Enums

Rust’s killer feature

Vineet Naik
Rust Bangalore, 9th March 2024

About me

- Rust novice
  - ~5 months experience writing rust
  - Limited to personal projects
- High level / dynamically typed / functional languages
  - Clojure, Python, Java, Erlang, Javascript

Outline

- Part 1: Introduction to Enums in Rust
- Part 2: Practical examples
  - From dupenukem, a personal project for file deduplication
  - Comparison with equivalent code in other languages
- Won’t cover
  - Low level details e.g. memory allocations and related optimizations, performance etc.

Part 1: Rust enum type

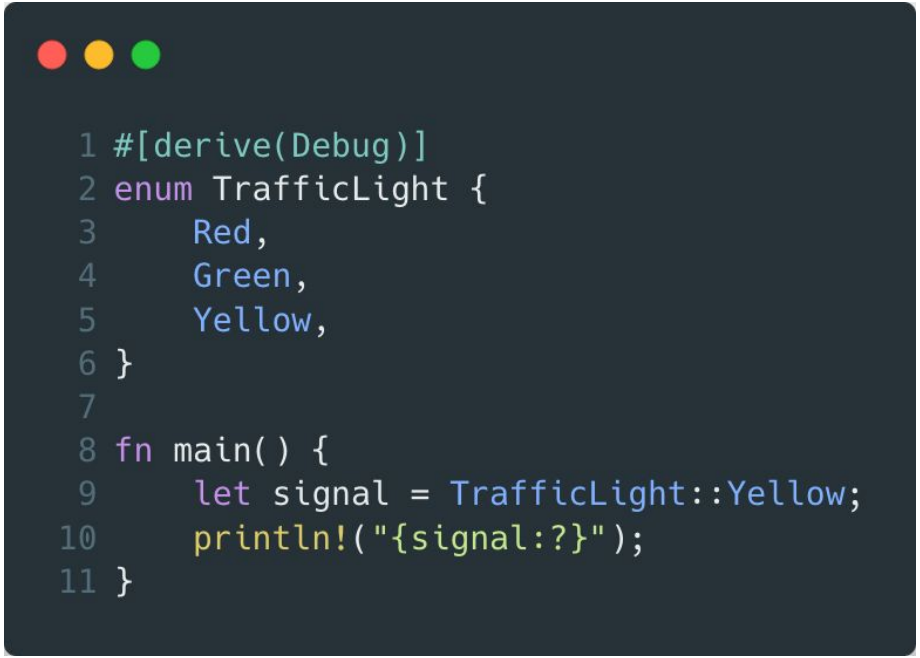
What are enums?

Types with a finite set of variants

Example: red, green, yellow colours in traffic lights

Sum type

A traffic light can be any of the three colours but at a time it can only be one colour



```rust
#[derive(Debug)]
enum TrafficLight {
    Red,
    Green,
    Yellow,
}

fn main() {
    let signal = TrafficLight::Yellow;
    println!("{signal:?}");
}
```

Pattern matching

- `match` construct for value comparison

```rust
#[derive(Debug)]
enum TrafficLight {
    Red,
    Green,
    Yellow,
}

fn action(signal: &TrafficLight) -> &str {
    match signal {
        TrafficLight::Red => "wait",
        TrafficLight::Green => "proceed",
        TrafficLight::Yellow => "slow down",
    }
}
```

Pattern matching

- `match` construct for value comparison
- Matches are exhaustive
  - All possibilities need to be covered for the code to compile

Enums in Python and Java

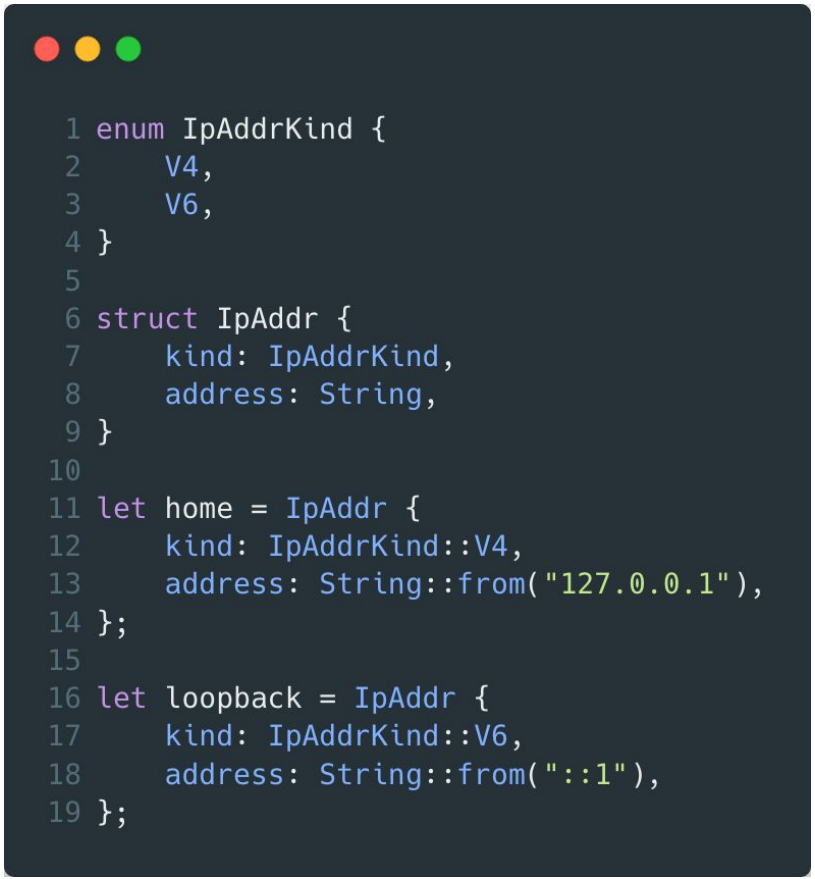
```python
from enum import Enum

class TrafficLight(Enum):
    RED = 1
    GREEN = 2
    YELLOW = 3

def action(signal):
    if signal == TrafficLight.RED:
        return "wait"
    elif signal == TrafficLight.GREEN:
        return "go"
    elif signal == TrafficLight.YELLOW:
        return "slow down"

if __name__ == '__main__':
    red = TrafficLight.RED
    green = TrafficLight["GREEN"]
    yellow = TrafficLight(3)

    print(action(red))
```

```java
enum TrafficLight {
    RED,
    GREEN,
    YELLOW;

    public String action() {
        return switch(this) {
        case RED -> "wait";
        case GREEN -> "go";
        case YELLOW -> "slow down";
        };
    }
}

public class TrafficLights {
    public static void main(String[] args) {
        TrafficLight t = TrafficLight.RED;
        System.out.println(t.action());
    }
}
```

Naive use of enums

- As a “kind” field inside a struct
- And other fields representing actual data

Not much different from enums in Java and Python



```
enum IpAddrKind {
    V4,
    V6,
}

struct IpAddr {
    kind: IpAddrKind,
    address: String,
}

let home = IpAddr {
    kind: IpAddrKind::V4,
    address: String::from("127.0.0.1"),
};

let loopback = IpAddr {
    kind: IpAddrKind::V6,
    address: String::from("::1"),
};
```

More than just a type to represent a “kind of” field

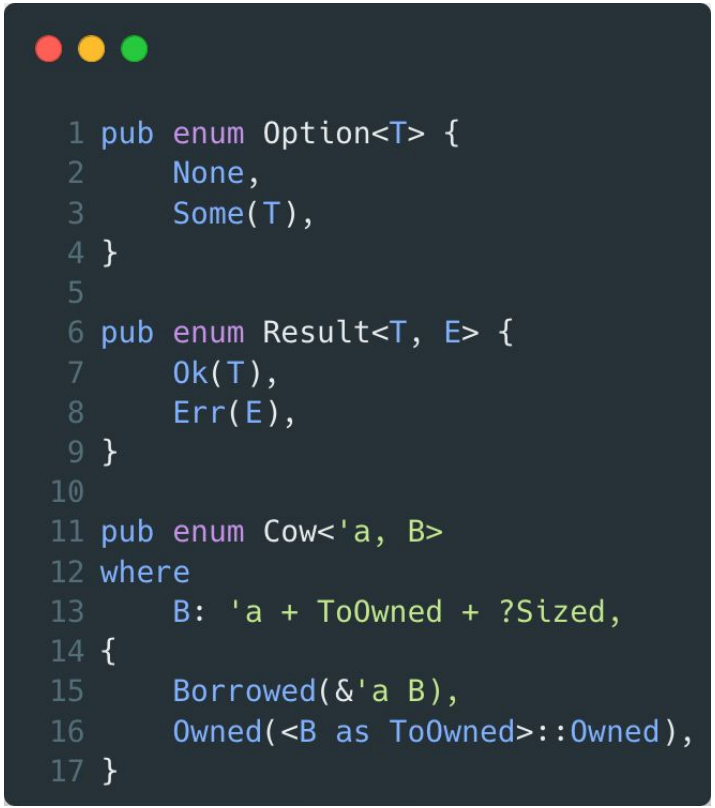
- Optionally “data-bearing”[1]
- Each variant can be attached data
- Data could be string, number, multiple values (tuple), struct
- Names of the variants becomes constructors
- Methods can be implemented

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

[1] The term “data-bearing” borrowed from Rust after the honeymoon by Bryan Cantrill

Enums in the rust stdlib

- Option
- Result
- Cow (clone-on-write)
- etc?



```rust
pub enum Option<T> {
    None,
    Some(T),
}

pub enum Result<T, E> {
    Ok(T),
    Err(E),
}

pub enum Cow<'a, B>
where
    B: 'a + ToOwned + ?Sized,
{
    Borrowed(&'a B),
    Owned(<B as ToOwned>::Owned),
}
```

Part 2: Practical examples

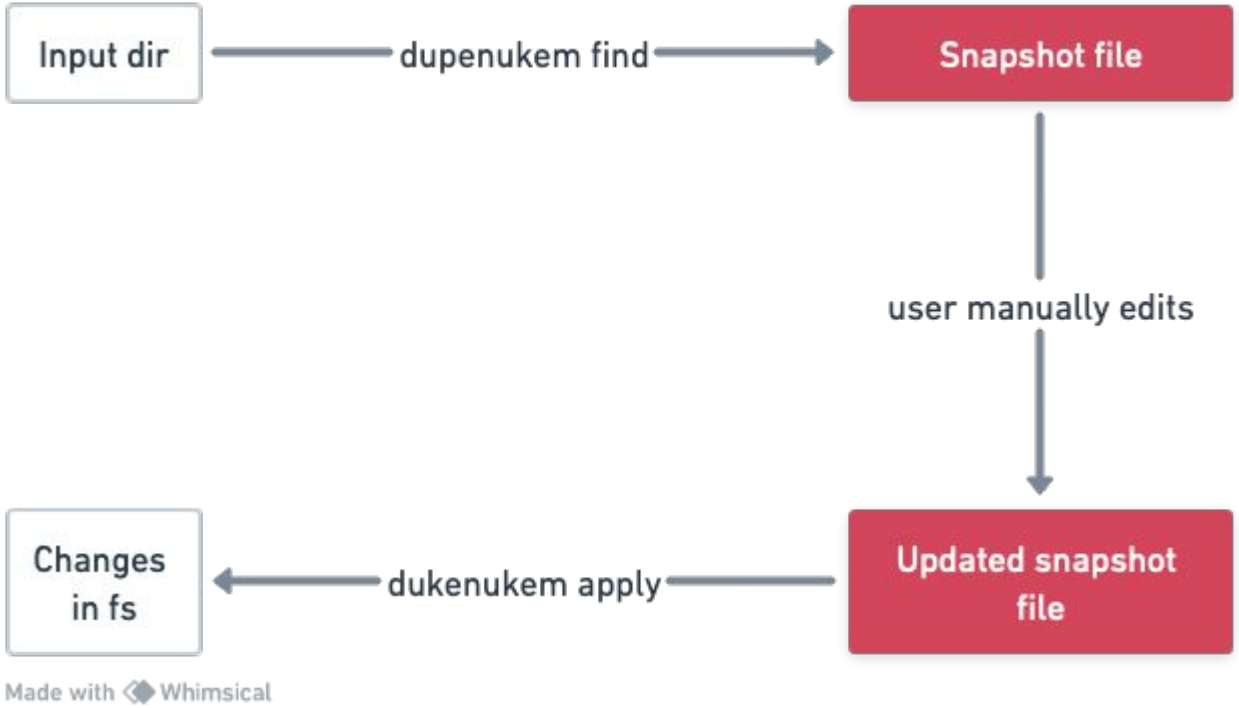
Criteria for evaluating the solutions

- Flexibility/Extensibility
  - How easy is it to adapt to changing requirements
- Robustness
  - How easy is it to make changes without regression
- Conciseness
  - How easy is it to avoid unnecessary complexity and verbosity

Non-criteria for this talk

- Performance
- Memory efficiency

Dupenukem - A file deduplication utility



Use case 1: Generate/Parse custom file format

```
$ dupenukem find ~/dpnktest | tee ~/dpnktest_snapshot.txt
#! Root Directory: /Users/vineet/dpnktest
#! Generated at: Tue, 16 Jan 2024 12:00:05 +0530

[13062064944137093030]
keep cat/2.txt
keep foo/2.txt

[10098984572146910405]
keep foo/1.txt
keep bar/1.txt

# Reference:
# keep <target> = keep the target path as it is
# delete <target> = delete the target path
# symlink <target> [-> <src>] = Replace target with a symlink
# .       If 'src' is specified, it can either be an absolute or
# .       relative (to 'target'). Else one of the duplicates marked
# .       as 'keep' will be considered. If 'src' is not specified,
# .       a relative symlink will be created.
#
# This section is a comment and will be ignored by the tool
```

Use case 1: Generate/Parse custom file format

- File = ordered collection of lines
  - `Vec<Line>`
- Our custom file format has different kinds of lines
- But, a vector is a homogenous collection in rust
- One workaround is to use traits
  - `Vec<Box<dyn Line>>` where `Line` is a trait
  - Over engineered for such a fixed use case
- Enums to the rescue!

Use case 1: Rust

```rust
#[derive(Debug, Eq, PartialEq)]
enum Line {
    Comment(String),
    MetaData {
        key: String,
        val: String,
    },
    Checksum(String),
    PathInfo {
        path: String,
        op: String,
        delim: Option<String>,
        extra: Option<String>,
    },
    Blank,
}
```

```rust
impl Line {
    fn encode(&self) -> String {
        match self {
            Self::Comment(comment) => format!("# {}", comment),
            Self::MetaData { key, val } => format!("#! {}: {}", key, val),
            Self::Checksum(hash) => format!("[{}]", hash),
            Self::PathInfo {
                path,
                op,
                delim,
                extra,
            } => {
                match &extra {
                    // @NOTE: Here we're not handling the case where
                    // delim is None. At this point it's not clear
                    // whether that would be a good idea.
                    Some(x) => format!("{} {} {} {}", op, path, delim.as_ref().unwrap(), x),
                    None => format!("{} {}", op, path),
                }
            }
            Self::Blank => String::from(""),
        }
    }
```

Use case 1: Rust

Flexibility/Extensibility	
Robustness	
Conciseness	

```rust
#[derive(Debug, Eq, PartialEq)]
enum Line {
    Comment(String),
    MetaData {
        key: String,
        val: String,
    },
    Checksum(String),
    PathInfo {
        path: String,
        op: String,
        delim: Option<String>,
        extra: Option<String>,
    },
    Blank,
}
```

```rust
impl Line {
    fn encode(&self) -> String {
        match self {
            Self::Comment(comment) => format!("# {}", comment),
            Self::MetaData { key, val } => format!("#! {}: {}", key, val),
            Self::Checksum(hash) => format!("[{}]", hash),
            Self::PathInfo {
                path,
                op,
                delim,
                extra,
            } => {
                match &extra {
                    // @NOTE: Here we're not handling the case where
                    // delim is None. At this point it's not clear
                    // whether that would be a good idea.
                    Some(x) => format!("{} {} {} {}", op, path, delim.as_ref().unwrap(), x),
                    None => format!("{} {}", op, path),
                }
            }
            Self::Blank => String::from(""),
        }
    }
}
```

Use case 1: Rust

Flexibility/Extensibility	✅
Robustness	
Conciseness	

```rust
#[derive(Debug, Eq, PartialEq)]
enum Line {
    Comment(String),
    MetaData {
        key: String,
        val: String,
    },
    Checksum(String),
    PathInfo {
        path: String,
        op: String,
        delim: Option<String>,
        extra: Option<String>,
    },
    Blank,
}
```

```rust
impl Line {
    fn encode(&self) -> String {
        match self {
            Self::Comment(comment) => format!("# {}", comment),
            Self::MetaData { key, val } => format!("#! {}: {}", key, val),
            Self::Checksum(hash) => format!("[{}]", hash),
            Self::PathInfo {
                path,
                op,
                delim,
                extra,
            } => {
                match &extra {
                    // @NOTE: Here we're not handling the case where
                    // delim is None. At this point it's not clear
                    // whether that would be a good idea.
                    Some(x) => format!("{} {} {} {}", op, path, delim.as_ref().unwrap(), x),
                    None => format!("{} {}", op, path),
                }
            }
            Self::Blank => String::from(""),
        }
    }
}
```

Use case 1: Rust

Flexibility/Extensibility	✅
Robustness	✅
Conciseness	

```rust
#[derive(Debug, Eq, PartialEq)]
enum Line {
    Comment(String),
    MetaData {
        key: String,
        val: String,
    },
    Checksum(String),
    PathInfo {
        path: String,
        op: String,
        delim: Option<String>,
        extra: Option<String>,
    },
    Blank,
}
```

```rust
impl Line {
    fn encode(&self) -> String {
        match self {
            Self::Comment(comment) => format!("# {}", comment),
            Self::MetaData { key, val } => format!("#! {}: {}", key, val),
            Self::Checksum(hash) => format!("[{}]", hash),
            Self::PathInfo {
                path,
                op,
                delim,
                extra,
            } => {
                match &extra {
                    // @NOTE: Here we're not handling the case where
                    // delim is None. At this point it's not clear
                    // whether that would be a good idea.
                    Some(x) => format!("{} {} {} {}", op, path, delim.as_ref().unwrap(), x),
                    None => format!("{} {}", op, path),
                }
            }
            Self::Blank => String::from(""),
        }
    }
}
```

Use case 1: Rust

Flexibility/Extensibility	✅
Robustness	✅
Conciseness	✅

```rust
#[derive(Debug, Eq, PartialEq)]
enum Line {
    Comment(String),
    MetaData {
        key: String,
        val: String,
    },
    Checksum(String),
    PathInfo {
        path: String,
        op: String,
        delim: Option<String>,
        extra: Option<String>,
    },
    Blank,
}
```

```rust
impl Line {
    fn encode(&self) -> String {
        match self {
            Self::Comment(comment) => format!("# {}", comment),
            Self::MetaData { key, val } => format!("#! {}: {}", key, val),
            Self::Checksum(hash) => format!("[{}]", hash),
            Self::PathInfo {
                path,
                op,
                delim,
                extra,
            } => {
                match &extra {
                    // @NOTE: Here we're not handling the case where
                    // delim is None. At this point it's not clear
                    // whether that would be a good idea.
                    Some(x) => format!("{} {} {} {}", op, path, delim.as_ref().unwrap(), x),
                    None => format!("{} {}", op, path),
                }
            }
            Self::Blank => String::from(""),
        }
    }
}
```

Equivalent code in other languages

Use case 1: Java

```java
interface ILine {
    String encode();
}

class Comment implements ILine {
    String value;

    public Comment(String value) {
        this.value = value;
    }

    public String encode() {
        return String.format("# %s", this.value);
    }
}

class Metadata implements ILine {
    String key;
    String val;

    public Metadata(String key, String val) {
        this.key = key;
        this.val = val;
    }

    public String encode() {
        return String.format("#! %s: %s", this.key, this.val);
    }
}
```

```java

class Checksum implements ILine { ... }

class PathInfo implements ILine { ... }

class Blank implements ILine { ... }

public class Lines {
    public static void main(String[] args) {
        Comment c = new Comment("this is a comment");
        Metadata m = new Metadata("foo", "bar");
        ArrayList<ILine> lines = new ArrayList<ILine>();
        lines.add(c);
        lines.add(m);
        for (ILine line: lines) {
            System.out.println(line.encode());
        }
    }
}
```

Use case 1: Java

```java
interface ILine {
    String encode();
}

class Comment implements ILine {
    String value;

    public Comment(String value) {
        this.value = value;
    }

    public String encode() {
        return String.format("# %s", this.value);
    }
}

class Metadata implements ILine {
    String key;
    String val;

    public Metadata(String key, String val) {
        this.key = key;
        this.val = val;
    }

    public String encode() {
        return String.format("#! %s: %s", this.key, this.val);
    }
}
```

Flexibility/Extensibility	✅
Robustness	✅
Conciseness	❌

```java

class Checksum implements ILine { ... }

class PathInfo implements ILine { ... }

class Blank implements ILine { ... }

public class Lines {
    public static void main(String[] args) {
        Comment c = new Comment("this is a comment");
        Metadata m = new Metadata("foo", "bar");
        ArrayList<ILine> lines = new ArrayList<ILine>();
        lines.add(c);
        lines.add(m);
        for (ILine line: lines) {
            System.out.println(line.encode());
        }
    }
}
```

Use case 1: Python

```python
def line_encode(line):
    if line["type"] == "comment":
        return "# {}".format(line["value"])
    elif line["type"] == "metadata":
        return "#! {}: {}".format(line["key"], line["val"])
    elif line["type"] == "checksum":
        return "[{}]".format(line["value"])
    elif line["type"] == "pathinfo":
        if line.get("extra"):
            return "{} {} {} {}".format(
                line["op"], line["path"], line["delim"], line["extra"]
            )
        else:
            "{} {}".format(line["op"], line["path"])
    elif line["type"] == "blank":
        return ""
    else:
        raise Exception("Invalid line")


line_encode({"type": "comment", "value": "this is a comment"})

line_encode({"type": "metadata", "key": "foo", "val": "bar"})
```

- Dynamically typed
- Heterogeneous collections
- Exceptions may happen at run time

Use case 1: Python

```python
def line_encode(line):
    if line["type"] == "comment":
        return "# {}".format(line["value"])
    elif line["type"] == "metadata":
        return "#! {}: {}".format(line["key"], line["val"])
    elif line["type"] == "checksum":
        return "[{}]".format(line["value"])
    elif line["type"] == "pathinfo":
        if line.get("extra"):
            return "{} {} {} {}".format(
                line["op"], line["path"], line["delim"], line["extra"]
            )
        else:
            "{} {}".format(line["op"], line["path"])
    elif line["type"] == "blank":
        return ""
    else:
        raise Exception("Invalid line")


line_encode({"type": "comment", "value": "this is a comment"})

line_encode({"type": "metadata", "key": "foo", "val": "bar"})
```

Flexibility/Extensibility	✅
Robustness	❌
Conciseness	✅

- Can we make it robust by using classes?
  - No
  - Can’t enforce homogeneous collections
  - Runtime exceptions
- Typed python?
  - May be but not sure
  - Only at the cost of conciseness

Use case 1: Clojure

```clojure
(defn line-encode
  [line]
  (case (:type line)
    :comment (format "# %s" (:value line))
    :metadata (format "#! %s: %s" (:key line) (:val line))
    :checksum (format "[%s]" (:value line))
    :path-info (let [{:keys [op path delim extra]} line]
                 (if extra
                   (format "%s %s %s %s" op path delim extra)
                   (format "%s %s" op path)))
    :blank ""))

(line-encode {:type :comment :value "this is a comment"})
;; "# this is a comment"

(line-encode {:type :metadata :key "foo" :val "bar"})
;; "#! foo: bar"

(line-encode {:type :unknown})
;; throws java.lang.IllegalArgumentException
```

- Dynamically typed
- Heterogeneous collections

Use case 1: Clojure

```clojure
(defn line-encode
  [line]
  (case (:type line)
    :comment (format "# %s" (:value line))
    :metadata (format "#! %s: %s" (:key line) (:val line))
    :checksum (format "[%s]" (:value line))
    :path-info (let [{:keys [op path delim extra]} line]
                 (if extra
                   (format "%s %s %s %s" op path delim extra)
                   (format "%s %s" op path)))
    :blank ""))

(line-encode {:type :comment :value "this is a comment"})
;; "# this is a comment"

(line-encode {:type :metadata :key "foo" :val "bar"})
;; "#! foo: bar"

(line-encode {:type :unknown})
;; throws java.lang.IllegalArgumentException
```

Flexibility/Extensibility	✅
Robustness	❌
Conciseness	✅

- Can we make it robust by using records and protocols?
  - No
  - Can’t enforce homogeneous collections
  - Runtime exceptions

Summary

	Rust	Java	Python	Clojure
Extensibility/Flexibility	✅	✅	✅	✅
Robustness	✅	✅	❌	❌
Conciseness	✅	❌	✅	✅

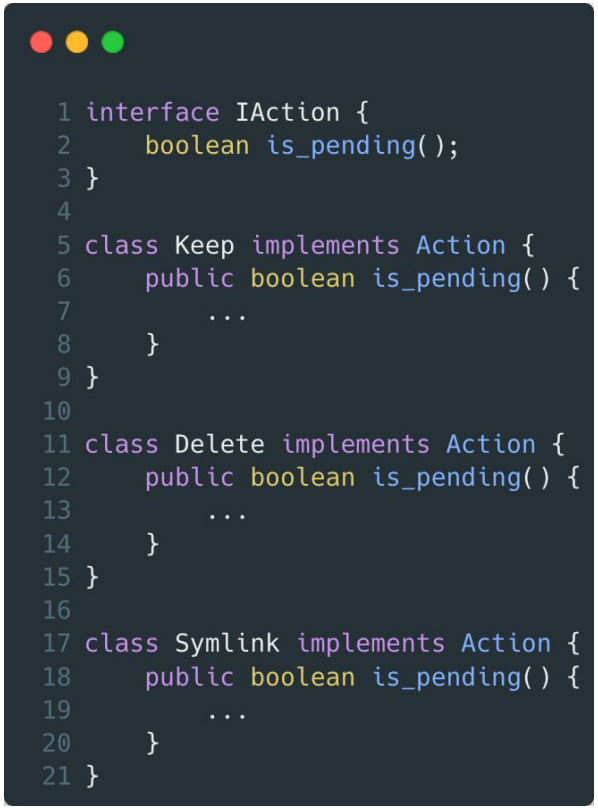
Use case 2: Different types of actions

```rust
#[derive(Debug)]
pub enum Action<'a> {
    Keep(&'a PathBuf),
    Symlink {
        path: &'a PathBuf,
        source: &'a PathBuf,
        is_explicit: bool,
        is_no_op: bool,
    },
    Delete {
        path: &'a PathBuf,
        is_no_op: bool,
    },
}
```

```rust
pub fn pending_actions<'a>(actions: &'a [Action], include_no_op: bool) -> Vec<&'a Action<'a>> {
    actions
        .iter()
        .filter(|action| match action {
            Action::Keep(_) => false,
            Action::Symlink {
                is_no_op,
                path: _,
                source: _,
                is_explicit: _,
            } => include_no_op || !is_no_op,
            Action::Delete { is_no_op, path: _ } => include_no_op || !is_no_op,
        })
        .collect::<Vec<&Action>>()
}
```

Use case 2: Java

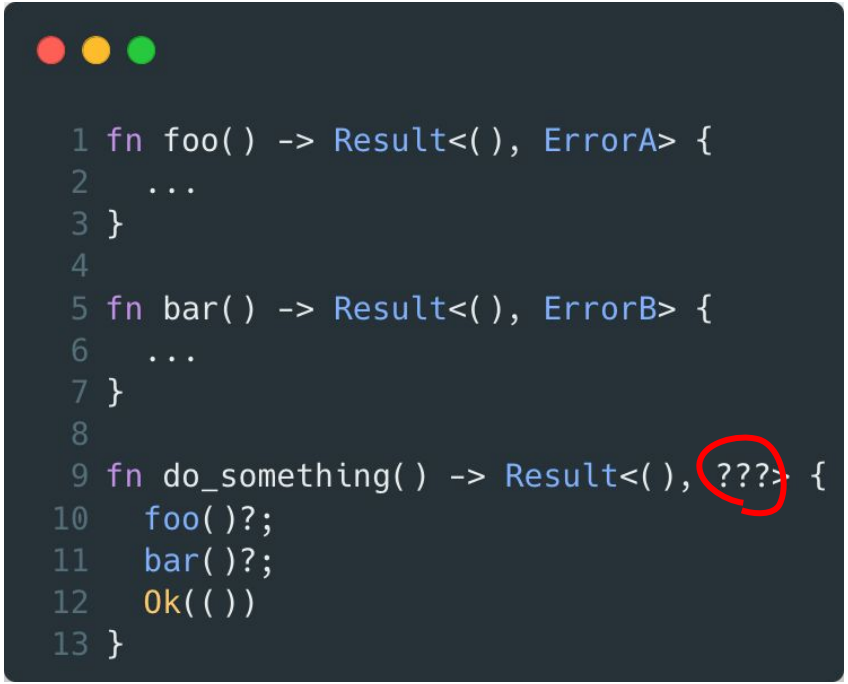
- Interfaces and classes that implement them
- The predicate fn for filtering is spread across multiple classes



```
interface IAction {
    boolean is_pending();
}

class Keep implements Action {
    public boolean is_pending() {
        ...
    }
}

class Delete implements Action {
    public boolean is_pending() {
        ...
    }
}

class Symlink implements Action {
    public boolean is_pending() {
        ...
    }
}
```

Use case 3: Error handling

Multiple error types can be wrapped inside a single enum



```
fn foo() -> Result<(), ErrorA> {
  ...
}

fn bar() -> Result<(), ErrorB> {
  ...
}

fn do_something() -> Result<(), ???> {
  foo()?;
  bar()?;
  Ok(())
}
```

Use case 3: Error handling

```rust
use crate::snapshot::validation;
use std::io;

#[derive(Debug)]
pub enum AppError {
    SnapshotParsing,
    SnapshotValidation(validation::Error),
    Cmd(String),
    Io(io::Error),
    Fs(String),
    ChecksumParsing,
}
```

```rust
fn take_backup(path: &Path, backup_dir: &Path, base_dir: &Path) -> Result<PathBuf, AppError> {
    // Find path relative to the rootdir
    let rel_path = path
        .strip_prefix(base_dir)
        .map_err(|_| AppError::Fs(String::from("Could not find path relative to the base dir")))?;
    let backup_path = backup_dir.join(rel_path);
    fs::create_dir_all(backup_path.parent().unwrap()).map_err(AppError::Io)?;
    fs::copy(path, &backup_path).map_err(AppError::Io)?;
    info!(
        "Backing up {} under {}",
        rel_path.display(),
        backup_dir.display()
    );
    Ok(backup_path)
}
```

Limitations of the enum type

- Extra memory allocation
- All variants need to be known (statically defined)
- ?What else?

Why I think enum is rust's killer feature?

- Rust code satisfies all the three criteria
- Trade offs
  - Dynamic languages => concise => quick prototyping
    - can result in brittle code
  - Static languages => compiler checks => high confidence
    - can get verbose
- Rust enums: quick prototyping with high confidence
- Enum-based solution seems like the Goldilocks[1] approach
  - It feels "just right"

[1]: Goldilocks Principle

Thank you