class Bag

Immutable collection of distinct objects with integer weights

```
class Bag does Baggy { }
```

A `Bag` is an immutable bag/multiset implementing Associative, meaning a collection of distinct elements in no particular order that each have an integer weight assigned to them signifying how many copies of that element are considered "in the bag". (For *mutable* bags, see BagHash instead.)

`Bag`s are often used for performing weighted random selections - see .pick and .roll.

Objects/values of any type are allowed as bag elements. Within a `Bag`, items that would compare positively with the `===` operator are considered the same element, with the number of how many there are as its weight. But of course you can also easily get back the expanded list of items (without the order):

```
my $breakfast = bag <spam eggs spam spam bacon spam>;

say $breakfast.elems;      # OUTPUT: «3»
say $breakfast.keys.sort;  # OUTPUT: «bacon eggs spam»

say $breakfast.total;      # OUTPUT: «6»
say $breakfast.kxxv.sort;  # OUTPUT: «bacon eggs spam spam spam spam»
```

`Bag`s can be treated as object hashes using the `{ }` postcircumfix operator, or the `< >` postcircumfix operator for literal string keys, which returns the corresponding integer weight for keys that are elements of the bag, and `0` for keys that aren't:

```
my $breakfast = bag <spam eggs spam spam bacon spam>;
say $breakfast<bacon>;    # OUTPUT: «1»
say $breakfast<spam>;     # OUTPUT: «4»
say $breakfast<sausage>;  # OUTPUT: «0»
```

Creating Bag objects

`Bag` s can be composed using the bag subroutine (or `Bag.new` , for which it is a shorthand). Any positional parameters, regardless of their type, become elements of the bag:

```
my $n = bag "a" => 0, "b" => 1, "c" => 2, "c" => 2;
say $n.keys.raku;        # OUTPUT: «(:c(2), :b(1), :a(0)).Seq»
say $n.keys.map(&WHAT);  # OUTPUT: «((Pair) (Pair) (Pair))»
say $n.values.raku;      # OUTPUT: «(2, 1, 1).Seq»
```

Alternatively, the `.Bag` coercer (or its functional form, `Bag()`) can be called on an existing object to coerce it to a `Bag` . Its semantics depend on the type and contents of the object. In general it evaluates the object in list context and creates a bag with the resulting items as elements, although for Hash-like objects or Pair items, only the keys become elements of the bag, and the (cumulative) values become the associated integer weights:

```
my $n = ("a" => 0, "b" => 1, "c" => 2, "c" => 2).Bag;
say $n.keys.raku;        # OUTPUT: «("b", "c").Seq»
say $n.keys.map(&WHAT);  # OUTPUT: «((Str) (Str))»
say $n.values.raku;      # OUTPUT: «(1, 4).Seq»
```

Furthermore, you can get a `Bag` by using bag operators (see next section) on objects of other types such as List , which will act like they internally call `.Bag` on them before performing the operation. Be aware of the tight precedence of those operators though, which may require you to use parentheses around arguments:

```
say (1..5) (+) 4;  # OUTPUT: «Bag(1 2 3 4(2) 5)»
```

Of course, you can also create a `Bag` with the `.new` method.

```
my $breakfast = Bag.new( <spam eggs spam spam bacon spam> );
```

Since 6.d (2019.03 and later) you can also use this syntax for parameterization of the `Bag` , to specify which type of values are acceptable:

```
# only allow strings (Str) in the Bag
my $breakfast = Bag[Str].new( <spam eggs spam spam bacon spam> );

# only allow whole numbers (Int) in the Bag
my $breakfast = Bag[Int].new( <spam eggs spam spam bacon spam> );
# Type check failed in binding; expected Int but got Str ("spam")
```

Finally, you can create Bag masquerading as a hash by using the `is` trait:

```
my %b is Bag = <a b c>;
say %b<a>;  # True
say %b<d>;  # False
```

Since 6.d (2019.03 and later), this syntax also allows you to specify the type of values you would like to allow:

```
# limit to strings
my %b is Bag[Str] = <a b c>;
say %b<a>;  # True
say %b<d>;  # False

# limit to whole numbers
my %b is Bag[Int] = <a b c>;
# Type check failed in binding; expected Int but got Str ("a")
```

Operators

See Operators with set semantics for a complete list of "set operators" applicable to, among other types, `Bag` .

Examples:

```
my ($a, $b) = bag(2, 2, 4), bag(2, 3, 3, 4);

say $a (<) $b;   # OUTPUT: «False»
say $a (<=) $b;  # OUTPUT: «False»
say $a (^) $b;   # OUTPUT: «Bag(3(2) 2)»
say $a (+) $b;   # OUTPUT: «Bag(2(3) 4(2) 3(2))»

# Unicode versions:
say $a ⊍ $b;  # OUTPUT: «False»
say $a ⊎ $b;  # OUTPUT: «False»
say $a ∖ $b;  # OUTPUT: «Bag(3(2) 2)»
say $a ∪ $b;  # OUTPUT: «Bag(2(3) 4(2) 3(2))»
```

Subroutines

sub bag

```
sub bag(*@args --> Bag)
```

Creates a new `Bag` from `@args` .

Note on `reverse` and ordering

This method is inherited from Any , however, `Mix` es do not have an inherent order and you should not trust it returning a consistent output.

See also

Sets, Bags, and Mixes