FABERNOVEL

What's to come with swift generics 

1 Generics

```
struct Array<Element> {
    /* Implementation */
}
```

```
struct Array<Element> {
    /* Implementation */
}
```

```
struct Optional<Wrapped> {
    /* Implementation */
}
```

```swift
// Here T can be any type we want
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

```swift
// Here T can be any type we want
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

```swift
// Here T has to conform to Comparable
func compare<T: Comparable>(a: T, b: T) -> Bool {
    return a > b
}
```

2 ○ Type-level VS Value-level abstraction

Type-level abstraction

```
// Any type conforming to Comparable can be used as parameter

func compare<T: Comparable>(a: T, b: T) -> Bool {
    return a > b
}
```

FABERNOVEL

Type-level abstraction

```
// Any type conforming to Comparable can be used as parameter

func compare<T: Comparable>(a: T, b: T) -> Bool {
    return a > b
}
```

Value-level abstraction

```
// Any value which has a conforming type to Protocol
// can be stored in the variable
var variable: Protocol
```

3

SE-0244 Opaque result type

Type-level abstraction for function returns

```
func foo<T: Constraints>() -> T {
    // Implementation
}
```

FABERNOVEL

Type-level abstraction for function returns

```
func foo<T: Constraints>() -> T {
    // Implementation
}
```

```
func foo() -> Constraints {
    // Implementation
}
```

FABERNOVEL

Type-level abstraction for function returns

```
func foo<T: Constraints>() -> T {
    // Implementation
}
```

```
func foo() -> Constraints {
    // Implementation
}
```

```
func foo() -> some Constraints {
    // Implementation
}
```

FABERNOVEL

FABERNOVEL

Why letting the callee decide of the type ?

Why letting the callee decide of the type ?

```
protocol Shape {
    func draw(to: Surface)
    func collides<Other: Shape>(with: Other) -> Bool
}

struct Rectangle: Shape { /* Implementation */ }
struct Circle: Shape { /* Implementation */ }

struct Union<A: Shape, B: Shape>: Shape {
    var a: A, b: B
    /* Implementation */
}

struct Intersect<A: Shape, B: Shape>: Shape {
    var a: A, b: B
    /* Implementation */
}

struct Transformed<S: Shape>: Shape {
    var shape: S
    var transform: Matrix3x3
    /* Implementation */
}
```

FABERNOVEL

Why letting the callee decide of the type ?

```
protocol GameObject {
    // The shape of the object
    associatedtype Shape: Shapes.Shape

    var shape: Shape { get }
}
```

FABERNOVEL

Why letting the callee decide of the type ?

```
protocol GameObject {
    // The shape of the object
    associatedtype Shape: Shapes.Shape

    var shape: Shape { get }
}


struct EightPointedStar: GameObject {
    var shape: Union<Rectangle, Transformed<Rectangle>> {
        return Union(Rectangle(), Transformed(shape: Rectangle(), transform: .fortyFiveDegrees)
    }
}
```

Why letting the callee decide of the type ?

```
protocol GameObject {
    // The shape of the object
    associatedtype Shape: Shapes.Shape

    var shape: Shape { get }
}


struct EightPointedStar: GameObject {
    var shape: Union<Transformed<Rectangle>, Rectangle> {
        return Union(Transformed(shape: Rectangle(), transform: .fortyFiveDegrees), Rectangle())
    }
}
```

FABERNOVEL

Why letting the callee decide of the type ?

```
protocol GameObject {
    // The shape of the object
    associatedtype Shape: Shapes.Shape

    var shape: Shape { get }
}


struct EightPointedStar: GameObject {
    var shape: some Shape {
        return Union(Transformed(shape: Rectangle(), transform: .fortyFiveDegrees), Rectangle())
    }
}
```

Key informations about opaque result types

FABERNOVEL

Key informations about opaque result types

Opaque result type is different from existential type

FABERNOVEL

Key informations about opaque result types

Opaque result type is different from existential type

The compiler knows what is the underlying type

FABERNOVEL

Key informations about opaque result types

Opaque result type is different from existential type

The compiler knows what is the underlying type

Opaque result types cannot be wrapped

FABERNOVEL

4 What is to come?

Clarifying existential types

```
// Any value which has a conforming type to Protocol
// can be stored in the variable
var variable: any Protocol
```

FABERNOVEL

Improving the notation for generics

```
func foo<T: Collection, U: Collection>(x: T, y: U) -> some Collection

func foo(x: Collection, y: Collection) -> some Collection
```

FABERNOVEL

Generalizing some syntax to arguments and returns

```
func concatenate<T>(
    a: some Collection<.Element == T>,
    b: some Collection<.Element == T>
) -> some Collection<.Element == T>
```

FABERNOVEL

Thank you

CONTACT
Denis POIFOL
Software engineer
+33 6 58 90 85 54
denis.poifol@fabernovel.com