Data Types

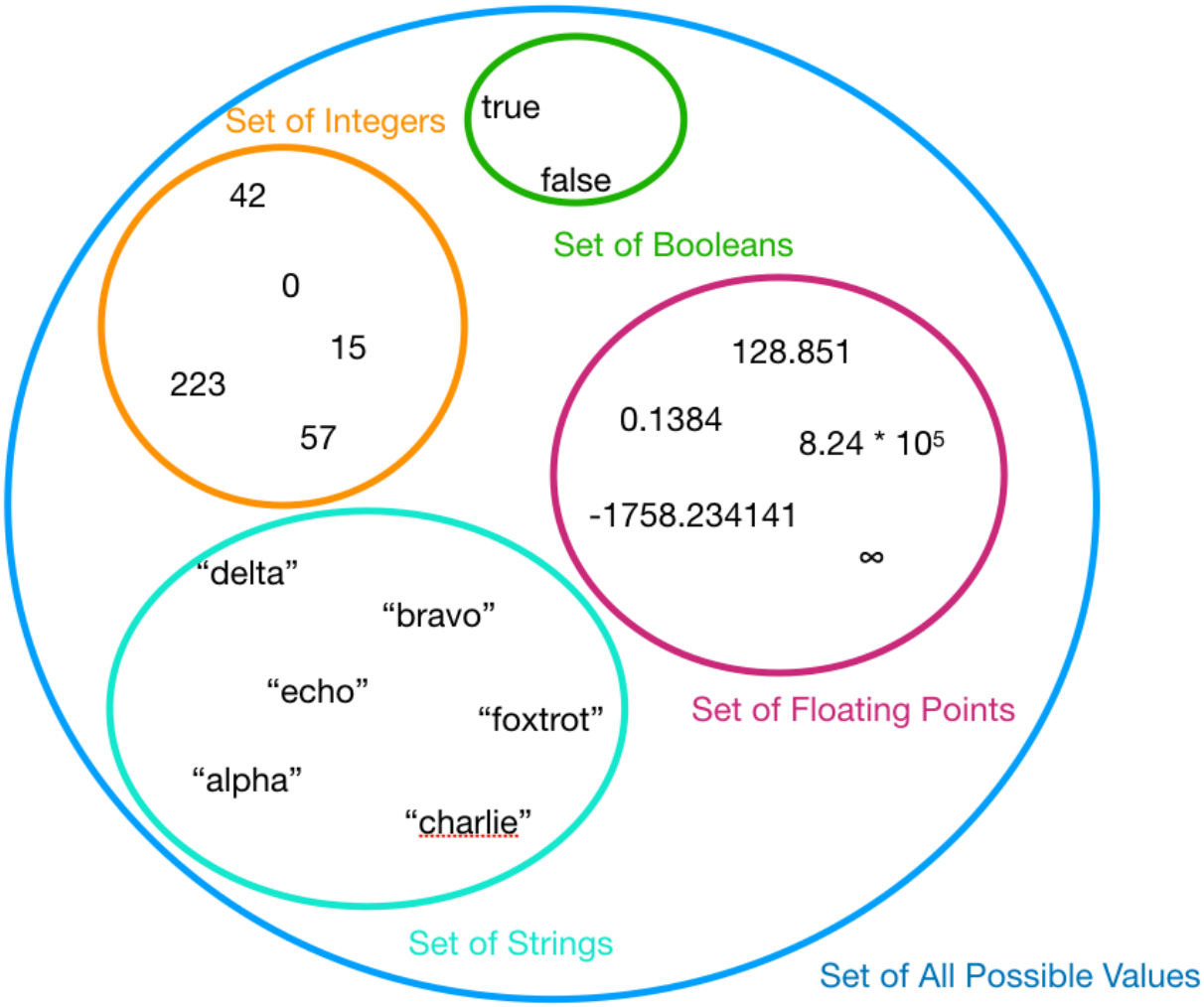
Data types (or sometimes just “types”) provide a means to classify data so that the REPL or ECRD environment can properly interpret the data. A data type provides the connection between the binary format in which all data is ultimately represented in a digital computer (whether that be in main memory, a register, on disk, or in transit across a network) and the representation of that data for humans.

Let’s consider a few bytes in memory:

07AE:	0	1	0	0	0	0	0	1

07AF:	0	1	0	1	1	0	0	0

The byte at memory location 0x07AE might represent the integer value 65. It also might represent the ASCII (a means of encoding characters into a byte) character ‘A’. The byte at memory location 0x07AF might represent the integer 88. It also might represent the ASCII character ‘X’. Or, perhaps, the two bytes form a single word representing a “half precision” floating point number. Another possibility is that the two bytes form a string beginning with the letters “AX”. Either may also be a Boolean value or part of an RGB (Red-Green-Blue) description of a pixel. Without context, the binary digits are meaningless. It is the **data type in conjunction with the binary digits** that provide meaning. Another way of thinking about this is that a data type **constrains the set of values** in the mapping from binary digits to a meaningful, human representation to a (relatively) small set of discrete elements, where each element is of the same type as all other members of the set. (See diagram on next page.)



L-Values and R-Values

An **L-Value** refers to an object that persists beyond a single expression, that is, it is addressble. The name comes from the typical manner in which assignment statements are written in most programming languages, with the variable receiving the value on the left:

```
x = 5 + 4
```

In this case, the variable "x" is an L-value which is being assigned the value 5+4, or 9. "x" refers to a location in memory which will contain the value 9, and that location will continue to persist after execution of this statement completes. For example, the next statement may again refer to "x", as in:

```
print(x)
```

Note that in most computer languages, the variable to be assigned is on the **left** of the assignment operator. In these cases, it would not be possible to assign to a variable on the right, as in:

```
5 + 4 = x
```

An **R-Value** refers to some object which **does not** persist after execution of the statement, that is, it is a temporary value. Think back to our original statement:

```
x = 5 + 4
```

In this case, 5+4 is an R-Value. It isn't possible to assign a value (that is, there is no persistent storage associated with) 5+4.

Constants

A **Constant** is a value which will not be altered as a program executes. A **Named Constant** is a constant associated with an identifier which can subsequently be used to refer to the value of the constant. In the below examples, *pi, e,* and g_0 are all named constants:

```
let pi = 3.14159265358979323846
```

```
let e = 2.718281828459
```

```
let g₀ = 9.80665  (in m/s²)
```

Named Constants assist programmers to add clarity to their code. It is clearer to both the programmer herself and others that later read the code that **pi** refers specifically to π and not another number that happens to start with 3.14. It also helps to eliminate mistakes due to repeatedly typing or copying and pasting a constant. Further, if a mistake is discovered in a named constant it is necessary only to correct the definition of the named constant in a single location rather than attempt to search for the value throughout all of the source code. Finally, declaring Named Constants help to protect the programmer by enabling compile-time checks.

While Named Constants are helpful for programmers they are also helpful to both REPL and ECRD environments as well by enabling optimizations that otherwise wouldn't be available.

Constants which are not named but rather expressed directly in the source code are called **Literal Constants.** For example: 145.6, "apple", 185, -123, and *false* are all examples of literal constants.

Operators

In programming, **Operators** provide us with the ability to write many expressions in a manner generally familiar to us from algebra. Operators can be grouped by **arity**, that is, the number of

operands on which the operator will function. For example, the expression 4+5 makes use of a binary operator (addition) which takes two operands, in this case a 4 and a 5. (An arity of one is often called unary while an arity of two is often called binary.)

Some common operators are listed in the table below:

Name	Symbol (Swift)	Arity	Type
Assignment	=	2	General
Unary Plus (Positive)	+	1	Arithmetic
Unary Minus (Negative)	-	1	Arithmetic
Addition	+	2	Arithmetic
Subtraction	-	2	Arithmetic
Multiplication	*	2	Arithmetic
Division	/	2	Arithmetic
Remainder	%	2	Arithmetic
Logical NOT	!	1	Boolean
Logical AND	&&	2	Boolean
Logical OR	||	2	Boolean
Equal to	==	2	Comparison
Not equal to	!=	2	Comparison
Greater than	>	2	Comparison
Greater than or equal to	>=	2	Comparison
Less than	<	2	Comparison
Less than or equal to	<=	2	Comparison

Some languages also support **Compound Assignment** operators, that is, they perform an operation on an L-Value and then store the result in the same L-Value. For example, the statements:

```
var x = 9
x += 2
```

assign the integer value of 9 to “x”, and then increment “x” by 2, resulting in a new value for “x” of 11. This is exactly equivalent to:

```
var x = 9
x = x + 2
```