```
###
    
    The programs in this document cover:
    
        • Objects / Complex Data
        • Object Literals
        • References

###

```

```
########################################
######################################## Objects
########################################

######################################## P1
# In addition to primitive data types there are "complex data types" or "data structures".
# The most common type is an "Object"
# You can think of it as a collection of variables.

pet = {}          # The curly brackets are literally an Object, an "Object Literal".
pet.name = "Sam"  # Create a variable, 'name', "on the object".
pet.type = "Cat"  # A variable on an object is called a "property"
                  # the Object.property notation is called "Dot Notation" or "Dot Syntax"
                  # It is common in many languages.

print(pet.name)   # Properties work EXACTLY like variables.
print(pet.type)   # reading the data
pet.type = "Old cat"    # overwriting the data
print(pet.type)

```

```
########################################## P2
# Objects allow us to create structured data.

pet1 = {}
pet1.type = "Cat"
pet1.name = "Sam"

pet2 = {}
pet2.type = "Dog"
pet2.name = "Ralph"

print(pet1.name)
print(pet1.type)
print(pet2.name)
print(pet2.type)
```

```
######################################## P3
# You can define an object's properties IN the Object Literal.
# HOWEVER, instead of using '=' we use ':'
# This is called "Object Literal Notation" opposed to "Dot Notation" or "Dot Syntax".

pet = {name:"Sam",type:"Cat"}    # each 'property:value' is called a "property value pair".
                                 # there are two "property value pairs" in this object literal.
print pet

# The reasons for using ':' instead of '=' are partly historical.
```

```
######################################## P4
# You can create nested object structures, or hierarchies.

pet = {name:"Sam Jr."}
pet.parent = {name:"Sam Sr."}
pet.parent.parent = {name:"Ol papa Sam"}

print(pet.parent.name)
print(pet.parent.parent.name)
```

```
70  ######################################## P5
71  # You can also create objects with nested structures on one line using "Object Literal Notation"
72
73  pet = {name:"Sam",type:"Cat",age:3,parent:{name: "Sam Sr."}}
74
75  print(pet.name)
76  print(pet.type)
77  print(pet.age)
78  print(pet.parent.name)
79
80
```

```
######################################## P6
# Nested object structures are common, and putting them on one line is messy.
# CoffeeScript has a shorthand.
# It looks like an outline.
# It uses tab-indents and line returns.
# The exact tab number of tab-indents are important
# Instead of using curly brackets and commas, 'property:value' pairs can go on a new line, with a tab-indent.

pet =          # variable declaration and assignment, the line return indicates 'start new object'
    name:"Sam"     # a property value pair, and the beginning of the object
    type:"Cat"     # another property value pair. It belongs to the same object
    age:3          # another
    parent:        # another property, but the value is...
        name: "Sam Sr."    # A NEW object

print(pet.name)
print(pet.type)
print(pet.age)
print(pet.parent.name)

# Note the tab indentations.
# This is another example of "significant whitespace"
```

```
105 ########################################
106 ######################################## Object Literals
107 ########################################
108
109 ######################################## P7
110 # Some functions accept objects as arguments
111
112 pet = {name:"Mike Hat",type:"Dog"}
113 print(pet)
114
115
```

```
116 ######################################## P8
117 # Here is the same program with the Object Literal Shorthand
118
119 pet =
120     name:"Mike Hat"
121     type:"Dog"
122
123 print(pet)
124
125
```

```
126 ######################################## P9
127 # We can also use an Object Literal directly as an argument
128
129 print({name:"Mike Hat",type:"Dog"})
130
131
```

```
######################################## P10
# We can also use an Object Literal Shorthand when doing this.

print(
    name:"Mike Hat"      # The line return AND the tab together indicate 'new object literal'
    type:"Dog")

# This program is EXACTLY the same as the previous program
```

```
######################################## P11
# In CofeeScript, it is very common to use function call shorthand AND Object Literal Shorthand together

print                   # The line return and tab indicate new object literal
    name:"Mike Hat"     # Because there is an argument, the print function needs no parenthesis.
    type:"Dog"


# This program is EXACTLY the same as the previous two programs
# Yes, it may seem weird that there are three ways of doing the same thing.
# Think about it like this...
# CoffeeScript programmers really just uses the third approach, but the reason that approach exists
# is because of the earlier ways of doing things.


```

```
156 ########################################
157 ######################################## Built in variables
158 ########################################
159
160 ######################################## P12
161 # Framer and JavaScript come with many built in variables.
162 # They are often organized into objects.
163 # In most cases, you can not put data in them, you can only read their values.
164 # Here are two of the more useful ones.
165
166 print Screen.width
167 print Screen.height
168
```

```
169 ########################################
170 ######################################## References
171 ########################################
172
173 ######################################## P13
174 # Assigning Objects to variables is DIFFERENT from assigning primitive data to variables.
175 # Object data is always given it's OWN UNIQUE memory area.
176 # When an Object is assigned to a variable, the variable "refers to" this memory.
177 # The variable has a "reference". A reference is NOT the data. It only "refers to" the data.
178 # This means multiple variables can "refer to" the SAME Object Data.
179
180 # Create an object literal with property 'power' that holds data "flight" and assign a REFERENCE to this object to the variable 'clark'.
181 clark = {power:"flight"}
182 superman = clark                # Copy the clark's REFERENCE and assign the copy to the new variable 'superman'
183 superman.weakness = "kryptonite" # Create new property 'weakness' on the object data.
184
185 print(superman.power)
186 print(clark.power)
187 print(superman.weakness)
188 print(clark.weakness)     # Clark and Superman "refer to" the SAME thing!
189
```

```
190 ######################################## P14
191 # Note the difference when using primitive data.
192
193 clark = "kryptonian"    # Assign data to variable 'clark'
194 superman = clark        # COPY data "kryptonian" to variable 'superman'
195 clark = "Kent"          # Assign new data to variable 'clark'.
196                         # superman variable unaffected.
197
198 print(clark)
199 print(superman)
200
```

```
######################################## P15
# If we assign primitive data to a variable that has REFERENCE,
# the REFERENCE is overwritten, NOT the object data being referred to.
# The DATA will still exist in memory,
# and the variable will have its own memory like normal.

clark = {power:"flight"}
superman = clark
clark = "Kent"  # This overwrites the -reference- NOT the -object data-.
                # The object still exists.
                # The variable superman still refers to it.

print(clark)
print(superman)


```

```
######################################## P16
# Advanced Technical Note:
# If we overwrite every reference to an object
# The data will STILL persist in memory,
# Even though we have no way to get at it!

holyGrail = {type:"cup"}
holyGrail = "Lost to us"

print(holyGrail)

# The {type:"cup"} object is still in memory, even though we can't access it!
# At this point, it's up to the computer to find and remove this data so we don't run out of memory!
# This find and remove is called "garbage collection".
# It is built into the environments that run Javascript/Coffeescript.
# However, other languages require you to manage memory more explicitly.



########################################
######################################## End
########################################
```