Item 64
git_comments:

git_commits:

1. **summary:** LUCENE-6879: Add missing null checks for parameters
   message: LUCENE-6879: Add missing null checks for parameters git-svn-id: https://svn.apache.org/repos/asf/lucene/dev/trunk@1713098 13f79535-47bb-0310-9956-ffa450edef68

github_issues:

github_issues_comments:

github_pulls:

github_pulls_comments:

github_pulls_reviews:

jira_issues:

1. **summary:** Allow to define custom CharTokenizer using Java 8 Lambdas/Method references
   description: As a followup from LUCENE-6874, I thought about how to generate custom CharTokenizers wthout subclassing. I had this quite often and I was a bit annoyed, that you had to create a subclass every time. This issue is using the pattern like ThreadLocal or many collection methods in Java 8: You have the (abstract) base class and you define a factory method named {{fromXxxPredicate}} (like {{ThreadLocal.withInitial(() -> value}}). {code:java} public static CharTokenizer fromTokenCharPredicate(java.util.function.IntPredicate predicate) {code} This would allow to define a new CharTokenizer with a single line statement using any predicate: {code:java} // long variant with lambda: Tokenizer tok = CharTokenizer.fromTokenCharPredicate(c -> !UCharacter.isUWhiteSpace(c)); // method reference for separator char predicate + normalization by uppercasing: Tokenizer tok = CharTokenizer.fromSeparatorCharPredicate(UCharacter::isUWhiteSpace, Character::toUpperCase); // method reference to custom function: private boolean myTestFunction(int c) { return (cracy condition); } Tokenizer tok = CharTokenizer.fromTokenCharPredicate(this::myTestFunction); {code} I know this would not help Solr users that want to define the Tokenizer in a config file, but for real Lucene users this Java 8-like way would be easy and elegant to use. It is fast as hell, as it is just a reference to a method and Java 8 is optimized for that. The inverted factories {{fromSeparatorCharPredicate()}} are provided to allow quick definition without lambdas using method references. In lots of cases, like WhitespaceTokenizer, predicates are on the separator chars ({{isWhitespace(int)}}, so using the 2nd set of factories you can define them without the counter-intuitive negation. Internally it just uses {{Predicate#negate()}}. The factories also allow to give the normalization function, e.g. to Lowercase, you may just give {{Character::toLowerCase}} as {{IntUnaryOperator}} reference.
2. **summary:** Allow to define custom CharTokenizer using Java 8 Lambdas/Method references
   description: As a followup from LUCENE-6874, I thought about how to generate custom CharTokenizers wthout subclassing. I had this quite often and I was a bit annoyed, that you had to create a subclass every time. This issue is using the pattern like ThreadLocal or many collection methods in Java 8: You have the (abstract) base class and you define a factory method named {{fromXxxPredicate}} (like {{ThreadLocal.withInitial(() -> value}}). {code:java} public static CharTokenizer fromTokenCharPredicate(java.util.function.IntPredicate predicate) {code} This would allow to define a new CharTokenizer with a single line statement using any predicate: {code:java} // long variant with lambda: Tokenizer tok = CharTokenizer.fromTokenCharPredicate(c -> !UCharacter.isUWhiteSpace(c)); // method reference for separator char predicate + normalization by uppercasing: Tokenizer tok = CharTokenizer.fromSeparatorCharPredicate(UCharacter::isUWhiteSpace, Character::toUpperCase); // method reference to custom function: private boolean myTestFunction(int c) { return (cracy condition); } Tokenizer tok = CharTokenizer.fromTokenCharPredicate(this::myTestFunction); {code} I know this would not help Solr users that want to define the Tokenizer in a config file, but for real Lucene users this Java 8-like way would be easy and elegant to use. It is fast as hell, as it is just a reference to a method and Java 8 is optimized for that. The inverted factories {{fromSeparatorCharPredicate()}} are provided to allow quick definition without lambdas using method references. In lots of cases, like

WhitespaceTokenizer, predicates are on the separator chars ({{isWhitespace(int)}}, so using the 2nd set of factories you can define them without the counter-intuitive negation. Internally it just uses {{Predicate#negate()}}. The factories also allow to give the normalization function, e.g. to Lowercase, you may just give {{Character::toLowerCase}} as {{IntUnaryOperator}} reference.

3. **summary:** Allow to define custom CharTokenizer using Java 8 Lambdas/Method references
   description: As a followup from LUCENE-6874, I thought about how to generate custom CharTokenizers wthout subclassing. I had this quite often and I was a bit annoyed, that you had to create a subclass every time. This issue is using the pattern like ThreadLocal or many collection methods in Java 8: You have the (abstract) base class and you define a factory method named {{fromXxxPredicate}} (like {{ThreadLocal.withInitial(() -> value)}}). {code:java} public static CharTokenizer fromTokenCharPredicate(java.util.function.IntPredicate predicate) {code} This would allow to define a new CharTokenizer with a single line statement using any predicate: {code:java} // long variant with lambda: Tokenizer tok = CharTokenizer.fromTokenCharPredicate(c -> !UCharacter.isUWhiteSpace(c)); // method reference for separator char predicate + normalization by uppercasing: Tokenizer tok = CharTokenizer.fromSeparatorCharPredicate(UCharacter::isUWhiteSpace, Character::toUpperCase); // method reference to custom function: private boolean myTestFunction(int c) { return (cracy condition); } Tokenizer tok = CharTokenizer.fromTokenCharPredicate(this::myTestFunction); {code} I know this would not help Solr users that want to define the Tokenizer in a config file, but for real Lucene users this Java 8-like way would be easy and elegant to use. It is fast as hell, as it is just a reference to a method and Java 8 is optimized for that. The inverted factories {{fromSeparatorCharPredicate()}} are provided to allow quick definition without lambdas using method references. In lots of cases, like WhitespaceTokenizer, predicates are on the separator chars ({{isWhitespace(int)}}, so using the 2nd set of factories you can define them without the counter-intuitive negation. Internally it just uses {{Predicate#negate()}}. The factories also allow to give the normalization function, e.g. to Lowercase, you may just give {{Character::toLowerCase}} as {{IntUnaryOperator}} reference.
   label: code-design
4. **summary:** Allow to define custom CharTokenizer using Java 8 Lambdas/Method references
   description: As a followup from LUCENE-6874, I thought about how to generate custom CharTokenizers wthout subclassing. I had this quite often and I was a bit annoyed, that you had to create a subclass every time. This issue is using the pattern like ThreadLocal or many collection methods in Java 8: You have the (abstract) base class and you define a factory method named {{fromXxxPredicate}} (like {{ThreadLocal.withInitial(() -> value)}}). {code:java} public static CharTokenizer fromTokenCharPredicate(java.util.function.IntPredicate predicate) {code} This would allow to define a new CharTokenizer with a single line statement using any predicate: {code:java} // long variant with lambda: Tokenizer tok = CharTokenizer.fromTokenCharPredicate(c -> !UCharacter.isUWhiteSpace(c)); // method reference for separator char predicate + normalization by uppercasing: Tokenizer tok = CharTokenizer.fromSeparatorCharPredicate(UCharacter::isUWhiteSpace, Character::toUpperCase); // method reference to custom function: private boolean myTestFunction(int c) { return (cracy condition); } Tokenizer tok = CharTokenizer.fromTokenCharPredicate(this::myTestFunction); {code} I know this would not help Solr users that want to define the Tokenizer in a config file, but for real Lucene users this Java 8-like way would be easy and elegant to use. It is fast as hell, as it is just a reference to a method and Java 8 is optimized for that. The inverted factories {{fromSeparatorCharPredicate()}} are provided to allow quick definition without lambdas using method references. In lots of cases, like WhitespaceTokenizer, predicates are on the separator chars ({{isWhitespace(int)}}, so using the 2nd set of factories you can define them without the counter-intuitive negation. Internally it just uses {{Predicate#negate()}}. The factories also allow to give the normalization function, e.g. to Lowercase, you may just give {{Character::toLowerCase}} as {{IntUnaryOperator}} reference.
   label: code-design
5. **summary:** Allow to define custom CharTokenizer using Java 8 Lambdas/Method references
   description: As a followup from LUCENE-6874, I thought about how to generate custom CharTokenizers wthout subclassing. I had this quite often and I was a bit annoyed, that you had to create a subclass every time. This issue is using the pattern like ThreadLocal or many collection methods in Java 8: You have the (abstract) base class and you define a factory method named {{fromXxxPredicate}} (like {{ThreadLocal.withInitial(() -> value)}}). {code:java} public static CharTokenizer fromTokenCharPredicate(java.util.function.IntPredicate predicate) {code} This would allow to define a new CharTokenizer with a single line statement using any predicate: {code:java} // long variant with lambda: Tokenizer tok = CharTokenizer.fromTokenCharPredicate(c -> !UCharacter.isUWhiteSpace(c)); // method reference for separator char predicate + normalization by uppercasing: Tokenizer tok = CharTokenizer.fromSeparatorCharPredicate(UCharacter::isUWhiteSpace, Character::toUpperCase); //

method reference to custom function: private boolean myTestFunction(int c) { return (cracy condition); } Tokenizer tok = CharTokenizer.fromTokenCharPredicate(this::myTestFunction); {code} I know this would not help Solr users that want to define the Tokenizer in a config file, but for real Lucene users this Java 8-like way would be easy and elegant to use. It is fast as hell, as it is just a reference to a method and Java 8 is optimized for that. The inverted factories {{fromSeparatorCharPredicate()}} are provided to allow quick definition without lambdas using method references. In lots of cases, like WhitespaceTokenizer, predicates are on the separator chars ({{isWhitespace(int)}}, so using the 2nd set of factories you can define them without the counter-intuitive negation. Internally it just uses {{Predicate#negate()}}. The factories also allow to give the normalization function, e.g. to Lowercase, you may just give {{Character::toLowerCase}} as {{IntUnaryOperator}} reference.

6. **summary:** Allow to define custom CharTokenizer using Java 8 Lambdas/Method references
   description: As a followup from LUCENE-6874, I thought about how to generate custom CharTokenizers wthout subclassing. I had this quite often and I was a bit annoyed, that you had to create a subclass every time. This issue is using the pattern like ThreadLocal or many collection methods in Java 8: You have the (abstract) base class and you define a factory method named {{fromXxxPredicate}} (like {{ThreadLocal.withInitial(() -> value}}). {code:java} public static CharTokenizer fromTokenCharPredicate(java.util.function.IntPredicate predicate) {code} This would allow to define a new CharTokenizer with a single line statement using any predicate: {code:java} // long variant with lambda: Tokenizer tok = CharTokenizer.fromTokenCharPredicate(c -> !UCharacter.isUWhiteSpace(c)); // method reference for separator char predicate + normalization by uppercasing: Tokenizer tok = CharTokenizer.fromSeparatorCharPredicate(UCharacter::isUWhiteSpace, Character::toUpperCase); // method reference to custom function: private boolean myTestFunction(int c) { return (cracy condition); } Tokenizer tok = CharTokenizer.fromTokenCharPredicate(this::myTestFunction); {code} I know this would not help Solr users that want to define the Tokenizer in a config file, but for real Lucene users this Java 8-like way would be easy and elegant to use. It is fast as hell, as it is just a reference to a method and Java 8 is optimized for that. The inverted factories {{fromSeparatorCharPredicate()}} are provided to allow quick definition without lambdas using method references. In lots of cases, like WhitespaceTokenizer, predicates are on the separator chars ({{isWhitespace(int)}}, so using the 2nd set of factories you can define them without the counter-intuitive negation. Internally it just uses {{Predicate#negate()}}. The factories also allow to give the normalization function, e.g. to Lowercase, you may just give {{Character::toLowerCase}} as {{IntUnaryOperator}} reference.
7. **summary:** Allow to define custom CharTokenizer using Java 8 Lambdas/Method references
   description: As a followup from LUCENE-6874, I thought about how to generate custom CharTokenizers wthout subclassing. I had this quite often and I was a bit annoyed, that you had to create a subclass every time. This issue is using the pattern like ThreadLocal or many collection methods in Java 8: You have the (abstract) base class and you define a factory method named {{fromXxxPredicate}} (like {{ThreadLocal.withInitial(() -> value}}). {code:java} public static CharTokenizer fromTokenCharPredicate(java.util.function.IntPredicate predicate) {code} This would allow to define a new CharTokenizer with a single line statement using any predicate: {code:java} // long variant with lambda: Tokenizer tok = CharTokenizer.fromTokenCharPredicate(c -> !UCharacter.isUWhiteSpace(c)); // method reference for separator char predicate + normalization by uppercasing: Tokenizer tok = CharTokenizer.fromSeparatorCharPredicate(UCharacter::isUWhiteSpace, Character::toUpperCase); // method reference to custom function: private boolean myTestFunction(int c) { return (cracy condition); } Tokenizer tok = CharTokenizer.fromTokenCharPredicate(this::myTestFunction); {code} I know this would not help Solr users that want to define the Tokenizer in a config file, but for real Lucene users this Java 8-like way would be easy and elegant to use. It is fast as hell, as it is just a reference to a method and Java 8 is optimized for that. The inverted factories {{fromSeparatorCharPredicate()}} are provided to allow quick definition without lambdas using method references. In lots of cases, like WhitespaceTokenizer, predicates are on the separator chars ({{isWhitespace(int)}}, so using the 2nd set of factories you can define them without the counter-intuitive negation. Internally it just uses {{Predicate#negate()}}. The factories also allow to give the normalization function, e.g. to Lowercase, you may just give {{Character::toLowerCase}} as {{IntUnaryOperator}} reference.
   label: documentation
8. **summary:** Allow to define custom CharTokenizer using Java 8 Lambdas/Method references
   description: As a followup from LUCENE-6874, I thought about how to generate custom CharTokenizers wthout subclassing. I had this quite often and I was a bit annoyed, that you had to create a subclass every time. This issue is using the pattern like ThreadLocal or many collection methods in Java 8: You have the (abstract) base class and you define a factory method named {{fromXxxPredicate}} (like {{ThreadLocal.withInitial(() -> value}}). {code:java} public static CharTokenizer

fromTokenCharPredicate(java.util.function.IntPredicate predicate) {code} This would allow to define a new CharTokenizer with a single line statement using any predicate: {code:java} // long variant with lambda: Tokenizer tok = CharTokenizer.fromTokenCharPredicate(c -> !UCharacter.isUWhiteSpace(c)); // method reference for separator char predicate + normalization by uppercasing: Tokenizer tok = CharTokenizer.fromSeparatorCharPredicate(UCharacter::isUWhiteSpace, Character::toUpperCase); // method reference to custom function: private boolean myTestFunction(int c) { return (cracy condition); } Tokenizer tok = CharTokenizer.fromTokenCharPredicate(this::myTestFunction); {code} I know this would not help Solr users that want to define the Tokenizer in a config file, but for real Lucene users this Java 8-like way would be easy and elegant to use. It is fast as hell, as it is just a reference to a method and Java 8 is optimized for that. The inverted factories {{fromSeparatorCharPredicate()}} are provided to allow quick definition without lambdas using method references. In lots of cases, like WhitespaceTokenizer, predicates are on the separator chars ({{isWhitespace(int)}}, so using the 2nd set of factories you can define them without the counter-intuitive negation. Internally it just uses {{Predicate#negate()}}. The factories also allow to give the normalization function, e.g. to Lowercase, you may just give {{Character::toLowerCase}} as {{IntUnaryOperator}} reference.

9. **summary:** Allow to define custom CharTokenizer using Java 8 Lambdas/Method references
   description: As a followup from LUCENE-6874, I thought about how to generate custom CharTokenizers wthout subclassing. I had this quite often and I was a bit annoyed, that you had to create a subclass every time. This issue is using the pattern like ThreadLocal or many collection methods in Java 8: You have the (abstract) base class and you define a factory method named {{fromXxxPredicate}} (like {{ThreadLocal.withInitial(() -> value}}). {code:java} public static CharTokenizer fromTokenCharPredicate(java.util.function.IntPredicate predicate) {code} This would allow to define a new CharTokenizer with a single line statement using any predicate: {code:java} // long variant with lambda: Tokenizer tok = CharTokenizer.fromTokenCharPredicate(c -> !UCharacter.isUWhiteSpace(c)); // method reference for separator char predicate + normalization by uppercasing: Tokenizer tok = CharTokenizer.fromSeparatorCharPredicate(UCharacter::isUWhiteSpace, Character::toUpperCase); // method reference to custom function: private boolean myTestFunction(int c) { return (cracy condition); } Tokenizer tok = CharTokenizer.fromTokenCharPredicate(this::myTestFunction); {code} I know this would not help Solr users that want to define the Tokenizer in a config file, but for real Lucene users this Java 8-like way would be easy and elegant to use. It is fast as hell, as it is just a reference to a method and Java 8 is optimized for that. The inverted factories {{fromSeparatorCharPredicate()}} are provided to allow quick definition without lambdas using method references. In lots of cases, like WhitespaceTokenizer, predicates are on the separator chars ({{isWhitespace(int)}}, so using the 2nd set of factories you can define them without the counter-intuitive negation. Internally it just uses {{Predicate#negate()}}. The factories also allow to give the normalization function, e.g. to Lowercase, you may just give {{Character::toLowerCase}} as {{IntUnaryOperator}} reference.
10. **summary:** Allow to define custom CharTokenizer using Java 8 Lambdas/Method references
    description: As a followup from LUCENE-6874, I thought about how to generate custom CharTokenizers wthout subclassing. I had this quite often and I was a bit annoyed, that you had to create a subclass every time. This issue is using the pattern like ThreadLocal or many collection methods in Java 8: You have the (abstract) base class and you define a factory method named {{fromXxxPredicate}} (like {{ThreadLocal.withInitial(() -> value}}). {code:java} public static CharTokenizer fromTokenCharPredicate(java.util.function.IntPredicate predicate) {code} This would allow to define a new CharTokenizer with a single line statement using any predicate: {code:java} // long variant with lambda: Tokenizer tok = CharTokenizer.fromTokenCharPredicate(c -> !UCharacter.isUWhiteSpace(c)); // method reference for separator char predicate + normalization by uppercasing: Tokenizer tok = CharTokenizer.fromSeparatorCharPredicate(UCharacter::isUWhiteSpace, Character::toUpperCase); // method reference to custom function: private boolean myTestFunction(int c) { return (cracy condition); } Tokenizer tok = CharTokenizer.fromTokenCharPredicate(this::myTestFunction); {code} I know this would not help Solr users that want to define the Tokenizer in a config file, but for real Lucene users this Java 8-like way would be easy and elegant to use. It is fast as hell, as it is just a reference to a method and Java 8 is optimized for that. The inverted factories {{fromSeparatorCharPredicate()}} are provided to allow quick definition without lambdas using method references. In lots of cases, like WhitespaceTokenizer, predicates are on the separator chars ({{isWhitespace(int)}}, so using the 2nd set of factories you can define them without the counter-intuitive negation. Internally it just uses {{Predicate#negate()}}. The factories also allow to give the normalization function, e.g. to Lowercase, you may just give {{Character::toLowerCase}} as {{IntUnaryOperator}} reference.
11. **summary:** Allow to define custom CharTokenizer using Java 8 Lambdas/Method references

description: As a followup from LUCENE-6874, I thought about how to generate custom CharTokenizers wthout subclassing. I had this quite often and I was a bit annoyed, that you had to create a subclass every time. This issue is using the pattern like ThreadLocal or many collection methods in Java 8: You have the (abstract) base class and you define a factory method named {{fromXxxPredicate}} (like {{ThreadLocal.withInitial(() -> value}}). {code:java} public static CharTokenizer fromTokenCharPredicate(java.util.function.IntPredicate predicate) {code} This would allow to define a new CharTokenizer with a single line statement using any predicate: {code:java} // long variant with lambda: Tokenizer tok = CharTokenizer.fromTokenCharPredicate(c -> !UCharacter.isUWhiteSpace(c)); // method reference for separator char predicate + normalization by uppercasing: Tokenizer tok = CharTokenizer.fromSeparatorCharPredicate(UCharacter::isUWhiteSpace, Character::toUpperCase); // method reference to custom function: private boolean myTestFunction(int c) { return (cracy condition); } Tokenizer tok = CharTokenizer.fromTokenCharPredicate(this::myTestFunction); {code} I know this would not help Solr users that want to define the Tokenizer in a config file, but for real Lucene users this Java 8-like way would be easy and elegant to use. It is fast as hell, as it is just a reference to a method and Java 8 is optimized for that. The inverted factories {{fromSeparatorCharPredicate()}} are provided to allow quick definition without lambdas using method references. In lots of cases, like WhitespaceTokenizer, predicates are on the separator chars ({{isWhitespace(int)}}, so using the 2nd set of factories you can define them without the counter-intuitive negation. Internally it just uses {{Predicate#negate()}}. The factories also allow to give the normalization function, e.g. to Lowercase, you may just give {{Character::toLowerCase}} as {{IntUnaryOperator}} reference.

12. **summary:** Allow to define custom CharTokenizer using Java 8 Lambdas/Method references
description: As a followup from LUCENE-6874, I thought about how to generate custom CharTokenizers wthout subclassing. I had this quite often and I was a bit annoyed, that you had to create a subclass every time. This issue is using the pattern like ThreadLocal or many collection methods in Java 8: You have the (abstract) base class and you define a factory method named {{fromXxxPredicate}} (like {{ThreadLocal.withInitial(() -> value}}). {code:java} public static CharTokenizer fromTokenCharPredicate(java.util.function.IntPredicate predicate) {code} This would allow to define a new CharTokenizer with a single line statement using any predicate: {code:java} // long variant with lambda: Tokenizer tok = CharTokenizer.fromTokenCharPredicate(c -> !UCharacter.isUWhiteSpace(c)); // method reference for separator char predicate + normalization by uppercasing: Tokenizer tok = CharTokenizer.fromSeparatorCharPredicate(UCharacter::isUWhiteSpace, Character::toUpperCase); // method reference to custom function: private boolean myTestFunction(int c) { return (cracy condition); } Tokenizer tok = CharTokenizer.fromTokenCharPredicate(this::myTestFunction); {code} I know this would not help Solr users that want to define the Tokenizer in a config file, but for real Lucene users this Java 8-like way would be easy and elegant to use. It is fast as hell, as it is just a reference to a method and Java 8 is optimized for that. The inverted factories {{fromSeparatorCharPredicate()}} are provided to allow quick definition without lambdas using method references. In lots of cases, like WhitespaceTokenizer, predicates are on the separator chars ({{isWhitespace(int)}}, so using the 2nd set of factories you can define them without the counter-intuitive negation. Internally it just uses {{Predicate#negate()}}. The factories also allow to give the normalization function, e.g. to Lowercase, you may just give {{Character::toLowerCase}} as {{IntUnaryOperator}} reference.
13. **summary:** Allow to define custom CharTokenizer using Java 8 Lambdas/Method references
description: As a followup from LUCENE-6874, I thought about how to generate custom CharTokenizers wthout subclassing. I had this quite often and I was a bit annoyed, that you had to create a subclass every time. This issue is using the pattern like ThreadLocal or many collection methods in Java 8: You have the (abstract) base class and you define a factory method named {{fromXxxPredicate}} (like {{ThreadLocal.withInitial(() -> value}}). {code:java} public static CharTokenizer fromTokenCharPredicate(java.util.function.IntPredicate predicate) {code} This would allow to define a new CharTokenizer with a single line statement using any predicate: {code:java} // long variant with lambda: Tokenizer tok = CharTokenizer.fromTokenCharPredicate(c -> !UCharacter.isUWhiteSpace(c)); // method reference for separator char predicate + normalization by uppercasing: Tokenizer tok = CharTokenizer.fromSeparatorCharPredicate(UCharacter::isUWhiteSpace, Character::toUpperCase); // method reference to custom function: private boolean myTestFunction(int c) { return (cracy condition); } Tokenizer tok = CharTokenizer.fromTokenCharPredicate(this::myTestFunction); {code} I know this would not help Solr users that want to define the Tokenizer in a config file, but for real Lucene users this Java 8-like way would be easy and elegant to use. It is fast as hell, as it is just a reference to a method and Java 8 is optimized for that. The inverted factories {{fromSeparatorCharPredicate()}} are provided to allow quick definition without lambdas using method references. In lots of cases, like WhitespaceTokenizer, predicates are on the separator chars ({{isWhitespace(int)}}, so using the 2nd set

of factories you can define them without the counter-intuitive negation. Internally it just uses {{Predicate#negate()}}. The factories also allow to give the normalization function, e.g. to Lowercase, you may just give {{Character::toLowerCase}} as {{IntUnaryOperator}} reference.
label: code-design

14. **summary:** Allow to define custom CharTokenizer using Java 8 Lambdas/Method references
description: As a followup from LUCENE-6874, I thought about how to generate custom CharTokenizers wthout subclassing. I had this quite often and I was a bit annoyed, that you had to create a subclass every time. This issue is using the pattern like ThreadLocal or many collection methods in Java 8: You have the (abstract) base class and you define a factory method named {{fromXxxPredicate}} (like {{ThreadLocal.withInitial(() -> value}}). {code:java} public static CharTokenizer fromTokenCharPredicate(java.util.function.IntPredicate predicate) {code} This would allow to define a new CharTokenizer with a single line statement using any predicate: {code:java} // long variant with lambda: Tokenizer tok = CharTokenizer.fromTokenCharPredicate(c -> !UCharacter.isUWhiteSpace(c)); // method reference for separator char predicate + normalization by uppercasing: Tokenizer tok = CharTokenizer.fromSeparatorCharPredicate(UCharacter::isUWhiteSpace, Character::toUpperCase); // method reference to custom function: private boolean myTestFunction(int c) { return (cracy condition); } Tokenizer tok = CharTokenizer.fromTokenCharPredicate(this::myTestFunction); {code} I know this would not help Solr users that want to define the Tokenizer in a config file, but for real Lucene users this Java 8-like way would be easy and elegant to use. It is fast as hell, as it is just a reference to a method and Java 8 is optimized for that. The inverted factories {{fromSeparatorCharPredicate()}} are provided to allow quick definition without lambdas using method references. In lots of cases, like WhitespaceTokenizer, predicates are on the separator chars ({{isWhitespace(int)}}, so using the 2nd set of factories you can define them without the counter-intuitive negation. Internally it just uses {{Predicate#negate()}}. The factories also allow to give the normalization function, e.g. to Lowercase, you may just give {{Character::toLowerCase}} as {{IntUnaryOperator}} reference.

jira_issues_comments:

1. Patch using Java 8s new functional APIs. Very cool and simple to define a new Tokenizer. I only don't like that CharTokenizer is in oal.analysis.util package. Maybe we should move the factories to a separate class in the oal.analysis,core pkg. The patch also has some tests showing how you would use them.
2. I think the tests are nice examples and like the separator vs tokenchar methods (it can be hard to think about opposites). Good improvement for java 8 on trunk.
3. **body:** We can improve the Javadocs by adding the examples. I just wanted to quickly write the patch to demonstrate how it could look like. We can also discuss about method names. The pattern follows convention used for all functional interfaces in Java 8 (method naming), but we can make it more readable. I am open to suggestions. In Lucene trunk we can also remove all the separate implementations like LetterTokenizer and just allow them to be produced by factories. This would be a slight break, but we could still provide the Solr/CustomAnalyzer factories as usual. The Tokenizer for ICU in LUCENE-6874 could also be a one-liner just provided by the Solr factory, but no actual instance :-) We could also provide a one-for all Solr/CustomAnalyzer factory using a Enum of predicate/normalizer functions to be choosen by string parameter.
label: code-design
4. **body:** Pretty cool, Uwe! bq. It is fast as hell I always thought hell was about slow and endless suffering? :)
label: code-design
5. bq. I always thought hell was about slow and endless suffering? Ähm, yes :-) But this video tells you different: https://www.youtube.com/watch?v=Uqa8MFSXZHM If you need to burn fat, fast as hell: http://www.amazon.com/ULTIMATE-CUTS-SECRETS-English-Edition-ebook/dp/B00HMQS8TA
6. +1 Nice Uwe.
7. **body:** New patch with improved Javadocs. Will commit this soon.
label: documentation
8. Commit 1712682 from [~thetaphi] in branch 'dev/trunk' [https://svn.apache.org/r1712682] LUCENE-6879: Allow to define custom CharTokenizer instances without subclassing using Java 8 lambdas or method references
9. Thanks for review!
10. Just FYI: I did some quick microbenchmark like this: {code:java} // init & warmup String text = "Tokenizer(Test)FooBar"; String[] result = new String[] { "tokenizer", "test", "foobar" }; final Tokenizer tokenizer1 = CharTokenizer.fromTokenCharPredicate(Character::isLetter, Character::toLowerCase); for

(int i = 0; i < 10000; i++) { tokenizer1.setReader(new StringReader(text)); assertTokenStreamContents(tokenizer1, result); } final Tokenizer tokenizer2 = new LowerCaseTokenizer(); for (int i = 0; i < 10000; i++) { tokenizer2.setReader(new StringReader(text)); assertTokenStreamContents(tokenizer2, result); } // speed test long [] lens1 = new long[100], lens2 = new long[100]; for (int j = 0; j < 100; j++) { System.out.println("Run: " + j); long start1 = System.currentTimeMillis(); for (int i = 0; i < 1000000; i++) { tokenizer1.setReader(new StringReader(text)); assertTokenStreamContents(tokenizer1, result); } lens1[j] = System.currentTimeMillis() - start1; long start2 = System.currentTimeMillis(); for (int i = 0; i < 1000000; i++) { tokenizer2.setReader(new StringReader(text)); assertTokenStreamContents(tokenizer2, result); } lens2[j] = System.currentTimeMillis() - start2; } System.out.println("Time Lambda: " + Arrays.stream(lens1).summaryStatistics()); System.out.println("Time Old: " + Arrays.stream(lens2).summaryStatistics()); {code} I was not able to find any speed difference after warmup: - Time Lambda: LongSummaryStatistics{count=100, sum=58267, min=562, average=582.670000, max=871} - Time Old: LongSummaryStatistics{count=100, sum=61489, min=600, average=614.890000, max=721}

11. Commit 1713098 from [~thetaphi] in branch 'dev/trunk' [https://svn.apache.org/r1713098] LUCENE-6879: Add missing null checks for parameters
12. Commit 1713099 from [~thetaphi] in branch 'dev/branches/branch_5x' [https://svn.apache.org/r1713099] Merge additional null check from LUCENE-6879