JAVA

ANTICIPATING THE UNEXPECTED

OLEG ŠELAJEV

@SHELAJEV

ZEROTURNAROUND

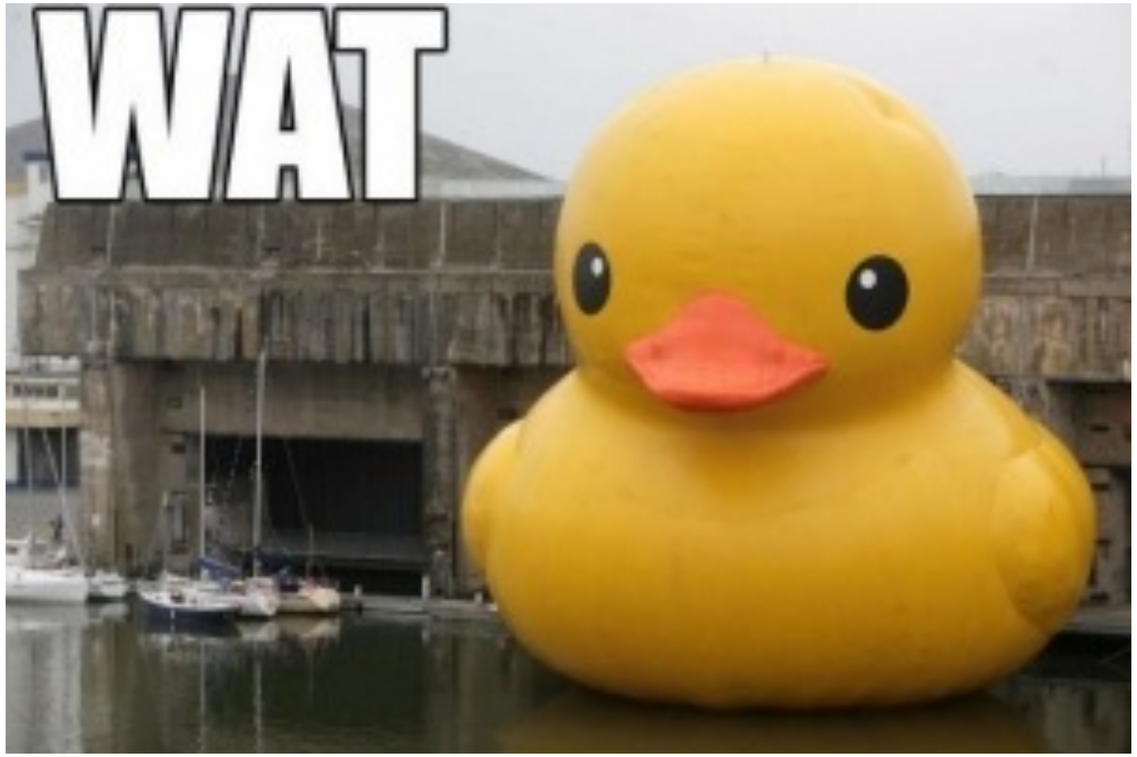
OLEG


OLEG SHELAJEV
THE PRODUCT ENGINEER
PAST JOB: PRIVATE DETECTIVE
you can't hide
hmm
+372 5518336
OLEG@ZEROTURNAROUND.COM
LINKEDIN.COM/IN/SHELAJEV
@SHELAJEV


upgrade
product
occasional
Standalone
tools
support
test
dev
LiveRebel
stable
ZeroTurnaround
Rebellabs
frameworks
strategy
Productivity
Responsibility
jenkins
legacy
servers
immediate
JRebel
nodes
Production
Disaster

INSPIRATION: **WAT**

- Wat — Destroy All Software Talks
  - Java
  - JVM
  - JEE



FORK-JOIN

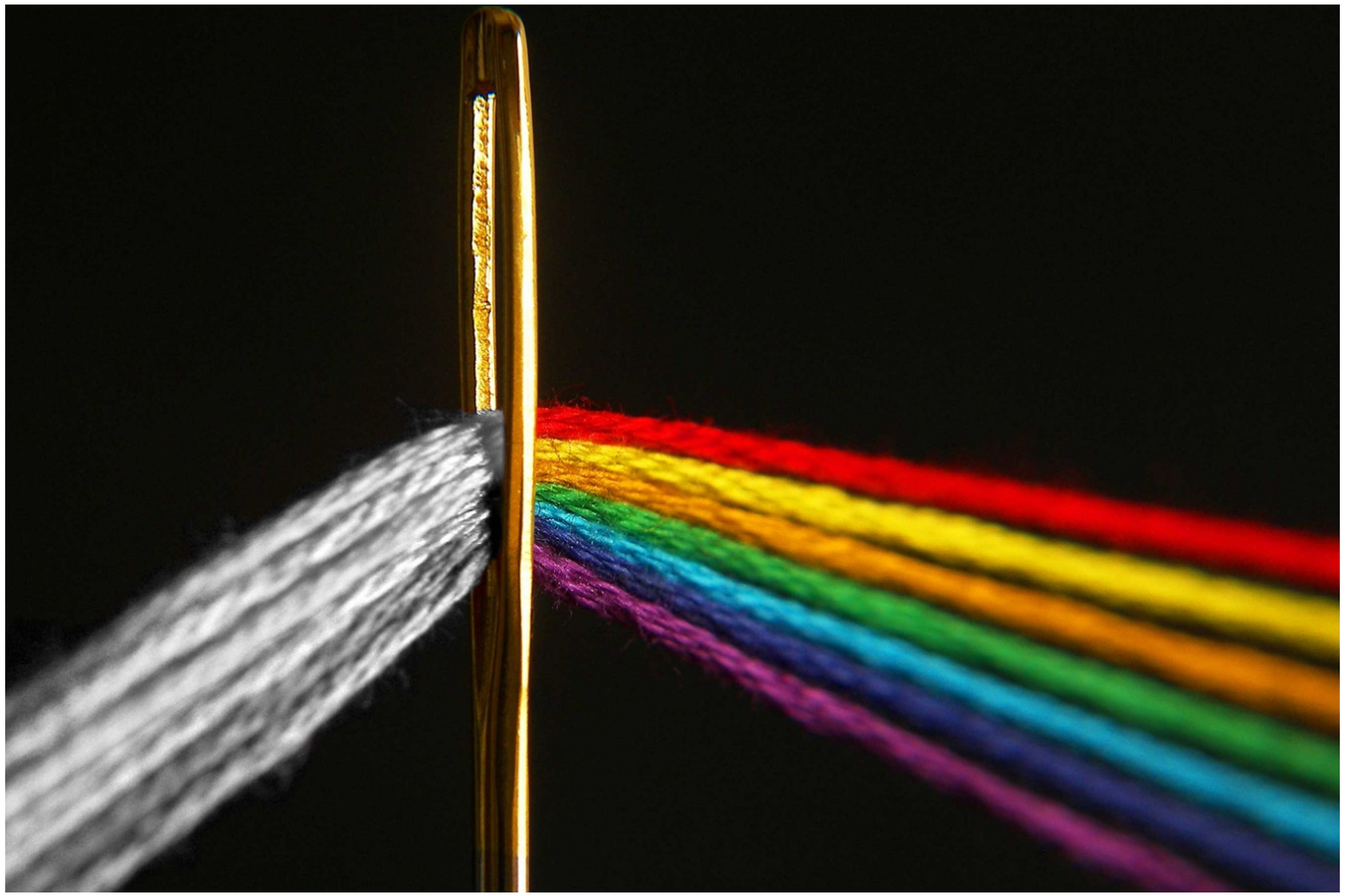
FORK-JOIN

- Parallelism
- Divide & Conquer
- Worker **threads**
- Work item queues
- Work Stealing

CLASSLOADERS

CLASSLOADERS

- Hierarchy
- Loading & Initialization
- Application isolation
- Avoiding duplication
- Painful debugging experience
- Reason for **JRebel** existence



THREADLOCALS

THREADLOCALS

- Object confined to 1 thread
- Every thread is supposed to have it’s own instance
- **SimpleDateFormat**, anyone?

RANDOM

RANDOM

- We will talk **JDK 1.8**
- Local private seed
- Initialised on demand
- Provides entropy

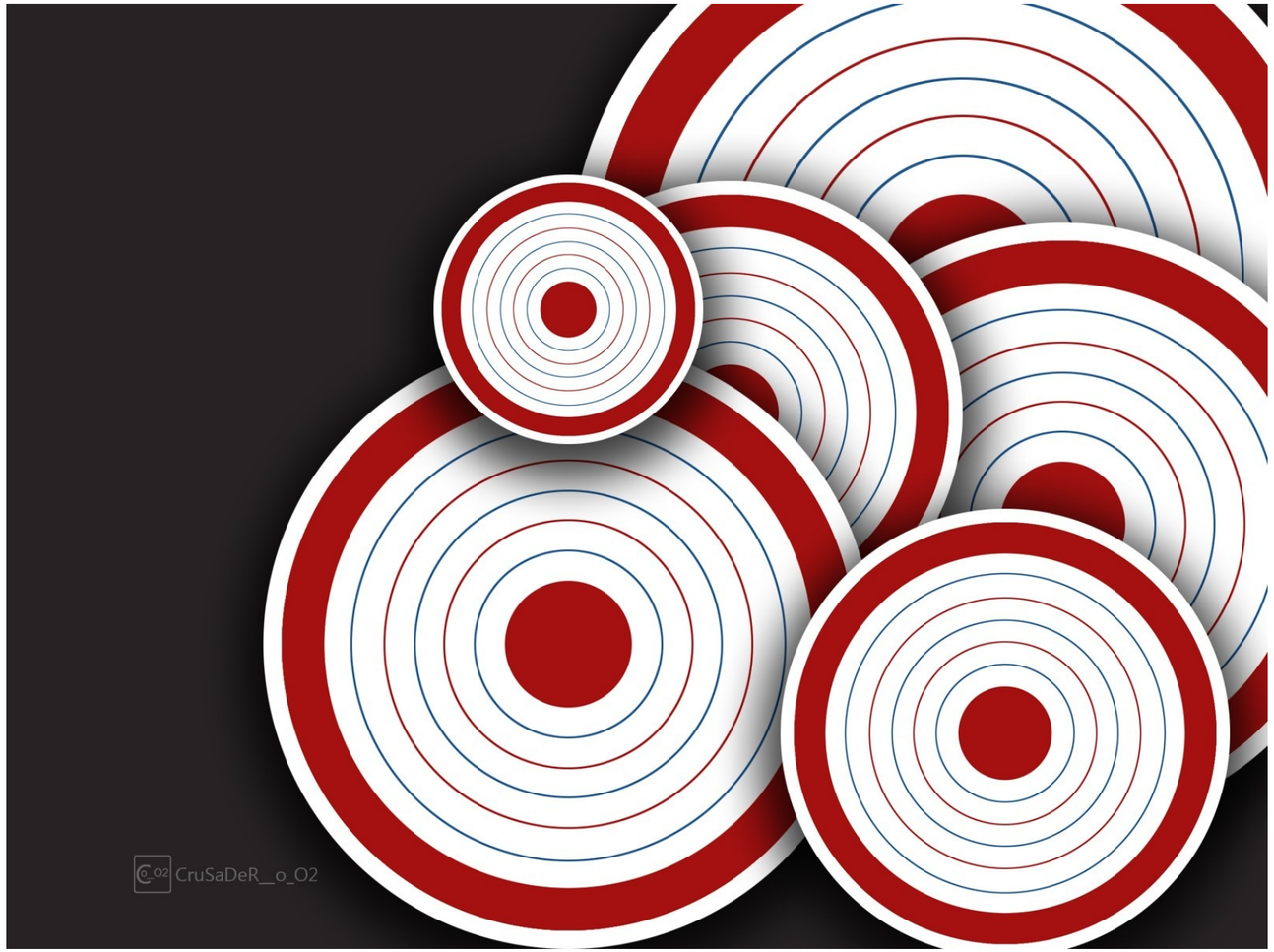
STACK OVERFLOW

STACK OVERFLOW

- Method calls
- Memory limits

- also a website

CASE OF: **CACHE MISS**



CASE OF: **CACHE MISS**

- Fork-Join pool
- Extremely recursive tasks
- Agressive caching

CASE OF: **CACHE MISS**

- Fork-Join pool
- Extremely recursive tasks
- Agressive caching

- Stumbled upon it in **Java Specialists' Newsletter** Archive

  http://goo.gl/vnlS2

CASE OF: **CACHE MISS** (SETUP)

- Computing **large** Fibonacci
- Dijkstra's sum of the squares (logarithmic)
- Karatsuba for multiplication (faster than BigInteger)

  O(3 n^1.6)

CASE OF: **CACHE MISS** (SETUP)

KaratsubaTask

MultiplyTask => 3 smaller MultiplyTasks

SquareTask => 3 smaller SquareTasks

CASE OF: **CACHE MISS** (SETUP)

KaratsubaTask

SquareTask => 3

FibonacciTask

FibonacciTask => 2 smaller FibonacciTasks

+ some multiplications after joins

CASE OF: **CACHE MISS** (GOTCHAS)

- Shared **ForkJoinPool**
- Agressive cache
- check previously computed result
- check F(n - 1) & F(n - 2), check F(n + 1) & F(n + 2)
- timed wait if result is computed currently

CASE OF: **CACHE MISS** (GOTCHAS)

- Shared **ForkJoinPool**
- Agressive cache
- check previously computed result
- check F(n - 1) & F(n - 2), check F(n + 1) & F(n + 2)
- timed wait if result is computed currently

CASE OF: **CACHE MISS** (RESULT)

"ForkJoinPool-1-worker-0" #13 daemon prio=5 os_prio=31 tid=0x00007f8ac3417470 nid=0x5903 in Object.wait() [0x00000001239ff000]

java.lang.Thread.State: **WAITING** (on object monitor)

at java.lang.Object.wait(Native Method)

- waiting on <0x00000007959dcab8> (a ParallelKaratsuba$MultiplyTask)

at java.lang.Object.wait(Object.java:502)

at java.util.concurrent.ForkJoinPool.awaitJoin(ForkJoinPool.java:2019)

- locked <0x00000007959dcab8> (a ParallelKaratsuba$MultiplyTask)

CASE OF: **CACHE MISS** (RESULT)

- All other threads

  waiting in the cache

- Blocking is not communicated to FJP

  compensation threads are not started

CASE OF: **CACHE MISS** (TAKEAWAY)

- **ForkJoinPool.ManagedBlocker** is your friend
- Be careful with sharing FJP
- on JDK 1.7 that code is lucky (on small numbers)

CASE OF: **SHALLOW IFS**

CASE OF: **SHALLOW IFS**

- Safe and **defensive** code is important
- As is **restricting** user input

CASE OF: **SHALLOW IFS**

```
private static class Relativity {
  private static final long SPEED_OF_LIGHT = 3 * 100 * 1000; // km / h
  private long speed;

  private void accelerate(long x) {
    speed += x;
    if(violatesRelativity()) {
      speed = SPEED_OF_LIGHT;
    }
  }

  public boolean violatesRelativity() {
    return speed > SPEED_OF_LIGHT;
  }
}
```

CASE OF: **SHALLOW IFS**

```
private static class Relativity {
  private static final long SPEED_OF_LIGHT = 3 * 100 * 1000; // km / h
  private long speed;

  private void accelerate(long x) {
    speed += x;
    if(violatesRelativity()) {
      speed = SPEED_OF_LIGHT;
    }
  }

  public boolean violatesRelativity() {
    return speed > SPEED_OF_LIGHT;
  }
}
```

CASE OF: **SHALLOW IFS**

```
public static void main(String... args) {
    Relativity r = new Relativity();
    r.accelerate(Long.MAX_VALUE);
    System.out.println(r);
}
```

CASE OF: **SHALLOW IFS**

```
public static void main(String... args) {
    Relativity r = new Relativity();
    r.accelerate(Long.MAX_VALUE);
    System.out.println(r);
}
```

Nothing bad happens!

CASE OF: **SHALLOW IFS**

```
public static void main(String... args) {
  Relativity r = new Relativity();
  exhaust(r);
}

private static int depth = 0;
private static void exhaust(Relativity r) {
  try {
    exhaust(r);
  } catch (StackOverflowError ignored) {}

  if (depth++ == 7) {
    accelerateThenExhaust(r);
  }
}

private static void accelerateThenExhaust(Relativity r) {
  r.accelerate(Relativity.SPEED_OF_LIGHT + 2014);
  accelerateThenExhaust(r);
}
```

CASE OF: **SHALLOW IFS**

```
public static void main(String... args) {
  Relativity r = new Relativity();
  exhaust(r);
}

private static int depth = 0;
private static void exhaust(Relativity r) {
  try {
    exhaust(r);
  } catch (StackOverflowError ignored) {}

  if (depth++ == 7) {
    accelerateThenExhaust(r);
  }
}

private static void accelerateThenExhaust(Relativity r) {
  r.accelerate(Relativity.SPEED_OF_LIGHT + 2014);
  accelerateThenExhaust(r);
}
```

Get close to the stack limit

CASE OF: **SHALLOW IFS**

```
public static void main(String... args) {
  Relativity r = new Relativity();
  exhaust(r);
}

private static int depth = 0;
private static void exhaust(Relativity r) {
  try {
    exhaust(r);
  } catch (StackOverflowError ignored) {}

  if (depth++ == 7) {
    accelerateThenExhaust(r);
  }
}

private static void accelerateThenExhaust(Relativity r) {
  r.accelerate(Relativity.SPEED_OF_LIGHT + 2014);
  accelerateThenExhaust(r);
}
```

Get close to the stack limit

Find violating depth,
trial and error way

CASE OF: **SHALLOW IFS**

- Don’t write code like that
- Think of how code can be abused

  Application code

  Application server code

  JVM code and rules

CASE OF: **SHALLOW IFS**

- StackOverflowErrors
- Waiting on objects releases locks

  Synchronization can suffer
- Serialization overrides final fields
- Order of operations is important

CASE OF: **LATERAL DEFAULTS**

CASE OF: **LATERAL DEFAULTS**

- Default methods
- Static methods in interfaces
- Runnable interfaces

  web-server in a single interface

CASE OF: LATERAL DEFAULTS

```
class EmptyList<E extends Comparable<E>> extends AbstractList<E> implements RandomAccess, Serializable {

  public void reverse() {
    this.sort(Comparator.<E>reverseOrder());
  }

  public Iterator<E> iterator() { return Collections.emptyIterator(); }
  public ListIterator<E> listIterator() { return Collections.emptyListIterator(); }
  public int size() {return 0;}
  public boolean isEmpty() {return true;}
  public boolean contains(Object obj) {return false;}
  public boolean containsAll(Collection<?> c) { return c.isEmpty(); }
  public Object[] toArray() { return new Object[0]; }
  public <T> T[] toArray(T[] a) {
    if (a.length > 0)
      a[0] = null;
    return a;
  }
  public E get(int index) { throw new IndexOutOfBoundsException("Index: "+index); }
  public boolean equals(Object o) { return (o instanceof List) && ((List<?>)o).isEmpty(); }
  public int hashCode() { return 1; }
}
```

CASE OF: LATERAL DEFAULTS

```
class EmptyList<E extends Comparable<E>> extends AbstractList<E> implements RandomAccess, Serializable {

  public void reverse() {
    this.sort(Comparator.<E>reverseOrder());
  }

  public Iterator<E> iterator() { return Collections.emptyIterator(); }
  public ListIterator<E> listIterator() { return Collections.emptyListIterator(); }
  public int size() {return 0;}
  public boolean isEmpty() {return true;}
  public boolean contains(Object obj) {return false;}
  public boolean containsAll(Collection<?> c) { return c.isEmpty(); }
  public Object[] toArray() { return new Object[0]; }
  public <T> T[] toArray(T[] a) {
    if (a.length > 0)
      a[0] = null;
    return a;
  }
  public E get(int index) { throw new IndexOutOfBoundsException("Index: "+index); }
  public boolean equals(Object o) { return (o instanceof List) && ((List<?>)o).isEmpty(); }
  public int hashCode() { return 1; }
}
```

Extremely useful

CASE OF: LATERAL DEFAULTS

```
class AnotherEmpty<E extends Comparable<E>>
  extends EmptyList<E> implements RandomAccessInterface<E> {
}
```

All methods are inherited!

CASE OF: LATERAL DEFAULTS

```
public class LateralDefaults {
  public static void main(String[] args) {
    EmptyList<Integer> list1 = new EmptyList<>();
    list1.reverse();
    System.out.println("Empty 1 reversed: " + list1);
    AnotherEmpty<Integer> list2 = new AnotherEmpty<>();
    list2.reverse();
    System.out.println("Empty 2 reversed: " + list2);
  }
}
```

CASE OF: LATERAL DEFAULTS

```
Empty 1 reversed: []
Exception in thread "main" java.lang.OutOfMemoryError: Just kidding!
  at org.shelajev.throwaway.anticipating.RandomAccessInterface.sort(LateralDefaults.java:60)
  at org.shelajev.throwaway.anticipating.EmptyList.reverse(LateralDefaults.java:30)
  at org.shelajev.throwaway.anticipating.LateralDefaults.main(LateralDefaults.java:22)
```

CASE OF: LATERAL DEFAULTS

```
class AnotherEmpty<E extends Comparable<E>>
  extends EmptyList<E> implements RandomAccessInterface<E> {
}
```

```
interface RandomAccessInterface<E> extends List<E> {
  default void sort(Comparator<? super E> c) {
    throw new OutOfMemoryError("Just kidding!");
  }
}
```

CASE OF: LATERAL DEFAULTS

```
class EmptyList<E extends Comparable<E>> extends AbstractList<E> implements RandomAccess, Serializable {

  public void reverse() {
    this.sort(Comparator.<E>reverseOrder());
  }

  public Iterator<E> iterator() { return Collections.emptyIterator(); }
  public ListIterator<E> listIterator() { return Collections.emptyListIterator(); }
  public int size() {return 0;}
  public boolean isEmpty() {return true;}
  public boolean contains(Object obj) {return false;}
  public boolean containsAll(Collection<?> c) { return c.isEmpty(); }
  public Object[] toArray() { return new Object[0]; }
  public <T> T[] toArray(T[] a) {
    if (a.length > 0)
      a[0] = null;
    return a;
  }
  public E get(int index) { throw new IndexOutOfBoundsException("Index: "+index); }
  public boolean equals(Object o) { return (o instanceof List) && ((List<?>)o).isEmpty(); }
  public int hashCode() { return 1; }
}
```

Superclass

```
public interface List<E> extends Collection<E> {
  default void sort(Comparator<? super E> c) {
    Collections.sort(this, c);
  }
  ...
}
```

SuperInterface

Root of evil

```
class AnotherEmpty<E extends Comparable<E>>
  extends EmptyList<E> implements RandomAccessInterface<E> {
}
```

Subclass

```
interface RandomAccessInterface<E> extends List<E> {
  default void sort(Comparator<? super E> c) {
    throw new OutOfMemoryError("Just kidding!");
  }
}
```

CASE OF: **LATERAL DEFAULTS**

- Always check your **interfaces**
- Unseen flexibility with **JDK 8**
- Features that **can** be abused, **will** be abused!

CASE OF: **UNFORTUNATE CONDITIONALS**

CASE OF: **UNFORTUNATE CONDITIONALS**

- It’s all about the types:

  Ternary operator => type conversion

  Autoboxing => type conversion

CASE OF: **UNFORTUNATE CONDITIONALS**

```
public class UnfortunateConditionals {
  public static void main(String... args) {
    boolean immediate = false, later = false;

    Number answer = answerToEverything(immediate, later);
    System.out.println("Welcome, answer to everything is: " + answer);

  }

  private static Number answerToEverything(boolean primitive, boolean object) {
    return primitive ? 42 : object ? new Integer(42) : null;
  }
}
```

CASE OF: **UNFORTUNATE CONDITIONALS**

```
Exception in thread "main" java.lang.NullPointerException
  at org.shelajev.throwaway.anticipating.UnfortunateConditionals.main(UnfortunateConditionals.java:10)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
  at java.lang.reflect.Method.invoke(Method.java:483)
  at com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)
```

CASE OF: **UNFORTUNATE CONDITIONALS**

```
Exception in thread "main" java.lang.NullPointerException
  at org.shelajev.throwaway.anticipating.UnfortunateConditionals.main(UnfortunateConditionals.java:10)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
  at java.lang.reflect.Method.invoke(Method.java:483)
  at com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)
```

> javap -c UnfortunateConditionals

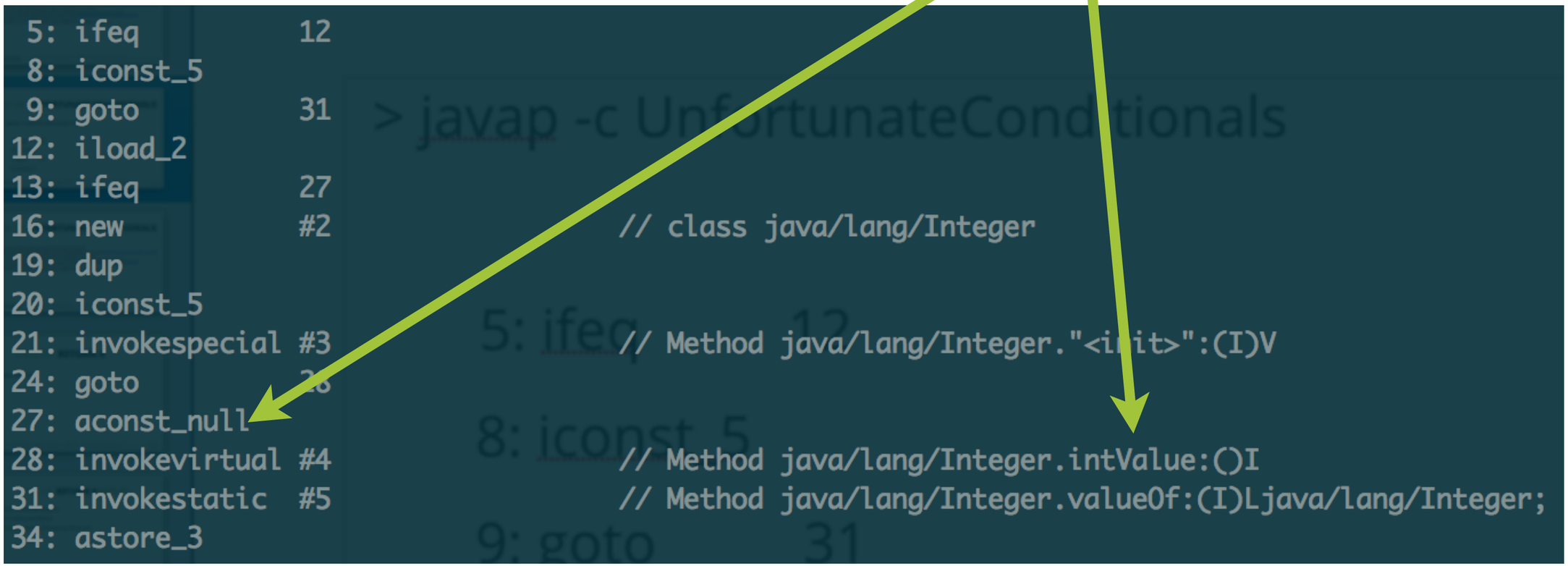
CASE OF: **UNFORTUNATE CONDITIONALS**

> javap -c UnfortunateConditionals

```
 5: ifeq          12
 8: iconst_5
 9: goto          31
12: iload_2
13: ifeq          27
16: new           #2                  // class java/lang/Integer
19: dup
20: iconst_5
21: invokespecial #3                  // Method java/lang/Integer."<init>":(I)V
24: goto          28
27: aconst_null
28: invokevirtual #4                  // Method java/lang/Integer.intValue:()I
31: invokestatic  #5                  // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
34: astore_3
```

CASE OF: **UNFORTUNATE CONDITIONALS**

> javap -c UnfortunateConditionals



CASE OF: **UNFORTUNATE CONDITIONALS**

JLS 15.25. Conditional Operator ? :

- If one is of **primitive** type T, and the other is **boxed** T, then the type of the conditional expression is T.
- If one is of the **null** type and the other is a **reference** type, then the type of the conditional expression is that **reference** type.

CASE OF: **UNFORTUNATE CONDITIONALS**

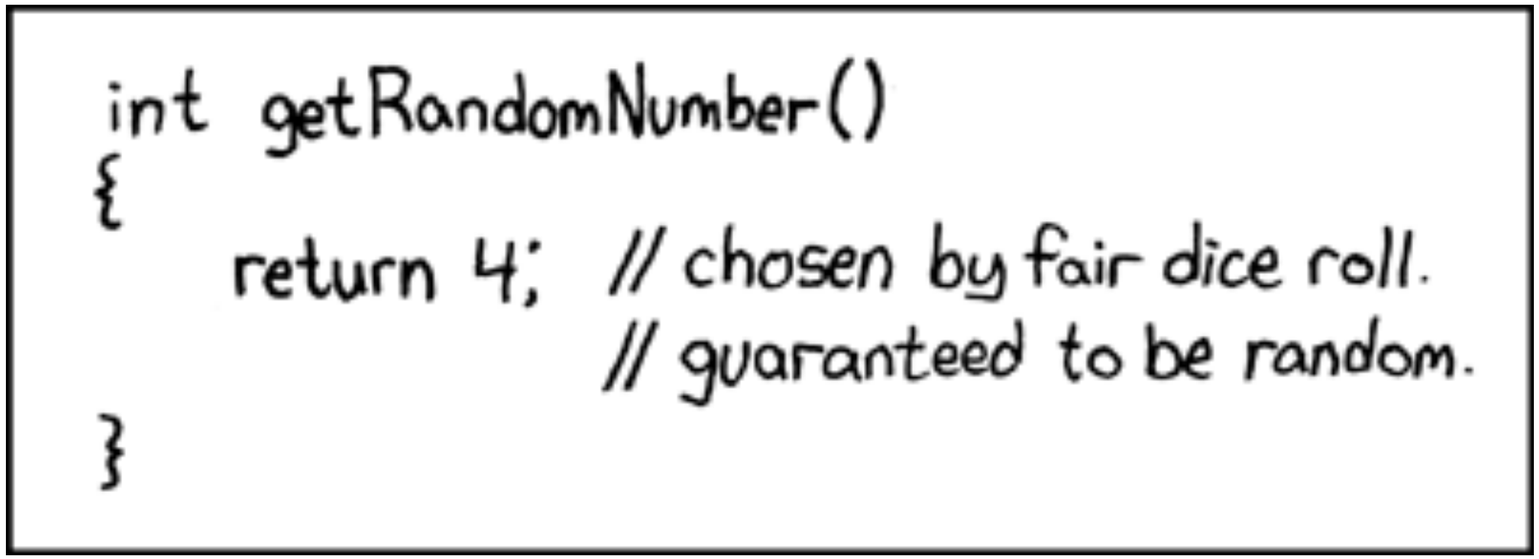
- condition ? Integer : null
- expression type: **Integer**

- condition ? int : Integer
- expression type: **int**

CASE OF: **UNFORTUNATE CONDITIONALS**

- Pay attention to IDE **warnings**
- When in doubt, check **bytecode**
- **Stop mixing** primitive and wrapper types already

CASE OF: **RETURN 4**



http://xkcd.com/221/

CASE OF: **RETURN 4** (SETUP)

- Not thread safe objects
- ThreadLocal<Object>
- **Danger:** leaking threadlocal instances

CASE OF: **RETURN 4**

```
public interface ReturnFour {
  public static void main(String... args) {
    ThreadLocalRandom tlr = ThreadLocalRandom.current();
    SwingUtilities.invokeLater(() -> {
      Supplier<String> text = () -> {
        char[] randomText = "Gt}F}\u00803u~f\u007F\u0081".toCharArray();
        for (int i = 0; i < randomText.length; i++) {
          randomText[i] -= tlr.nextInt(26);
        }
        return new String(randomText);
      };
      showWithSwing(text.get());
    });
  }
```

CASE OF: **RETURN 4**

Every single time!

CASE OF: **RETURN 4**

- ThreadLocalRandom is shared
- Seed init happens based on thread ID
- Swing thread sees default value
- Not so random output after all

CASE OF: **RETURN 4**

```
public interface ReturnFour {
  public static void main(String... args) {
    ThreadLocalRandom tlr = ThreadLocalRandom.current();
    SwingUtilities.invokeLater(() -> {
      Supplier<String> text = () -> {
        char[] randomText = "Gt}F}\u00803u~f\u007F\u0081".toCharArray();
        for (int i = 0; i < randomText.length; i++) {
          randomText[i] -= tlr.nextInt(26);
        }
        return new String(randomText);
      };
      showWithSwing(text.get());
    });
  }
```

CASE OF: **RETURN 4**

Leaking is easy!

```
public interface ReturnFour {
  public static void main(String... args) {
    ThreadLocalRandom tlr = ThreadLocalRandom.current();
    SwingUtilities.invokeLater(() -> {
      Supplier<String> text = () -> {
        char[] randomText = "Gt}F}\u00803u~f\u007F\u0081".toCharArray();
        for (int i = 0; i < randomText.length; i++) {
          randomText[i] -= tlr.nextInt(26);
        }
        return new String(randomText);
      };
      showWithSwing(text.get());
    });
  }
```

CASE OF: **RETURN 4**

```
public interface ReturnFour {
  public static void main(String... args) {
    ThreadLocalRandom tlr = ThreadLocalRandom.current();
    SwingUtilities.invokeLater(() -> {
      Supplier<String> text = () -> {
        char[] randomText = "Gt}F}\u00803u~f\u007F\u0081".toCharArray();
        for (int i = 0; i < randomText.length; i++) {
          randomText[i] -= tlr.nextInt(26);
        }
        return new String(randomText);
      };
      showWithSwing(text.get());
    });
  }
```

Leaking is easy!

Any lambda executed in some ExecutorService

CASE OF: **RETURN 4**

- This example shows a bug
- Leaking ThreadLocal is **easy**
- Leaking ThreadLocal is **bad**
- **You will do it, try not to**

TAKEAWAY

- Validate assumptions
- Bytecode is easier
- **JVM** doesn't think out-of-the-box
- Nothing beats experience

SOURCES OF AWESOMENESS

- **Vanilla # Java** by Peter Lawrey

  vanillajava.blogspot.com

- **Puzzles** by Wouter Coekaerts

  wouter.coekaerts.be/puzzles

- **Java Specialists' Newsletter** by Dr. Heinz Kabutz

  javaspecialists.eu/archive/archive.jsp

ZEROTURNAROUND