracking.selector.Repository: com.nastel.jkool.tnt4j.repository.FileTokenRepository
    ;activity.listener: com.nastel.jkool.tnt4j.examples.MyActivityHandler
}

;Stanza used for sources that start with org
{
    source: org
    tracker.factory: com.nastel.jkool.tnt4j.tracker.DefaultTrackerFactory
    dump.sink.factory: com.nastel.jkool.tnt4j.dump.DefaultDumpSinkFactory
    event.sink.factory: com.nastel.jkool.tnt4j.logger.DefaultEventSinkFactory
    event.formatter: com.nastel.jkool.tnt4j.format.DefaultFormatter
    tracking.selector: com.nastel.jkool.tnt4j.selector.DefaultTrackingSelector
    tracking.selector.Repository: com.nastel.jkool.tnt4j.repository.FileTokenRepository
}
```

Error Handling: What happens when logging fails?

- TNT4J provides error handling via **SinkErrorListener** interface
  - Create a class that implements **SinkErrorListener**
  - Register the listener with TNT4J framework
  - Sink error listener gets called whenever writing to the sink fails
  - Scope of the error handling is per thread (TrackingLogger.register())

Register logger with a socket sink factory where events are written to a socket using JSON

```
// register with the TNT4J framework
TrackerConfig config = TrackerConfig.defaultConfig(new Application(args[0], args[1])).setEventFormatter(new JSONFormatter(false));
config.setEventSinkFactory(new SocketEventSinkFactory(System.getProperty("tnt4j.sink.factory.socket.host", "localhost"),
        Integer.getInteger("tnt4j.sink.factory.socket.port", 6400)));
TrackingLogger.register(config.build());
TrackingLogger.addSinkErrorListener(new MySinkErrorHandler());
```

Create and register a sink error handler

```
class MySinkErrorHandler implements SinkErrorListener {
    public void sinkError(SinkError event) {
        System.out.println("onSinkError: " + event);
        if (event.getCause() != null) event.getCause().printStackTrace();
    }
}
```

Actual implementation of the sink error handler

Activity Notifications

- Get notified when activities are started/stopped
  - Define a listener which implement **ActivityListener** interface
  - Pre and post activity processing
  - Adding custom metrics as snapshots to each stopped activity

```java
// register with the TNT4J framework
TrackerConfig config = TrackerConfig.defaultConfig(new Source(args[0], args[1])).setEventFormatter(new JSONFormatter(false));
// TrackerConfig config = TrackerConfig.defaultConfig(new Source(args[0], args[1]));
config.setEventSinkFactory(new SocketEventSinkFactory(System.getProperty("tnt4j.sink.factory.socket.host", "localhost"),
        Integer.getInteger("tnt4j.sink.factory.socket.port", 6400)));
config.setActivityListener(new MyActivityHandler());
TrackingLogger.register(config.build());
TrackingLogger.addSinkErrorListener(new MySinkErrorHandler());
```

```java
class MyActivityHandler implements ActivityListener {
    @Override
    public void started(Activity activity) {
        System.out.println("activity.uuid=" + activity.getSignature() + ", started=" + activity.getStartTime());
    }

    @Override
    public void stopped(Activity activity) {
        // post processing of activity: enrich activity with application metrics
        PropertySnapshot snapshot = new PropertySnapshot("APPL_METRICS");
        snapshot.add(new Property("appl.activity.count", TNT4JTest.activityCount));
        snapshot.add(new Property("appl.event.count", TNT4JTest.eventCount));
        activity.add(snapshot); // add property snapshot to activity
        System.out.println("activity.uuid=" + activity.getSignature() + ", stopped=" + activity.getElapsedTime() + ", items=" + activity.getItemCount());
    }
}
```

Register an activity listener, which is triggered when activity timing events occur.

TNT4J vs. LOG4J

Features and Performance

Capabilities: TNT4J vs LOG4J

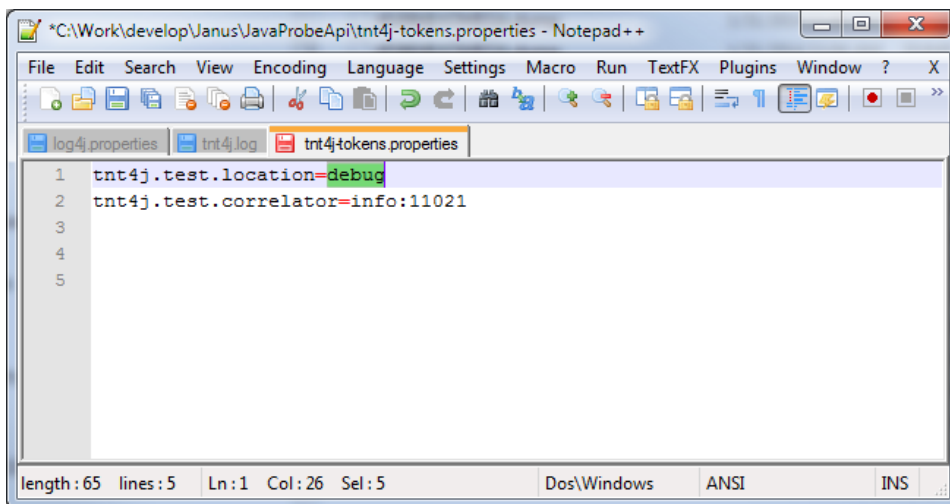| Capability	TNT4J	LOG4J
Activity tracking, timing, performance	X	NA (requires coding, log message)
Correlators, tags, geo location	X	NA (requires coding, log message)
Application state, dumps	X	NA (requires coding, log message)
Logging behavior based on user defined tokens, regular expressions	X	NA (requires coding, log message)
Cross application correlation	X	NA (requires coding, log message)
User defined metrics, properties	X	NA (requires coding, log message)
Plugin architecture (event sinks, trackers, formatters, listeners)	X	X
Pluggable with other logging frameworks	X	NA (requires coding, log message)
Error handling when logging fails	X	NA (silent failure)

Diagnostic Task Comparison

Diagnostic Activity	JKOOL TNT4J	LOG4J
Show all related events that handle a specific request	grep correlator file.log	**Not possible** unless developer explicitly writes correlator as part of each message.
How long does it take to execute activities	grep ElapsedTimeUsec file.log	**Not possible** unless developer explicitly write timing information and you need to know what that is.
Show all related events that handle a specific request across multiple application logs	grep correlator file1.log file2.log … fileN.log	**Not possible** unless developer maintains correlators across all involved applications and logs.

Enable tracing only for server requests that match a specific pattern	Add token to tnt4j-tokens.properties or shared cache.
	    tnt4j.test.location=debug:1102.*
	    tnt4j.test.correlator=info:11021
		Not possible at all. Developer can only turn on tracing for categories and as a result generate traces for all requests, thus adding much more overhead and generating too much data which makes it hard to analyze.

Enable tracing only for server requests that match a specific pattern across multiple apps that handle this request	Add token to tnt4j-tokens.properties or shared cache, makes sure all applications are sharing the underlying token repository	**Not possible at all.** Developer can only turn on tracing for categories. Developer would have to turn debug on for all involved applications and as a result generate traces for all requests across all apps, thus adding much more overhead and generating too much data which makes it hard to analyze.
Find all SEND requests for a server application	grep "Type:SEND" file.log	**Not possible** unless developer explicitly identifies and writes SEND/RECV identifiers as part of each trace message
Find all requests with a GEO location matching a specific locator (zip code)	grep "Location:zip-code" file.log … fileN.log	**Not possible** unless developer is recording a GEO location consistently across all logs.
What was CPU utilization for all traced activities	grep "Name:CPU" file.log .. fileN.log	**Not possible** unless developer measures and logs CPU usage during logging process.

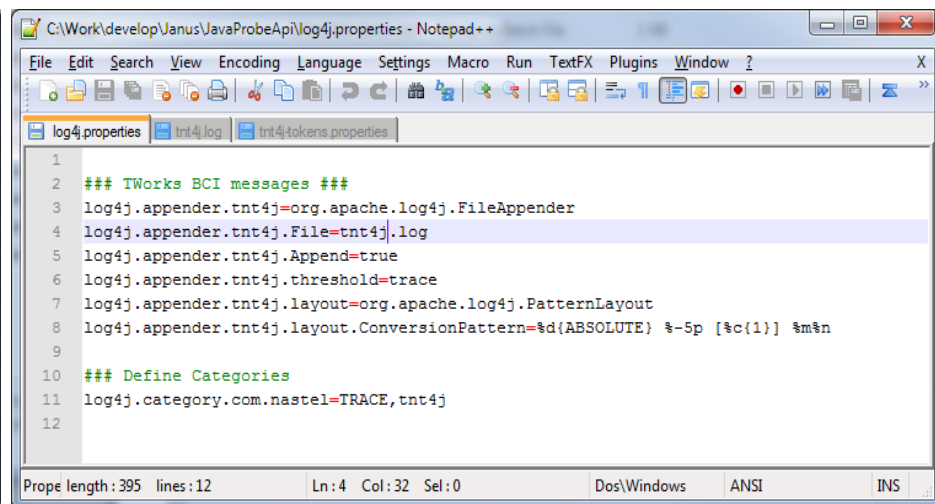# Logging Behavior Performance Comparison: No RegEx Pattern

Performance	TrackingLogger.isSet(…)	LOG4J.isDebugEnabled()
10,000,000 calls	256000 (usec)	145000 (usec)
Per call	0.0256 usec/call	0.0145 usec/call

TNT4J selector call is slower (~1.77 times) but offers more flexibility for conditional logging and pattern matching for severity/token/value combinations not offered in log4j and therefore significantly reducing overall logging output.

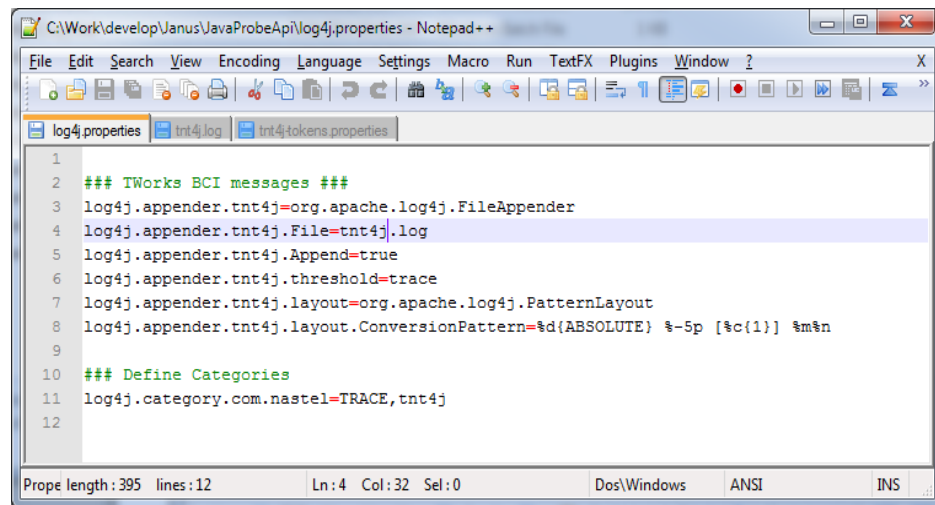# Logging Behavior Performance Comparison: TNT4J RegEx Pattern

Performance	TrackingLogger.isSet(…)	LOG4J.isDebugEnabled()
10,000,000 calls	2601000 (usec)	145000 (usec)
Per call	0.2601 usec/call	0.0145 usec/call

In this test TNT4J selector call is much slower than LOG4J due to additional pattern matching using regular expressions, but the upside is dramatically less logging output due to more granular tracking condition. jKool will only log messages that match ("tnt4j.test.location") vs. LOG4J all message for a specific category (java package scope).

Send questions or comments to amavashev@nastel.com

Q&A