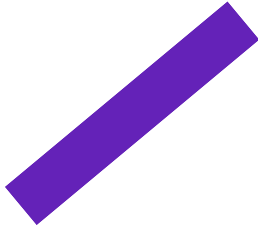




"Error Handling in Ruby"

Huy Du – Senior Software Engineer
@ Ascenda Loyalty

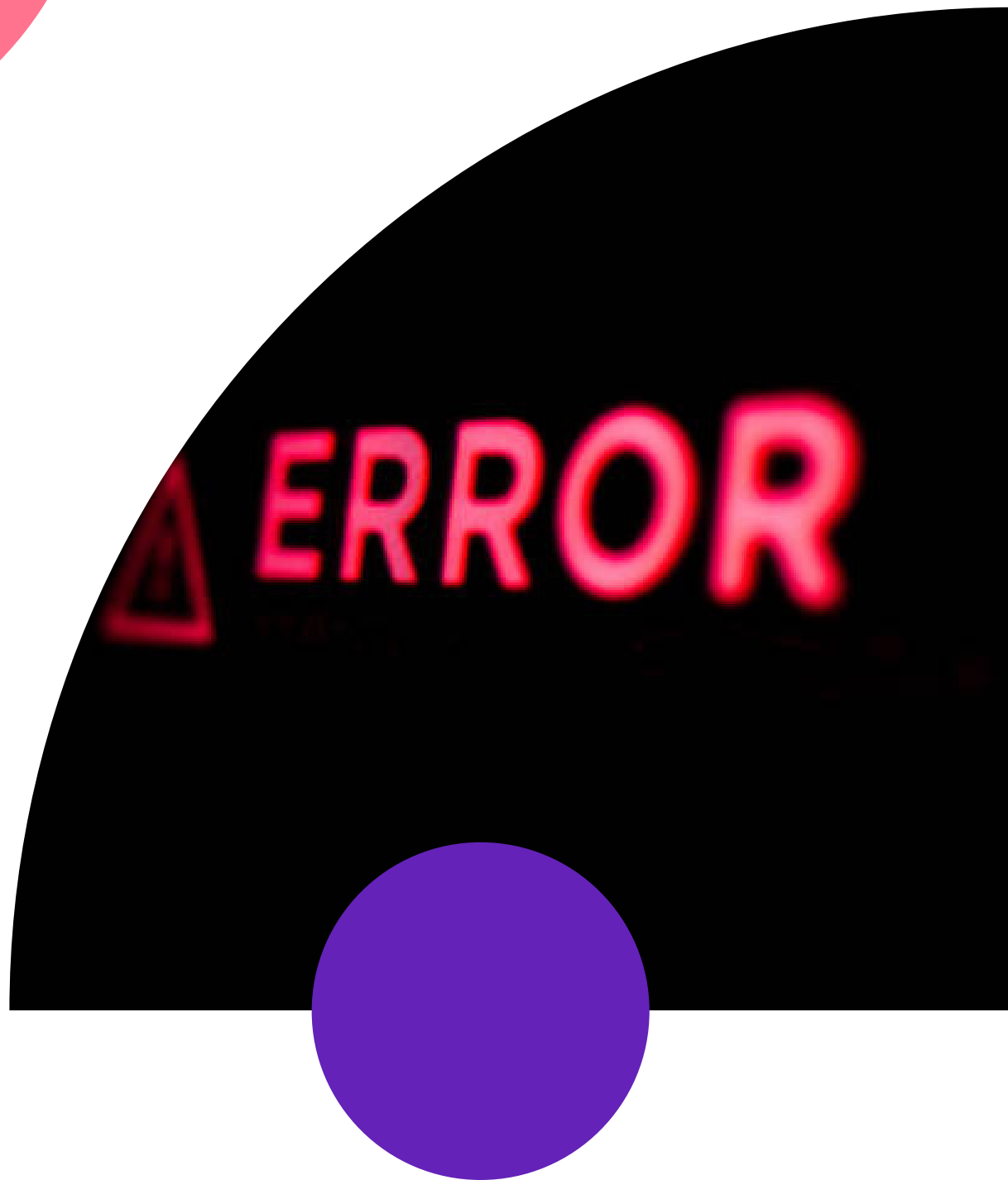




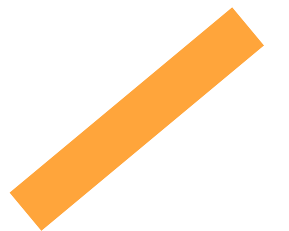
“The correct way of error
handling will stop your
lifespan for a couple of
seconds”

Huy Du






- Error
- Error Handling
- Error Handling in Ruby





What is an error?

Programming error means an error which occurs during the development or encoding of a computer program, software, or application, which would, when in operation, result in a malfunction or incorrect operation of a computer system.



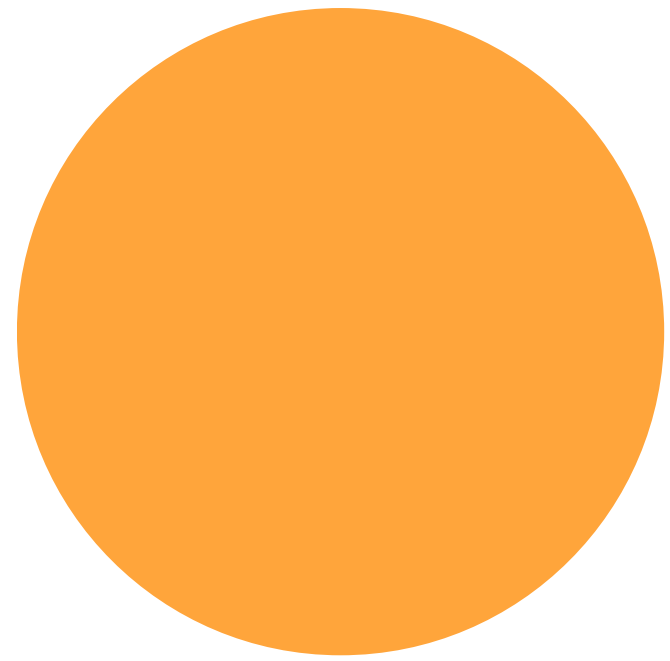


Error

- Mistake
- Misconception
- Misunderstanding

Of Developers

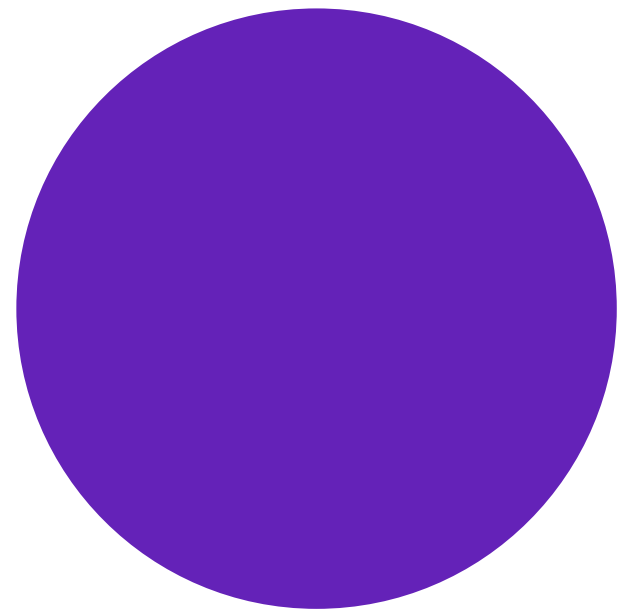




Error from a
programming
language-defined
problem

Error from a
developer-defined
problem

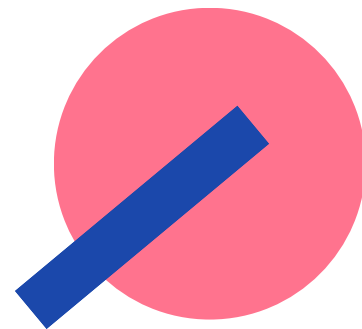




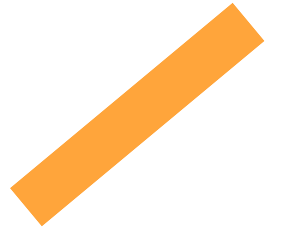
Error Handling

Detect and respond to errors that occur during the execution of software.

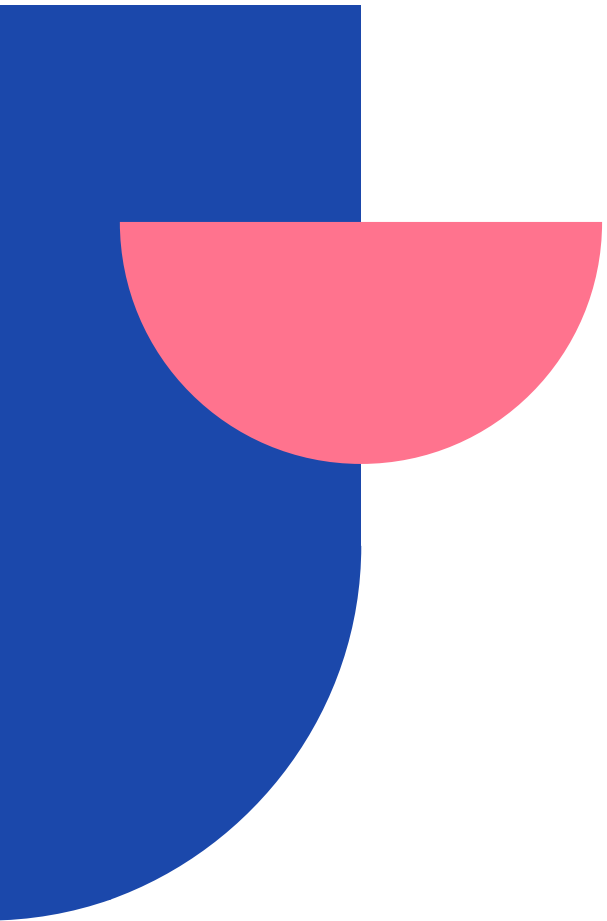
Prevent and minimize the impact of errors during the execution of software



How to error handling?



1. Define the problem
2. Define the error of the problem
3. Raise or log error
4. Handle error
5. Test it!



Example

As a User, I want to have a shopping cart that add/update/remove so that I can checkout my item

As a User, I want to checkout my shopping cart so that I can receive my item

Define the problem

Check for completeness: ensure documentation is complete and includes all necessary information

Review for consistency: does the documentation consistent between every parts of it?

Identify an ambiguous: unclear requirement or ambiguity can lead to misunderstanding, misconception

Look for edge cases: any scenarios that may not have been considered?

Review for feasibility: does it feasible and can be implemented?

Define the problem

Is there any other feature we need to support for Shopping Cart?

Are all user stories consistent? Is there any conflict between those user stories?

What if the item is not available anymore?

What if the selected item is out of stock?

Can we able to implement full-feature shopping cart like Shopee?

Define the error

After identified the problem, try to classify it into 2 main concepts

Validation

- What will be required?
- What will be forbidden?
- What will be permitted?

Perform operation

- What will be caused it to stop?
- Will any potentially unexpected things happen?

```
# Validation
#
if order.cash_payment?
  if validate_payment_intent.failed?
    order.fail!("Stripe payment intent validation failed")
    cancel_payment_intent!

    return error_from(validate_payment_intent)
  end

  cash_transaction.update(status: :on_hold)
end

if checkout_validation.failed?
  order.fail!(checkout_validation.errors.flat_message)
  cancel_payment_intent! if order.cash_payment?

  return error_from(checkout_validation)
end

if order.applied_promo_code? && redeem_promo_code.failed?
  order.fail!("Could not redeem promo code")
  cancel_payment_intent! if order.cash_payment?

  return error_from(redeem_promo_code)
end
```

```
# Operation
#
if order.points_payment? && redemption.failed?
  order.fail!("Redemption failed: #{redemption.errors.flat_message}")
  cancel_payment_intent! if order.cash_payment?

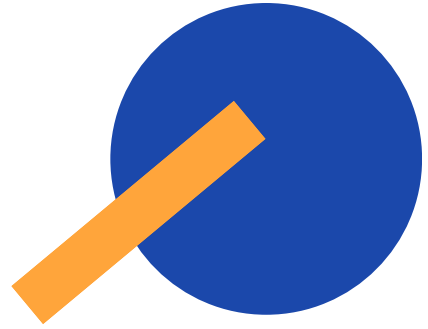
  return error_from(redemption)
end

if order.cash_payment?
  if capture_payment_intent.failed?
    order.fail!("Stripe payment intent capture failed")
    cash_transaction.update(
      status: :failed,
      error_message: capture_payment_intent.errors.flat_message
    )

    schedule_refund! if order.points_payment?

    stripe_errors = error(:stripe_error, capture_payment_intent.errors.flat_message)
    ReportError.(stripe_errors)

    return stripe_errors
  end
  cash_transaction.update(status: :succeeded)
end
```



Raise or log error

The error should clearly reflect what happened. The concise error message is very useful for debugging and resolving problem as fast as possible

After the error is raised, the normal control flow is interrupted and the exceptional control flow begins.

```
class Fulfillment < ApplicationService
  class OrderItemNotFound < StandardError; end
  result :credit

  option :order

  def call
    raise OrderItemNotFound, "Order #{order.id}" if crypto_order_item.blank?

    fulfillment = Api::Int::FulfillService.transfer_crypto(**crypto_params)

    if fulfillment.successful?
      crypto_order_item.process!(fulfillment.credit.id)

      success(credit: fulfillment.credit)
    else
      crypto_order_item.fail!(fulfillment.errors.flat_message)

      error_from(fulfillment)
    end
  end
end
```

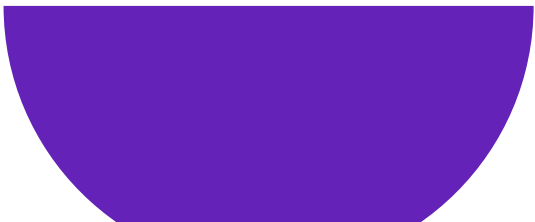



Handle error

Who will handle it?

- Developer
- System

How to handle it?

- Special Return Value
 - Try-Catch
 - Type System
- 

Test it!



```
context "with every successful checking" do
  it do
    expect { service.() }.to succeed_with(:order)
      .and change { order.status }.from("pending").to("fulfilling")
      .and change { order.cash_transactions.count }.from(0).to(1)
      .and change { cart.cart_items.count }.from(1).to(0)
      .and have_enqueued_job(SendCheckoutResultEventJob).with(**success_event_payload)
    end
  end

  context "when order is attached with promo code" do
    let(:promo_code_feature) { true }
    let(:points_paid_1) { 250 }
    let(:points_paid_2) { 250 }
    let(:amount) { 1000 }

    it do
      expect { service.() }.to succeed_with(:order)
        .and change { order.cash_transactions.count }.from(0).to(1)
        .and change { order.reload.status }.from("pending").to("fulfilling")
        .and change { order.promo_code_usage.status }.from("attached").to("redeemed")
        .and change { order.promo_code.number_of_uses }.from(0).to(1)
        .and change { cart.cart_items.count }.from(1).to(0)
        .and not_change { order.promo_code.quantity }
        .and have_enqueued_job(SendCheckoutResultEventJob).with(**success_event_payload)
      end
    end
  end
end
```

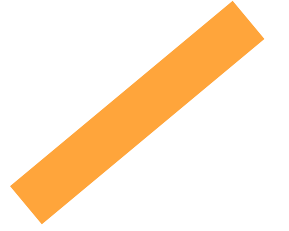
```
context "with validate payment intent failure" do
  let(:validate_payment_intent_result) do
    mock_service_error_with("NN0002", "External Error", 500)
  end

  it do
    expect { service.() }
      .to fail_with_error("NN0002", "External Error", 500)
      .and change { OrderItem.distinct.pluck(:status) }.from(["pending"]).to(["failed"])
      .and have_enqueued_job(SendCheckoutResultEventJob).with(**failure_event_payload)
  end

  context "when order is attached with promo code" do
    let(:promo_code_feature) { true }

    it do
      expect { service.() }
        .to fail_with_error("NN0002", "External Error", 500)
        .and change { OrderItem.distinct.pluck(:status) }.from(["pending"]).to(["failed"])
        .and not_change { order.promo_code_usage.status }.from("attached")
        .and have_enqueued_job(SendCheckoutResultEventJob).with(**failure_event_payload)
      end
    end
  end
end
```


How to error handling?



1. Define the problem
2. Define the error of the problem
3. Raise or log error
4. Handle error
5. Test it!



Error Handling in Ruby



Exception represents an error or unexpected condition that occurs in the Ruby program.

Exception objects carry information about:

- Type (exception's class name)
- Descriptive string (error message)
- Backtrace information

Built-in subclasses of Exception 2.5.1 ✓

- NoMemoryError
- ScriptError
- SecurityError
- SignalError
- SystemExit
- SystemStackError

- StandardError

- RuntimeError

– > default for "rescue"

– > default for "raise"

```
# frozen_string_literal: true
```

```
class ApplicationService
```

```
  Result = Struct.new(:successful?, :failed?, :result, keyword_init: true)
```

```
  Errors = Struct.new(:successful?, :failed?, :errors, keyword_init: true)
```

```
  def self.call(*arguments, **keywords)
```

```
    | new(*arguments, **keywords).call
```

```
  end
```

```
  def call
```

```
    | raise NotImplementedError
```

```
  end
```

```
  private
```

```
  def error(identifier, message)
```

```
    | errors = ApplicationError.from_config(identifier, message)
```

```
    | Errors.new(successful?: false, failed?: true, errors: errors)
```

```
  end
```

```
  def success(result)
```

```
    | Result.new(successful?: true, failed?: false, result: result)
```

```
  end
```

```
end
```



```
# frozen_string_literal: true
```

```
class Checkout < ApplicationService
  class PaymentNotFound < StandardError; end

  def initialize(order)
    @order = order
  end

  def call
    return error(:validation_error, "Order expired") if order.expired?
    return error(:validation_error, "Product is out of stock") if order.product.outstock?
    return error(:external_error, "Payment method unavailable") if order.payment.unavailable?

    make_payment!
    order.mark_as_confirmed!
    order.schedule_fulfillment!

    success
  end


  private

  def make_payment!
    payment = order.get_payment
    raise PaymentNotFound, "Order #{order.id}" if payment.blank?

    begin
      payment.submit!
    rescue ConnectionError
      do_something
    end
  end
end
```



Recap

- Effective error handling is a critical part of writing high-quality Ruby code.
 - Properly handling errors can help prevent crashes, improve application stability, and provide a better user experience.
 - Don't overlook the importance of error handling in your code.
- 



Thank you!