orphan: Swift supports what type theory calls "algebraic data types", or ADTs, which are an amalgam of two familiar C-family language features, enums and unions. They are similar to enums in that they allow collections of independent symbolic values to be collected into a type and switched over:

```
enum Color {
  case Red, Green, Blue, Black, White
}

var c : Color = .Red
switch c {
case .Red:
  ...
case .Green:
  ...
case .Blue:
  ...
}
```

They are also similar to C unions in that they allow a single type to contain a value of two or more other types. Unlike C unions, however, ADTs remember which type they contain, and can be switched over, guaranteeing that only the currently inhabited type is ever used:

```
enum Pattern {
  case Solid(Color)
  case Outline(Color)
  case Checkers(Color, Color)
}

var p : Pattern = .Checkers(.Black, .White)
switch p {
case .Solid(var c):
  print("solid \(c)")
case .Outline(var c):
  print("outlined \(c)")
case .Checkers(var a, var b):
  print("checkered \(a) and \(b)")
}
```

Given the choice between two familiar keywords, we decided to use 'enum' to name these types. Here are some of the reasons why:

Why 'enum'?

The common case works like C

C programmers with no interest in learning about ADTs can use 'enum' like they always have.

"Union" doesn't exist to Cocoa programmers

Cocoa programmers really don't think about unions at all. The frameworks vend no public unions. If a Cocoa programmer knows what a union is, it's as a broken C bit-bangy thing. Cocoa programmers are used to more safely and idiomatically modeling ADTs in Objective-C as class hierarchies. The concept of closed-hierarchy variant value types is new territory for them, so we have some freedom in choosing how to present the feature. Trying to relate it to C's 'union', a feature with negative connotations, is a disservice if anything that will dissuade users from wanting to learn and take advantage of it.

It parallels our extension of 'switch'

The idiomatic relationship between 'enum' and 'switch' in C is well-established--If you have an enum, the best practice for consuming it is to switch over it so the compiler can check exhaustiveness for you. We've extended 'switch' with pattern matching, another new concept for our target audience, and one that happens to be dual to the concept of enums with payload. In the whitepaper, we introduce pattern matching by starting from the familiar C case of switching over an integer and gradually introduce the new capabilities of Swift's switch. If all ADTs are 'enums', this lets us introduce both features to C programmers organically, starting from the familiar case that looks like C:

```
enum Foo { case A, B, C, D }

func use(_ x:Foo) {
  switch x {
  case .A:
  case .B:
  case .C:
  case .D:
  }
}
```

and then introducing the parallel new concepts of payloads and patterns together:

```
enum Foo { case A, B, C, D, Other(String) }

func use(_ x:Foo) {
  switch x {
  case .A:
  case .B:
  case .C:
  case .D:
  case .Other(var s):
  }
}
```

People already use 'enum' to define ADTs, badly

Enums are already used and abused in C in various ways as a building block for ADT-like types. An enum is of course the obvious choice to represented the discriminator in a tagged-union structure. Instead of saying 'you write union and get the enum for free', we can switch the message around: 'you write enum and get the union for free'. Aside from that case, though, there are many uses in C of enums as ordered integer-convertible values that are really trying to express more complex symbolic ADTs. For example, there's the pervasive LLVM convention of 'First_*' and 'Last_*' sigils:

```
/* C */
enum Pet {
  First_Reptile,
    Lizard = First_Reptile,
    Snake,
  Last_Reptile = Snake,

  First_Mammal,
    Cat = First_Mammal,
    Dog,
  Last_Mammal = Dog,
};
```

which is really crying out for a nested ADT representation:

```
// Swift
enum Reptile { case Lizard, Snake }
enum Mammal { case Cat, Dog }
enum Pet {
  case Reptile(Reptile)
  case Mammal(Mammal)
}
```

Or there's the common case of an identifier with standardized symbolic values and a 'user-defined' range:

```
/* C */
enum Language : uint16_t {
  C89,
  C99,
  Cplusplus98,
  Cplusplus11,
  First_UserDefined = 0x8000,
  Last_UserDefined = 0xFFFF
};
```

which again is better represented as an ADT:

```
// Swift
enum Language {
  case C89, C99, Cplusplus98, Cplusplus11
  case UserDefined(UInt16)
}
```

Rust does it

Rust also labels their ADTs 'enum', so there is some alignment with the "extended family" of C-influenced modern systems programming languages in making the same choice

Design

Syntax

The 'enum' keyword introduces an ADT (hereon called an "enum"). Within an enum, the 'case' keyword introduces a value of the enum. This can either be a purely symbolic case or can declare a payload type that is stored with the value:

```
enum Color {
  case Red
  case Green
```

```
  case Blue
}

enum Optional<T> {
  case Some(T)
  case None
}

enum IntOrInfinity {
  case Int(Int)
  case NegInfinity
  case PosInfinity
}
```

Multiple 'case' declarations may be specified in a single declaration, separated by commas:

```
enum IntOrInfinity {
  case NegInfinity, Int(Int), PosInfinity
}
```

Enum declarations may also contain the same sorts of nested declarations as structs, including nested types, methods, constructors, and properties:

```
enum IntOrInfinity {
  case NegInfinity, Int(Int), PosInfinity

  constructor() {
    this = .Int(0)
  }

  func min(_ x:IntOrInfinity) -> IntOrInfinity {
    switch (self, x) {
    case (.NegInfinity, _):
    case (_, .NegInfinity):
      return .NegInfinity
    case (.Int(var a), .Int(var b)):
      return min(a, b)
    case (.Int(var a), .PosInfinity):
      return a
    case (.PosInfinity, .Int(var b)):
      return b
    }
  }
}
```

They may not however contain physical properties.

Enums do not have default constructors (unless one is explicitly declared). Enum values are constructed by referencing one of its cases, which are scoped as if static values inside the enum type:

```
var red = Color.Red
var zero = IntOrInfinity.Int(0)
var inf = IntOrInfinity.PosInfinity
```

If the enum type can be deduced from context, it can be elided and the case can be referenced using leading dot syntax:

```
var inf : IntOrInfinity = .PosInfinity
return inf.min(.NegInfinity)
```

The 'RawRepresentable' protocol

In the library, we define a compiler-blessed 'RawRepresentable' protocol that models the traditional relationship between a C enum and its raw type:

```
protocol RawRepresentable {
  /// The raw representation type.
  typealias RawType

  /// Convert the conforming type to its raw type.
  /// Every valid value of the conforming type should map to a unique
  /// raw value.
  func toRaw() -> RawType

  /// Convert a value of raw type to the corresponding value of the
  /// conforming type.
  /// Returns None if the raw value has no corresponding conforming type
  /// value.
  class func fromRaw(_:RawType) -> Self?
}
```

Any type may manually conform to the RawRepresentable protocol following the above invariants, regardless of whether it supports

compiler derivation as underlined below.

Deriving the 'RawRepresentable' protocol for enums

An enum can obtain a compiler-derived 'RawRepresentable' conformance by declaring "inheritance" from its raw type in the following circumstances:

- The inherited raw type must be ExpressibleByIntegerLiteral, ExpressibleByExtendedGraphemeClusterLiteral, ExpressibleByFloatLiteral, and/or ExpressibleByStringLiteral.
- None of the cases of the enum may have non-void payloads.

If an enum declares a raw type, then its cases may declare raw values. raw values must be integer, float, character, or string literals, and must be unique within the enum. If the raw type is ExpressibleByIntegerLiteral, then the raw values default to auto-incrementing integer literal values starting from '0', as in C. If the raw type is not ExpressibleByIntegerLiteral, the raw values must all be explicitly declared:

```
enum Color : Int {
  case Black   // = 0
  case Cyan    // = 1
  case Magenta // = 2
  case White   // = 3
}

enum Signal : Int32 {
  case SIGKILL = 9, SIGSEGV = 11
}

enum NSChangeDictionaryKey : String {
  // All raw values are required because String is not
  // ExpressibleByIntegerLiteral
  case NSKeyValueChangeKindKey = "NSKeyValueChangeKindKey"
  case NSKeyValueChangeNewKey = "NSKeyValueChangeNewKey"
  case NSKeyValueChangeOldKey = "NSKeyValueChangeOldKey"
}
```

The compiler, on seeing a valid raw type for an enum, derives a RawRepresentable conformance, using 'switch' to implement the fromRaw and toRaw methods. The NSChangeDictionaryKey definition behaves as if defined:

```
enum NSChangeDictionaryKey : RawRepresentable {
  typealias RawType = String

  case NSKeyValueChangeKindKey
  case NSKeyValueChangeNewKey
  case NSKeyValueChangeOldKey

  func toRaw() -> String {
    switch self {
    case .NSKeyValueChangeKindKey:
      return "NSKeyValueChangeKindKey"
    case .NSKeyValueChangeNewKey:
      return "NSKeyValueChangeNewKey"
    case .NSKeyValueChangeOldKey:
      return "NSKeyValueChangeOldKey"
    }
  }

  static func fromRaw(_ s:String) -> NSChangeDictionaryKey? {
    switch s {
    case "NSKeyValueChangeKindKey":
      return .NSKeyValueChangeKindKey
    case "NSKeyValueChangeNewKey":
      return .NSKeyValueChangeNewKey
    case "NSKeyValueChangeOldKey":
      return .NSKeyValueChangeOldKey
    default:
      return nil
    }
  }
}
```