Variables

Creating Variables

Swift uses variables to store and refer to values by an identifying name. Swift also makes extensive use of variables whose values cannot be changed. These are known as constants. Note that a constant is a kind of variable, with the limitation that its value cannot change.

A variable or constant must be declared before it can be used.

```
// Declaring a constant
let freezing = 32

// Declaring a variable
var currentTemperature = 71
```

A constant is declared using the keyword *let*, while a variable is declared using the keyword *var*.

Once a variable is declared, it can be used in subsequent instructions to refer to a stored value. For example, here a new variable degreesAboveFreezing is declared using the previously declared freezing, and currentTemperature. The value of degreesAboveFreezing is set to the result of the expression “value of currentTemperature minus value of freezing.”

```
var degreesAboveFreezing = currentTemperature - freezing
```

The value of a constant cannot be changed once it is set, whereas a variable can be set to a different value in the future.

```
var friendlyWelcome = “hello!”

friendlyWelcome = “bonjour!”
```

Constants and variables associate a name with a value of a particular type. Swift is a type-safe language, which means the language helps programmers to be clear about the types of values code can work with in order to avoid errors.

```
// Create a Double (Floating point) value
var angle = 0.0

// Create an Int (Integer) value
var iconWidth = 100
```

If you don’t specify the type of value you need directly, Swift uses ***type inference*** to work out the appropriate type. Type inference enables a compiler to deduce the type of a particular

expression automatically when it compiles your code, and set the variable's type based on that deduction.

A nice characteristic of type inference is its brevity. There are few characters and words required. However, it isn't explicit. A reader of the code must look at the expression and understand the result type to know the variable type. Sometimes the type result should be obvious, however there are times when it may be unclear.

You can optionally provide a ***type annotation*** when you declare a variable or constant in order to be clear about the kind of value the variable can store. A type annotation consists of a colon after the variable name, followed by a space, followed by the name of the type to use.

```
// Create a variable with type String
var welcomeMessage: String

// Create a variable with type Int
var playerScore: Int
```

An initial value can still be set when using type annotation.

```
// Create a variable with type annotation and set value.
var playerScore: Int = 0
```

Variable names (or other Identifiers) in Swift cannot contain whitespace characters, mathematical symbols, arrows, private-use (or invalid) Unicode code points, or line and box drawing characters. Nor can they begin with a number, although numbers may be included elsewhere within the name.

Because of this restriction, it is common to see "CamelCase" formatting of names. In this convention multiple words are smashed together without spaces, but the first letter of each word is capitalized to increase readability. In addition, although not required by Swift, a common idiom is to start all variable names with a lowercase letter. This is done to differentiate the variables names from type names, which are always capitalized.

```
// A couple CamelCase name examples

var avatarHeight: String

var rotationRate: Int
```

Three aspects of every variable

Every variable has three characteristics. It has a name, a type, and a value. The name is the identifier that we use to access the value. The type describes what kind of value the variable can store. And the value is the actual prize - it is the object that is stored for some future use in the program.

Types

Here we will discuss a few common types. There are many more types than discussed here, and later in the semester you will learn how to create your own custom types.

Int

Integers are whole numbers with no fractional component, such as 42 and -23. Int is the standard integer type in Swift. There are other integer types with different storage sizes and providing unsigned values.

Integers are either signed (positive, zero, or negative) or unsigned (positive or zero). Int provides signed values, while the type UInt is the standard unsigned type.

Integer computation does not create a fractional amount. Consider this expression

6 / 4

Read that as “six divided by four”. You might expect the result would be 1.5, however, 1.5 is a floating point value. Integer division will throw the remainder value away, and the answer will be 1 (a whole number). It is possible to find out the remainder value. (“What is the remainder of six divided by four”) That expression is written:

6 % 4

And the result is 2. We will discuss operators more in a later assignment.

Double

Floating point numbers are numbers with a fractional component, such as 3.14159, 0.1, and -273.15. Floating point types can represent a much wider range of values than integer types, and can store numbers that are much larger or smaller than the same sized integer type. However, there are always tradeoffs! CPUs are able to perform integer computations faster than floating point computations due to the additional complexity involved for floats. Also, floating point values only provide an approximate precision for some values, which can be an important factor for some applications.

Double is the Swift default type for floating point values. It provides a 64-bit floating point value. Another type you may run into is called Float, which provides a 32-bit floating point value.

Why use the name Double for the default floating point type? The reason is because of historical use of this name. The extra size and computation time for floating point values is significant, and was commonly a greater concern in the past when computers were slower than they are today. (It is still a concern for many applications.) Float was (historically) commonly the name for the standard floating point value - 32 bits in size. If applications needed better precision, they would use the “Double precision” type that is 64 bits in size. This is the Double type. Modern microprocessors process 64 bit values by default, so there isn’t the same incentive to make 32 bit floating point values the default standard.

Bool

Swift has a basic Boolean type called Bool. Boolean values are referred to as logical, because they can only ever be the value true or false.

```
let orangesAreOrange = true
let turnipsAreDelicious = false
```

String

A String is a sequence of characters. We will look into the details of characters and strings at a later point in time.

```
let city = “Tempe”
```

Type Safety

Swift performs type checks when compiling your code and flags any mismatched types, or any types that can’t be combined without ambiguity, and marks them as errors. This enables you to catch and fix errors as early as possible when writing your program.

```
// alpha is of type Int
let alpha = 3
// beta is of type Double
let beta = 0.14159

// This will produce an error
let pi = alpha + beta
```

This example produces an error.

> **Binary operator ‘+’ cannot be applied to operands of type ‘Int’ and ‘Double’**

I’ll attempt to provide a more verbose explanation for this somewhat terse error message.

> **You are using the addition operator ‘+’ with two values. However, the left value is an Int and the right value is a Double. This is an ambiguous situation - do you want to do integer math, or floating point math?**

Here is one potential solution, if I want to perform floating point math in this expression.

```
let pi = Double(alpha) + beta
```

The part that says Double(alpha) will convert the Int value of alpha into a Double value.

Names

Names are surprisingly important in creating good, readable programs. It has been said, anyone can write a program that a computer can understand. The challenge is writing a program that a person can understand. The use of good names is critical to writing understandable code. You will need to make names when making variables, functions, structures, and classes. Here are a few notes regarding names and code.

You are required to avoid “key” words that are reserved by the programming language. For example, Swift reserves the use of the word “let” for declaring a constant.

Be thoughtful about your names, and choose names that indicate the purpose or intent of the variable (or function).

Picking names is an art, not a science. You are trying to choose a good name to be read by another human. The Swift compiler will not be confused or lose track of the names that you select. Here is an illustration of the thought process in selecting names. Suppose you need a variable for the horizontal position of the front wheel in a drawing of a vehicle. First, you might choose a simple name.

```
x
```

Many mathematicians might choose this name. It is a common name to use in a math formula, and X commonly refers to the X-axis, which usually is horizontal. However, that name x is very generic. It doesn’t indicate much regarding the purpose in your drawing, except an idea about its dimensional orientation. Another choice could be

```
circleX
```

Because the wheel is a circle, and that is the shape that will be used in drawing the wheel. Again, circleX is also somewhat generic. It doesn’t indicate the purpose of the circle in the drawing. I like this choice better

```
frontWheelX
```

This name is longer, but it makes a clear statement about the purpose of the value. It will make the program more readable.