|  |
| --- |
| trygve |
| an overview  Fall 2015 |

# The What and Why of Trygve

Programming is always about producing illusions through a reality that the programmer creates in a computer and between a computer and a human being. We can profitably use some of these illusions, but many of them must fight through layers of history and prejudice to touch anything deep inside the end user, or her surrogate, the “programmer.” A lot of what we know about programming gets in the way of engagement between end users and the objects they playfully interact with to achieve some task. For example, while they think in objects, we script execution in classes. While they think in scenarios and problem-solving approaches, we focus on individual methods and on the state of individual objects.

To get the most enjoyment from this tutorial reference you might best willfully suspend your disbelief and forget a lot of what you think you know about programming. The **trygve** language is in some sense designed for a child-like mentality that hasn’t yet been polluted by computer science, but which retains the keen problem-solving skills that children can exercise through eyes that perhaps see the world in a less stilted way than a programmer does. The approach took its original inspiration from what we can glean from Alan Kay’s early ideas of how his vision of the Dynabook computer could aid children in what we can refer to in vulgar terms as *learning*, but which entails much more, to the point of thinking of the machine as an extension of self.

The **trygve** language allows you to think about your program more in terms of the mental models of some phenomenon that you bring to the table. It’s called object-oriented programming, so we’re going to talk about classes as a supporting cast rather than as the star players. Here, objects are the star of the show. We talk about objects in terms of the names our compressed mental models give them as they interact to solve some problem. Those names are Roles. Any given Role is just a name but at the same time is much more. The Role “fireman” is just a name for some human being in a particular Context, but it also elicits a host of associated behaviors. Those behaviors that are germane to the role itself, without much regard for the role-player, are called methods — little recipes for doing very small tasks.

While Roles link to the left brain, classes link to the right brain. We filter billions of bits of information in the world around us to make sense of it, and our massively parallel right brain is really good at doing that. We use classification to compress the common aspects of many related objects into one concept. Though we see Hamlet a 100 times we know just a single script. We may start by organizing our world into classes, which will provide the basic building blocks of local state and ways of interacting with that state.

Programming becomes an act of writing a script. A “program” in the sense that we usually use it, for the source code, isn’t the real program at all — it’s just the script that the enactment will follow. The **trygve** language is designed to reduce the transition from run-time to scripting and back again. The “programmer’s” mental model should be focused on the action and, in particular, on the interaction between the end-user and the objects which represent that user’s interests inside the machine.

The **trygve** language takes a cue from Java syntax mainly for reasons of broad familiarity. With the Java legacy comes a legacy of arithmetic types — to a Martian, it would seem that all programmers are working in the domain of mathematics. The fantasies of academia aside, it simply isn’t so. Very few computers compute — i.e., very few of them produce an “answer” or “result.” They run machinery, fly planes, process telephone calls, with any of the numbers of those domains playing only an incidental part in the structure of the program.

But **trygve** departs from Java in many subtle ways. You can write Java in any programming language, and **trygve** is very accommodating for those who offer Java syntax. But experience with the language will help you discover that it has a few features that make it feel at times like Erlang, and at times like Ruby, or maybe some other language that has graced your programming history.

And **trygve** departs from Java in yet more fundamental ways. Java tried to be a pure OO language by outlawing global functions, but that is a simplistic hope at best. It ended being only a class-oriented programming language. Like most languages of its kind it has many features to finesse class relationships. These features encourage class-oriented thinking, overuse of inheritance, and programmer convenience over end-user mental models. So you will find neither friends, or static objects, or the concept of super, nor the protected access property in **trygve**.

The **trygve** language is just one part of a system design that supports end user mental models. In the end, trygve’s main contribution is to the left-brained side of computation — the enactment of scripts. Users still engage their right brain during program enactment but in modern computing, such activity is usually associated with the visual cortex. Identifying the right entities (objects) happens on the screen, and Model-View Controller (MVC) has been designed as the bridge between end user and computer in that regard. MVC and **trygve** can powerfully be combined to provide the most expressive links between the end user and the machine.

# Enactment Building Blocks

The **trygve** language, as a language, has its writings appear on two-dimensional surfaces such as your computer screen or a sheet of paper. We usually call this sequence of characters a program. In **trygve** we call it a *script*, to evoke a metaphor of theatre with a lot going on in the framework of defined Roles. The **trygve** environment is designed to help you think about what exists and happens during the enactment of the script — the actual performance of the play. The program that you write in **trygve** is a set of instructions for bringing about a certain enactment. The script describes what can happen at run time and, in many instances, describes the ordering constraints on run-time activities.

The main run-time building blocks are called *objects*. Each object works with other objects to solve some business problem together. Objects exist only during script enactment (in computing terms, when we are “running the program”). Each one carries out the work that it “knows” how to do, and the sum of all the small activities from individual objects add up to a fully performing program.

These smaller activities that take place inside objects are small scripts in their own right so we call them scripts, too. In the programming vernacular they are called *methods*: a kind of prescription for a way of doing something. Here is a **trygve** script for computing factorial:

int fact(int n) {

return if (n <= 1) 1 else n \* fact(n-1)

}

Each object has the ability to run one of its scripts at a time. At a low level a script is a collection of expressions that are enacted in a specified order.

Though **trygve** supports the illusion that each object has its own copy of its scripts, we need to script the activity only once — in the object’s *class*. This is done before program enactment. It will be in classes, and their close cousins Context and Role, where most of the scripting activity takes place. But it’s important always to keep in mind that a real program is a living, breathing thing, interacting with a (less metaphorically) living, breathing end user out there. The mindset during scripting should always be in terms of the objects in play and in terms of their interactions with the end user during program enactment.

So we now have the context to move on to scripting.

# Script Building Blocks: Declarations and Expressions

As a problem-solver you will be writing scripts that bring the program and its enactment into being at run time. These scripts are the two-dimensional text you write on a piece of paper or type into the source window of the **trygve** environment. The **trygve** language is a textual language, mainly because we felt that, for whatever reason, textually based programming languages seem to thrive while graphical programming languages founder or never gain any footing. The most likely reason is that we simply don’t yet know how to express complex solutions well in graphical form. Text can be highly expressive, even with a limited vocabulary, because of the meaning that is generated by the sequential arrangement of textual content on a page.

Scripts in **trygve** (the **trygve** source language constructs) have two basic building blocks:

* Declarations
* Expressions

There is no concept of “statement” in **trygve**. *Declarations* are the foundation and the scaffolding for your program, and *expressions* are the things you write to instruct your program how to do stuff. Declarations include, for example:

* Classes — A class specifies the makeup of an object and is the place where we write the basic scripts to interact with the state of that object. We say that an object is an *instance* of its class. The class captures the properties of some concept from our business, where that object corresponds to something we name, manipulate, or manage in the real world. We can use a class to describe the concept of a rectangle in the domain of geometry:

class Rectangle {

public double area() { return height\_ \* width\_; }

public Point center() {

return new Point(upperLeft\_.x() + lowerRight\_.x() / 2,

upperLeft\_.y() + lowerRight\_.y() / 2);

}

. . . .

Point upperLeft\_, lowerRight\_;

double height\_, width\_;

}

This class happens to contain the methods area and center, and also includes declarations for the data height\_ and width\_. Data are how we remember stuff. Data encode information that is important to support our problem-solving; how they encode it should be of no concern to the user of the data. The representation, or encoding, of information as data is encapsulated inside of a class, and the users of the corresponding information interact with the data through the *methods* of that class. In the above example, Rectangle is a class — but so are Point and double. We access them through those declarations that it chooses to make public. By the way, a class exists in a condensed and largely hidden form during program enactment.

* Contexts — a Context is a script for a complex set of interactions involving several objects. The overall script is broken down into individual scripts called *Roles*. A given set of objects may play a given set of Roles to enact some system-level script. In computerese we call this higher-level script a *system operation*; the requirements people call it a *use case*. This level of enactment is akin to a scene in a play. Like a scene in a play, a given Context enactment has a beginning and an end (though in theory a given system operation could go on forever).
* Roles — A Role is the script for an object participating in a particular system operation or “scene.” Roles exist only inside Context scripts. Each object participating in a given Context enactment (system operation) “knows” how to play its Role. Role-playing knowledge is bestowed on an object when it becomes engaged in a Context script, and it forgets all of that Role-playing ability at the close of the Context action. A Role comprises *methods* which an object enacts when receiving a message from another Role; that object in turn can elect to send messages to one or more objects that are also playing roles within the same Context. An object playing a Role is called that Role’s *role-player*.
* Methods are like the lines spoken by a given role-player. A method is a collection of expressions, and possibly supporting declarations, that describe how it carries out its part of a dialogue that is elicited by a message from another object. Most Roles have multiple methods, just as a Role in a play may speak and do several things in a given scene of a play.

Declarations may use expressions, as well as other declarations, as building blocks. (The same is true for expressions in that they may use declarations as well as other expressions as building blocks.)

Expressions include, for example,

* Arithmetic or boolean computations (addition, comparison,etc)
* Invoking a message (the way we script interaction between objects)

A **trygve** program comprises a list of declarations followed by a single expression (though that expression may be quite elaborate). Enactment starts with the evaluation of that expression. Of course, that expression interacts with the preceding declarations, and a large, complex enactment may unfold from these interactions.

|  |
| --- |
|  |

|  |
| --- |
|  |

# Scripts

You write a script to orchestrate the solving of your business problem. A script is a sequence of Declarations followed by a single Expression that kicks things off — usually by interacting with the expressions in the Declarations. A very simple program may not have any programmer-supplied declarations at all but only an expression. So this is a perfectly good, complete **trygve** program:

System.out.println("Hello World")

This expression uses a built-in declaration of class System that lives inside the **trygve** environment, which publishes an interface to an object called out that programs can use to send output to the user. The object can be asked (we always *ask* objects — never *tell* them) to evaluate the expressions of its script called println. We send along some information in our message to the out object requesting the performance of the built-in println script, which is a string that we want out to print: “Hello World”. And out will do just that.

Start up the **trygve** environment on your machine. Copy the above script into the text pad (the editing area on the left half of the screen). You can either copy and paste the text or simply type it into the text pad. Or you can select “Example Text” from the **Edit** menu.

Next, press the **parse** button. This causes the **trygve** environment to read the script and to “understand” what it is supposed to do. During this process **trygve** will look for errors in your script: it’s pretty fussy about the grammar you use. If all is well **trygve** will draw a blue line beneath the welcome message.

Last, touch **run**. This will cause **trygve** to evaluate the last (and only) expression in your script. The result is that the message “Hello World” appears on the output pad.

|  |
| --- |
|  |

System.out.println is one kind of expression. Like all expressions, it can be combined with other expressions into a larger script. Declarations can also be used to group together scripts related to some real-world object, or to group together scripts for a given Role.

## Mathematical Expressions

Mathematicians were some of the first programmers and, for better or worse, most contemporary programming languages bear their mark. I remember that when I started studying ancient Hebrew that our instructor advised us that there is really nothing magical about the Hebrew language and that, contrary to belief of some, if wasn’t the language spoken by the Deity or anything like that. Computer scientists tend to have the same belief about mathematics: that it is somehow fundamental to what computers were designed to do. In fact, few computers “compute” (i.e., evaluate formal mathematical expressions to give a certain number or set as a result), but rather serve as a way to extend the user’s processing of information. Numeric data have proven themselves over history as a convenient way to represent information. So we might represent a given color as 164 parts red, 210 parts green and 237 parts blue (which represents the color of part of the shape in the header of this page). Since it is useful to have a single common underlying infrastructure that is common across the industry (because of the economies of scale and scope that come with building on a single technology rather than trying to support several technologies in parallel), numbers have found a home in computing. Numbers form the alphabet from which we craft the “words” and, ultimately, the expressions of computation. But we really just use numbers and rarely use mathematics. Mathematics is a language for talking about numbers and, by inference, real-world things that we choose to represent by numbers (such as the size of your paycheck). Here we are not using the language of mathematics, but rather a language called **trygve**. One might use the language of mathematics to talk about **trygve** programs (e.g., to prove that they have certain properties) but we won’t do that here.

This is just a long way of saying that 1. We can’t avoid dealing with numbers and arithmetic, and 2. There are many **trygve** expressions to talk about ideas from arithmetic. So 1 + 2 is a valid **trygve** expression. As an expression, we can combine it together with our earlier println expression to say:

System.out.println(1 + 2)

which of course prints 3. The common arithmetic operators are all there: ­–, \*, +, and /. There are also less common binary operators like \*\* for exponentiation (e.g., 2 \*\* 128 yields 2147483647).

You can use parenthesis to group smaller expressions into larger expressions. For example, the expression

(-b + sqrt(b \*\* 2 – (4.0 \* a \* c)) ) / (a + a)

might be used to find one of the roots of a simple polynomial. The expression b \*\* 2 means “raise b to the second power;” 4 \* a \* c means to multiply 4 by a, and they to multiply the result by c. Evaluation normally proceeds from left to right.

## Identifiers

But what are a, b, and c? Expression like the one above became familiar to you if you graduated from arithmetic to algebra. In algebra, we use letters to represent some value that can vary every time we use any given expression. Algebra calls them *variables*. In **trygve** we call them *object identifiers*, or just *identifiers* for short.

Mathematicians tend to use letters of the alphabet to name numbers (and they’ll sometimes use a variety of alphabets depending on their goal, so you may know π, for example). We find single letters a bit boring, so we use ordinary English words as identifier names. So we might more clearly describe the quadratic formula like this:

(-coefficientOfSecondOrderTerm + sqrt(coefficientOfFirstOrderTerm \*\* 2 –

(4.0 \* coefficientOfFirstOrderTerm \* constant)) ) /

(coefficientOfFirstOrderTerm + coefficientOfFirstOrderTerm )

This looks long and you probably are thinking that it’s too much typing. However, programs are read dozens of time for each time you write them, and we want code to be above all understandable, even by those who didn’t write the code.

An identifier is a name for an object. Identifiers behave like PostIt® notes that can be affixed to objects. We associate an identifier with an object like this:

a = 1

This creates a PostIt® note with “a” written on it and affixes it to the object for which *1* is the symbol. *1* is just a symbol in the **trygve** language that represents a very simple mathematical object. *2* is another symbol that represents another object. These two objects are similar in many ways: they can be added and multiplied in the same ways, and the “things we can do to them” are the same (all the abelian operators). These objects belong with each other, and many others, as members of a set: the set of *integers*.

By the way, if we say:

b = a

then both b and a end being names for the same object. If we then say:

b = 2

then b ceases to be a name for the object represented by the symbol *1* and becomes a name for the object for which *2* is the symbol. The identifier a remains to be a name for the object representing the integer *1*.

Expressions become much more powerful when we can write them in terms of identifiers. We’ll learn more about identifiers in the next section.

## Types and Simple Declarations

The **trygve** language knows about a small number of sets, including the set of integers. We have a name for this set, which is Integer. (We can also use the shorter name int for historical reasons). Instead of using the word “set” in **trygve**, we use the word *type*. This word (in English) emphasizes that its members have similar characteristics. We use types to constrain the kinds of objects that identifiers can associate with, like this:

Integer a, b

Integer is a type. This code tells **trygve** that the identifiers a and b may be associated only with Integer objects. This helps **trygve** help you keep things straight in a program. The **trygve** language also understands the additional small set of types String, double, and boolean.

The above line is called a *declaration*. It joins *expressions* as one of the major building blocks of **trygve** programs. By itself, a simple declaration doesn’t do any work or run any script. It doesn’t create any objects. It takes no memory in the machine. It merely “declares” some properties of its identifiers, for future use. You have to declare an identifier in order to use it. Within a script the declaration for an identifier must appear before its use. In general, it’s a good idea always to have the declaration for an identifier appear above its use in a program, but the **trygve** language is flexible for some declarations and may allow them to come out of order. We’ll see some examples later.

## Objects

Objects are the key building blocks of a running **trygve** program. The **trygve** scripts bring objects into existence and the objects can have access to each other, to invoke each others’ scripts, to achieve some overall business goal.

For you computer scientists: you’re probably thinking that I misspoke about 1 when I used the word “object,” because most programming languages distinguish between objects and values. The **trygve** language has no such distinction because, for all practical purposes, it deals only with objects. There is only one value in the **trygve** language, which is null, and we’ll talk more about it later. In short, it is the object that is no object.

Objects are agents inside our program that can do basic, simple, local computations. They are like individual actors that we hire if we are producing a play. Individually an actor may not be able to do much that is impressive, bus several actors exchanging lines and working in concert can rise to complex action.

An object has three distinguishing properties. The first is behavior. An object is a locus of related behaviors that together model the behaviors of a concept that is part of our model of the world. So an object in our program helps us use the computer to extend the processing of our brain according to how we program it. We implement an object’s behaviors as a collection of scripts. Each script has a name and a few other properties that help us to use it in a flexible way.

The second is state. When a script executes in an object it usually changes its state. For example, if we have an object that represents a triangle on a place, and the object executes its moveTo script, the object’s location changes. If the object is charged with “knowing” its location we say that location are part of its state. State has meaning only under the possibility of changing. An object’s state is usually implemented in a specific small collection of computer data that are allocated when an object comes into existence.

The third is identity. If an object exists and we associate an identifier with it, and then associate another identifier with the same object, then we can send a message to that object through either of those identifiers and the end result will be the same.

When the symbol 1 appears in a **trygve** program, the program creates a new object representing the integer value 1. So the statement:

a = 1

first creates an object representing the object 1, and then associates it with the identifier a.

## Classes, Scripts and Cues

We have talked about the type Integer. Integer has its own scripts for adding, subtracting, multiplying, and dividing numbers, and maybe a few other arithmetic tasks. And Integer has its own data to store its state. Someone has written the code for Integer. The construct that holds that code is called a *class.* Classes are the most common building blocks of **trygve** programs. A class looks like this:

class ClassMate {

public String familyName() {

return familyName\_;

}

public String firstName() {

return firstName\_;

}

public String nickName() {

return nickName\_;

}

public int age() {

Date today = new Date();

return today.getYear() – birthYear\_;

}

private int birthYear\_;

private String familyName\_, firstName\_, nickname\_;

}

We usually capitalize class names. This simple class represents one of our classmates. Its scripts so far are very simple, just giving us some insight into the data used to represent the information in the class’s state.

A class is the DNA of an object. If an object is like an actor, the class is what gives it is basic, simple functionality. All new objects are created with the “DNA” of their class. That DNA includes the general structure of the object as well as all of its scripts.

Before talking about the class’s scripts, let’s look at its state. The class has declared three identifiers: birthYear\_, familyName\_, and firstName\_. These identifiers can each be associated with an object. (In a bit we’ll describe a convenient way to set up the associations between these identifiers and their objects). Note that each name ends with an underscore: a convenient convention. Each declaration gives its identifier a type. In this case all the identifiers are declared with a private attribute. That means that they may be accessed only by the scripts inside of ClassMate: as far as any external script is concerned, these identifiers don’t exist.

The name attributes familyName\_, firstName\_, and nickName\_ are all names of objects of class String. A String holds a sequence of characters that we usually associate with text that can appear on a screen or on a printed page. Its contents are the closest we get to an object that can be "digested" by human senses (though we still need the machinery inside the computer to make the object visible on the screen or on the page — but that's just part of the machine, and it's the same for all programming languages). You can't read an Integer — it's just a mathematical concept, and the machine may choose to internally represent it in binary-coded decimal, in binary, or in a more efficient representation like ternary. But we should leave those matters to the machine. Therefore, like most other built-in types, Integer provides a way for us to convert it into something more digestable by humans: a String. We do this with the String script inside Integer:

Integer i = 123;

String s = i.toString();

System.out.println(s)

Note that if we just say:

System.out.println(i)

we get the same result; that's because println automatically applies the toString script for us so it can display it on output. The println script does this for all built-in types. When you define your own types (we'll talk about that later) you should usually include a toString script as part of your implementation.

These lines declare and define the script familyName:

public String familyName() {

return familyName\_;

}

Altogether, this is another kind of declaration. This declaration describes a script. In computerese these are called *methods*. We declare the script as public, meaning that any other code that has access to an object of class ClassMate is allowed to ask that the script be run.

A script declaration typically starts with a public or private access modifier, followed by the type of the object that the script delivers back to the client when it is done. (If the script provides no value in return for the cue, then we use the type “void” here). Then comes the name of the script, followed by a pair of parenthesis whose importance will soon become apparent. Then comes the real script: the collection of expressions that the script will execute. These expressions are grouped together inside a pair of curly braces. The lines between any pair of curly braces together are called a *block* of code. The main block for a script is called the script’s *body*.

The familyName script belongs to the class ClassMate: we think of it as being *inside* of a ClassMate object. We think of every ClassMate object as having the ability to run the familyName script; because it’s public, anyone with a handle on a ClassMate object can cue the familyName script to be enacted. While it is being enacted we think of the action happening inside the ClassMate object. Inside the object we find other scripts as well as identifiers that have become associated with other objects. We see the familyName script refer to the identifier familyName\_ (note the trailing underscore — a manually maintained convention for naming identifiers that belong to an object, and which are declared in those objects’ class). Because the familyName script is “running inside the object” and because the identifier familyName\_ is also inside the object, the script can just casually invoke the identifier name to interact with the object for which it is a name. The same would be true for birthYear\_. And the same would be true if one method wanted to cue another method within the same object.

Let’s assume that some client of ClassMate has become a client by somehow receiving an object of class ClassMate. It associates that object with the identifier studyPartner. That client can ask the ClassMate object to run its familyName script like this:

// cue the study partner

String familyNameOfMyStudyPartner = studyPartner.familyName();

We say that the client *sends a cue* to studyPartner requesting that it perform its familyName script. This is reminiscent of players in a play saying their piece or performing some action in response to some words or action directed to them from some other actor on the stage. In computerese, what we here call a *cue* is called a *message*. In this example, the studyPartner object (of class ClassMate) runs its script — which in this case is very simple. When the script runs it takes the object that was previously associated with its local identifier familyName\_ (identifiers local to a class usually end in an underscore, but that is only a convention) and “returns” it to the client. The return expression delivers the object to the client in the place where the cue appears in the code. In this example, the identifier familyNameOfMyStudyPartner in the client also becomes bound to the same object that ClassMate had already bound to its familyName\_ identifier. So familyNameOfMyStudyPartner and familyName\_ are now both names for the same object.

Most of the other scripts for ClassMate are similar — they serve just to give information to some client. The age script actually needs to do some work — a simple arithmetic calculation. Date is another class that knows, obviously, about dates. We can create a Date object like this:

Date today = new Date();

The text new Date is an expression whose result is a new Date object. That expression can be used anywhere in the program where a Date object is expected. Here, we also declare the identifier today and we associate today with the new object, as a means for asking the object to run its scripts. We can cue the new object to give us the current year:

int thisYear = today.getYear();

### Constructors and Method Parameters

You are probably wondering how and when the identifiers birthYear\_, familyName\_ and firstName\_ first become associated with an object. They are declared private and are inside ClassMate so they are inaccessible to the outside world. We need to add a new script to initialize the object, so that these identifiers are all bound to objects. This script is called a *constructor*. We can recognize a constructor because the script name is the same as the class name:

class ClassMate {

public ClassMate() {

familyName\_ = "Smith";

firstName\_ = “John”;

birthYear\_ = 2000

}

. . . .

Of course, it would be a boring world if all of our classmates where named John Smith. So we allow the client to send information along with their cue to create a new ClassMate object — information about the individual for whom the object is being created. Like other scripts, constructor scripts are allowed to have parameters. A declaration of a constructor script with parameters looks like this:

class ClassMate {

public ClassMate(String familyName, String firstName, int birthYear) {

familyName\_ = familyName;

firstName\_ = firstName;

birthYear\_ = birthYear

}

. . . .

That is, the method declaration starts with a list of parameter declarations, separated by commas and enclosed in those parentheses you’ve been wondering about. (The syntax is supposed to be reminiscent of the way mathematicians write functions with their arguments.) We send a message to create a new ClassMate object and to enact this initialization script by saying:

ClassMate someClassMate = new ClassMate("Lastname", "firstname", 1975);

Of course, we need only bind those identifiers that we think need to be handled when the ClassMate object comes into being; maybe the rest can be bound later by other scripts. However, it is usually good practice to bind all of a class’s identifiers to some object inside the constructor.

Putting it all together, we can make a complete running program using the ClassMate class:

class ClassMate {

public ClassMate(String familyName, String firstName, int birthYear) {

familyName\_ = familyName;

firstName\_ = firstName;

birthYear\_ = birthYear

}

public String familyName() {

return familyName\_

}

public String firstName() {

return firstName\_

}

public String nickName() {

return nickName\_

}

public int age() {

Date today = new Date();

return today.getYear() - birthYear\_

}

private int birthYear\_;

private String familyName\_, firstName\_, nickname\_

}

{

ClassMate myStudyPartner = new ClassMate("Funch", "Rune", 1975);

String familyNameOfMyStudyPartner = myStudyPartner.familyName();

System.out.print("Family name of my study partner is ")

.print(familyNameOfMyStudyPartner);

System.out.print(", age of ").println(myStudyPartner.age())

}

This is actually a pretty serious program! The last six lines of the program form a block, and a block is just another kind of expression. Since it is the last expression in the program, it is where **trygve** starts enactment. Computer people sometimes call this the “main program.” It should be very small and tidy — just enough to kick off execution, so the rest of the objects can get about working together to perform the overall pattern of system enactment.

This program also does a bit of showing off with respect to how we can chain together print and println requests. We’ll explain later exactly what’s going on to make it work. For now, just trust that it works and use it as a handy convenience to save some typing.

### Scripts at several levels

You’ll notice that the above program has several scripts that we can see inside class ClassMate, as well as the final script that starts things off. These are the two most common kinds of **trygve** scripts. Most **trygve** scripts live inside a class and become available to users of objects of that class.

We’ll later learn about other concepts which, like classes, group scripts together, sometimes with data and sometimes not. These concepts are:

* The Context, which represents a graph of interactions between objects
* Roles, which contain scripts for archetypes that we carry around in our head

### Identifiers Inside a Script

A script can declare its own identifiers for its own use, inaccessible to any other code. The script can associate these identifiers with the objects passed as parameters, with objects that are returned when cueing other scripts, or with objects that the script creates.

We might create a script to evaluate one of the roots of a simple polynomial. To make the code easier to understand we can break the evaluation into parts. We can declare an identifier for each one of the sub-expressions and combine them into an answer:

double polyRoot(double a, double b, double c) {

double four\_a\_c = 4.0 \* a \* c;

double denominator = a + a;

double rootArg = b \*\* 2 – four\_a\_c;

double root = (-b + Math.sqrt(rootArg)) / denominator;

return root

}

As we showed above, we can put several scripts inside of a class. We can break down the polynomial evaluation into small scripts, each one of which contributes its part to the answer:

class Polynomial {

public Polynomial(double a, double b, double c) {

a\_ = a; b\_ = b; c\_ = c

}

private double four\_a\_c() {

return 4.0 \* a\_ \* c\_;

}

private double denominator() {

return a\_ + a\_;

}

private double rootArg() {

return b\_ \*\* 2 – this.four\_a\_c();

}

public double root() {

return (-b\_ + Math.sqrt(this.rootArg())) / this.denominator();

}

private double a\_, b\_, c\_;

}

First, you’ll notice that many of the scripts are declared as private — they are private to class Polynomial. That is because they are of no interest or use outside the class. Only other scripts within Polynomial may cue a request to these scripts.

Second, you’ll notice that the script called root cues two other scripts: rootArg and denominator. It does so by sending a cue within the same object, which we designate with the identifier this. The identifier this is implicitly declared for you by **trygve** within every script, and it always refers to the object containing the script.

## Flow Expressions

A script may need to improvise its behavior when it is enacted, perhaps because it has been designed to handle multiple related situations and it has to wait until enactment to know which one applies. This means that we need to be able to steer the flow of expressions during enactment. Or maybe there is some sequence of expressions that we want to do evaluate several times: we want to be able to write it just once but enact it many times, like the chorus of a song.

### The IF expression

The IF expression lets you ask a yes/no question derived from some state of an object and then to evaluate either one expression or another depending on the answer. So you might write code for a car in China that looks at the license plate number: its oddness (licenseNumber % 2 == 1) and the evenness or oddness of the date (Date.getDate() % 2 == 1) and to allow the driver to drive only if the evenness of both the date and the license number are the same:

boolean canDrive = if (licenseNumber % 2 == 1)

Date.getDate() % 2 == 1

else

Date.getDate() % 2 == 0

(Of course, logicians could state this same condition in a much more compact way, but this way is easy to read and helps illustrate what we’re trying to demonstrate here.) The IF expression is called a *conditional expression*. It is one of the key building blocks of business logic. Its use also arises in mathematics. Here’s an elegant formulation of factorial in **trygve**:

class Fact {

public int fact(int n) {

return if (n <= 1) 1 else n \* fact(n-1)

}

}

Let’s go back to our polynomial roots example. We’ve already seen the code:

public double root() {

return (-b\_ + Math.sqrt(this.rootArg())) / this.denominator();

}

This works as long as this.rootArg() is non-negative. If it is negative, then the polynomial has a complex root with an imaginary component.

Instead of writing a script that returns the root to another script in response to a message, we might instead write a method that just prints both of the roots of the polynomial — including any complex ones.

Let’s say that the square root argument is negative: that would cause trygve to encounter the uncomfortable position of trying to take the square root of a negative number, and we want to save it that embarrassment. We’ll assume for the time being that we aren’t interested in imaginary or complex solutions. We can write:

public void printRoot() {

double rootArgument = this.rootArg();

if (rootArgument > 0) {

System.out.print("root is ")

.println(-b\_ +

Math.sqrt(rootArgument) / this.denominator())

} else {

System.out.println("root is complex")

}

}

**Side-Effects**

Like all expressions the IF expression yields a result. Like most other expressions, it’s a fusion of other expressions working together. We can divide **trygve** expressions into two kinds: those that have *side-effects*, and those that don’t. A side-effect is usually a change in some state that persists after the expression has completed its enactment. If one actor in a play just speaks a part, we can’t tell by looking at the scene whether that player has spoken that line or not: there is no residual side-effect. But if one player spits at another we can know that we had come to that point in the script.The player left a side-effect.

The IF expression is often used for its side-effect alone in many programming languages, and it can be used this way in **trygve**. Let’s return to our polynomial example, and revert the printRoot example back to the original version that returned a result.

### For loops and arrays

Sometimes we want to group many items of the same kind together, like a flock of geese or a collection of test scores or a list of attendees at a party. And we may want to enact some script for each of them in turn: e.g., to print an invitation for each of the people we want to invite to the party. If we know the number of items we'll have to deal with, we can use an array. We can create an array of ints or of Strings or of ClassMates of a given size.

We must declare an array before we can use it. An array of ints to hold all the scores from a class test might be declared like this:

int numberOfClassMembers = 23;

int [] scores = new int[numberOfClassMembers]

The above array is called scores and has 23 *elements*. Each element of the array is just a name, and we can associated each of those elements with an object, like this:

int joesScore = 96

scores[5] = 96

These expressions first cause joesScore to become a name for the int object whose value is 96, and then causes scores[5] to be another name for that same object. If we had instead said:

int joesScore = 96

int joesIndex = 5

scores[joesIndex] = 96

the result would be exactly the same: both scores[5] and joesScore would become names for the same object carrying Joe's score.

We normally “fill up” an entire array with values before it is used; however, there is nothing in **trygve** that says you must do this. If you do fill up the entire array, then you can very expressively process each of its elements, one at a time. We do this with another expression called a *for expression*. It looks like this:

for (int aScore : scores) total = total + aScore

So, for example, we could compute the average score like this:

int scoreCount = 0

double total = 0

for (int aScore : scores) {

total = total + aScore

scoreCount++

}

double averageScore = total / scoreCount

The expression scoreCount++ simply increments the value of scoreCount by 1. You notice that we use a double instead of an int to total the scores. If we had summed them as an integer and then divided by an integer, **trygve** would give us an integer result: we would lose the precision in the fractional part. This follows a common convention used in Java and in most modern programming languages.

There is an alternative form of the for expression that isn't limited to working with collections like arrays. It is a general way of running an index over a predefined range with a regular increment, and it looks like this:

for (int i = 0; i <= 22; i++) {

System.out.print(i)

}

This loop will print the integers from 0 to 22 inclusive. It works like this: The for statement in this form ties together four expressions:

for ( *initialization expression* ; *test expression* ; *increment* *expression* ) *body expression*

The for loop starts by evaluating the *initialization expression*. It can be any expression or, as indicated in the above example, it may alternatively even be a declaration. By tradition and convention it declares and initializes a simple integer, often called the *loop variable*. The for expression will cause the loop variable to successively take on increasing values so that it can be used, for example, as an index into an array.

Then the *test expression* is evaluated. It must evaluate to a boolean result. If the result is false, then the for expression is done and enactment will pick up at the next part of the script in the flow. Otherwise, the *body expression* is evaluated. The body expression is most often a *block*: a sequence of expressions grouped together by surrounding curly braces so that, from the outside, they are all treated as one large expression. Once the *body expression* has been enacted, the for expression evaluates the *increment expression*. Again, this can be any expression, but by tradition and convention it increments the loop variable. Last, we return again to the *test expression*, and the loop continues again through the *body expression* *and increment expression* and back again, until the *test expression* causes the looping to come to an end.

So revisiting the loop from above,

int scoreCount = 0

double total = 0

for (int aScore : scores) {

total = total + aScore

scoreCount++

}

double averageScore = total / scoreCount

we could just have well have written it like this:

int scoreCount = scores.size()

double total = 0

for (int i = 0; I < scoreCount; i++) {

total = total + scores[i]

}

double averageScore = total / scoreCount

**A Puzzle**

It's perhaps time for a little quiz! Given what you now know about objects and arrays, what is the output of the following program?

class ArrayTest {

public void test() {

int [] intArray = new int[5];

for (int i = 0; i < 5; i++) {

intArray[i] = i

}

for (int i = 0; i < 5; i++) {

System.out.println(intArray[i])

}

}

}

(new ArrayTest()).test()

### While loops

The for loop is just one kind of loop in **trygve**, and there are two others: while loops, and do/while loops. These kinds of loops are all basically the same in how they structure computation; the difference is in the boundary conditions, in relative complexity, and in whether they check for completion at the beginning or the end of the interation.

The while loop is perhaps the simplest kind of loop in **trygve**. Generically, it is of the form:

while ( *condition expression* ) *body* *expression*

It works by first evaluating the *condition expression*, which can be any expression that evaluates to a boolean. If it evaluates to false then enactment continues at whatever expression follows the *body expression*. Otherwise, the *body expression* is executed once. Then we just start over again, and go back to evaluate the *condition expression*, and then the *body expression* again if the result evaluates to true, and so forth. The iteration stops when then *condition expression* finally evaluates to false.

Note that this implies that the *body expression* has some side-effect on which the *condition expression* depends; otherwise, you will loop forever!

The while loop can be used with any expressions that require doing something again and again. Whereas the for loop is usually used when the number of iterations is known ahead of time, the while loop is typically known when we can know only just-in-time when we are done.

Consider implementing Newton's Method, which takes an unpredictable number of iterations to find a root (adapted from: http://www.cplusplus.com/forum/general/94191/):

double root(double number, double lower, double upper, double guess) {

double ACCURACY=0.001;

if (number < 1){

lower = number;

upper = 1.0

} else {

lower = 1.0;

upper = number

}

while ((upper-lower) > ACCURACY){

guess = (lower + upper) / 2.0;

if (guess\*guess > number)upper = guess

else lower = guess;

}

return (lower + upper)/ 2.0;

}

### Do / While loops

A do / while loop is like the while loop above except that the test expression comes at the end:

do *body* *expression* while ( *condition expression* )

In a do / while loop the *body* *expression* is always executed at least once. We can use the worst sorting algorithm ever, Bubble Sort, to illustrate (courtesy of <http://www.algolist.net/Algorithms/Sorting/Bubble_sort>, slightly modified):

public void bubbleSort(int [] array) {

      boolean swapped = true;

      int j = 0;

      int tmp;

      do {

            swapped = false;

            j++;

            for (int i = 0; i < array.size() - j; i++) {

                  if (array[i] > array[i + 1]) {

                        tmp = array[i];

                        array[i] = array[i + 1];

                        array[i + 1] = tmp;

                        swapped = true

                  }

            }

      } while (swapped)

}

# Built-in Classes

**Class String**

All implemented interfaces:

None

Direct known subclasses:

None

public class **String**

extends Object

The class String is an interface to the Java String facility. Most of its methods are faithful to Java semantics. The reader is referred to suitable Java documentation for details (e.g., <http://docs.oracle.com/javase/7/docs/api/java/lang/String.html>).

An expression of the form “abc” is a String literal in **trygve**. It is a shorthand for a String object whose contents are the characters *a*, *b*, and *c*, catenated together.

Most built-in types have a toString operation that leaves the original object unmodified, and returns a String representation of its contents.

**Since**:

trygve 1.0

**See Also:**

**Constructor Summary**

|  |  |  |
| --- | --- | --- |
| |  | | --- | | **Constructor and Description** | | **String()**  Allocates a String object and initializes it so that it contains no characters | |

**Method Summary**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Modifier and Type** | **Method and Description** | | **int** | **length()**  Get the number of characters in the String | | **String** | **substring(int startindex, int endindex)**  Extract the String starting at (zero-based) index startindex and ending at endindex. It is a fatal error if the indeces are outside the bounds of the cued object | | **String** | **toString()**  Returns the object | | **String** | **+**  A binary operator used to catenate two strings. The expression *a + b* results in a new String that is the catenation of the strings *a* and *b*. | | **int** | **indexOf(String aString)**  Get the index (zero-based) of the first instance of the designated String aString in the object. | | **boolean** | **contains(String aString)**  Returns true or false according to whether the object contains, or does not contain, the argument aString | |

**Class Date**

All implemented interfaces:

None

Direct known subclasses:

None

public class **Date**

extends Object

The class Date is an interface to the Java Date facility. Most of its methods are faithful to Java semantics. The reader is referred to suitable Java documentation for details (e.g., <https://docs.oracle.com/javase/7/docs/api/java/util/Date.html>).

**Since**:

trygve 1.0

**See Also:**

**Constructor Summary**

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | **Constructor and Description** | | **Date()**  Allocates a Date object and initializes it so that it represents the time at which it was allocated | | **Date(int year, int month, int date)**  Allocates a Date object and initializes it so that it represents the specified date | |

**Method Summary**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Modifier and Type** | **Method and Description** | | **int** | **getDate()**  Get the day of the month | | **int** | **getMonth()**  Get the month of the year | | **int** | **getYear()**  Get the Gregorian year | | **int** | **getDay()**  Get the day of the week | | **void** | **setDate(int date)**  Set the day of the month | | **void** | **setMonth(int month)**  Set the month of the year | | **void** | **setYear(int year)**  Set the Gregorian year | | **void** | **setDay(int day)**  Set the day of the week | | **String** | **toString()**  Render the date as a String. See the Java documentation for a description of the String format | |

**Class List<T>**

All implemented interfaces:

None

Direct known subclasses:

None

public class template **List<T>**

extends Object

The class template List simulates the Java List<T> interface. Most of its methods are faithful to Java semantics. The reader is referred to suitable Java documentation for details (e.g., https://docs.oracle.com/javase/8/docs/api/java/util/List.html).

**Since**:

trygve 1.0

**See Also:**

**Constructor Summary**

|  |  |  |
| --- | --- | --- |
| |  | | --- | | **Constructor and Description** | | **List()**  Allocates a List object and initializes it to the empty list | |

**Method Summary**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Modifier and Type** | **Method and Description** | | **void** | **add(T element)**  Append element to the end of the list | | **T** | **get(int theIndex)**  Get the List element at the indicated (zero-based) index | | **int** | **indexOf(T element)**  Get the (zero-based) index of the first appearance of the indicated element in the List | | **boolean** | **contains(T element)**  Return true or false according to whether the List contains element, or not, respectively | | **int** | **size()**  The number of elements in the List | | **boolean** | **isEmpty()**  Returns true if the size() is 0, and false otherwise | |

**Class Map<K,V>**

All implemented interfaces:

None

Direct known subclasses:

None

public class template **Map<K,V>**

extends Object

The class template Map simulates the Java HashMap<T> class interface template. Most of its methods are faithful to Java semantics. The reader is referred to suitable Java documentation for details (e.g., <http://docs.oracle.com/javase/7/docs/api/java/util/Map.html>).

WARNING: Map implementation is in progress and has not yet been released for general use.

**Since**:

trygve 1.0

**See Also:**

**Constructor Summary**

|  |  |  |
| --- | --- | --- |
| |  | | --- | | **Constructor and Description** | | **Mao()**  Allocates a List object and initializes it have no associations | |

**Method Summary**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Modifier and Type** | **Method and Description** | | **void** | **put(K key, V value)**  Associates the specified value with the specified key in this map | | **V** | **get(K key)**  Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key. | | **boolean** | **containsKey(K key)**  Returns true if this map contains a mapping for the specified key. | | **boolean** | **containsValue(V value)**  Returns true if this map maps one or more keys to the specified value. | | **int** | **size()**  Returns the number of key-value mappings in this map. | | **V** | **remove(K key)**  Removes the mapping for a key from this map if it is present (optional operation). | |

**Class Math**

All implemented interfaces:

None

Direct known subclasses:

None

public class **Math**

extends Object

The class Math provides a small fraction of the Java Math class facility. For all intents and purposes it is never instantiated but is used for its static methods. Most of its methods are faithful to Java semantics. The reader is referred to suitable Java documentation for details (e.g., https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html).

**Since**:

trygve 1.0

**See Also:**

**Method Summary**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Modifier and Type** | **Method and Description** | | **static double** | **random()**  Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0. | | **static double** | **sqrt(double theIndex)**  Returns the correctly rounded positive square root of a double value. | |

**Class PrintStream**

All implemented interfaces:

None

Direct known subclasses:

None

public class **PrintStream**

extends Object

The class PrintStream is a superficial interface to the Java PrintStream class. It is most commonly used in conjunction with its preallocated static member of class System, System.out. There is never a good reason to instantiate one’s own instance of PrintStream in the current implementation.

**Since**:

trygve 1.0

**See Also:**

**Method Summary**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Modifier and Type** | **Method and Description** | | **PrintStream** | **print(int value)**  Print an integer to the stream | | **PrintStream** | **print(double value)**  Print a double to the stream | | **PrintStream** | **print(String value)**  Print a String to the stream | | **PrintStream** | **print(boolean value)**  Print a boolean to the stream | | **PrintStream** | **println(int value)**  Print an integer to the stream, followed by a newline | | **PrintStream** | **println(double value)**  Print a double to the stream, followed by a newline | | **PrintStream** | **println(String value)**  Print a String to the stream, followed by a newline | | **PrintStream** | **println(boolean value)**  Print a boolean to the stream, followed by a newline | | **PrintStream** | **println()**  Print a newline | |

# Simple Configuration Instructions

Most of you should be able to get by with these simple configuration instructions from Andreas Söderlund. He uses them to bring up **trygve** on Windows but I'm guessing it will also work just as well on a Mac. It's a great way to get started if you only want to explore the language, and not yet to make changes to the environment or language itself.

I've got it up and running on windows now, it works very well to just clone the git repo and execute "gradlew run" in a command prompt.

For windows user with not even java and git installed, I recommend the following install process:

1. Download Java from http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html (one of the windows x64/x86 versions) and install

2. Download the full package from http://cmder.net/ (a nice console emulator) and install

3. Start cmder and execute:

git clone https://github.com/jcoplien/trygve.git

cd trygve

gradlew run

If you don't have Office to read the documentation in trygve1.docx (great introduction, Cope!), this is a freeware alternative: http://www.kingsoftstore.com/kingsoft-office-freeware.html

It's always a long install process, no matter how simple you make it. :) But anyway, this is great! It feels like I'm back with a good old ABC-80 but with a language I wish existed back then...!

Configuring for Eclipse

This section describes how to download **trygve** from the Git repository at <https://github.com/jcoplien/trygve/> and configure it for use under Eclipse Luna (4.4.2). Preconditions: Eclipse, gradle, and git are installed. Eventually we’ll add instructions here for bringing those two tools up on your machine

1. Create a directory in your machine’s file system that will be your working space, and enter that directory:

$ mkdir workingspace

$ cd workingspace

1. Clone the trygve Git workspace in the working space you just created:

$ git clone git@github.com:jcoplien/trygve.git

*Cloning into 'trygve'...*

*remote: Counting objects: 1604, done.*

*remote: Compressing objects: 100% (354/354), done.*

*remote: Total 1604 (delta 202), reused 0 (delta 0), pack-reused 1119*

*Receiving objects: 100% (1604/1604), 2.26 MiB | 383.00 KiB/s, done.*

*Resolving deltas: 100% (925/925), done.*

*Checking connectivity... done.*

*Checking out files: 100% (159/159), done.*

1. Start up eclipse. In the File menu, switch eclipse to the new workspace you created (f you select “Other” you will be given the opportunity to browse to your directory):





Eclipse will restart.

1. Import the files from the workspace. In the File menu, select Import, near the bottom of the menu:



A popup window will appear. Open the Gradle selection item, and then select Gradle Project.



Press Next.

1. A file selection menu will appear. Navigate to the trygve folder in your workspace:



1. Press the Build Model button. The popup window will show progress on building the model from Gradle:



1. Select the Project tickbox next to the trygve project:



1. Press the Finish button near the bottom of the page. The screen will display progress as it refreshes the trygve project:



1. Go to your workbench in the workspace by tapping the Workbench icon in the upper-right of the screen:



You should see the environment with its views:



1. Open the trygve list at the upper left of the screen, and then open into the src/main/java tab:



1. Type the word “Preferences” into the help text box at the upper part of the window near the right-hand side. A popup window will appear:



1. Find the entry for the Preferences menu, and select it (see above). You will come to this popup:



1. Open the Run/Debug tab and select “Launching.”



1. In the bottom pane, select the radio box button “Always launch the previously launched application.”



Press “Apply” and then press “OK”.

1. In the Run menu, open up “Run Configurations…”:



1. Open the Java Application item and select “New configuration (1)”



1. Select the Arguments tab.
2. Add “-ea” as a VM argument:



1. Pres s”Apply” and then press “Close.”
2. Repeat steps 15 – 19 for “Debug Configurations…” instead of “Run Configurations...”. (You may find that the   “-ea” text has already been entered — that’s O.K.).
3. Open the info.fulloo.trygve.editor tab under the Package Explorer on the left:



1. CTRL-click on Main.java and run it as a Java application:



1. The trygve console will appear. Close it out by clicking on the red close button at the top-left of the window.
2. In the Eclipse window, again open Run Configurations:



1. Add the “-ea” option to the Main configuration:



1. Press Apply, then press Close or Run.
2. Do steps 24 and 25 again for the Debug Configurations, if necessary.
3. In the Run menu, select “Run last lauched”:



The trygve app should appear:



1. Press “Run Tests.” Test should run successfully.



1. You are done.