EECS3311 Introductory Tutorial: Accompanying Notes

Jackie Wang

September 14, 2020

Video Tutorials

Each section in this set of notes:

- Summarizes main points discussed in the corresponding part of the following introductory tutorial on Eiffel (a programming language and a design method) and its tool:

  https://www.youtube.com/playlist?list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN

- Contains a link to the corresponding video.

How Should You Use this Set of Notes?

It is advised that after watching/studying each part of the tutorial series above, go over its written summary to review and reflect, before you proceed to the next video.

Roadmap

1 Unit Testing & Console Outputting, Run vs. Run Workbench System

Link: https://www.youtube.com/watch?v=vINZxvljR3c&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=2

1. Generate a starter project `birthday_book` from:

   https://www.eecs.yorku.ca/~eiffel/eiffel-new/

   (1.1) Assumption: `mathmodels.zip` already downloaded and unzipped, and the location of directory `mathmodels` has been set to variable `MATHMODELS`.

   (1.2) Download and unzip the project archive `birthday_book.zip`.

   (1.3) Launch Eiffel Studio: `estudio &`, add the project, and compile.

2. For the generated `ROOT` class:

   (2.1) Multiple inheritance: it inherits from both

   `ES_SUITE` (for unit testing) and `ARGUMENTS_32` (for console outputs).

   (2.2) Delete `inherit ARGUMENTS_32` and `print(...)` from the `make` command.

   (2.3) Instead, create an alternative root class: `ROOT_CONSOLE`.

   (2.4) Delete the generated test cases in `TEST_EXAMPLE`. Then type a default boolean case, which always passes, there:

   ```
   t0: BOOLEAN
     do
       comment("t0: a test always passing")
       Result := true
     end
   ```

3. Show how to set the root class between `ROOT` and `ROOT_CONSOLE`:

   (3.1) When root class is set to `ROOT_CONSOLE` (for console outputs), always hit `Run` to see the execution result (contract violations, expected or not, will break the execution flow).

   (3.2) When the root class is set to `ROOT` (for unit testing):

   - Hit `Run Workbench System` to see the test report (some contract violations may be expected and will not cause a *red bar*).
   - Hit `Run` to execute each test case as a normal feature (contract violations, even if expected, will be considered as "exceptions" and break the execution flow). This option is useful when you need to debug because there's a *red bar* from your `ES_TEST` test report.

   Notes:

   - To debug, you should use breakpoints/debugger in the unit-testing mode. Simply trying console outputs does not scale to larger projects.

2 Setting Tools Layout, Navigating Library Classes

Link: https://www.youtube.com/watch?v=fsz8yiNzvcY&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=3

1. Set panels:
   (1.1) Right: `Groups` (≈ package explorer in Eclipse) & `Features` (≈ outline in Eclipse).
   (1.2) Bottom: `Feature` (for setting break points)
   (1.3) To reset the tools layout: `View` → `Tools Layout` → `Reset Tools Layout`
2. Navigate the Eiffel base library:
   (2.1) Search for a library class (verbatim vs. regular expression e.g., `*SORT*`, `*LIST*`) from the `Class` textfield.
   (2.2) Skim through the available features in the `Features` panel.
   (2.3) Click on the feature in `Features` panel, or type it in the `Feature` textfield.
   (2.4) Show the *contract* view to see the interface of the class.
   (2.5) Show the *flat* view to see the "flattened" version, accumulating features and contracts from ancestor classes (e.g., flat view of `ARRAY`).
   - On the `Features` panel, see what the ancestors are.
   - On the editor panel, see what the accumulated contracts are (e.g., {`ARRAY`}`.item`).

   (2.6) Show *ancestor* and *descendants* of a library class (e.g., `ARRAY`).
   - Pay attention to the iconic hint about a class being deferred (abstract, partially implemented) or effective (concrete, fully implemented).
   - Exercise: Get a feel about the inheritance hierarchy of `LINKED_LIST`.
   - Exercise: Say you declare a variable `container: LIST`, what are the possible *dynamic* types of `container`?

3 BIRTHDAY: Variable Declarations, Features, Invariant

Link: https://www.youtube.com/watch?v=wDg-8frItEk&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=4

1. Create a new class: `BIRTHDAY`.
2. Create customized feature sections: e.g.,

   `Commands`, `Attributes`, `Queries`, `Equality`, `String Representation`

   Note. Each `feature` clause creates a bookmark for navigating within the class.
3. Add attributes `month` and `day`.

   (3.1) `INTEGER` denotes the set of integer values.

   (3.2) When you write `month: INTEGER`, it has the mathematical meaning:

   $$\texttt{month} \in \texttt{INTEGER}$$

   That is, at runtime, `month` stores a value from the `INTEGER` set.
4. Taxonomy:

   - *features*
     - *attributes* (storage)
     - *routines* (computation)
       - commands (no return values, side effects)
       - queries (return values, no side effects)
5. Always first think about the ***class invariant***: how should a legitimate `BIRTHDAY` object be characterized (in terms of its attribute values and query return values)?

   (5.1) `valid_day`, `valid_month`

4 BIRTHDAY: Static Queries, TDD, Boolean Test Case

Link: https://www.youtube.com/watch?v=OZJZITCh7PI&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=5

1. `valid_month` and `valid_day` are **not** enough because invalid dates such as June 31 will be considered as valid (i.e., June [6] is a valid month and 31 is a valid day).
2. Define class-level queries `is_month_with_31_days` and `is_month_with_30_days`.
   (2.1) Define preconditions and postconditions of these two queries.
   (2.2) Specially, writing `class` as a postcondition makes a feature class-level (static).
   (2.3) We supply implementations that satisfy the postconditions.
   - In this simple case, we may simply change `=` (equality) to `:=` (assignment).
   - Alternatively, we can implement using an array.
3. These two class-level queries can be invoked without having to create a `BIRTHDAY` object (e.g., `{BIRTHDAY}.is_month_with_30_days(...)`).
4. *Test Driven Development* (TDD):
   - Test as soon as a feature becomes *executable*.
   - Re-run all tests when a *change* is made. [**regression testing**]
5. Write a Boolean test case for these two class-level queries.
   - To add a boolean test (testing if a *normal scenario* happens as expected):
     In the constructor of a sub-class of `ES_TEST`, write: `add_boolean_case(agent f)`
     `f` is the boolean test query.

5 BIRTHDAY: Assertions, Logical Operator Precedence

Link: https://www.youtube.com/watch?v=1NX3ryQWV_g&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=6

1. When there are multiple, consecutive re-assignments to the `Result` of a Boolean test query, intermediate assertions are necessary.
   (1.1) If no, then the mistake of a poor supplier (e.g., the `is_month_with_31_days` query with a faulty implementation and with no postcondition) cannot be caught by the test case.
   (1.2) Intermediate `check ... end` assertions can catch such errors by causing *check assertion violations*.
   (1.3) Alternatively, If the `is_month_with_31_days` query had an appropriate postcondition, then the faulty implementation would have been caught as a *postcondition violation*.
   (1.4) But in general, we may **not** assume that an appropriate postcondition always exists, so intermediate `check` assertions are necessary.
2. Use these two class-level queries to write the class invariant:

   `valid_day_month_combination`.

   (2.1) For operator precedence (tightest to loosest): `and`, `or`, `implies`

6 BIRTHDAY: Syntax Overview of Classes and Features

Link: https://www.youtube.com/watch?v=Nyk7yaH2d5M&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=7

1. Classes are divided into various `feature` sections.
2. Always think first about the ***class invariant***: Under what circumstances should instances of a class be considered as valid (e.g., a valid bank account, a valid birthday).
3. A routine is either a command or a command:
   (3.1) A command or a query may have a list of input parameter declarations, separated by semi-columns (`;`).
   (3.2) A command has no return type.
   (3.3) A query has return type.
   - Imaginatively, for each query with a return type `T`:
   - There is a first line: `Result: T`
   - There is a last line: `return Result` [never write this, just imagine it!]
   - Consequently:
     - You are not allowed to use `Result` as the name of your own variable.
     - To implement a query, manipulate the pre-defined variable `Result`, in such way that it stores the desired value.

7 BIRTHDAY: Using a Command as a Constructor

Link: https://www.youtube.com/watch?v=ikz2LogoKI4&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=8

1. Declare command `make` and add its implementation.
2. Add a `create` clause and put `make` as one of the valid constructors.
3. Rename `TEST_EXAMPLE` as `TEST_BIRTHDAY`.
4. To create an object, you must use the `create` keyword:
   - (Version 1) e.g., `create bd.make(1, 31)`

     Alternatively, create an object using an anonymous object on the RHS:
   - (Version 2) e.g., `bd := create BIRTHDAY.make (1, 31)`

     The RHS of `:=` (assignment) is an anonymous object: `create BIRTHDAY.make (1, 31)`
   - Constrast how an anonymous is created in Java: `new Birthday(1, 31)`

     That is, the way a new object is created in Java is similar to Version 2 above:

     ```
     Birthday bd = new Birthday(1, 31)
     ```
   - *Cross Reference.* See Section 17 for specifying a dynamic type.
5. On the other hand, writing `bd.make(1, 31)` means: `bd` already points to some `BIRTHDAY` object (i.e., `bd` is *attached*), a change is to be made on that object, and no new object is to be created.
6. Show the difference between including and excluding `make` in the `create` clause.

   (6.1) Without explicitly adding `make` under a `create` clause:
   - Its use as a constructor is no longer valid (e.g., previously-compiled test no longer compiles).
   - It can **only** be used as a command with a context object (by writing e.g., `bd.make(...)`). In this case, **no** new object is being created.

   (6.2) By explicitly adding `make` under a `create` clause, it can then be used **both** as a command and as a constructor (by writing e.g., `create bd.make(...)`).

8 BIRTHDAY: Setting Breaking Points & Launching Debugger

Link: https://www.youtube.com/watch?v=5H-eOezK_QI&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=9

1. Set breakpoints within a boolean test case query on creating a valid birthday.

   e.g., {TEST_BIRTHDAY}.t_create_new_birthday

2. Debugging Scenarios:

   - Set a break point on a test case that fails.
   - Set a break point on a feature which you want to see how that line is called (from the stack trace).

3. To set a breakpoint:

   - (Approach 1): Switch to the `flat view` and type the feature in the `Feature` textfield.
   - (Approach 2): Right-click on the feature and drop it to the bottom `Feature` panel.

4. To launch the debugger, just click on `Run` and you're given 3 options to execute the code:

   - One step at a time
   - Step into a routine (query or command)
   - Step out of a routine (e.g., within the check of class invariant)

9 BIRTHDAY: Writing a Precondition Violation Test

Link: https://www.youtube.com/watch?v=kX1Idyj_ZgI&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=10

1. Write a **boolean test case** (a boolean query with no parameters) which creates a `BIRTHDAY` with invalid day and month.
2. A class invariant violation occurs (⇒ a *red bar*).
   (2.1) `Run` to show this class invariant violation on the stack trace.
   (2.2) To prevent the class invariant from happening, add ***precondition*** to `make`.
3. Re-running the test (via `Workbench System`),
   a precondition violation occurs (⇒ a *red bar*).
   (3.1) `Run` to show this precondition violation on the stack trace.
   (3.2) The precondition violation occurs as expected ⇒ We want a *green bar*.
4. Instead, we add a violation case for testing this expected precondition violation.
   (4.1) Re-running the test (via `Workbench System`), a precondition violation occurs but it's expected (⇒ a *green bar*).

10 BIRTHDAY: Writing a Postcondition Test

Link: https://www.youtube.com/watch?v=v2c6MBPhEz0&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=11

1. As an example, in order to test that the `day_set` postcondition violation occurs when it should, we need an incorrect implementation such that:
   - It violates the Boolean expression for `day_set`: `month = m`
2. Define a **violation test case** (a command with no parameters) which calls this version of incorrect implementation.
3. To add a violation test (testing if an *exceptional scenario* happens as expected):
   In the constructor of a sub-class of `ES_TEST`, write:
   ```
   add_violation_case_with_tag("tag_name", agent f)
   ```
   where `f` is the violation test command, and `tag_name` is the tag of the postcondition under test (spelt *verbatim*, e.g., `day_set`).
4. Test Driven Development (TDD): write a test for a routine as soon as it becomes executable.
   - violation tests for its precondition and postcondition
   - boolean tests for its implementation

11 BIRTHDAY: Reference Equality vs. Object Equality

Link: https://www.youtube.com/watch?v=AiiJSPLwBSk&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=12

1. Write a test that compare references (addresses) of two `BIRTHDAY` class using `=`.
2. Redefine the `is_equal` query from the `ANY` class (≈ the `boolean equals` method of the `Object` class in Java).
3. The top class `ANY` (≈ `Object` in Java) declares the equality query:

   ```
   is_equal(other: like Current)
   ```

   (3.1) It is different from what happens in Java: `boolean equals(Object obj)`.

   (3.2) It uses an *anchor type*: the type of `other` depends on what the current class is. This is a design choice of Eiffel base library, so as to avoid the trouble of doing what you have to do in Java:

   ```
   1 boolean equals(Object obj) {
   2   if(this == obj) { return true; }
   3   if(obj == null) { return false; }
   4   if(this.getClass() != obj.getClass()) { return false; }
   5   BIRTHDAY other = (BIRTHDAY) obj;
   6   return this.month == other.month && this.day == other.day;
   7 }
   ```

   (3.3) In Eiffel's case, the use of an *anchor type* allows you to focus on just **L6** in the Java code (assuming that the context object `Current` and the argument object `other` have the same dynamic type).

   (3.4) *Cross Reference.* See how anchor type is used in Section 23.

4. Show the use of ∼:

   (4.1) When `BIRTHDAY` does not redefine (override) `{ANY}.is_equal`,

   `bd1 ∼ bd2` is equivalent to `bd1 = bd2`.

   (Show this in debugger that the version of `{ANY}.is_equal` is called.)

   (4.2) When `{ANY}.is_equal` is redefined in `BIRTHDAY`,

   `bd1 ∼ bd2` is equivalent to `bd1.is_equal(bd2)`.

   (Show this in debugger that the version of `{BIRTHDAY}.is_equal` is called.)

12 `BIRTHDAY`: Logical Pattern for Invariant

Link: https://www.youtube.com/watch?v=VyDau1Vt77A&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=13

1. Can we replace `implies` by `and` in invariant `valid_day_month_combination`?

   That is, can the following revised invariant work?

```
valid_combination:
   (is_month_with_31_days (month) and 1 <= day and day <= 31)
   and
   (is_month_with_30_days (month) and 1 <= day and day <= 30)
   and
   (month = 2 implies 1 <= day and day <= 29)
```

2. No. Conceptually valid dates such as June 23 and January 12 will cause ***class invariant violations***.

   (2.1) `b1 and b2` evaluates to `false` if `b1` is `false`.

   (2.2) `b1 implies b2` evaluates to `true` if `b1` is `false`.

3. Write a test case to demonstrate this class invariant violation.

13 Basic Operations of Arrays

Link: https://www.youtube.com/watch?v=pv4AZcWYmmk&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=14

1. Use of `ARRAY` (with a test case):
   - (1.1) `make_empty`, `force` vs. `put`
   - (1.2) `lower`, `upper`, `count`, `valid_index` , `[...]`, `item(...)`
   - (1.3) By default, indices start with 1.
   - (1.4) `from ... until` loop using a loop counter `i`
2. Use of `object_comparison`:
   - (2.1) It affects queries related to *membership*: e.g., {`ARRAY`}`.has` and {`ARRAY`}`.occurrences`
   - (2.2) See definitions of {`ARRAY`}`.has` and {`ARRAY`}`.occurrences`
   - (2.3) Say there's an array `a1` with starting index 1. The behaviour `a1.has(x)` depends on the value of `a1.object_comparison`:
     - *a1.object_comparison* is `true`: `a1.has(x)` is equivalent to `(a1[1]` $\sim$ `x)` $\vee$ `(a1[2]` $\sim$ `x)` $\vee$ ... $\vee$ `(a1[a1.upper]` $\sim$ `x)`
     - *a1.object_comparison* is `false`: `a1.has(x)` is equivalent to `(a1[1] = x)` $\vee$ `(a1[2] = x)` $\vee$ ... $\vee$ `(a1[a1.upper] = x)`
3. Use of $\sim$ for collections:
   - See the definition of {`ARRAY`}`.is_equal`
   - Say two arrays `a1` and `a2`: `a1` $\sim$ `a2` evaluates to `true` if:
     - *a1.lower* = *a2.lower*
     - *a1.count* = *a2.count*
     - *a1.object_comparison* = *a2.object_comparison*
       Say *a1.lower* is 1.
     - `a1[1]` $\sim$ `a2[1]` $\wedge$ `a1[2]` $\sim$ `a2[2]` $\wedge$ ... `a1[a1.upper]` $\sim$ `a2[a2.upper]`

14 Basic Operations of Linked Lists

Link: https://www.youtube.com/watch?v=1FGHBmilVP8&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=15

1. Use of `LINKED_LIST` (with a test case):
   - (1.1) `make`, `extend`
   - (1.2) `count`, `valid_index`, `is_empty`, `[...]`, `item`,
   - (1.3) `start`, `forth`, `after`
   - (1.4) By default, indices start with 1.
   - (1.5) `from ... until` loop using cursor operations: `start`, `forth`, `after`
2. Use of `object_comparison`:
   - (2.1) It affects queries related to *membership*: e.g., {`ARRAY`}`.has` and {`ARRAY`}`.occurrences`
   - (2.2) See definitions of {`ARRAY`}`.has` and {`ARRAY`}`.occurrences`
   - (2.3) Say there's a list `l1` with starting index 1. The behaviour `l1.has(x)` depends on the value of `l1.object_comparison`:
     - *l1.object_comparison* is **true**: `l1.has(x)` is equivalent to `(l1[1] ∼ x) ∨ (l1[2] ∼ x) ∨ ... ∨ (l1[l1.count] ∼ x)`
     - *l1.object_comparison* is **false**: `l1.has(x)` is equivalent to `(l1[1] = x) ∨ (l1[2] = x) ∨ ... ∨ (l1[l1.count] = x)`
3. Use of ∼ for collections:
   - See the definition of {`LINKED_LIST`}`.is_equal`
   - Say two arrays `l1` and `l2`: `l1 ∼ l2` evaluates to **true** if:
     - *l1.count = l2.count*
     - *l1.object_comparison = l2.object_comparison*
     - `l1[1] ∼ l2[1] ∧ l1[2] ∼ l2[2] ∧ ... l1[l1.count] ∼ l2[l2.upper]`

15 Use of `across` as Loop Instructions

Link: https://www.youtube.com/watch?v=zKT8UlqxIn0&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=16

1. Use of `across` (with test cases):
   (1.1) Code completion:
   `across as` (default, which denotes a cursor) vs. `across is` (denotes a value)
   (1.2) across over integer interval (e.g., `1 .. (a.count - 1)`)
   (1.3) across over iterable collection
   (1.4) Instruction: `across ... is ... loop ... end`

16 Use of `across` as Boolean Expressions

Link: https://www.youtube.com/watch?v=DYnrsLbj5CY&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=17

1. Use of `across` (with test cases):
   (1.1) Code completion:
   `across as` (default, which denotes a cursor) vs. `across is` (denotes a value)
   (1.2) across over integer interval (e.g., `1 .. (a.count - 1)`)
   (1.3) across over iterable collection
   (1.4) Boolean Expressions:
   `across ... is ... all ...` ($\approx \forall$) vs. `across ... is ... some ...` ($\approx \exists$)

17 BIRTHDAY_BOOK Class: Attributes, Constructor, Void Safety

Link: https://www.youtube.com/watch?v=jcpFE5FrsVA&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=18

1. Add attributes `count`, `names`, and `birthdays`.

   (1.1) `LIST[BIRTHDAY]` denotes the set of all possible lists (of size 0, 1, 2, ...), where each slot stores the reference of a *valid* `BIRTHDAY` object (i.e., one satisfying the invariant of `BIRTHDAY`).

   (1.2) When you write `birthdays: LIST[BIRTHDAY]`, it has the mathematical meaning:

   $$\texttt{birthdays} \in \texttt{LIST[BIRTHDAY]}$$

   That is, at runtime, `birthdays` stores the beginning address of an array from the `LIST[BIRTHDAY]` set.

2. Design principle: *Program from the Interface, Not from the Implementation*

   (2.1) The static type of `birthdays` is the deferred (abstract) class `LIST`.

   (2.2) In order to instantiate `birthdays`, we must use an effective (concrete) class that is a descendant of `LIST` (recall the exercise done previously on looking up the descendants of `LIST`!).

   (2.3) Say we instantiate it to a `LINKED_LIST` (called an ***implementation secret***), by writing:

   ```
   create LINKED_LIST[BIRTHDAY] birthdays.make
   ```

   - What goes between the curly brackets denotes the dynamic type of `birthdays`: `LINKED_LIST[BIRTHDAY]`.
   - Contrast how object creations are done (particularly for reference types) in Java:

     ```
     List<Birthday> birthdays;
     birthdays = new LinkedList<Birthday>();
     ```

     and in Eiffel:

     ```
     birthdays: LIST[BIRTHDAY]
     ...
     create {LINKED_LIST[BIRTHDAY]} birthdays.make
     -- equivalent to:
     -- birthdays := create {LINKED_LIST[BIRTHDAY]}.make
     -- (anonymous type)
     ```

3. ***Void Safety***. All referene attributes (`names` and `birthdays`, not `count`) must be initialized.

4. ***Void Safety***. Object variables are `attached` by default:

   (4.1) When we declare a variable of some reference type, e.g.:

   ```
   name: STRING
   ```

   ```
   names: ARRAY[STRING]
   ```

   (4.2) We implicitly declare to the Eiffel compiler—so that the compiler will perform *automatic* checks for us before allowing a given piece of code to be *executed*—that `name` and `names` will always be `attached`.

   (4.3) An object variable `x` being `attached` means that `x` is not `void` (i.e., non-null, or pointing to some object stored in the memory).

   (4.4) This is different from Java, where object variables are not always non-null (hence the notorious phenomenon of *NullPointerException*!). Moreover, there is no way for you to ask the Java compiler to perform static checks on your Java code, so that a *compilable* Java code is free from *NullPointerException*.

   (4.5) On the other hand, the *void safety* feature of the Eiffel compiler guarantees that, if your code passes it checks and compiles, there will be no *runtime errors* related to null pointers.

   ⇒ What's the catch?

   - In Java, you run into *NullPointerException* at runtime (meaning that there is no way for you to be aware of such an issue during development).
   - In Eiffel, you run into *compile-time* errors related to void safety, and you are forced to fix these compile-time errors before you can run your software.

     A typical *void-safety error* is the use of an uninitialized object variable `obj` as the context object of some call: e.g., `obj.f(...)`.
   - **Analogy**.
     - In Java, a problem of a plane's landing gear is discovered after its takeoff.
     - In Eiffel, the landing gear's problem would have been discovered and forced to fix before the takeoff.

   (4.6) More precisely:

   - Declaring object variables such as `String name` and `String[] names` in Java is as if you added an explicit modifier `detachable` in the case of Eiffel:

     `name: detachable STRING` and `names: detachable ARRAY[STRING]`.
     - In this case, the Java compiler let problems manifest themselves when a *NullPointerException* causes your program to crash at *runtime*, whereas the Eiffel compiler imposes constraints related to *void safety* at the *compile-time*.
     - In this case, both Java and Eiffel allow the developer to perform manual checks.
       - In Java, you write Boolean expressions such as `name != null` and `names != null`.
       - In Eiffel, you write Boolean expressions such as `attached name` and `attached names`.
   - Declaring objet variables that are always `attached` in Eiffel (e.g., `name: STRING` and `names: ARRAY[STRING]`) has no equivalent support in Java.

18 BIRTHDAY_BOOK: Class Invariant

Link: https://www.youtube.com/watch?v=3MnIO3oJWew&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=19

1. Always first think about the ***class invariant***: how should a legitimate BIRTHDAY object be characterized (in terms of its attribute values and query return values)?
2. consistent_counts
3. no_duplicate_names uses nested across.

19 BIRTHDAY_BOOK: Command add – Debugging Precondition

Link: https://www.youtube.com/watch?v=ubrFBmbURKI&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=20

1. Add precondition: name_does_not_exist.
   - **(Approach 1)** Use of {ARRAY}.has
   - Add a precondition violation test (e.g., adding a name which already exists).
   - In order to inquire membership in the names array, names.object_comparison must be set to true.
   - **(Approach 2)** Use of across ... is ... some ...
   - Conversion between $\forall$ and $\exists$: $(\forall x : P(X)) \equiv \neg(\exists x : \neg P(X))$.
2. ***The video also demonstrates a valuable process of debugging a test. Please make sure you take the time to understand and re-produce the process.***

20 BIRTHDAY_BOOK: Command add – Testing Postcondition

Link: https://www.youtube.com/watch?v=0PwlNxt9Lqc&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=21

1. Add postconditions, assuming that the new name and new birthday are always added to the end of the two linear structures.
2. Add an implementation that satisfies the postconditions.
3. Add a boolean test for testing the implementation of command add.
4. Add a postcondition violation test.
   - In order to test that the name_added postcondition violation occurs when it should, we need an incorrect implementation such that:
     - It satisfies all postconditions are satisfied: one_more_name and one_more_birthday
     - It violates the Boolean expression for name_added: `names[count] ~ p_n`

21 BIRTHDAY_BOOK: Get Birthday – detachable? [Supplier]

Link: https://www.youtube.com/watch?v=TTpbrlvDIOw&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=22

1. `get_birthday_book(name: STRING): BIRTHDAY`
   (1.1) ***Void Safety***. The return value `Result` is always *attached* and must thus be explicitly initialized.
   (1.2) The `from ... until ...` loop requires `Result` to be initialized.
2. `get_birthday_book(name: STRING): detachable BIRTHDAY`
   (2.1) ***Void Safety*** The return value `Result` is `detachable` and may thus be uninitialized, i.e., storing `void` by default.
   (2.2) The `from ... until ...` loop does not require `Result` to be initialized.
3. We define an auxiliary query `index_of_name` to help specify the postcondition.

22 BIRTHDAY_BOOK: Get Birthday – detachable? [Client]

Link: https://www.youtube.com/watch?v=eCxI48msewU&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=23

1. Write tests on `add` using `get_birthday`.
2. Write tests on `add` using `get_detachable_birthday`.
   (2.1) ***Void Safety*** Calling a feature on the return value of `get_detachable_birthday` requires the use of a developer's assertion:

```
check attached get_detachable_birthday(...) as ... then ... end
```

23 BIRTHDAY_BOOK: Exercise – Implement & Specify celebrate

Link: https://www.youtube.com/watch?v=ZeCqOIJulFM&list=PL5dxAmCmjv_4bxISJrwPoBrzVdCfVgVnN&index=24

1. The return value uses the anchor type: `remind (today: BIRTHDAY): like names`
   (1.1) It is as if you declared `remind (today: BIRTHDAY): ARRAY[STRING]`
   (1.2) What's the advantage of using an anchor type here?
   - The use of anchor type for `remind`'s return type indicates a designer's intention that `names` and `remind(...)` are always of the same type.
   - When the type of `names` changes, say from `ARRAY[NAME]` to `LIST[NAME]`, is there any manual change necessary to still satisfy this designer's intention?
   - No. It will be updated automatically.

   (1.3) *Cross Reference.* See how anchor type is used in Section 11.
2. `TODO`: Complete the implementation of this query.
3. `TODO`:
   - `every_name_in_result_is_an_existing_name`
   - `every_name_in_result_has_birthday_today`
4. You should write boolean test queries for testing the implementation.
5. You should write violation test commands for testing each of the postconditions.