beetee17 / NUS_Y1S2 Public

Code Issues Pull requests Actions Projects Wiki

main NUS_Y1S2 / CS2030S / Past Year Practices / PE2 / pe2-2021s2 / cs2030s / fp / Try (solution).java / Go to file ...

Jump to

beetee17 Update 8 Apr Latest commit 0b0d5e6 1 minute ago History

1 contributor

177 lines (151 sloc) 4.79 KB Raw Blame

```java
package cs2030s.fp;

public abstract class Try<T> {
  /**
   * Return a failed Try with a given Throwable.
   *
   * @param <T> The type of the value contained in this Throwable.
   * @param a The Throwable to initialize this failure with.
   *
   * @return The new failure.
   */
  public static <T> Try<T> failure(Throwable a) {
    return new Failure<T>(a);
  }

  /**
   * Return a success Try with the given value.
   *
   * @param <T> The type of the value contained in this Throwable.
   * @param value The Throwable to initialize this success with.
   *
   */
  public static <T> Try<T> success(T value) {
    return new Success<T>(value);
  }

  /**
   * Return a Try after running the producer.  If the producer produces
   * without error, return a success with the value produced.  Else,
   * return a failure with the error/exception.
   *
   * @param <T> The type of the value contained in this Throwable.
   * @param producer The producer to initialize this Try with.
   *
   * @return The new Try.
   */
  public static <T> Try<T> of(Producer<? extends T> producer) {
    try {
      return success(producer.produce());
    } catch (Throwable e) {
      return failure(e);
    }
  }

  public abstract T get() throws Throwable;

  public abstract <R> Try<R> map(Transformer<? super T, ? extends R> mapper);

  public abstract <R> Try<R> flatMap(Transformer<? super T, ? extends Try<? e

  /**
   * If the calling Try is a failure, return itself after running the consume
   * on the throwabale; otherwise if the calling Try is a success, just retur
   * it self.  If the consumer fails the throwable is replaced with the new
   * error.
   *
   * @param consumer The consumer to consume the throwable with.
   *
   * @return The new Try
   */
  public abstract Try<T> onFailure(Consumer<? super Throwable> consumer);

  /**
   * If the calling Try is a failure, return a success Try after running the
   * given Transformer on the value; if the calling Try is a success, return
   * itself without running the Transformer.
   * If the transformer fails the throwable is replaced with the new
   * error.
   *
   * @param transformer The transformer to transform the value with.
   *
   * @return The new Try
   */
  public abstract Try<T> recover(Transformer<? super Throwable, ? extends T>

  private static class Success<T> extends Try<T> {
    T value;

    Success(T value) {
      this.value = value;
    }

    public String toString() {
      return "Success: " + String.valueOf(value);
    }

    public <R> Try<R> map(Transformer<? super T, ? extends R> mapper) {
      try {
        return success(mapper.transform(value));
      } catch (Throwable e) {
        return failure(e);
      }
    }

    public <R> Try<R> flatMap(Transformer<? super T, ? extends Try<? extends
      @SuppressWarnings("unchecked")
      Try<R> t =  (Try<R>) mapper.transform(value);
      return t;
    }

    public T get() throws Throwable {
      return value;
    }

    public Try<T> onFailure(Consumer<? super Throwable> consumer) {
      return this;
    }

    public Try<T> recover(Transformer<? super Throwable, ? extends T> transfo
      return this;
    }

    public boolean equals(Object o) {
      if (o != null && o instanceof Success) {
        Success<?> success = (Success<?>) o;
        if (success.value != null) {
          return success.value.equals(this.value);
        }
        return this.value == null;
      }
      return false;
    }
  }

  private static class Failure<T> extends Try<T> {
    Throwable throwable;

    Failure(Throwable value) {
      this.throwable = value;
    }

    public String toString() {
      return "Failure: " + String.valueOf(throwable);
    }

    public <R> Failure<R> map(Transformer<? super T, ? extends R> mapper) {
      @SuppressWarnings("unchecked")
      Failure<R> t =  (Failure<R>) this;
      return t;
    }

    public <R> Failure<R> flatMap(Transformer<? super T, ? extends Try<? exte
      @SuppressWarnings("unchecked")
      Failure<R> t =  (Failure<R>) this;
      return t;
    }

    public T get() throws Throwable {
      throw throwable;
    }

    public Try<T> onFailure(Consumer<? super Throwable> consumer) {
      try {
        consumer.consume(this.throwable);
        return this;
      } catch (Throwable t) {
        return failure(t);
      }
    }

    public Try<T> recover(Transformer<? super Throwable, ? extends T> transfo
      try {
        return success(transformer.transform(this.throwable));
      } catch (Throwable t) {
        return failure(t);
      }
    }

    public boolean equals(Object o) {
      if (o != null && o instanceof Failure) {
        Failure<?> that = (Failure<?>) o;
        return that.throwable.toString().equals(this.throwable.toString());
      }
      return false;
    }
  }
}
```