Generics

- Sometimes we just don't care

  We may want to manipulate data structures that we are "type agnostic" about.
  In other words, we don't know what type something is and we don't care.



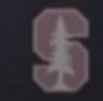
Generics

- Sometimes we just don't care
  - We may want to manipulate data structures that we are "type agnostic" about.
  - In other words, we don't know what type something is and we don't care.
  - But Swift is a strongly-typed language, so we don't use variables and such that are "untyped."



Generics

- Sometimes we just don't care
  - We may want to manipulate data structures that we are "type agnostic" about.
  - In other words, we don't know what type something is and we don't care.
  - But Swift is a strongly-typed language, so we don't use variables and such that are "untyped."
  - So how do we specify the type of something when we don't care what type it is?
  - We use a "don't care" type (we call this feature "generics") ...

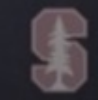


Generics

- Example of a user of a "don't care" type: `Array`

  Awesome example of generics: `Array`.

  An `Array` contains a bunch of things and it doesn't care at all what type they are!



Generics

- Example of a user of a "don't care" type: `Array`

  Awesome example of generics: `Array`.

  An `Array` contains a bunch of things and it doesn't care at all what type they are!

  But inside `Array`'s code, it has to have variables for the things it contains. They need types.

  And it needs types for the arguments to `Array` functions that do things like adding items to it.



Generics

- Example of a user of a "don't care" type: `Array`

  Awesome example of generics: `Array`.

  An `Array` contains a bunch of things and it doesn't care at all what type they are!

  But inside `Array`'s code, it has to have variables for the things it contains. They need types.

  And it needs types for the arguments to `Array` functions that do things like adding items to it.

  Enter ... GENERICS.



Generics

- How Array uses a "don't care" type

  Array's declaration looks something like this ...

```
struct Array<Element> {
    ...
    func append(_ element: Element) { ... }
}
```

  The type of the argument to `append` is `Element`. A "don't care" type.



Generics

- How Array uses a "don't care" type

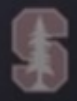
  Array's declaration looks something like this ...

```
struct Array<Element> {
    ...
    func append(_ element: Element) { ... }
}
```

  The type of the argument to append is Element. A "don't care" type.

  Array's implementation of append knows nothing about that argument and it does not care.

  Element is not any known struct or class or protocol, it's just a placeholder for a type.



Generics

- How Array uses a "don't care" type

  Array's declaration looks something like this ...

```
struct Array<Element> {
    ...
    func append(_ element: Element) { ... }
}
```

  The type of the argument to append is Element. A "don't care" type.

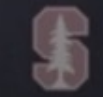
  Array's implementation of append knows nothing about that argument and it does not care.

  Element is not any known struct or class or protocol, it's just a placeholder for a type.

  The code for using an Array looks something like this ...

```
var a = Array<Int>()
a.append(5)
a.append(22)
```

  When someone uses Array, that's when Element gets determined (by Array<Int>).



Generics

- How Array uses a "don't care" type

  Array's declaration looks something like this ...

```
struct Array<Element> {
    ...
    func append(_ element: Element) { ... }
}
```

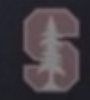
  Note that Array has to let the world know the names of all of its "don't care" types in its API.

  It does this with the < > notation on its struct declaration Array<Element> above.

  That's how users of Array know that they have to say what type Element actually is.

```
var a = Array<Int>()
```

  It is perfectly legal to have multiple "don't care" types in the above (e.g. <Element, Foo>)



Generics

- Type Parameter

  I will often refer to these types like `Element` in `Array` as a "don't care" type.

  If you ask `Array` what type its elements are, it will say "I don't care".

  But its



Generics

- Type Parameter

  I will often refer to these types like `Element` in `Array` as a "don't care" type.
  If you ask `Array` what type its elements are, it will say "I don't care".
  But its actual name is Type Parameter.
  Other languages most of you may know (e.g. Java) have a similar feature.
  However, Swift combines this with `protocols` to take it all to the next level.
  We'll talk about that next week!

