Grace Documentation:
Built-in Objects and Basic Libraries

Kim B. Bruce Andrew P. Black

August 10, 2015

1 Introduction

Grace has several built-in types and objects, and a growing selection of dialects and libraries that define other useful types and objects. Built-ins can be used as-is, dialects must be declared with the **dialect** statement, and libraries must be imported with the **import** statement.

2 Built-in Types

Grace supports built-in objects with types Object, Number, Boolean, and String.

2.1 Object

All Grace objects understand the methods in type Object. These methods will often be omitted when other types are described.

```
type Object = {

  == (other: Object) −> Boolean
  // true if other is equal to self

  != (other: Object) −> Boolean
  /= (other: Object) −> Boolean
  ≠ (other: Object) −> Boolean
  // the inverse of ==.   All three variants have the same meaning.

  hash −> Number
  // the hash code of self, a Number in the range 0..2^32

  match (other: Object) −> SucccessfulMatch | FailedMatch
  // returns a SuccessfulMatch if self "matches" other
  // returns FailedMatch otherwise.
  // The exact meaning of "matches" depends on self.

  asString −> String
  // a string describing self

  asDebugString −> String
  // a string describing the internals of self

  :: (other:Object) −> Binding
  // a Binding object with self as key and other as value.
}
```

We will also not discuss or include the pattern combinators | (or) and & (and) used in generating compound patterns.

2.2 Number

Number describes all numeric values in *minigrace*, including integers and numbers with decimal fractions. (Thus, *minigrace* Numbers are what some other languages call floating point numbers, floats or double-precision). Numbers are represented with a precision of approximately 51 bits.

```
type Number = {

  + (other: Number) -> Number
  // sum of self and other

  - (other: Number) -> Number
  // difference of self and other

  * (other: Number) -> Number
  // product of self and other

  / (other: Number) -> Number
  // quotient of self and other

  % (other: Number) -> Number
  // modulus of self and other (remainder after division)

  .. (last: Number) -> Sequence
  // the Sequence of numbers from self to last

  < (other: Number) -> Boolean
  // true iff self is less than other

  <= (other: Number) -> Boolean
  ≤(other: Number) -> Boolean
  // true iff self is less than or equal to other

  > (other: Number) -> Boolean
  // true iff self is greater than other

  >= (other: Number) -> Boolean
  ≥ (other: Number) -> Boolean
  // true iff self is greater than or equal to other

  prefix- -> Number
  // negation of self

  compare (other: Number) -> Number
  // a three-way comparison: -1 if (self < other), 0 if (self == other), and +1 if (self > other).

  inBase (base:Number) -> String
  // a string representing self as a base number (e.g., base 2)

  truncated -> Number
  // number obtained by throwing away self's fractional part
```

```
rounded −> Number
// whole number closest to self

floor −> Number
// largest whole number less than or equal to self

ceiling −> Number
// smallest number greater than or equal to self

abs −> Number
// the absolute value of self

sqrt −> Number
// the square root of self

sgn −> Number
// the signum function: 0 when self == 0, // −1 when self < 0, and +1 when self > 0

isNan −> Boolean
// true if this Number is a NaN

sin −> Number
// trigonometric sine (self in radians)

cos −> Number
// cosine (self in radians)

tan −> Number
// tangent (self in radians)

asin −> Number
// arcsine of self (result in radians)

acos −> Number
// arccosine of self (result in radians)

atan −> Number
// arctangent of self (result in radians)

lg −> Number
// log base 2 of self

ln −> Number
// the natural log of self

exp −> Number
// e raised to the power of self

log10 (n: Number) −> Number
// log base 10 of self
}
```

2.3 String

String constructors are written surrounded by double quote characters. There are three commonly-used escape characters:

- \n means the newline character
- \\ means a single backslash chracter
- \" means a double quote character.

There are also escapes for a few other characters and for arbitrary Unicode codepoints; for more information, see the Grace language specification.

String constructors can also contain simple Grace expressions[1] enclosed in braces, like this: `"count = {count}."` These are called string interpolations. The value of the interpolated expression is calculated, converted to a string (by requesting its asString method), and concatenated between the surrounding fragments of literal string. Thus, if the value of count is 7, the above example will evaluate to the string `"count = 7."`

Strings are immutable. Methods like replace()with always return a new string; they never change the receiver.

```
type String = {
  * (n: Number) −> String
  // returns a string that contains n repetitions of self, so "abc"∗3 = "abcabcabc"

  ++ (other: Object) −> String
  // returns a string that is the concatenation of self and other.asString

  < (other: String)
  // true if self precedes other lexicographically

  <= (other: String)
  ≤ (other: String)
  // (self == other) || (self < other)

  == (other: Object)
  // true if other is a String and is equal to self

  != (other: Object)
  ≠ (other: Object)
  // (self == other).not

  > (other: String)
  // true if self follows other lexicographically

  >= (other: String)
  ≥ (other: String)
  // (self == other) || (self > other)

  at (index: Number) −> String
  // returns the character in position index (as a string of size 1); index must be in the range 1..size

  first −> String
  // returns the first character of the string, as a String of size 1. String must
```

[1] It is a limitation of *minigrace* that expressions containing {braces} and "quotes" cannot be interpolated into strings.

//not be empty

asDebugString −> String
// returns self enclosed in quotes, and with embedded special characters quoted.
// See also quoted.

asLower −> String
// returns a string like self, except that all letters are in lower case

asNumber −> Number
// attempts to parse self as a number; returns that number, or NaN if it can't

asString −> String
// returns self, naturally

asUpper −> String
// returns a string like self, except that all letters are in upper case

capitalized −> String
// returns a string like self, except that the initial letters of all words are in upper case

compare (other: String) −> Number
// a three−way comparison: −1 if (self < other), 0 if (self == other), and +1 if (self > other)

contains (other: String) −> Boolean
// returns true if other is a substring of self

endsWith (possibleSuffix: String)
// true if self ends with possibleSuffix

filter (predicate: Block1[[String, Boolean]]) −> String
// returns the String containing those characters of self for which predicate returns true

fold [[U]] (binaryFunction: Block2[[U, String, U]]) startingWith(initial: U) −> U
// performs a left fold of binaryFunction over self, starting with initial.
// For example, fold {a, b −> a + b.ord} startingWith 0 will compute the sum
// of the ords of the characters in self

hash −> Number
// the hash of self

indexOf (pattern: String) −> Number
// returns the leftmost index at which pattern appears in self, or 0 if it is not there.

indexOf [[W]] (pattern: String) ifAbsent (absent: Block0[[W]]) −> Number | W
// returns the leftmost index at which pattern appears in self; applies absent if it is not there.

indexOf (pattern: String) startingAt(offset) −> Number
// like indexOf(pattern) except that it returns the first index ≥ offset, or 0 if there is no such index.

indexOf [[W]] (pattern:String) startingAt(offset) ifAbsent(action:Block0[[W]]) −> Number | W
// like the above, except that it applies action if there is no such index.

indices −> Sequence [[T]]
// an object representing the range of indices of self (1..self.size)

isEmpty −> Boolean
// true if self is the empty string

iterator −> Iterator[[String]]
// an iterator over the characters of self

lastIndexOf(sub:String) −> Number
// returns the rightmost index at which sub appears in self, or 0 if it is not there.

lastIndexOf [[W]] (sub:String) ifAbsent (absent:Block0[[W]]) −> Number | W
// returns the rightmost index at which sub appears in self; applies absent if it is not there.

lastIndexOf [[W]] (pattern:String) startingAt (offset) ifAbsent (action:Block0[[W]]) −> Number | W
// like the above, except that it returns the last index $\leq$ *offset.*

map[[U]] (function:Block[[String,U]]) −> Iterable[[U]]
// returns an Iterable object containing the results of successive applications of function to the
// individual characters of self. Note that the result is not a String, even if type U happens to be String.
// If a String is desired, use fold()startingWith "" with a function that concatenates.

match (other: Object) −> SuccessfulMatch | FailedMatch
// returns SuccessfulMatch match if self matches other, otherwise FailedMatch

ord −> Number
// a numeric representation of the first character of self, or NaN if self is empty.

replace (pattern: String) with (new: String) −> String
// a string like self, but with all occurrences of pattern replaced by new

size −> Number
// returns the size of self, i.e., the number of characters it contains.

startsWith (possiblePrefix: String) −> Boolean
// true when possiblePrefix is a prefix of self

startsWithDigit −> Boolean
// true if the first character of self is a (Unicode) digit.

startsWithLetter −> Boolean
// true if the first character of self is a (Unicode) letter

startsWithPeriod −> Boolean
// true if the first character of self is a period

startsWithSpace −> Boolean
// true if the first character of self is a (Unicode) space.

substringFrom (start: Number) size (max: Number) −> String
// returns the substring of self starting at index start and of length max characters,
// or extending to the end of self if that is less than max. *If start = self.size + 1, or*
// stop < start, the empty string is returned. *If start is outside the range*
// 1..self.size+1, BoundsError is raised.

```
    substringFrom (start: Number) to (stop: Number) -> String
    // returns the substring of self starting at index start and extending
    // either to the end of self, or to stop.    If start = self.size + 1, or
    // stop < start, the empty string is returned.   If start is outside the range
    // 1..self.size+1, BoundsError is raised.

    substringFrom (start: Number) -> String
    // returns the substring of self starting at index start and extending
    // to the end of self.    If start = self.size + 1, the empty string is returned.
    // If start is outside the range 1..self.size+1, BoundsError is raised.

    trim -> String
    // a string like self except that leading and trailing spaces are omitted.

    quoted -> String
    // returns a quoted version of self, with internal characters like " and \ and newline escaped,
    // but without surrounding quotes.
  }
```

2.4 Boolean

The Boolean literals are true and false.

```
type Boolean = {

    not −> Boolean
    prefix ! −> Boolean
    // the negation of self

    && (other: Boolean) −> Boolean
    // return true when self and other are both true

    || (other: Boolean) −> Boolean
    // return true when either self or other (or both) are true
}
```

In conditions in ‘if‘ statements, and in the operators && and ||, a Block returning a boolean may be used instead of a Boolean. This means that && and || can be used as “short-circuit”, also known as “non-commutative”, operators: if the argument is a block, it will be evaluated only if necessary.

```
type BlockBoolean = { apply −> Boolean }
type BlockOrBoolean = BlockBoolean | Boolean
```

2.5 Blocks

Blocks are anonymous functions that take zero or more arguments and may return a result. There is a family of ‘Block‘ types that describe block objects.

```
type Block0[[R]] = {
// type of parameterless functions that return a value of type R
  apply −> R
}

type Block1[[T,R]] = {
  // type of functions that take a single argument of type T and return a value of type R
  apply(a:T) −> R
}

type Block2[[S,T,R]] = {
// type of functions that takes two parameters of types S and T, and returns a value of type R
  apply(a:S, b:T) −> R
}
```

2.6 Point

Points can be thought of as locations in the cartesian plane, or as 2-dimensional vectors from the origin to a specified location. As a result they can also be used to represent the width and height of an object. Points are created from Numbers using the @ infix operator. Thus, 3 @ 4 represents the point with coordinates $(3, 4)$.

```
type Point = {

    x −> Number
    // the x−coordinates of self
```

```
    y −> Number
    // the y−coordinate of self

    + (other:Point) −> Point
    // the Point that is the vector sum of self and other, i.e. (self.x+other.x) @ (self.y+other.y)

    − (other:Point) −> Point
    // the Point that is the vector difference of self and other, i.e. (self.x−other.x) @ (self.y−other.y)

    prefix − −> Point
    // the point that is the negation of self

    * (factor:Number) −> Point
    // this point scaled by factor,  i.e. (self.x*factor) @ (self.y*factor)

    / (divisor:Number) −> Point
    // this point scaled by 1/factor, i.e. (self.x/divisor) @ (self.y/divisor)

    length −> Number
    // distance from self to the origin

    distanceTo(other:Point) −> Number
    // distance from self to other
}
```

2.7 Binding

A binding is an immutable pair comprising a key and a value. Bindings are created with the infix :: operator, as in k::v, or by requesting binding.key(k) value(v).

```
type Binding[[K, T]] = {
    key −> K
    // returns the key
    value −> T
    // returns the value
}
```

3 Collection objects

The objects described in this section are made available to all standard Grace programs. (This means that they are defined as part of the *standardGrace* dialect.) As is natural for collections, the types are parameterized by the types of the elements of the collection. Type arguments are enclosed in [[and]] used as brackets. This enables us to distinguish, for example, between Set[[Number]] and Set[[String]]. In Grace programs, type arguments and their brackets can be omitted; this is equivalent to using Unknown as the argument, which says that the programmer either does not know, or does not care to state, the type.

3.1 Common Abstractions

The major kinds of collection are sequence, list, set, and dictionary. Although these objects differ in their details, they share many common methods, which are defined in a hierarchy of types, each extending the one above it in the hierarchy.

The simplest is the type Iterable[[T]], which captures the idea of a (potentially unordered) collection of *elements*, each of type T, over which a client can iterate:

```
type Iterable[[T]] = {
    iterator −> Iterator[[T]]
    // Returns an iterator over my elements. It is an error to modify self while iterating over it.
    // Note: all other methods can be defined using iterator. Iterating over a dictionary
    // yields its values.

    isEmpty −> Boolean
    // True if self has no elements

    size −> Number
    // The number of elements in self; raises SizeUnknown if size is not known.

    sizeIfUnknown (action: Block0[[Number]]) −> Number
    // The number of elements in self; if size is not known, then action is evaluated and its value returned.

    first −> T
    // The first element of self; raises BoundsError if there is none.
    // If self is unordered, then first answers an arbitrary element.

    do (action: Block1[[T,Unknown]]) −> Done
    //  Applies action to each element of self.

    do (action:Block1[[T, Unknown]]) separatedBy (sep: Block0[[Unknown]]) −> Done
    // applies action to each element of self, and applies sep (to no arguments) in between.

    map[[R]] (unaryFunction: Block1[[T, R]]) −> Iterable[[T]]
    // returns a new collection whose elements are obtained by applying unaryFunction to
    // each element of self. If self is ordered, then the result is ordered.

    fold[[R]] (binaryFunction: Block2[[R, T, R]]) startingWith(initial:R) −> R
    // folds binaryFunction over self, starting with initial. If self is ordered, this is
    // the left fold. For example, fold {a, b −> a + b} startingWith 0
    // will compute the sum, and fold {a, b −> a ∗ b} startingWith 1 the product.

    filter (condition: Block1[[T, Boolean]]) −> Iterable[[T]]
    // returns a new collection containing only those elements of self for which
    // condition holds. The result is ordered if self is ordered.

    ++ (other: Iterable[[T]]) −> Iterable[[T]]
    // returns a new object whose elements include those of self and those of other.
}
```

The type ‘Collection‘ adds some conversion methods to ‘Iterable‘:

```
type Collection[[T]] = Iterable[[T]] & type {
    asList −> List[[T]]
        // returns a (mutable) list containing my elements.

    asSequence −> Sequence[[T]]
    // returns a sequence containing my elements.

    asSet −> Set[[T]]
    // returns a (mutable) Set containing my elements, with duplicates eliminated.
    // The == operation on my elements is used to identify duplicates.
```

```
}
```

Additional methods are available in the type Enumerable; an Enumerable is like a Sequence, but where the elements must be *enumerated* one by one, in order, using a computational process, rather than being stored explicitly. For this reason, operations that require access to all of the elements at one time are not supported, except for conversion to other collections that store their elements. The key difference between an Iterable and an Enumerable is that Enumerables have a natural order, so lists are Enumerable, whereas sets are just Iterable.

```
type Enumerable[[T]] = Collection[[T]] & type {

    values −> Enumerable[[T]]
    // an enumeration of my values: the elements in the case of  sequence or list,
    // the values in the case of a dictionary.

    asDictionary −> Dictionary[[Number, T]]
    // returns a dictionary containing my indices as keys and my elements as values, so that
    // my self.at(i) element is self.asDictionary.at(i)}.

    keysAndValuesDo (action: Block2[[Number, T, Object]]) −> Done
    // applies action, in sequence, to each of my keys and the corresponding element.

    into(existing: Collection[[T]]) −> Collection[[T]]
    // adds my elements to existing, and returns existing.

    sorted −> List[[T]]
    // returns a new List containing all of my elements, but sorted by their < and == operations.

    sortedBy (sortBlock: Block2[[T, T, Number]]) −> Sequence[[T]]
    // returns a new List containing all of my elements, but sorted according to the ordering
    // established by sortBlock, which should return −1 if its first argument is less than its second
    //  argument, 0 if they are equal, and +1 otherwise.
}
```

3.2 Lineups

The Grace language uses brackets as a syntax for constructing lineup objects. For example, [2,4,6,8] is the line-up consisting of the first four even positive integers. [ ] constructs the empty lineup.

Lineup objects have type Iterable[[T]] for the appropriate T. They are not indexable, so can’t be used like arrays or lists. They are primarily intended for initializing more capable collections, as in list [2, 3, 4], which creates a list, or set ["red", "green", "yellow"], which creates a set. Notice that a space must separate the name of the method from the lineup.

3.3 Sequence

The type Sequence[[T]] describes sequences of values of type T. Sequence objects are immutable; they can be constructed either explicitly, using sequence [1, 3, 5, 7], or as ranges like 1 .. 10.

```
type Sequence[[T]] = Enumerable[[T]] & type {

    at (ix:Number) −> T
    // returns my ix^th element, provided ix is integral and 1 ≤ ix ≤ size
```

first −> T
// returns my first element

second −> T
// returns my second element

third −> T
// returns my third element

fourth −> T
// returns my fourth element

fifth −> T
// returns my fifth element

last −> T
// returns my last element

indices −> Sequence[[Number]]
// returns the sequence of my indices.

keys −> Sequence[[Number]]
// same as indices; the name keys is for compatibility with dictionaries.

indexOf (sought:T) −> Number
// returns the index of my first element v such that v == sought. Raises NoSuchObject if there is none.

indexOf[[W]] (sought:T) ifAbsent(action:Block0[[W]]) −> Number | W
// returns the index of the first element v such that v == sought.
// Performs action if there is no such element.

reversed −> Sequence[[T]]
// returns a Sequence containing my values, but in the reverse order.

contains (sought:T) −> Boolean
// returns true if I contain an element v such that v == sought

}

3.4 Ranges

Ranges are sequences of consecutive integers. They behave exactly like other sequences, but are stored compactly. Ranges are created by two methods on the range class:

range.from (lower: Number) to (upper: Number)
// the sequence of integers from lower to upper, inclusive. If lower = upper, the range contains a single value.
// if lower > upper, the range is empty. It is an error for lower or upper not to be an integer.

range.from (upper: Number) downTo (lower: Number)
// the sequence from upper to lower, inclusive. If upper = lower, the range contains a single value.
// if upper < lower, the range is empty. It is an error for lower or upper not to be an integer.

The .. operation on Numbers can also be conveniently used to create ranges. Thus, 3 .. 9 is the same as range.from (3) to (9), and (3 .. 9).reversed is the same as range.from (9) downTo (3).

3.5 List

The type List[[T]] describes objects that are mutable lists of elements that have type T. List objects can be constructed using the list request, as in list[[T]] [a, b, c, ...] or list (existingCollection). An empty list is created by emptyList.

```
type List[[T]] = Sequence[[T]] & type {

    at (n: Number) put(new:T) −> List[[T]]
    // updates self so that my n^th element is new. Returns self.
    // Requires 1 ≤ n ≤ size+1; when n =size+1, equivalent to addLast(new).

    add (new: T) −> List[[T]]
    addLast(new:T) −> List[[T]]
    // adds new to end of self. (The first form can be also be applied to sets, which are not Indexable.)

    addFirst (new: T) −> List[[T]]
    // adds new as the first element of self. Changes the index of all of the existing elements.

    addAllFirst (news: Iterable[[T]]) −> List[[T]]
    // adds news as the first elements of self. Change the index of all of the existing elements.

    removeFirst −> T
    // removes and returns first element of self. Changes the index of the remaining elements.

    removeLast −> T
    // remove and return last element of self.

    removeAt (n: Number) −> T
    // removes and returns n^th element of self

    remove (element: T) −> List[[T]]
    // removes element from self. Raises a NoSuchObject exception if not.self.contains(element).
    // Returns self

    remove (element: T) ifAbsent (action: Block0[[Unknown]]) −> List[[T]]
    // removes element from self; executes action if any of them is not contained in self. Returns self

    removeAll (elements: Iterable[[T]]) −> List[[T]]
    // removes elements from self. Raises a NoSuchObject exception if any one of
    // them is not contained in self. Returns self

    removeAll (elements: Iterable[[T]]) ifAbsent(action:Block0[[Unknown]]) −> List[[T]]
    // removes elements from self; executes action if any of them is not contained in self. Returns self

    ++ (other:List[[T]]) −> List[[T]]
    // returns a new list formed by concatenating self and other

    addAll (extension: Iterable[[T]]) −> List[[T]]
    // extends self by appending extension; returns self.

    contains (sought: T) −> Boolean
    // returns true when sought is an element of self.
```

```
    == (other: Object) −> Boolean
    // returns true when other is a Sequence of the same size as self, containing the same elements
    // in the same order.

    sort −> List[[T]]
    // sorts self, using the < and == operations on my elements. Returns self.
    // Compare with sorted, which constructs a new list.

    sortBy (sortBlock: Block2[[T, T, Number]]) −> List[[T]]
    // sorts self according to the ordering determined by sortBlock, which should return −1 if its first
    // argument is less than its second argument, 0 if they are equal, and +1 otherwise. Returns self.
    // Compare with sortedBy, which constructs a new list.

    copy −> List[[T]]
    // returns a list that is a (shallow) copy of self

    reverse −> List[[T]]
    // mutates self in−place so that its elements are in the reverse order. Returns self.
    // Compare with reversed, which creates a new collection.
}
```

3.6 Sets

Sets are unordered collections of elements without duplicates. The == method on the elements is used to detect and eliminate duplicates; it must be symmetric.

```
type Set[[T]] = Collection[[T]] & type {
    size −> Number
    // the number of elements in self.

    add (element:T) −> Set[[T]]
    // adds element to self. Returns self.

    addAll (elements: Iterable[[T]]) −> Set[[T]]
    // adds elements to self. Returns self.

    remove (element: T) −> Set[[T]]
    // removes element(s) from self. It is an error if element is not present. Returns self.

    remove (element: T) ifAbsent(block: Block0[[Done]]) −> Set[[T]]
    // removes element from self. Executes action if element is not present. Returns self.

    removeAll (elems: Iterable[[T]])
    // removes elems from self. It is an error if any of the elems is not present. Returns self.

    removeAll (elems: Iterable[[T]]) ifAbsent (action: Block1[[T,Done]]) −> Set[[T]]
    // removes elems from self. Executes action apply(e) for each e in elems that is
    // not present. Returns self.

    contains (elem: T) −> Boolean
    // true if self contains elem

    includes (predicate: Block1[[T,Boolean]]) −> Boolean
    // true if predicate holds for any of the elements of self

    find (predicate: Block1[[T,Boolean]]) ifNone (notFoundBlock: Block0[[T]]) −> T
```

// returns an element of self for which predicate holds, or the result of applying notFoundBlock is there is none.

copy −> Set[[T]]
// returns a copy of self

** (other:Set[[T]]) −> Set[[T]]
// set intersection; returns a new set that is the intersection of self and other

−− (other:Set[[T]]) −> Set[[T]]
// set difference (relative complement); the result contains all of my elements that are not also in other.

++ (other:Set[[T]]) −> Set[[T]]
// set union; the result contains elements that were in self or in other (or in both).

isSubset (s2: Set[[T]]) −> Boolean
// true if I am a subset of s2

isSuperset (s2: Iterable[[T]]) −> Boolean
// true if I contain all the elements of s2

into (existing: Collection[[T]]) −> Collection[[T]]
// adds my elements to existing, and returns existing.
}

3.7 Dictionary

The type Dictionary[[K, T]] describes objects that are mappings from *keys* of type K to *values* of type T. Like sets and sequences, dictionary objects can be constructed using the class dictionary , but the argument to dictionary must be of type Iterable[[Binding]]. This means each element of the argument must have methods key and value. Bindings can be conveniently created using the infix :: operator, as in dictionary[[K, T]](k::v, m::w, n::x, ...). An empty dictionary can be created by evaluating emptyDictionary.

type Dictionary[[K,T]] = Collection[[T]] & **type** {
size −> Number
// the number of key::value bindings in self

at (key: K) put (value:T) −> Dictionary[[K,T]]
// puts value at key; returns self

at (k: K) −> T
// returns my value at key k; raises NoSuchObject if there is none.

at (k: K) ifAbsent (action:Block0[[T]]) −> T
// returns my value at key k; returns the result of applying action if there is none.

containsKey (k: K) −> Boolean
// returns true if one of my keys == k

contains (v: T)
containsValue (v: T)
// returns true if one of my values == v

removeAllKeys (keys: Iterable[[K]]) −> Self

```
    // removes all of the keys from self, along with the corresponding values. Returns self.

    removeKey (key: K) −> Self
    // removes key from self, along with the corresponding values. Returns self.

    removeAllValues (removals: Iterable[[T]]) −> Self
    // removes from self all of the values in removals, along with the corresponding keys. Returns self.

    removeValue (removal: T) −> Self
    // removes from self removal, along with the corresponding keys. Returns self.

    keys −> Iterable[[K]]
    // returns my keys as a lazy sequence

    values −> Iterable[[K]]
    // returns my values as a lazy sequence

    bindings −> Iterable[[Binding[[K, T]]]]
    // returns my bindings as a lazy sequence

    keysAndValuesDo (action: Block2[[K, T, Object]]) −> Done
    // applies action, in arbitrary order, to each of my keys and the corresponding value.

    keysDo (action: Block2[[K, Object]]) −> Done
    // applies action, in arbitrary order, to each of my keys.

    valuesDo (action: Block2[[T, Object]]) −> Done
    do (action:Block2[[T, Object]]) −> Done
    // applies action, in arbitrary order, to each of my values.

    copy −> Self
    // returns a new dictionary that is a(shallow) copy of self

    asDictionary −> Dictionary[[K,T]]
    // returns self

    ++ (other:Dictionary[[K,T]]) −> Dictionary[[K,T]]
    // returns a new dictionary that merges the entries from self and other.
    // A value in other at key k overrides the value in self at key k.

    −− (other:Dictionary[[K,T]]) −> Dictionary[[K,T]]
    // returns a new dictionary that contains all of my entries except for those whose keys are in other

}
```

3.8 Iterators and for loops

Collection that implement the type Iteratable[[T]] implement the internal and external iterator patterns, which provide for iteration through a collection of elements of type T, one element at a time. The method do() and its variant do()separatedBy() implement internal iterators, and method iterator returns an external iterator object, with the following interface:

```
type Iterator[[T]] = type {
    next −> T
```

```
    // returns the next element of the collection over which I am the iterator;
    // raises the Exhausted exception if there are no more elements.
    // Repeated request of this method will yield all of the elements of
    // the underlying collection, one at a time.

    hasNext −> Boolean
    // returns true if there is at least one more element,
    // i.e., if next will not raise the Exhausted exception.
}
```

Multiple iterators can exist on the same collection, for example, multiple iterator objects and multiple dos can be used to read through a file. Requesting next on one iterator advances its conceptual position, but does not affect other iterators over the same collection; nor does requesting do on a collection disturb any iterator objects. However, *it is an error to modify a collection object while iterating through it*. If you implement your own iterator, it is good practice to detect this error and raise ConcurrentModification.

For-do loops on Iterable objects are provided by standard Grace. The method for()do() takes two arguments, an Iterable collection and a one-parameter block body. It repeatedly applies body to the elements of collection. For example:

```
  def fruits = sequence ["orange", "apple", "mango", "guava"]
  for (fruits) do { each −>
     print(each)
}
```

The elements of the sequence fruits are bound in turn to the parameter each of the block that follows do, and the block is then executed. This continues until all of the elements of fruits have been supplied to the block, or the block terminates the surrounding method by executing a **return**.

for()do() is precisely equivalent to requesting the do method of the Iterable, which is usually both faster and clearer:

```
  fruits.do { each −>
     print(each)
  }
```

A variant for()and()do() allows one to iterate through two collections in parallel, terminating when the smaller is exhausted:

```
  def result = list [ ]
  def xs = [1, 2, 3, 4, 5]
  def ys = ["one", "two", "three"] for (xs) and (ys) do { x, y −>
  result.add(x::y) }
```

After executing this code, result == [1::"one", 2::"two", 3::"three"].

The need for external iterators becomes apparent when it is necessary to iterate through two collections, but not precisely in parallel. For example, this method merges two sorted Iterables into a sorted list:

```
method merge [[T]](cs: List[[T]]) and (ds: List[[T]]) −> List[[T]] {
    def cIter = cs.iterator
    def dIter = ds.iterator
    def result = emptyList

    if (cIter.hasNext.not) then { return result.addAll(ds) }
    if (dIter.hasNext.not) then { return result.addAll(cs) }

    var c := cIter.next
    var d := dIter.next
```

```
    while {cIter.hasNext && dIter.hasNext} do {
        if (c <= d) then {
            result.addLast (c)
            c := cIter.next
        } else {
            result.addLast (d)
            d := dIter.next
        }
    }

    if (c <= d) then {
        result.addAll [c,d]
    } else {
        result.addAll [d,c]
    }
    while {cIter.hasNext} do { result.addLast (cIter.next) }
    while {dIter.hasNext} do { result.addLast (dIter.next) }
    result
}
```

3.9 Primitive Array

Primitive arrays can be constructed using primitiveArray.new(size) where size is the number of slots in the array. Initially, the contents of the slots are undefined. Primitive arrays are indexed from 0 though size − 1. They are intended as building blocks for more user-friendly objects. Most programmers should use list, set or dictionary rather than primitiveArray.

```
type Array[[T]] = {
    size −> Number
    // return the number of elements in self

    at (index: Number) −> T
    // return the element of array at index

    at (index: Number) put (newValue: T) −> Done
    // update element of list at given index to newValue

    sortInitial (n: Number) by (sortBlock: Block2[[T, T, Number]]) −> Boolean
    // sorts elements 0..n. The ordering is determined by sortBlock, which should return −1
    // if its first argument is less than its second argument, 0 if they are equal, and +1 otherwise.

    iterator −> Iterator[[T]]
    // returns iterator through the elements of self. It is an error to modify the array while
    // iterating through it.
}
```

4 Built-In Libraries

4.1 Random

The random module object can be imported using **import** "random" **as** rand, for any identifier of your choice, e.g. rand. The object rand responds to the following methods.

```
between0And1 -> Number
// A pseudo-random number in the interval [0..1)
between (m: Number) and (n: Number) -> Number // A pseudo-random number in the interval
    [m..n)
integerIn (m: Number) to (n: Number) -> Number // A pseudo-random integer in the interval [m
    ..n]
```

4.2 Option

The option module object can be imported using **import** "option" **as** option, for any identifier of your choice, e.g. option. The object option responds to the following methods.

```
type Option[[T]] = type {
  value -> T
  do (action:Block1 T, Done ) -> Done isSome -> Boolean
  isNone -> Boolean
}
some T (contents:T) -> Option T
// creates an object s such that s.value is contents, s.do(action)
// applies action to contents, isSome answers true and isNone answers false

none T -> Option T
// creates an object s such that s.value raises a ProgrammingError,
// s.do(action) does nothing, isSome answers false and isNone answers true
```

4.3 Sys

The sys module object can be imported using **import** "sys" **as** system, for any identifier of your choice, e.g. system. The object system responds to the following methods.

```
type Environment = type {
  at (key: String) -> String
  at (key: String) put (value: String) -> Boolean
  contains (key: String) -> Boolean
}

argv -> Sequence String
// the command-line arguments to this program

elapsedTime -> Number
// the time in seconds, since an arbitrary epoch. Take the difference of two

elapsedTime
// values to measure a duration.

exit (exitCode: Number) -> Done
// terminates the whole program, with exitCode.

execPath -> String
```

```
// the directory in which the currently-running executable was found.

environ -> Environment
// the current environment.
```