Types, Polymorphisms, & Composition

Nick Sumner
wsumner@sfu.ca
Why do we care about types?

- They have detractors
Why do we care about *types*?

- They have detractors
  - Many languages got by without them: Python, Ruby, JavaScript, ...
  - Some languages are pretty flexible about them: C
  - They may involve extra typing
  - *They limit what a program can do*
Why do we care about *types*?

- They have detractors
  - Many languages got by without them: Python, Ruby, JavaScript, ...
  - Some languages are pretty flexible about them: C
  - They may involve extra typing
  - *They limit what a program can do*

- But there are benefits
Why do we care about *types*?

- They have detractors
  - Many languages got by without them: Python, Ruby, JavaScript, ...
  - Some languages are pretty flexible about them: C
  - They may involve extra typing
  - *They limit what a program can do*

- But there are benefits
  - Fewer bugs
  - Easier readability
  - Better toolability
  - Many languages have incorporated them: Python, Ruby, JavaScript, ...
  - *They limit what a program can do*
Why do we care about *types*?

- They have detractors
  - Many languages got by without them: Python, Ruby, JavaScript, ...
  - Some languages are pretty flexible about them: C
  - They may involve extra typing
  - *They limit what a program can do*

- But there are benefits
  - Fewer bugs
  - Easier readability
  - Better toolability
  - Many languages have incorporated them: Python, Ruby, JavaScript, ...
  - *They limit what a program can do*

- To understand why the last point is good, let us consider what a type *is*
What are types?

- A type comprises
  - a set of values and
  - how those values may be used
What are types?

- A *type* comprises
  - a set of values and
  - how those values may be used
What are types?

- A type comprises
  - a set of values and
  - how those values may be used
What are types?

• A type comprises
  – a set of values and
  – how those values may be used

• By **limiting** the values/operations possible at a program point, we make it easier to **prove** a program correct, at least to a degree
What are types?

- A type comprises
  - a set of values and
  - how those values may be used

- By *limiting* the values/operations possible at a program point, we make it easier to *prove* a program correct, at least to a degree
  - Superficially this is obvious but maybe unconvincing.
    We shall explore stronger arguments & examples over the rest of the term.
What are types?

- A type comprises
  - a set of values and
  - how those values may be used

- By *limiting* the values/operations possible at a program point, we make it easier to *prove* a program correct, at least to a degree
  - Superficially this is obvious but maybe unconvincing. We shall explore stronger arguments & examples over the rest of the term.

- In a *statically typed* language, we can describe the set of values ahead of time, without running the code
What are types?

- A type comprises
  - a set of values and
  - how those values may be used

- By *limiting* the values/operations possible at a program point, we make it easier to *prove* a program correct, at least to a degree
  - Superficially this is obvious but maybe unconvincing. We shall explore stronger arguments & examples over the rest of the term.

- In a *statically typed* language, we can describe the set of values ahead of time, without running the code
  - This enables problems to be found in advance
  - It also enables tools to provide better assistance
Goals & trade offs

- Writing out types can be complex
Goals & trade offs

• Writing out types can be complex
  – There could be extra typing
  – Type inference helps significantly in modern Java, C++, C#, ...
Goals & trade offs

- Writing out types can be complex
  - There could be extra typing
  - Type inference helps significantly in modern Java, C++, C#, ...
  - Capturing all valid & only valid types can be tricky
  - Fair point. We will see more design trade offs for an engineer
Goals & trade offs

- Writing out types can be complex
  - There could be extra typing
  - Type inference helps significantly in modern Java, C++, C#, ...
  - Capturing all valid & only valid types can be tricky
  - Fair point. We will see more design trade offs for an engineer

- Expressing static types can be limiting
  - Only defining each function for a single type limits reuse & extensibility
Goals & trade offs

- Writing out types can be complex
  - There could be extra typing
  - Type inference helps significantly in modern Java, C++, C#, ...
  - Capturing all valid & only valid types can be tricky
  - Fair point. We will see more design trade offs for an engineer

- Expressing static types can be limiting
  - Only defining each function for a single type limits reuse & extensibility

```java
min(3, 5)
min("aardvark"'s, "easyvark"'s)
...
```
Goals & trade offs

● Writing out types can be complex
  – There could be extra typing
  – Type inference helps significantly in modern Java, C++, C#, ...
  – Capturing all valid & only valid types can be tricky
  – Fair point. We will see more design trade offs for an engineer

● Expressing static types can be limiting
  – Only defining each function for a single type limits reuse & extensibility

  \[
  \begin{align*}
  \text{min}(3, 5) \\
  \text{min}("aardvark"s, "easyvark"s) \\
  \ldots
  \end{align*}
  \]

  – One solution was through *polymorphism* – types comprising sets of types
Polymorphisms

- We have seen 2 forms, but at least 4 major forms are classic & common
Polymorphisms

- We have seen 2 forms, but at least 4 major forms are classic & common
  - Runtime polymorphism (subtyping & inheritance)
  - Parametric polymorphism (templates, generics, ...)

Polymorphisms

- We have seen 2 forms, but at least 4 major forms are classic & common
  - Runtime polymorphism (subtyping & inheritance)
  - Parametric polymorphism (templates, generics, ...)
  - Overloading
  - Coercion

There are more, but we won’t discuss them
Polymorphisms

- We have seen 2 forms, but at least 4 major forms are classic & common
  - Runtime polymorphism (subtyping & inheritance)
  - Parametric polymorphism (templates, generics, ...)
  - Overloading
  - Coercion

[Universal]

[Ad hoc]
Polymorphisms

- We have seen 2 forms, but at least 4 major forms are classic & common
  - Runtime polymorphism (subtyping & inheritance)
  - Parametric polymorphism (templates, generics, ...)
  - Overloading
  - Coercion

- Universal polymorphisms define types that can comprise an infinite number of other types with a common structure

Universal  Ad hoc
Polymorphisms

- We have seen 2 forms, but at least 4 major forms are classic & common:
  - Runtime polymorphism (subtyping & inheritance)
  - Parametric polymorphism (templates, generics, ...)
  - Overloading
  - Coercion

Universal polymorphisms define types that can comprise an infinite number of other types with a common structure.

```cpp
template<typename T>
T&
min(T& first, T& second) {
    return (first < second) ? first : second;
}
```
Polymorphisms

- We have seen 2 forms, but at least 4 major forms are classic & common:
  - Runtime polymorphism (subtyping & inheritance)
  - Parametric polymorphism (templates, generics, ...)
  - Overloading
  - Coercion

Universal polymorphisms define types that can comprise an infinite number of other types with a common structure.

```cpp
template<typename T>
T&
min(T& first, T& second) {
    return (first < second)
        ? first : second;
}
```

```java
public class Cat
    extends Comparable<Cat> {
    ...
    @Override
    boolean compareTo(Cat other)
    ...
}
```

```java
public static <T extends Comparable<T>>
T
min(T first, T second) {
    return (first.compareTo(second) < 0)
        ? first : second;
}
```

common structure
Polymorphisms

- We have seen 2 forms, but at least 4 major forms are classic & common
  - Runtime polymorphism (subtyping & inheritance)
  - Parametric polymorphism (templates, generics, ...)
  - Overloading
  - Coercion

- Universal polymorphisms define types that can comprise an infinite number of other types with a common structure
- Ad hoc polymorphisms define types that can comprise a finite set of explicitly specified types with even disparate structure
Polymorphism

We have seen 2 forms, but at least 4 major forms are classic & common:

- Runtime polymorphism (subtyping & inheritance)
- Parametric polymorphism (templates, generics, ...)
- Overloading
- Coercion

Universal polymorphisms define types that can comprise an infinite number of other types with a common structure.

Ad hoc polymorphisms define types that can comprise a finite set of explicitly specified types with even disparate structure.
Polymorphisms

- We have seen 2 forms, but at least 4 major forms are classic & common:
  - Runtime polymorphism (subtyping & inheritance)
  - Parametric polymorphism (templates, generics, ...)
  - Overloading
  - Coercion

- Universal polymorphisms define types that can comprise an infinite number of other types with a common structure

- Ad hoc polymorphisms define types that can comprise a finite set of explicitly specified types with even disparate structure

```
int add(int first, int second) {
    return first + second;
}

String add(const String& s1, const String& s2) {
    String result{s1};
    result.append(s2);
    return result;
}

auto x = add(1, 2);
auto y = add("hello", " world");
```
Polymorphisms

- We have seen 2 forms, but at least 4 major forms are classic & common
  - Runtime polymorphism (subtyping & inheritance)
  - Parametric polymorphism (templates, generics, ...)
  - Overloading
  - Coercion

Universal polymorphisms define types that can comprise an infinite number of other types with a common structure

Ad hoc polymorphisms define types that can comprise a finite set of explicitly specified types with even disparate structure

```cpp
class string_view {
    string_view(const char *);
    string_view(const std::string&);

    template <size_t N>
    string_view(const char[N]);

    template <size_t N>
    string_view(const std::array<char,N>&);

    ...
};
```
Polymorphisms

- We have seen 2 forms, but at least 4 major forms are classic & common
  - Runtime polymorphism (subtyping & inheritance)
  - Parametric polymorphism (templates, generics, ...)
  - Overloading
  - Coercion

Universal polymorphisms define types that can comprise an infinite number of other types with a common structure.

Ad hoc polymorphisms define types that can comprise a finite set of explicitly specified types with even disparate structure.

```cpp
class string_view {
    string_view(const char *);
    string_view(const std::string&);
    template <size_t N>
    string_view(const char[N]);
    template <size_t N>
    string_view(const std::array<char,N>&);
    ...
};

bool endsInING(string_view view) {
    return view.ends_with("ing");
}
```
Polymorphisms

- We have seen 2 forms, but at least 4 major forms are classic & common
  - Runtime polymorphism (subtyping & inheritance)
  - Parametric polymorphism (templates, generics, ...)
  - Overloading
  - Coercion

Universal polymorphisms define types that can comprise an infinite number of other types with a common structure.

Ad hoc polymorphisms define types that can comprise a finite set of explicitly specified types with even disparate structure.

```cpp
class string_view {
    string_view(const char *);
    string_view(const std::string&);
    template <size_t N>
    string_view(const char[N]);
    template <size_t N>
    string_view(const std::array<char,N>&);
    ...

    bool endsInING(string_view view) {
        return view.ends_with("ing");
    }
}
```

```cpp
bool endsInING("reading");
endsInING(std::string{"writing"});
std::array act = {'a','c','t','i','n','g'};
endsInING(acting);
```
Polymorphisms

- We have seen 2 forms, but at least 4 major forms are classic & common
  - Runtime polymorphism (subtyping & inheritance)
  - Parametric polymorphism (templates, generics, ...)
  - Overloading
  - Coercion

- Universal polymorphisms define types that can comprise an infinite number of other types with a common structure

- Ad hoc polymorphisms define types that can comprise a finite set of explicitly specified types with even disparate structure

- All forms of polymorphism have benefits & costs, but junior developers often struggle with inheritance vs parametricity
Runtime vs Parametric Polymorphism (commonly)

- Parametric polymorphism
  - Defines a fresh type for new parameters

```cpp
std::array<int, 5> != std::array<int, 6>
```
Runtime vs Parametric Polymorphism (commonly)

- Parametric polymorphism
  - Defines a fresh type for new parameters

This means:
Their elements can be taken as references.
They may have different sizes.
They cannot be stored in a single collection.
...
### Runtime vs Parametric Polymorphism (commonly)

- **Parametric polymorphism**
  - Defines a fresh type for new parameters
  - Statically type checked & bound
Runtime vs Parametric Polymorphism (commonly)

- Parametric polymorphism
  - Defines a fresh type for new parameters
  - Statically type checked & bound
    - More errors can be found at compile time

\[
\text{std::array<int,5> \neq \text{std::array<int,6>}}
\]
Runtime **vs** Parametric Polymorphism *(commonly)*

- Parametric polymorphism
  - Defines a fresh type for new parameters
  - Statically type checked & bound
    - More errors can be found at compile time
    - The parameters must be resolved at compile time (not dynamically linked in)

std::array<int,5> != std::array<int,6>
Runtime **vs** Parametric Polymorphism (commonly)

- Parametric polymorphism
  - Defines a fresh type for new parameters
  - Statically type checked & bound
    - More errors can be found at compile time
    - The parameters must be resolved at compile time (not dynamically linked in)
    - Significant performance gains are achievable

\[
\text{std::array<int,5> != std::array<int,6>}
\]
Runtime vs Parametric Polymorphism (commonly)

- **Parametric polymorphism**
  - Defines a fresh type for new parameters
  - Statically type checked & bound
    - More errors can be found at compile time
    - The parameters must be resolved at compile time (not dynamically linked in)
    - Significant performance gains are achievable

- **Runtime polymorphism**
  - Resolves operations dynamically (at runtime) through indirection
    - Indirection supports more flexibility & provides a uniform view

\[ \text{std::array<int,5>} \neq \text{std::array<int,6>} \]
Runtime vs Parametric Polymorphism

- **Parametric polymorphism**
  - Defines a fresh type for new parameters
  - Statically type checked & bound
    - More errors can be found at compile time
    - The parameters must be resolved at compile time (not dynamically linked in)
    - Significant performance gains are achievable

- **Runtime polymorphism**
  - Resolves operations dynamically (at runtime) through indirection
    - Indirection supports more flexibility & provides a uniform view
  - Hides the specific type from users of that type (decoupling)

```c++
void foo(Base&);
Derived1 d1;
foo(d1);
Derived2 d2;
foo(d2);
```
Runtime vs Parametric Polymorphism (commonly)

- **Parametric polymorphism**
  - Defines a fresh type for new parameters
  - Statically type checked & bound
    - More errors can be found at compile time
    - The parameters must be resolved at compile time (not dynamically linked in)
    - Significant performance gains are achievable

- **Runtime polymorphism**
  - Resolves operations dynamically (at runtime) through indirection
    - Indirection supports more flexibility & provides a uniform view
  - Hides the specific type from users of that type (decoupling)
  - Subtypes can be compiled separately (dynamically loaded, plug-in based, ...)

\[ \text{std::array<int,5>} \neq \text{std::array<int,6>} \]
Runtime & Parametric Polymorphism

• Combining them carefully leads to powerful results
  – Done well, you get the strengths of both (powerful good)
  – Done poorly, you get the weaknesses of both (powerful bad)
Runtime & Parametric Polymorphism

- Combining them carefully leads to powerful results
  - Done well, you get the strengths of both \(\text{-(powerful good)}\)
  - Done poorly, you get the weaknesses of both \(\text{-(powerful bad)}\)

- Parametric derived classes create a family of types satisfying an interface

```cpp
class Base {
    virtual void foo() = 0;
};

template<typename T>
class Derived : public Base {
    virtual void foo() = 0;
};
```
Runtime & Parametric Polymorphism

- Combining them carefully leads to powerful results
  - Done well, you get the strengths of both (powerful good)
  - Done poorly, you get the weaknesses of both (powerful bad)

- Parametric derived classes create a family of types satisfying an interface

```cpp
class Base {
    virtual void foo() = 0;
};

template<typename T>
class Derived : public Base {
    virtual void foo() = 0;
};
```
Runtime & Parametric Polymorphism

- Combining them carefully leads to powerful results
  - Done well, you get the *strengths* of both (powerful good)
  - Done *poorly*, you get the *weaknesses* of both (powerful bad)
- Parametric *derived* classes create a family of types satisfying an interface
- Parametric *base* classes support passing information from derived to based to improve safety & performance

```
class Base {
    virtual void foo(Base&) = 0;
};

template<typename Derived>
class Base {
    virtual void foo(Derived&) = 0;
};
```
Runtime & Parametric Polymorphism

- Combining them carefully leads to powerful results
  - Done well, you get the strengths of both (powerful good)
  - Done poorly, you get the weaknesses of both (powerful bad)
- Parametric *derived* classes create a family of types satisfying an interface
- Parametric *base* classes support passing information from derived to based to improve safety & performance

```cpp
class Base {
    virtual void foo(Base&) = 0;
};

template<typename Derived>
class Base {
    virtual void foo(Derived&) = 0;
};
```

What do the different sets of values mean?
Runtime & Parametric Polymorphism

- Combining them carefully leads to powerful results
  - Done well, you get the strengths of both (powerful good)
  - Done poorly, you get the weaknesses of both (powerful bad)
- Parametric derived classes create a family of types satisfying an interface
- Parametric base classes support passing information from derived to based to improve safety & performance
- Hiding inheritance behind a parametric interface can provide consistent usage while reducing complexity for a user
Runtime & Parametric Polymorphism

- Combining them carefully leads to powerful results
  - Done well, you get the strengths of both (powerful good)
  - Done poorly, you get the weaknesses of both (powerful bad)

- Parametric derived classes create a family of types satisfying an interface

- Parametric base classes support passing information from derived to based to improve safety & performance

- Hiding inheritance behind a parametric interface can provide consistent usage while reducing complexity for a user

- Problems with poor inheritance usage are exacerbated by parametricity (significant additional overheads & complexity)
Runtime & Parametric Polymorphism

- Both enable the open/closed principle
  - Code should be
    - open to extension (easy to customize)
    - closed to modification (original code should not need modification)
Runtime & Parametric Polymorphism

- Both enable the open/closed principle
  - Code should be
    - *open to extension* (easy to customize)
    - *closed to modification* (original code should not need modification)

```cpp
class CrosswordGenerator {
    CrosswordGenerator(... clues)
        : clues{std::move(clues)}
            { }

private:
        std::unique_ptr<Clues> clues;
};

auto englishClues = ...
CrosswordGenerator cg{englishClues};

auto frenchClues = ...
CrosswordGenerator cg{frenchClues};
```
Runtime & Parametric Polymorphism

- Both enable the open/closed principle
  - Code should be
    - open to extension (easy to customize)
    - closed to modification (original code should not need modification)

```cpp
template <typename WallCarver>
class MazeGenerator {
  MazeGenerator(WallCarver carver)
    : carver{std::move(carver)}
  { }

private:
  WallCarver carver;
};
```
Both enable the open/closed principle
- Code should be
  \textit{open to extension} \ (easy to customize)
  \textit{closed to modification} \ (original code should not need modification)

Both enable \textit{programs with holes} [Meyer 1996]
- Portions of a design are abstracted out & meant to be filled by a user
Runtime & Parametric Polymorphism

- Both enable the open/closed principle
  - Code should be
    
    open to extension (easy to customize)
    closed to modification (original code should not need modification)

- Both enable programs with holes [Meyer 1996]
  - Portions of a design are abstracted out & meant to be filled by a user
  - This allows you to defer some design decisions to a later point in time!
Runtime & Parametric Polymorphism

• Both enable the open/closed principle
  – Code should be
    *open to extension* (easy to customize)
    *closed to modification* (original code should not need modification)

• Both enable *programs with holes* [Meyer 1996]
  – Portions of a design are abstracted out & meant to be filled by a user
  – This allows you to defer some design decisions to a later point in time!

Polymorphism makes designing around decisions easier!
Runtime & Parametric Polymorphism

- Both enable the open/closed principle
  - Code should be
    - *open to extension* (easy to customize)
    - *closed to modification* (original code should not need modification)

- Both enable *programs with holes* [Meyer 1996]
  - Portions of a design are abstracted out & meant to be filled by a user
  - This allows you to defer some design decisions to a later point in time!
  - This is one form of *inversion of control*
Runtime & Parametric Polymorphism

- Both enable the open/closed principle
  - Code should be
    - open to extension (easy to customize)
    - closed to modification (original code should not need modification)

- Both enable *programs with holes* [Meyer 1996]
  - Portions of a design are abstracted out & meant to be filled by a user
  - This allows you to defer some design decisions to a later point in time!
  - This is one form of inversion of control
  - We have seen this before with higher order functions & lambdas!
Both enable *programs with holes* [Meyer 1996]
- Portions of a design are abstracted out & meant to be filled by a user
- This allows you to defer some design decisions to a later point in time!
- This is one form of inversion of control
- We have seen this before with higher order functions & lambdas!
Both enable programs with holes [Meyer 1996]

- Portions of a design are abstracted out & meant to be filled by a user
- This allows you to defer some design decisions to a later point in time!
- This is one form of inversion of control
- We have seen this before with higher order functions & lambdas!
Both enable programs with holes [Meyer 1996]
- Portions of a design are abstracted out and meant to be filled by a user
- This allows you to defer some design decisions to a later point in time!
- This is one form of inversion of control
- We have seen this before with higher order functions & lambdas!

```cpp
template <typename Collection, typename Predicate>
bool any_of(const Collection& c, Predicate p) {
    for (const auto& element : c) {
        if (p(c)) {
            return true;
        }
    }
    return false;
}

any_of(elements, [] (const auto& e) { return e == 3; });
```
Composition

- Plain composition is still simpler than polymorphism, but it makes satisfying the open/closed principle harder.
Composition

- Plain composition is still simpler than polymorphism, but it makes satisfying the open/closed principle harder.
- By thinking of types as sets of values it still offers some tactics.
Composition

- Plain composition is still simpler than polymorphism, but it makes satisfying the open/closed principle harder.
- By thinking of types as sets of values it still offers some tactics.

char
256 values
Composition

- Plain composition is still simpler than polymorphism, but it makes satisfying the open/closed principle harder.
- By thinking of types as sets of values it still offers some tactics.

```cpp
char
256 values

class Base {
    virtual void foo() = 0;
};
∞ values
Composition

- Plain composition is still simpler than polymorphism, but it makes satisfying the open/closed principle harder.
- By thinking of types as sets of values it still offers some tactics.

```cpp
struct Pair {
    char a;
    char b;
};

256^2 = 65536 values
```

```cpp
class Base {
    virtual void foo() = 0;
};

∞ values
```
Composition

- Plain composition is still simpler than polymorphism, but it makes satisfying the open/closed principle harder.
- By thinking of types as sets of values it still offers some tactics.

```cpp
enum Colors {
  RED, ORANGE, YELLOW,
  GREEN, BLUE, PURPLE
};

6 values

char
256 values

struct Pair {
  char a;
  char b;
};

256^2 = 65536 values

class Base {
  virtual void foo() = 0;
};

∞ values
Composition

- Plain composition is still simpler than polymorphism, but it makes satisfying the open/closed principle harder.
- By thinking of types as sets of values it still offers some tactics.

```cpp
enum Colors {
    RED, ORANGE, YELLOW, GREEN, BLUE, PURPLE
};

6 values

char 256 values

OR → +

struct Pair {
    char a;
    char b;
};

256^2 = 65536 values

AND → *

class Base {
    virtual void foo() = 0;
};

∞ values
Composition

- Plain composition is still simpler than polymorphism, but it makes satisfying the open/closed principle harder.
- By thinking of types as sets of values it still offers some tactics.
- *Algebraic data types* can be constructed through basic relational compositions of values.
Composition

- Plain composition is still simpler than polymorphism, but it makes satisfying the open/closed principle harder.
- By thinking of types as sets of values it still offers some tactics.
- **Algebraic data types** can be constructed through basic relational compositions of values.
  - *Product types* are records.

```c
struct Pair {
    char a;
    char b;
};
```
Composition

- Plain composition is still simpler than polymorphism, but it makes satisfying the open/closed principle harder.
- By thinking of types as sets of values it still offers some tactics.
- **Algebraic data types** can be constructed through basic relational compositions of values:
  - *product types* are records
  - *sum types* are discriminated unions

```c
struct Pair {
    char a;
    char b;
};
```

```c
enum Colors {
    RED, ORANGE, YELLOW,
    GREEN, BLUE, PURPLE
};
```
Composition

- Plain composition is still simpler than polymorphism, but it makes satisfying the open/closed principle harder.
- By thinking of types as sets of values it still offers some tactics.
- **Algebraic data types** can be constructed through basic relational compositions of values:
  - *product types* are records
  - *sum types* are discriminated unions

```rust
class Pair {
    var a: char;
    var b: char;
}
enum Colors {
    RED, ORANGE, YELLOW,
    GREEN, BLUE, PURPLE
}
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String)
}
[From the Rust Book]
Composition

- Plain composition is still simpler than polymorphism, but it makes satisfying the open/closed principle harder.
- By thinking of types as sets of values it still offers some tactics.
- **Algebraic data types** can be constructed through basic relational compositions of values:
  - *product types* are records
  - *sum types* are discriminated unions
- Operations on sum types use **pattern matching** to require that all possible values are handled:
  - This is even enforced by the compiler!
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String)
}

let msg = Message::Quit;
match msg {
    Message::Quit => {
        println!("The Quit variant has no data to destruct."),
    },
    Message::Move { x, y } => {
        println!("Move {} and {}", x, y);
    },
    Message::Write(text) => println!("Text message: {}", text),
}
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String)
}

let msg = Message::Quit;
match msg {
    Message::Quit => {
        println!("The Quit variant has no data to destructe.")
    },
    Message::Move { x, y } => {
        println!("Move {} and {}", x, y);
    },
    Message::Write(text) => println!("Text message: {}", text),
}
```rust
double Message { Quit,
    Move { x: i32, y: i32 },
    Write(String)
}

let msg = Message::Quit;
match msg {
    Message::Quit => {
        println!("The Quit variant has no data to destructure."");
    },
    Message::Move { x, y } => {
        println!("Move {} and {}", x, y);
    },
    Message::Write(text) => println!("Text message: {}", text),
}
```

[From the Rust Book]
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String)
}

let msg = Message::Quit;
match msg {
    Message::Quit => {
        println!("The Quit variant has no data to destruct.");
    },
    Message::Move { x, y } => {
        println!("Move {} and {}", x, y);
    },
    Message::Write(text) => println!("Text message: {}", text),
}
What do sum types look like in e.g. C++ or Java?

```cpp
using Message = std::variant<Quit, Move, Write>;

struct Action {
    void operator()(const Quit&) {...}
    void operator()(const Move&) {...}
    void operator()(const Write&) {...}
};

... 
Message m = Quit{};
std::visit(Action{}, m);
```
What do sum types look like in e.g. C++ or Java?

```cpp
using Message = std::variant<Quit, Move, Write>;

struct Action {
    void operator()(const Quit&) {...}
    void operator()(const Move&) {...}
    void operator()(const Write&) {...}
};

Message m = Quit{};
std::visit(Action{}, m);
```

```java
public enum Message {
    QUIT,
    MOVE,
    WRITE {
        void bar() {...}
        @Override
        void foo() {...}
    }

    Message() {...}
}

public void foo() {}
```
What do sum types look like in e.g. C++ or Java?

```c++
using Message = std::variant<Quit, Move, Write>;

struct Action {
    void operator()(const Quit&) {...}
    void operator()(const Move&) {...}
    void operator()(const Write&) {...}
};
...
Message m = Quit{};
std::visit(Action{}, m);
```

```java
public enum Message {
    QUIT,
    MOVE,
    WRITE {
        void bar() {...}
        @Override
        void foo() {...}
    }
    Message() {...}
    ...
    public void foo() {};
}
```

But both languages are moving toward full pattern matching!
What *may* pattern matching look like in e.g. C++ or Java?

```cpp
int get_area(const Shape& shape) {
    return inspect (shape) {
        <Circle>    [r]    => 3.14 * r * r,
        <Rectangle> [w, h] => w * h
    }
}

Message m = ...
inspect (m) {
    <Quit> q:    ...
    <Move> o:    ...
    <Write> w:   ...
}
```

[Pattern Matching, p1371r0]
• What *may* pattern matching look like in e.g. C++ or Java?

```cpp
int get_area(const Shape& shape) {
    return inspect (shape) {
        <Circle> [r] => 3.14 * r * r,
        <Rectangle> [w, h] => w * h
    }
}
```

```java
Message m = ...
Result r = switch (m) {
    case QUIT q -> ...;
    case MOVE o -> ...;
    case WRITE w -> ...;
};
```

[Pattern Matching for Java]
By thinking of types as sets of values, we can carefully design types that help ensure correctness, flexibility, & performance.
Summary

- By thinking of types as sets of values, we can carefully design types that help ensure correctness, flexibility, & performance

- Four(!) major forms of polymorphism have been in use for decades that give us significant power when designing our types
Summary

- By thinking of types as sets of values, we can carefully design types that help ensure correctness, flexibility, & performance

- Four(!) major forms of polymorphism have been in use for decades that give us significant power when designing our types

- Algebraic data types use composition of types to provide safe and convenient handling of finite sets of types
Summary

- By thinking of types as sets of values, we can carefully design types that help ensure correctness, flexibility, & performance.
- Four(!) major forms of polymorphism have been in use for decades that give us significant power when designing our types.
- Algebraic data types use composition of types to provide safe and convenient handling of finite sets of types.
- All of these approaches have tradeoffs.