




**The complete guide to**

# **Solving Java Application Errors in Production**

**Alex Zhitnitsky**

# Table of Contents



<b>Introduction.....</b>	<b>Page 2</b>
<b>Chapter 1.....</b>	<b>Page 3</b>
<b>Keep it Actionable: What's the Purpose of Using Exceptions?</b>	
<i>Let's break some myths. Exceptions are mostly ignored (and not that exceptional). This chapter covers practical tips for producing meaningful and actionable exceptions.</i>	
<b>Chapter 2.....</b>	<b>Page 10</b>
<b>Source Code Crunch: Lessons from Analyzing over 600,000 Java projects</b>	
<i>An overview of exception handling in over 600,000 Java projects. In this chapter you'll see data about how exceptions are actually used (and misused).</i>	
<b>Chapter 3.....</b>	<b>Page 17</b>
<b>Production Data Crunch: 1,000 Java Applications, 1 Billion Logged Errors</b>	
<i>Now it's time to see what happens in production. This chapter introduces the Pareto logging principle showing that 97% of logged errors originate from 3% of unique events.</i>	
<b>Chapter 4.....</b>	<b>Page 23</b>
<b>Know Your Enemy: The Top 10 Exceptions Types in Production</b>	
<i>In this chapter we dive into the most common exceptions in production, their frequency, and possible solution strategies. The infamous NullPointerException is obviously #1.</i>	
<b>Final Thoughts.....</b>	<b>Page 30</b>

## Introduction

# What's an APM and Why Do You Even Need It?

Production exception handling is a dark science. Once your code gets out to production, any minor imperfection or glitch can translate to millions of log events. Especially if you're using exceptions as part of the control flow. It's the least explored subject matter, with the largest impact on how your application behaves - And the dark patterns used to handle and solve the errors it produces are quite unpleasant.

That is why we felt the urgency to explore this issue where operations and developers meet. In this eBook, we present data gathered from over 600,000 Java projects, and a 1,000 production applications generating over 1 Billion events, with the new knowledge that it encouraged us to create.

Let's dig in.

## Chapter 1

# Keep it Actionable: What's the Purpose of Using Exceptions?

**Exceptions are probably the most misused Java language feature. Here's why**

Let's break some myths. There is no tooth fairy. Santa isn't real. TODO comments. [finalfinalversion-final.pdf](#). Soapless soap. And... Exceptions are in fact exceptions. The latter might need some more convincing, but we got you covered.

For this chapter, we asked [Avishai Ish-Shalom](#), an experienced systems architect and a longtime friend of OverOps (most importantly, a big fan of [furry hats](#)), to join us for a quick chat about the current state of exceptions in Java applications. Here's what we found out.

### Exceptions are by definition far from normal

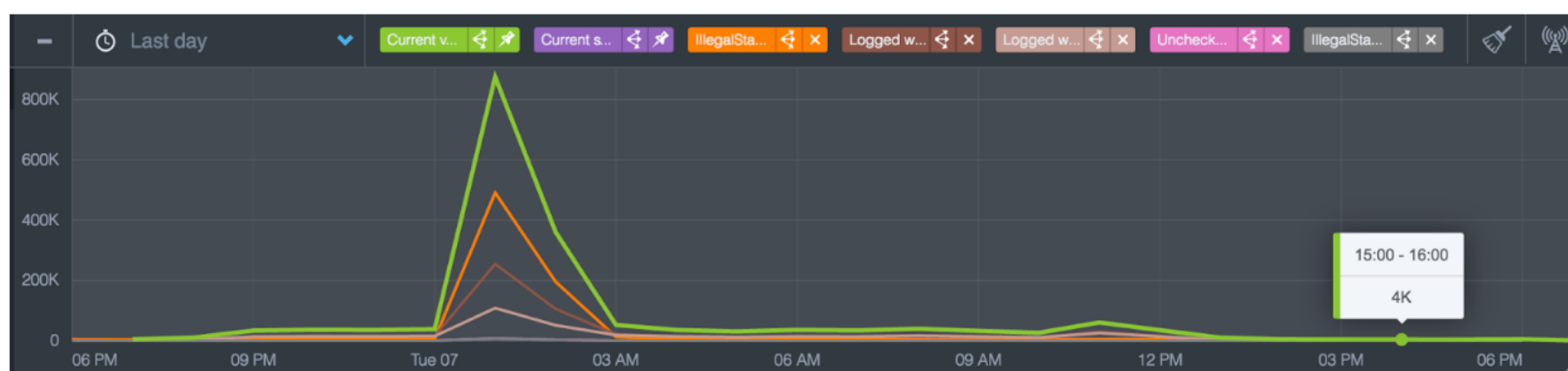
Let's kick off with a quote from the [official Java documentation](#): "An exception is an event that occurs during the execution of a program that DISRUPTS the normal flow of instructions". Honest disclosure: we've added the caps ourselves.

In practice, the normal flow of instructions in most applications is filled with "normal" recurrences of these so called "normal" exceptions, that cause "normal" disruptions.

There's an increasing high level of noise in most applications, with exceptions thrown, logged, then indexed and analyzed which... are mostly meaningless.

This operational noise, apart from creating unnecessarily stress on the system, makes you lose touch with the exceptions that really matter. Imagine an eCommerce application with a new important exception that started happening, signaling that something has gone wrong and affected, say, a 100 users that weren't able to complete their checkout. Now, cover it up with thousands of useless "normal" exceptions and try to understand what went wrong.

For example, most applications have a "normal" level of error events. In the following screenshot, we can see it's about 4k events per hour:



### OverOps's error analysis dashboard – Error trends

If we're "lucky", a new error would show itself as a spike in the graph, like we have right here with an `IllegalStateException` occurring hundreds of thousands of times around 1am (Ouch). We can immediately see what caused a spike.

The green line indicates the total number of events, and the rest of the lines indicate specific exceptions and logged errors / warnings.

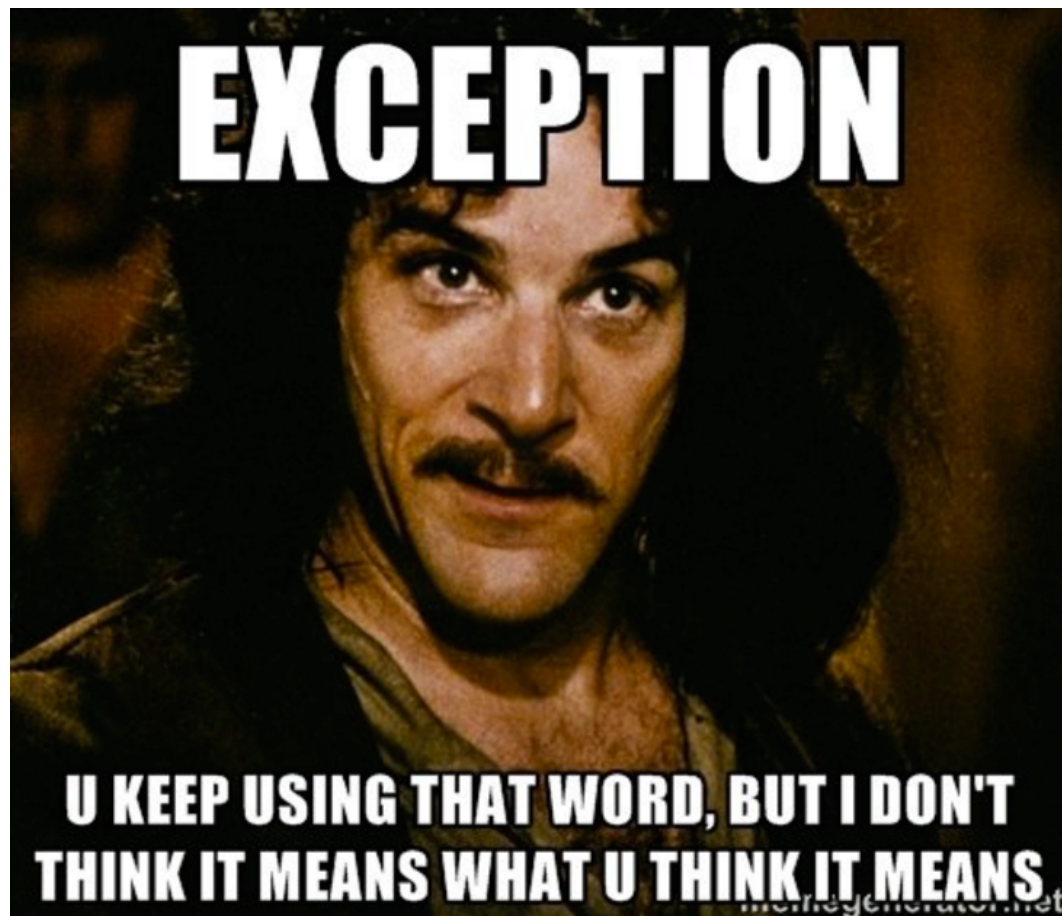
The danger comes from exceptions with only a few, small, but lethal instances that are buried within the so called "normal" level of exception.

## What are these “normal” exceptions you’re talking about?

Unlike real errors that require code changes to fix, exceptions today indicate a plethora of other scenarios that really don’t carry any actionable insights. They only weigh down on the system.

Consider these 2 scenarios that any experienced developer can anticipate:

1. **Business Errors** – Anything the user / data might do which the business flow does not permit. Like any kind of form validation, filling in text inside a phone number form field, checking out with an empty cart, etc.
2. **System Errors** – Anything you ask from the OS and it might say no, things that are out of your control. Like, trying to access a file you don’t have permissions for.



Real exceptions on the other hand, are things you weren't aware of when writing the code, like an `OutOfMemoryException`, or even a `NullPointerException` that messes things up unexpectedly. Issues that require you to take action to resolve them.

## Exceptions are



## events

### Exceptions are designed to crash & burn

Uncaught exceptions kill your thread, and might even crash the whole application or put it in some “zombie state” when an important thread is dead and the rest are stuck waiting for it. Some applications know how to handle that, most don't.

The exception's main purpose in Java is to help you catch the bug and solve it, not crossing lines into application logic land. They were meant to help in debugging which is why they try to contain as much info as possible from the application's perspective.

Another issue this can create is inconsistent state, when the application flow gets... jumpy, it's even worse than a goto statement. It has the same shortcomings, with some twists of its own:

1. It breaks the flow of the program
2. It's hard to track and understand what will happen next
3. Hard to cleanup, even with finally blocks
4. Heavyweight, carrying all the stack and additional extra data with it

## Use “error” flows without exceptions

If you try to use an exception to deal with predictable situations that should be handled by application logic, you’re in trouble. The same trouble most Java applications are in.

Issues that can be expected to happen, aren’t really exceptions by the book. An interesting solution comes from Futures in Scala – handling errors without exceptions.

Scala example from official Scala docs:

```

1  import scala.util.{Success, Failure}
2
3  val f: Future[List[String]] = Future {
4      session.getRecentPosts
5  }
6
7  f onComplete {
8      case Success(posts) => for (post <- posts) println(post)
9      case Failure(t) => println("An error has occurred: " + t.getMessage)
10 }

```

ScalaFuture.scala hosted with ❤ by GitHub view raw

Exceptions may be thrown by the code run inside the future, but they are contained and don’t leak outside. The possibility of failure is made explicit by the Failure(t) branch and it’s very easy to follow code execution.

In the new Java 8 CompletableFuture feature ([of which we also recently wrote](#)), we can use `completeExceptionally()` although it’s not as pretty.

## The plot thickens with APIs

Let’s say we have a system that uses a library for database access, how would the DB library expose its errors to the outside world? Welcome to the wild wild west. And keep in mind the library may still throw generic errors, like *java.net.UnknownHostException* or *NullPointerException*.

One real life example of how this can go wrong is a library that wraps JDBC, and just throws a generic DBException without giving you a chance to know what’s wrong. Maybe it’s all just fine and there’s just a connectivity error, or maybe... you actually need to change some code.



A common solution is the DB library using a base exception, say, `DBException`, from which library exceptions inherit. This allows the library user to catch all library errors with one try block. But what about the system errors that may have caused the library to err? The common solution is to wrap any exception happening inside it. So if it's unable to resolve a DNS address, which is more of a system error than a library error, it will catch it and rethrow this higher level exception – which the user of the library should know to catch. Try-catch nightmare, with a hint of nested exceptions wrapping other exceptions.

If we put Actors into the mix, the control flow even gets messier. Async programming with exceptions is a mess. It can kill an Actor, restart it, a message will be sent to some other Actor with the original error and you lose the stack.

## So... What can you do about it?

Starting from scratch and avoiding unnecessary exceptions is always easier, however most likely that it's not the case. With an existing system, like a 5 year old application, you're in for a lot of plumbing work (If you're lucky, and get managerial approval to fix the noise).

Ideally we'd want all exceptions to be actionable, meaning, drive actions that would prevent them from happening again, and not just acknowledge that these things sometimes happen.

To sum up, un-actionable exceptions cause a lot of mess around:

- Performance
- Stability
- Monitoring / log analysis
- And... Hide real exceptions that you want to see and act on



The solution is... doing the hard work of pruning away the noise and creating control flows that make more sense. Another creative solution is changing the log levels, if it's not an actionable exception, don't log it as an error. That's only a cosmetic solution but might get you to 80% of the work.

Ultimately, logs and dashboards are only cosmetics, there's a need to fix the issue at its core and avoid unactionable exceptions altogether.

To check out the current state of exceptions and logged errors in your application, attach the OverOps agent and you'll have a complete understanding of how code behaves in your production environment (and how to fix it) in a matter of minutes.

[Check it out.](#)

## Final Thoughts

The bottom line is, do you have an Exception that doesn't result in code changes? You shouldn't even be wasting time looking at it.

## Chapter 2

# Source Code Crunch: Lessons from Analyzing over 600,000

Exception handling in over 600,000 Java projects on Github and Sourceforge

Java is one of the few languages that use checked exceptions. They are enforced during compile time, and require handling of some sort. But... what happens in practice? Do most developers actually handle anything? And how do they do that?

In this chapter we'll go over the data from a recent research study by the university of Waterloo that covered the use of exceptions in over 600,000 Java projects from GitHub and sourceforge.

Let's answer some questions.

## The top 10 exception types in catch clauses

In this instance of the data crunch, the researchers analyzed Java projects on Github and Sourceforge, looking into the catch clauses and reporting on the findings.

Let's see how the dataset looks like:

ExceptionType	Number	Checked Exception	Unchecked Exception
Exception	3,859,217	Superclass	Superclass
IOException	2,170,519	Yes	
SQLException	681,638	Yes	
Throwable	670,214	Superclass	Superclass
InterruptedException	555,555	Yes	
IllegalArgumentException	460,872		Yes
NumberFormatException	364,362		Yes
NullPointerException	326,193		Yes
RemoteException	263,920	Yes	
RecognitionException	203,892		Yes
TOTAL		3,671,632	1,355,319

*The top 10 exception types in catch clauses, source: "Analysis of Exception Handling Patterns in Java"*

Well well, what do we have here? The research found that checked exceptions account for **almost three times** the number of unchecked exceptions in Java projects. Can't upset the compiler here.

Another insight is that developers often catch checked exceptions at the top level, using the Throwable and Exception classes.

To learn more about how checked exceptions are handled, the researchers examined the Exception and Throwable handlers. 78% of the methods that caught Exception did not catch any of its subclasses, same as 84% of Throwable.

### Meaningless catch clauses.

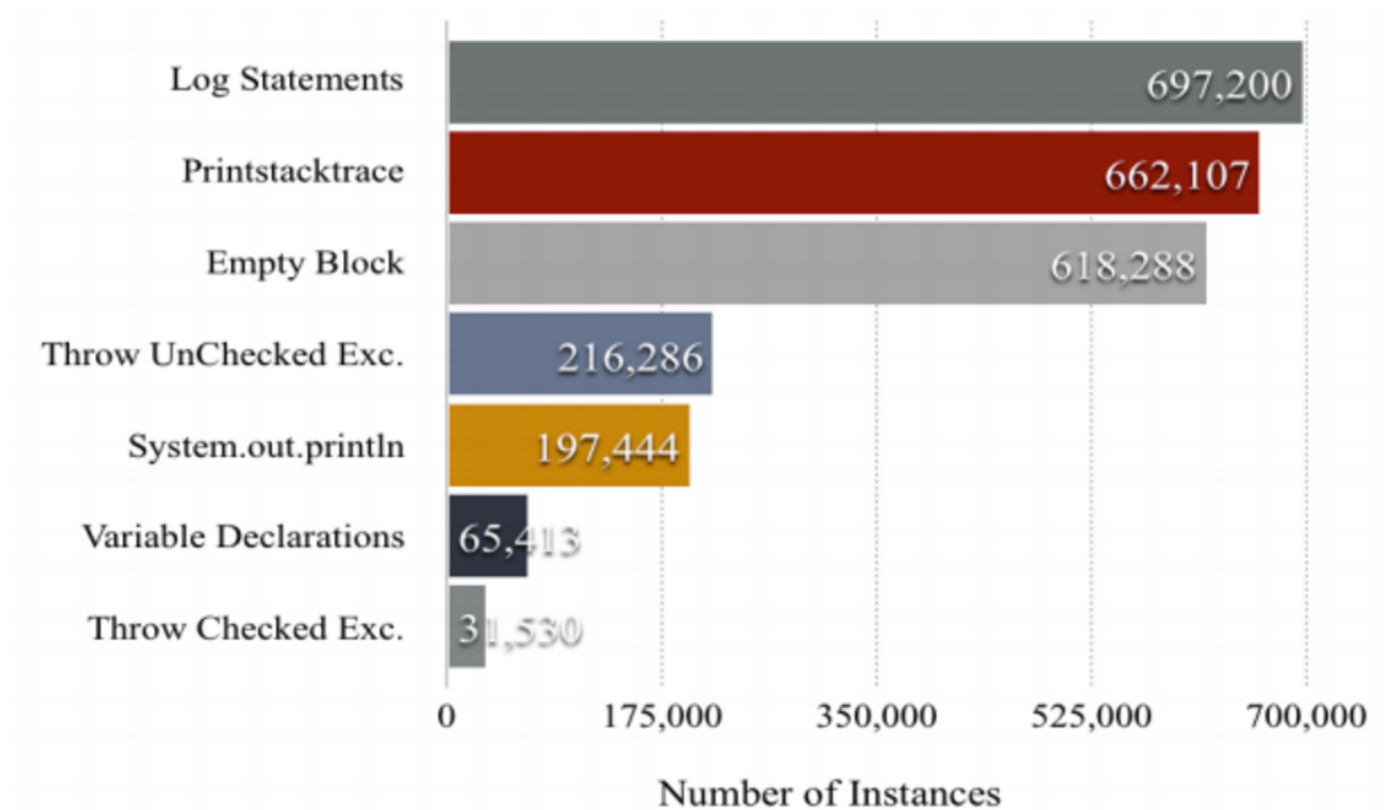
Next up, let's find out what's going on inside these catch clauses.

Maybe there's hope.

## “Most programmers ignore checked exceptions and leave them unnoticed”

Sounds bad? Keep reading. It’s an actual, real, official takeaway from the study. Many of us had this tingling spidey-sense feeling about checked exceptions, but in software development it’s unusual to have data that provides a cold hard proof to issues around actual code style. Apart from personal experiences and qualitative rather than quantitative type of studies.

The following chart shows the top operations performed in the top 3 checked exception catch blocks:



*Top operations in checked exception catch clauses, source: “Analysis of Exception Handling Patterns in Java”*

We see that log statements and `e.printStackTrace()` are at the top, making them the top operations used in checked exception catch blocks, which helps debug the situation and understand what happened.

Sneaking up on them are the notorious empty catch blocks. Joshua Bloch describes in [“Effective Java”](#) what would ideally happen, “To capture the failure, the detail message of an exception should contain the values of all parameters and fields that contributed to the exceptions”. Empty catch blocks are defeating this purpose.

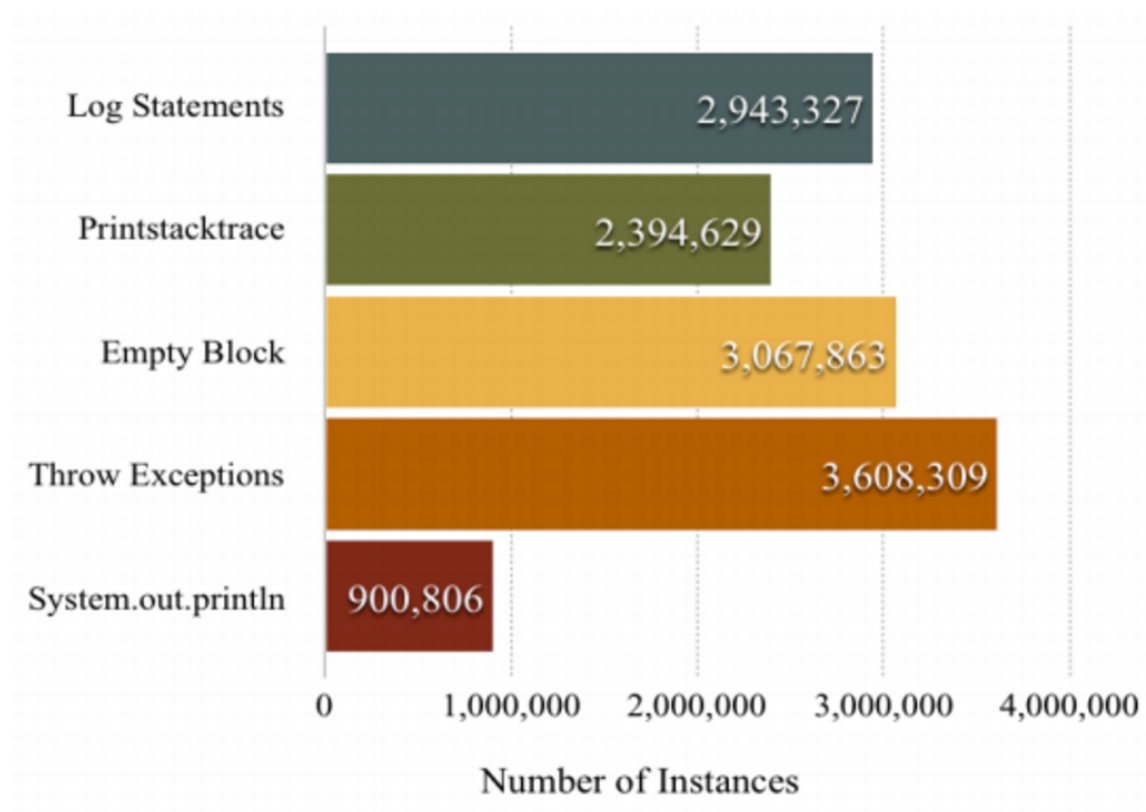
Another common use case is throwing an unchecked exception that replaces the checked exception.

[Mario Fusco](#) summed it up pretty good on his twitter feed:



## But wait, there's more

Looking at the bigger picture of both checked and unchecked exceptions, only on Github this time, we see a similar picture with rethrows gaining some more popularity:



*Top operations used in exception handling (Github), source: "Analysis of Exception Handling Patterns in Java"*

20% of the total (6,172,462) catch blocks are empty. This is quite bad. Connecting the dots with the fact that exceptions which are higher at the hierarchy are used more frequently than specific types, the researchers arrived to the conclusion that “most participants seemed to give a low priority to exception handling as a task, or included exceptions in their code only when the language forced them to handle checked exceptions”.

Eventually, the product quality suffers.

## What’s going on with the re-throws?

Since throwing exceptions up the call stack hierarchy is the most popular catch clause operation, the researchers further looked into what kind of conversions are most popular. The results are summed up in the following table:

**Table 4: Top 15 ‘A -> B’ calls made inside a catch construct (GH).**

A -> B Expression	Number of Throws
Exception -> RuntimeException	145,916
IOException -> RuntimeException	120,586
Throwable -> IOException	30,504
Exception -> IOException	22,851
Exception -> SystemException	20,230
Exception -> IllegalArgumentException	19,435
Exception -> Exception	17,818
InterruptedException -> RuntimeException	17,791
UnsupportedEncodingException -> RuntimeException	17,724
Throwable -> RuntimeException	17,507
SQLException -> RuntimeException	16,949
IllegalAccessException -> RuntimeException	16,252
HibernateException -> SystemException	16,154
RemoteException -> RuntimeException	15,332
IOException -> IllegalStateException	15,122

*Top exception transformations, source: source: “Analysis of Exception Handling Patterns in Java”*

In at #1, transforming Exception to RuntimeException. Most conversions from any exception type were made to RuntimeException, making checked exceptions unchecked.

## Exception best practices

In addition to the data crunch and its insights, the article mentions Joshua Bloch's guidelines for dealing with exceptions from [the famous 2nd edition of his book: "Effective Java"](#). We thought it would be a good idea to list them here as well:

### 1. *"Use exceptions only for exceptional scenarios"*

Exceptions cause considerable overhead on the JVM, using exceptions for normal flow control is a source for trouble (Yes, even though many developers abuse it).

### 2. *"Use checked exceptions for recoverable conditions and runtime exceptions for programming errors"*

This implies that if you find a checked exception unrecoverable, it's ok to wrap it in an unchecked exception and throw it up the hierarchy for logging and handling.

### 3. *"Avoid unnecessary use of checked exceptions"*

Use checked exceptions only when the exception cannot be avoided by properly coding the API and there's no alternate recovery step.

### 4. *"Favor the use of standard exceptions"*

Using standard exceptions from the already extensive Java API promotes readability.

### 5. *"Throw exceptions appropriate to the abstraction"*

As you go higher in the hierarchy, use the appropriate exception type.

### 6. *"Document all exceptions thrown by each method"*

No one likes surprises when it comes down to exceptions.

### 7. *"Include failure-capture information in detail messages"*

Without information about the state the JVM was in, there's not much you can do to solve the exception. Not everyone has [OverOps](#) in place to cover their back.

### 8. *"Don't ignore exceptions"*

All exceptions should lead to some action, what else do you need them for?



## Whaaaaaat exactly were we looking at here?

The data for this study comes from a [research paper](#) by [Suman Nakshatri](#), [Maithri Hegde](#), and [Sahithi Thandra](#) from the David R. Cheriton School of Computer Science at the University of Waterloo Ontario, Canada.

The researcher team looked through [a database of 7.8m Github projects and 700k Sourceforge projects](#), extracted the Java projects, and examined the use of catch blocks with the [BOA domain specific language](#) for mining software repositories.

**Table 1: Dataset Statistics for GitHub**

Total projects	7,830,023
Number of Java projects	554,864
Methods with atleast one catch block	10,862,172
Total number of Catch blocks	16,172,462

**Table 2: Dataset Statistics for SourceForge**

Total projects	699,331
Number of Java projects	50,692
Methods with atleast one catch block	6,657,595
Total number of Catch blocks	9,956,760

*The dataset by the numbers*

## Final Thoughts

Exceptions should be reserved to exceptional situations, but... other things happen in practice. Checked exceptions become unchecked, empty catch blocks are all over the place, control flow is mixed with error flow, there's lots of noise and critical data goes missing. It's a mess.

This was a main motivation to us for building [OverOps](#), a Java agent that monitors JVMs in production and takes care of filling in the blanks with everything you need to know about how exceptions behave (and how to solve them).

## Chapter 3

# Production Data Crunch: 1,000 Java Applications, 1 Billion Logged Errors

### 97% of Logged Errors are Caused by 10 Unique Errors

It's 2016 and one thing hasn't changed in 30 years. Dev and Ops teams still rely on log files to troubleshoot application issues. For some unknown reason we trust log files implicitly because we think the truth is hidden within them. If you just grep hard enough, or write the perfect regex query, the answer will magically present itself in front of you.

Yep, tools like [Splunk](#), [ELK](#) and Sumologic have made it faster to search logs but all these tools suffer from one thing – operational noise. Operational noise is the silent killer of IT and your business today. It's the reason why application issues go undetected and take days to resolve.

## Log Reality

Here's a dose of reality, you will only log what you think will break an application, and you're constrained by how much you can log without incurring unnecessary overhead on your application. This is why debugging through logging doesn't work in production and why most application issues go undetected.

Let's assume you do manage to find all the relevant log events, that's not the end of the story. The data you need isn't usually not in there, and leaves you adding additional logging statements, creating a new build, testing, deploying and hoping the error happens again. Ouch.

## Time for Some Analysis

At [OverOps](#) we capture and analyze every error or exception that is thrown by Java applications in production. Using some cheeky data science this is what I found from analyzing over 1,000 applications monitored by OverOps.

High-level aggregate findings:

- Avg. Java application will throw **9.2 million errors/month**
- Avg. Java application generates about **2.7TB of storage/month**
- Avg. Java application contains **53 unique errors/month**
- **Top 10 Java Errors by Frequency** were
  - NullPointerException
  - NumberFormatException
  - IllegalArgumentException
  - RuntimeException
  - IllegalStateException
  - NoSuchMethodException
  - ClassCastException
  - Exception
  - ParseException
  - InvocationTargetException

So there you have it, the pesky `NullPointerException` is to blame for all that's broken in log files. Ironically, checking for null was the first feedback I got in my first code review back in 2004 when I was a Java developer.

Right, here are some numbers from a randomly selected enterprise production application over the past 30 days:

- 25 JVMs
- 29,965,285 errors
- ~8.7TB of storage
- 353 unique errors
- Top Java errors by frequency were:
  - `NumberFormatException`
  - `NoSuchMethodException`
  - Custom Exception
  - `StringIndexOutOfBoundsException`
  - `IndexOutOfBoundsException`
  - `IllegalArgumentException`
  - `IllegalStateException`
  - `RuntimeException`
  - Custom Exception
  - Custom Exception

## Time for Trouble (shooting)

So, you work in development or operations and you've been asked to troubleshoot the above application which generates a million errors a day, what do you do? Well, let's zoom in on when the application had an issue right?

Let's pick, say a 15 minute period. However, that's still 10,416 errors you'll be looking at for those 15 minutes. You now see this problem called operational noise? This is why humans struggle to detect and troubleshoot applications today... and it's not going to get any easier.

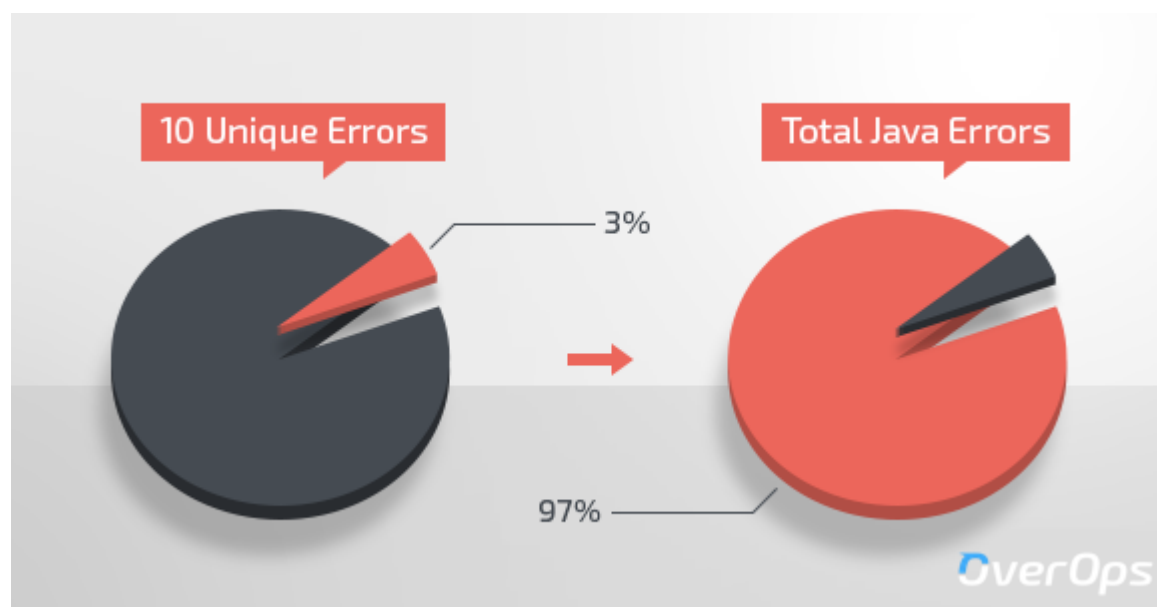
## What if we Just Fixed 10 Errors?

Now, let's say we fixed 10 errors in the above application. What percent reduction do you think these 10 errors would have on the error count, storage and operational noise that this application generates every month?

**1%, 5%, 10%, 25%, 50%?**

How about **97.3%**. Yes, you read that. Fixing just 10 errors in this application would reduce the error count, storage and operational noise by **97.3%**.

The top 10 errors in this application by frequency are responsible for 29,170,210 errors out of the total 29,965,285 errors thrown over the past 30 days.



## Take the Crap Out of Your App

The vast majority of application log files contain duplicated crap which you're paying to manage every single day in your IT environment.

You pay for:

- Disk storage to host log files on servers
- Log management software licenses to parse, transmit, index and store this data over your network
- Servers to run your log management software
- Humans to analyze and manage this operational noise

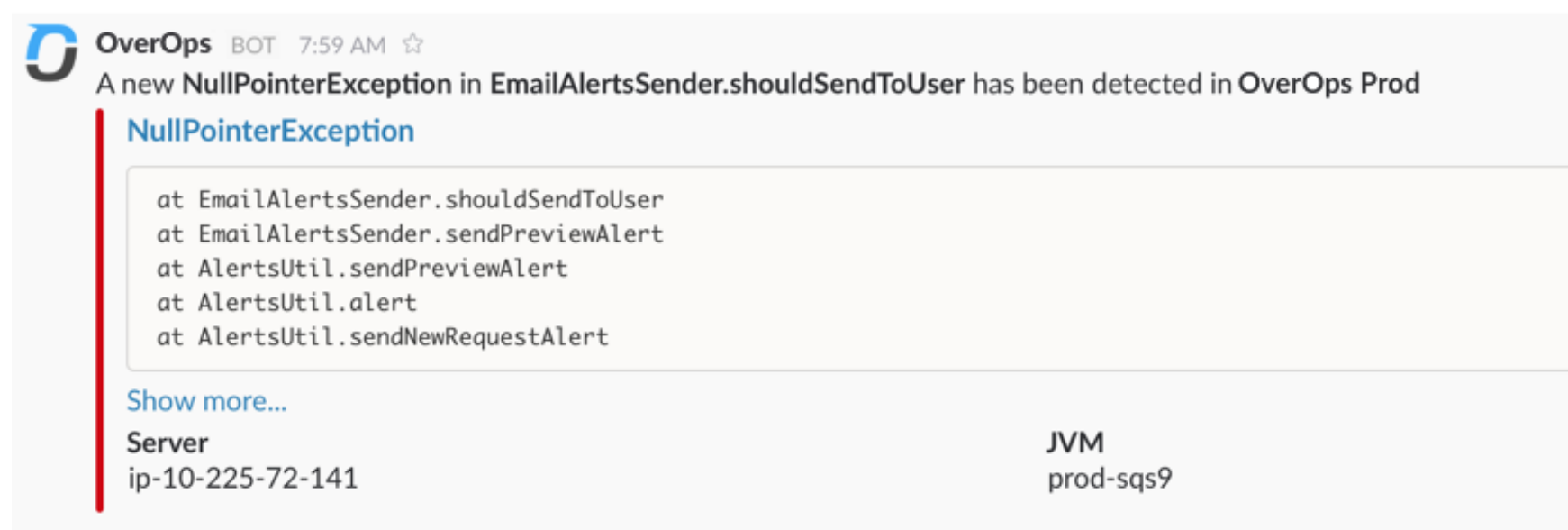
The easiest way to solve operational noise is to fix application errors versus ignore them. Not only will this dramatically improve the operational insight of your teams, you'll help them detect more issues and troubleshoot much faster because they'll actually see the things that hurt your applications and business.

## The Solution

If you want to identify and fix the top 10 errors in your application, [download OverOps for free](#), stick it on a few production JVMs, wait a few hours, sort the errors captured by frequency and in one-click OverOps will show you the exact source code, object and variable values that caused each of them. Your developers in a few hours should be able to make the needed fixes and Bob will be your uncle.

The next time you do a code deployment in production OverOps will instantly notify you of new errors which were introduced and you can repeat this process. Here's two ways we use OverOps at OverOps to detect new errors in our SaaS platform:

**Slack Real-time Notifications** which inform our team of every new error introduced in production as soon as it's thrown, and a one-click link to the exact root cause (source code, objects & variable values that caused the error).



**OverOps** BOT 7:59 AM ☆  
A new **NullPointerException** in `EmailAlertsSender.shouldSendToUser` has been detected in **OverOps Prod**

**NullPointerException**

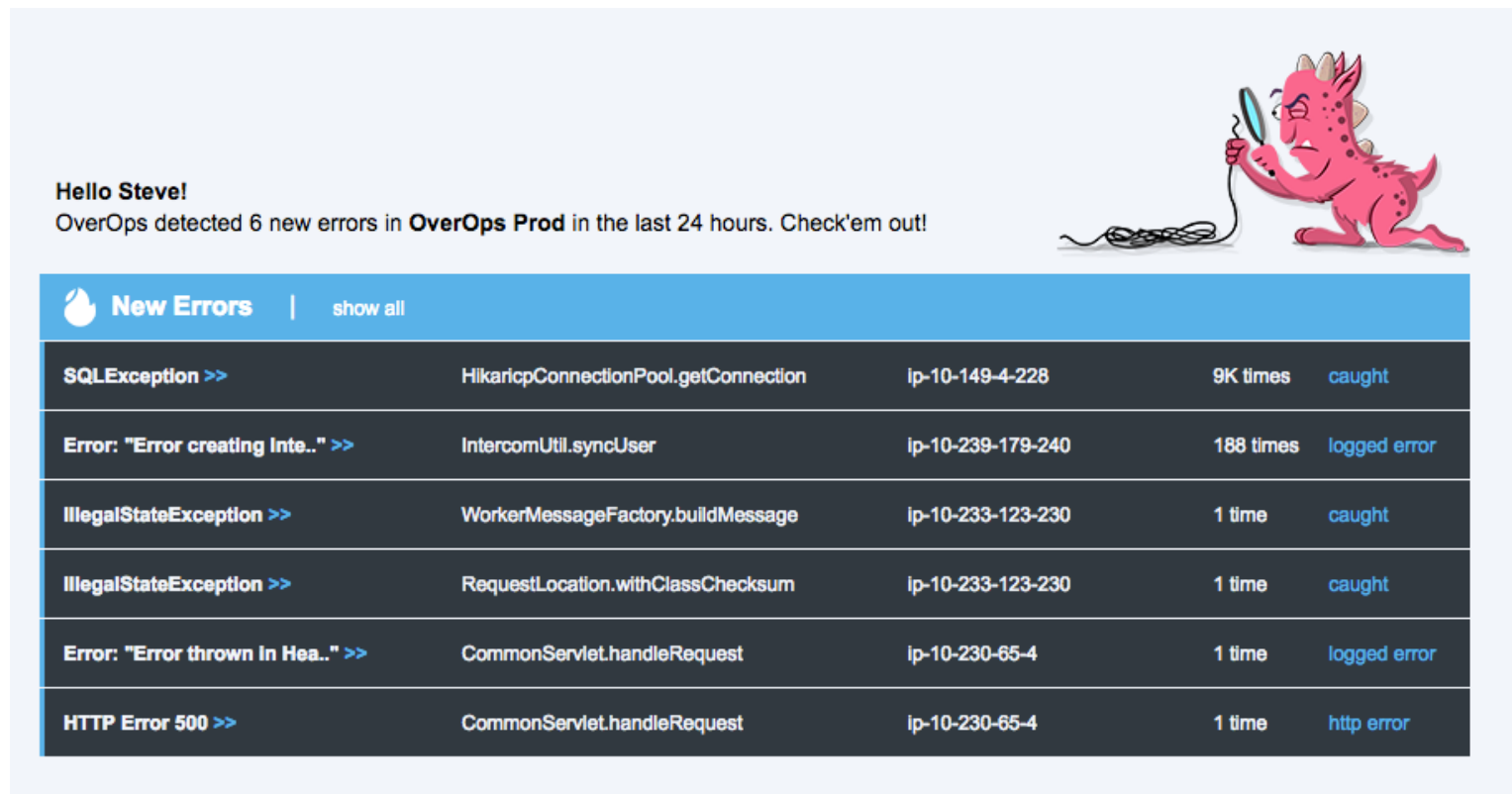
```
at EmailAlertsSender.shouldSendToUser
at EmailAlertsSender.sendPreviewAlert
at AlertsUtil.sendPreviewAlert
at AlertsUtil.alert
at AlertsUtil.sendNewRequestAlert
```

[Show more...](#)

**Server**  
ip-10-225-72-141

**JVM**  
prod-sqs9

Email Deployment Digest Report showing the top 5 new errors introduced with direct links to the exact root cause.



**Hello Steve!**  
OverOps detected 6 new errors in **OverOps Prod** in the last 24 hours. Check'em out!

New Errors		show all
<a href="#">SQLException &gt;&gt;</a>	HikaricpConnectionPool.getConnection	ip-10-149-4-228 9K times <a href="#">caught</a>
<a href="#">Error: "Error creating Inte.." &gt;&gt;</a>	IntercomUtil.syncUser	ip-10-239-179-240 188 times <a href="#">logged error</a>
<a href="#">IllegalStateException &gt;&gt;</a>	WorkerMessageFactory.buildMessage	ip-10-233-123-230 1 time <a href="#">caught</a>
<a href="#">IllegalStateException &gt;&gt;</a>	RequestLocation.withClassChecksum	ip-10-233-123-230 1 time <a href="#">caught</a>
<a href="#">Error: "Error thrown In Hea.." &gt;&gt;</a>	CommonServlet.handleRequest	ip-10-230-65-4 1 time <a href="#">logged error</a>
<a href="#">HTTP Error 500 &gt;&gt;</a>	CommonServlet.handleRequest	ip-10-230-65-4 1 time <a href="#">http error</a>

## Final Thoughts

We see time and time again that the top few logged errors in production are pulling away most of the time and logging resources. The damage these top few events cause, each happening millions of times, is disproportionate to the time and effort it takes to solve them.

## Chapter 4

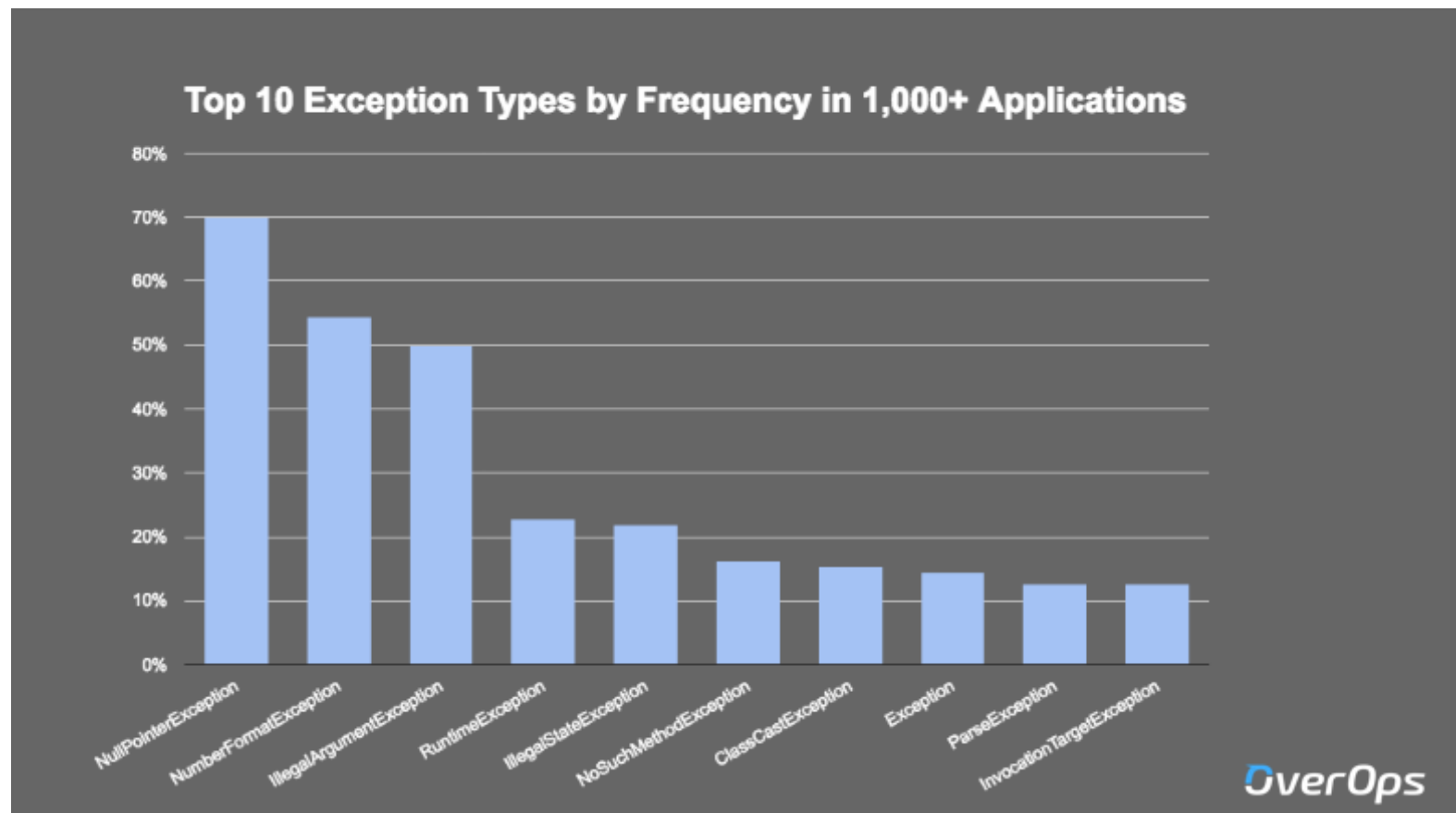
# Know Your Enemy: The Top 10 Exceptions Types in Production

The Pareto logging principle: 97% of logged error statements are caused by 3% of unique errors

We received a lot of feedback and questions following the data crunching project in the previous chapter where we showed that 97% of logged errors are caused by 10 unique errors. By popular demand, this chapter goes a step deeper into the top exceptions types in over a 1,000 applications that were included in this research.

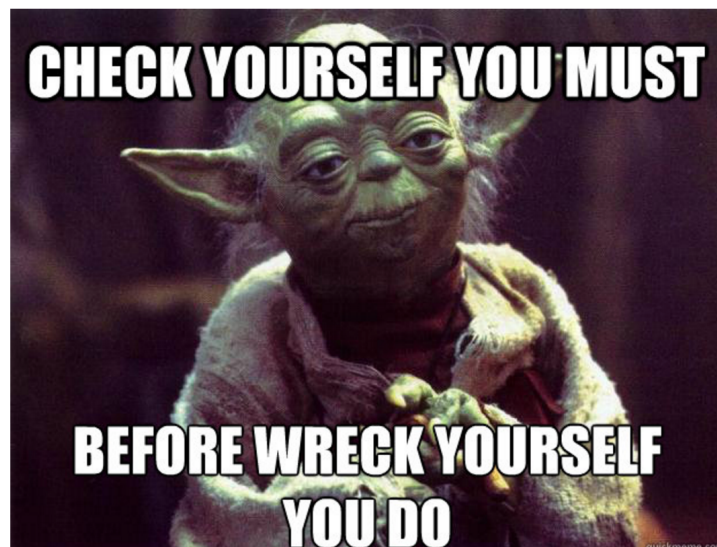


## Without Further Ado: The Top Exceptions by Types



To pull out the data, we crunched anonymized stats from over a 1,000 applications monitored by OverOps's error analysis micro-agent, and checked what were the top 10 exception types for each company. Then we combined all the data and came up with the overall top 10 list.

Every production environment is different, R&D teams use different 3rd party libraries, and also have custom exception types of their own. Looking at the bigger picture, the standard exceptions stand out and some interesting patterns become visible.



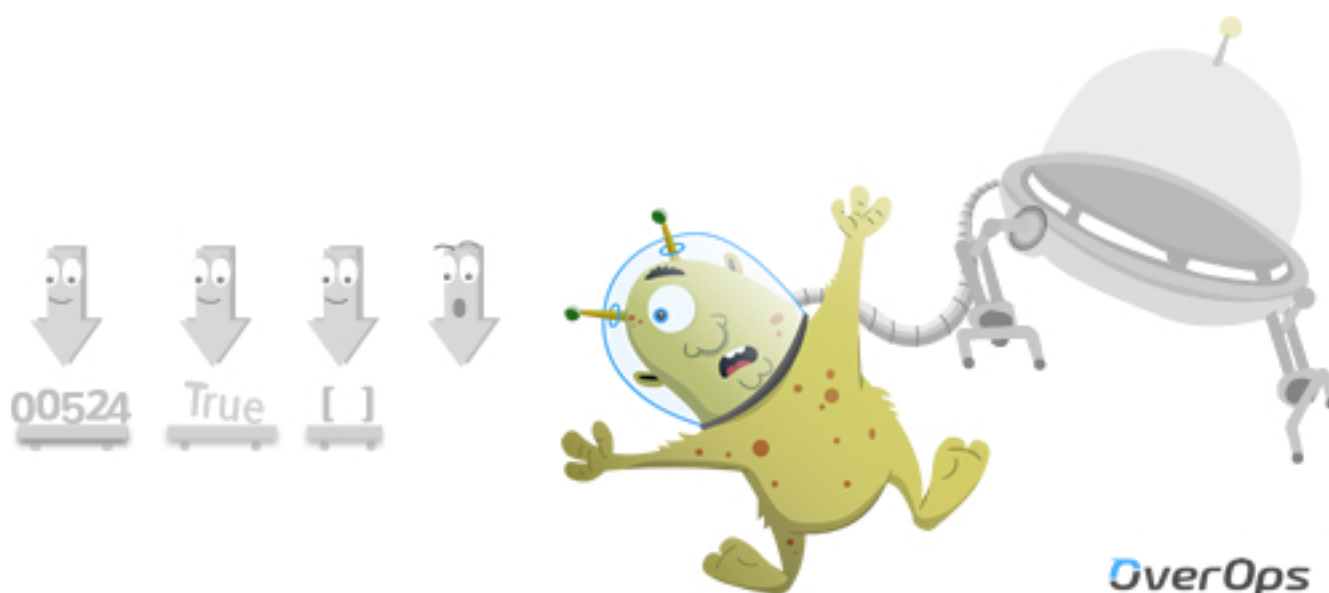
*True story*

## 1. NullPointerException – 70% of Production Environments

Yes. The infamous NullPointerException is in at #1. Sir Charles Antony Richard Hoare, inventor of the Null Reference was not mistaken when he said:

“I call it my billion-dollar mistake. It was the invention of the null reference in 1965... This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years”.

With a top 10 spot at 70% of the production environments that we looked at, NPEs take the first place. At [OverOps](#), we actually have a special alert that lets us know whenever a new NullPointerException is introduced on our system, [this is how you can set it up yourself](#).



*OverOps NPE Monster*

## 2. NumberFormatException – 55% of Production Environments

In at #2 is the NumberFormatException which happens when you try to convert a string to a numeric value and the String is not formatted correctly. It extends IllegalArgumentException which also makes an appearance here at #3.

One easy fix to make sure that the input you're passing to the parse method passes these regular expression:

1. For integer values: “-?\d+”
2. For float values: “-?\d+\.\d+”

## 3. IllegalArgumentException – 50% of Production Environments

Next up at #3, IllegalArgumentException, appearing at the top 10 exceptions in 50% of the production environments in this survey.

An IllegalArgumentException actually saves you from trouble, and thrown when you're passing arguments from an unexpected type to your methods. For example, some method that expects type X and you're calling it with type Y as an argument. Once again, an error that's caused by not checking what you're sending out as input to other methods.



OverOps

*IllegalArgumentException OverOps Monster*

## 4. RuntimeException – 23% of Production Environments

All exception objects in the top 10 list (Apart from Exception) are unchecked and extend RuntimeException. However, at #4 we're facing a "pure" RuntimeException, where Java the language actually doesn't throw any itself. So what's going on here?

There are 2 main use cases to explicitly throw a RuntimeException from your code:

1. Throwing a new "generic" unchecked exception
2. Rethrows:
  - "Wrapping" a general unchecked exception around another exception that extends RuntimeException
  - Making a checked exception unchecked

[One famous story](#) around checked vs. unchecked and the last use case we described here comes from Amazon's AWS SDK which ONLY throws unchecked exceptions and refuses to use checked exceptions.

### RuntimeException



*OverOps RuntimeExceptionMonster*

## 5. IllegalStateException – 22% of Production Environments

In at #5, featured at the top 10 exceptions in 22% of over a 1,000 applications covered in this post is the `IllegalStateException`.

An `IllegalStateException` is thrown when you're trying to use a method in an inappropriate time, like... [this scene with Ted and Robin in the first episode of How I Met Your Mother](#).

A more realistic Java example would be if you use `URLConnection`, trying to do something assuming you're not connected, and get “`IllegalStateException: Already Connected`”.

## 6. NoSuchMethodException – 16% of Production Environments

[Such Method, Much Confusion](#). 16% of the production environments in this data crunch had `NoSuchMethodException` in their top 10.

Since most of us don't write code while drunk, at least during day time, this doesn't necessarily mean that we're that delirious to think we're seeing something that's not there. That way the compiler would have caught that way earlier in the process.

This exception is thrown when you're trying to use a method that doesn't exist, which happens when you're using reflection and getting the method name from some variable or when you're building against a version of a class and using a different one at production.

## 7. ClassCastException – 15% of Production Environments

A `ClassCastException` occurs when we're trying to cast a class to another class of which it is not an instance. 15% of production environments have it in their top 10 exceptions, quite troublesome.

The rule is that you can't cast an object to a different class which it doesn't inherit from. Nature did it once, when no one was looking, and that's how we got the... [Java mouse-deer](#). Yep, that's a real creature.

## 8. Exception – 15% of Production Environments

In at #8 is the mother of all exceptions, Exception, [DUN DUN DUUUUN](#) (grandmother is Throwable).

Java never throws plain Exceptions, so this is another case like RuntimeException where it must be... you, or 3rd party code, that throws it explicitly because:

1. You need an exception and just too lazy to specify what it actually is.
2. Or... More specifically, you need a checked exception to be thrown for some reason

## 9. ParseException – 13% of Production Environments

Parsing errors strike again! Whenever we're passing a string to parse into something else, and it's not formatted the way it's supposed to, we're hit by a ParseException. Bummer.

It's more common than you might have thought with 13% of the production environments tested in this posted featuring this exception in their top 10.

The solution is... yet again, check yo' self.

## 10. InvocationTargetException – 13% of Production Environments

Another exception that's thrown at us from the world of Java Reflection is the InvocationTargetException. This one is actually a wrapper, if something goes wrong in an invoked method, that exception is then wrapped with an InvocationTargetException.

To get the original exception, you'd have to use the `getTargetException` method.

We see that 13% of production environments tested in this post had it in their list of top 10 exceptions. The second exception type here that's directly related to Java's reflection features.

## Conclusion

The world of Java exceptions is indeed quite colorful, and it's amazing to see how much impact the top 10 exceptions have on our logs. 97% of all logged errors come from 10 unique exceptions.

# Final Thoughts

We hope that you've found this guide useful. The research and insights originate from our team's experience building [OverOps](#) and supporting thousands of Java applications in production. The only monitoring tool that provides full state, source, and stack for every error in production.

Comments or questions? Please get in touch through [hello@overops.com](mailto:hello@overops.com)



[OverOps](#) is a Java agent designed to monitor your JVMs in real time without adding excess overhead.

```
String dataTableSuffix = TablesUtil.
calculateBestSuffix(minutes); String
dataTable = TablesUtil.
request
  userAgent null
  device "iPad"
  retry false
useExpandedTables ? TablesUtil. agents
TimeToMinutes.getAgentsTime :
useExpandedTables ? TablesUtil. agents
TimeToMinutes(frame.getAgentsTime())
String dataTableSuffix = TablesUtil.
calculateBestSuffix(minutes); String
dataTable = TablesUtil.
```

OverOps detects 100% of errors, caught and uncaught exceptions, HTTP and log errors.

```
String dataTableSuffix = TablesUtil.
calculateBestSuffix(minutes); String
dataTable = TablesUtil.
useExpandedTables ? TablesUtil. agents
TimeToMinutes.getAgentsTime :
useExpandedTables ? TablesUtil. agents
TimeToMinutes(frame.getAgentsTime())
String dataTableSuffix = TablesUtil.
calculateBestSuffix(minutes); String
dataTable = TablesUtil.
```

With detailed error analyses, you can see the exact variable values and code state that led to an error.