|  |
| --- |
| trygve |
| an overview  Winter 2016 |

# 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 stars 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 persists as 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.

This is probably a bit more than most people want or need to know, so you skip ahead to 4.3 if you like… There is a longstanding tradition in programming, most notably those that spring from the BCPL-, B- and C- based cultures and from UN\*X programming, that leading underscores are reserved for internal and system use. It long predates C++ and is much broader than just C. The same sentiment holds broadly for most Microsoft software and the use of leading underscores is explicitly discouraged in the CLS. It’s outright illegal in Delphi (yes, there are still a lot of Delphi programmers, particularly in the Balkans).

I don’t want **trygve** code to look like computer code — at least to the extent that is possible within Java confines.

There are tougher restrictions that are somewhat more conventional. Many internal **trygve** variables use embedded “$” characters. In one case, I use a leading space (the name of the class from which Class is instantiated starts with a space, as I recall). The lexical analyzer doesn’t accept any of these so it’s impossible to alias internal program names.

There are other restrictions on naming, but that’s an Easter egg and I’m going to leave it to others to discover… No fair looking at the code!

## 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 other's 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
* Interfaces, which are just a way of grouping classes with similar behaviour

### 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] = joesScore

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)

}

### Exiting a Loop

The **trygve** language lets you jump out of a loop prematurely if you should find the need. The break statement causes any looping construct to stop enactment immediately. Enactment ensues immediately following the loop.

Use break statements sparingly. They interrupt the normal flow of execution and can make a program slightly more difficult to comprehend than if they weren't there.

### Switch Expressions

A switch expression is kind of a poor man's table lookup. Here is an example from a program by Andreas Söderlund that uses a switch statement to map a chord name to a position:

public int position() const {

int pos =

switch (name()) {

case "C": 1; break;

case "C#": 2; break;

case "Db": 2; break;

case "D": 3; break;

case "D#": 4; break;

case "Eb": 4; break;

case "E": 5; break;

case "F": 6; break;

case "F#": 7; break;

case "Gb": 7; break;

case "G": 8; break;

case "G#": 9; break;

case "Ab": 9; break;

case "A": 10; break;

case "A#": 11; break;

case "Bb": 11; break;

case "B": 12; break;

default: 0; break

}

In general the switch statement is of the form:

switch(*switch-expression*) { case *constant*: *expression* case *constant*: *expression* default: *expression* }

The statement first evaluates the *switch-expression* to get an object. It then checks each of the *case labels* (those parts of the above description labeled as *constant*) to look for an exact match. When it finds a match, it continues enactment with the corresponding *expression*.

Case labels are transparent with respect to enactment: that is, enactment will proceed right through them without doing anything. So to keep the above example from executing every expression we add a break expression at the end of each one. This causes enactment to jump out of the switch statement and to continue at whatever follows. That means that the above switch statement can be made a little less redundant by rewriting it as follows:

public int position() const {

int pos =

switch (name()) {

case "C": 1; break;

case "C#":

case "Db": 2; break;

case "D": 3; break;

case "D#":

case "Eb": 4; break;

case "E": 5; break;

case "F": 6; break;

case "F#":

case "Gb": 7; break;

case "G": 8; break;

case "G#":

case "Ab": 9; break;

case "A": 10; break;

case "A#":

case "Bb": 11; break;

case "B": 12; break;

default: 0; break

}

}

Of course, you can use a switch expression just for its side effects instead of to generate a value. Again, here we unnecessarily introduce a variable that represents state, when, in fact, this is a table lookup that has no state:

int weekdayIndex(String weekdayName) {

int retval;

switch (weekdayName) {

case "mandag": retval = 0;

case "tirsday": retval = 1;

case "onsdag": retval = 2;

case "torsdag": retval = 3;

case "fredag": retval = 4;

case "lordag": retval = 5;

case "sonday": retval = 6

}

return retval;

}

The expression weekdayName is evaluated and then enactment ensues at the expression matching the case label associated with the corresponding value. If none of the values match the expression, you can include a default label to catch outliers. If there is no default label then the switch statement effectively does nothing if no cases match the expression. So in this function, weekends would be caught by the default clause:

int workingDayIndex(String weekdayName) {

int retval;

switch (weekdayName) {

case "mandag": retval = 0;

case "tirsday": retval = 1;

case "onsdag": retval = 2;

case "torsdag": retval = 3;

case "fredag": retval = 4;

default: System.out.print(weekdayName).println(" is not a working day")

}

return retval;

}

It's always a good idea to include a default case! And unless you want to use the switch statement just for its side effects, all of the case expressions have to be of the same type (which means that the last expression in each expression list after the case label, is of the same type.)

Note that the switch statement works a bit different than in most C-based languages. In C, the case expressions may be in any enclosed scope of the switch. In **trygve** they must be in the scope immediately enclosed by the switch expression. Also, C limits the case types to ints; **trygve** supports a much broader set of types, including Strings.

# Class-Oriented Programming

The most popular programming languages today include Java, C++ and C#. While most people call most of these object-oriented languages, in fact, one writes objects in none of them. Most of the focus of these languages is on *classes*. (One can view Javascript as a more true object-oriented language, but we won't discuss that in detail here.) Classes go back at least to the Simula 67 programming language which many herald as the first, er, object-oriented programming language.

Classes are an extremely important construct whose main value is to capture the primary structure of a program. In most programming languages, classes are the factories from which objects are built, and the structure of any given object can be traced to its class. To a first approximation this is true in **trygve**. Classes represent our conceptualization of what the system *is*: they are the building blocks of modern programs. In most languages, they exist in the source code; i.e., classes can exist before you have a running program. In many programming languages (like **trygve**, but unlike C++) these classes live on into run-time.

Classes incorporate many of the building blocks of previous-generation programming languages, combining procedural scripts with declarations into one tidy package where the parts fit well together and build on each other to offer more complex chunks of reasoning than procedures alone can. So if we need a stack, we can actually create a single entity that quacks like a stack, walks like a stack, and flies like a stack. We implement that entity as a class, where the class is the assembly of all of the scripts and declarations we would otherwise use to implement stack functionality in an old language like FORTRAN or Pascal, or in C or some other assembly language.

Classes have a close relationship to another important programming building block called the *abstract data type*, exemplified in a language called Clu. Much of the focus of early C++ work was to provide abstract data types (ADTs) to C programmers. The "abstract" in *abstract data type* refers to the usual way of articulating them as just a collection of APIs that distanced the client from having to worry about how things were implemented internally. These APIs are like contracts for users of an object.

In some sense ADTs exist only in our mind (or they may show up in system architecture documentation where we want to conceptualize the big pieces). However, like all languages, computing languages should be good at communicating the stuff we care about and work with, so it would be good to be able to actually *write* an ADT. And in **trygve** we can: the artefact is called an *interface*. **An *interface* is a contract by which some client uses an object**.

There has been much confusion around this notion over the years. It's important to keep thinking of an abstract data type as something that's kind of abstract: about whose internals we do not have to bother. Of course, there must be something concrete behind that abstraction if we are to execute real programs! Sometimes ideal computer scientists get lost in the abstraction and forget about the need for concrete stuff. And that's what classes are. **Classes are the implementations of abstract data types**.

Let's start concretely and then climb up the abstraction ladder.

## A concrete, complex class

Those of who who have started to take algebra know about complex numbers. A complex number is a number with two parts: a *real part* and an *imaginary part*. Engineers and other folks in the physical sciences use them to model many physical phenomenæ. If you don't have experience with complex numbers, for the time being just take our word for it that they're useful and that each one is really two numbers rolled into one.

We can write a class for a complex number like this:

class Complex {

public Complex(double realPart, double imaginaryPart) {

this.realPart\_ = realPart;

this.imaginaryPart\_ = imaginaryPart

}

public Complex(double realPart) {

realPart\_ = realPart; imaginaryPart\_ = 0.0

}

public Complex() { realPart\_ = imaginaryPart\_ = 0.0 }

public Complex +(Complex other) {

double resultingRealPart = other.realPart() + realPart\_;

double resultingImaginaryPart = other.imaginaryPart() + imaginaryPart\_;

Complex retval =

new Complex(resultingRealPart, resultingImaginaryPart);

return retval

}

. . . .

public String toString() const { . . . . }

public double realPart() const { return realPart\_ }

public double imaginaryPart() const { return imaginaryPart\_ }

private double realPart\_, imaginaryPart\_;

}

Most (but not all) class names start with an upper case letter. (The **trygve** language currently does not enforce this but it's good to be attentive to this very common convention.)

### Classes and Constructors

Let's take apart this class one piece at a time. At the outermost level is a wrapper:

class Complex {

. . . .

}

These lines wrap a declaration, where the declaration, well, *declares* the class Complex. We fill in the declaration with our instructions to the **trygve** language about what this class should do. The first script is a special script called a *constructor*, and it is used to initialize objects of class Complex:

public Complex(double realPart, double imaginaryPart) {

this.realPart\_ = realPart;

this.imaginaryPart\_ = imaginaryPart

}

The **trygve** environment calls this constructor on our behalf when we create new objects of class Complex. You notice that the constructor has two parameters, realPart and imaginaryPart. We need to supply these parameters when we create the object. We do that like this:

Complex myComplexNumber = new Complex(1.0, -1.0)

This will create a new Complex object with a real part of 1.0 and an imaginary part of ­–1.0. When we make this enactment request, realPart becomes a name for the object representing 1.0 and imaginaryPart becomes a name for the object representing -1.0. It remembers these numbers by associating them with identifiers inside Complex itself, and we find their declaration at the very bottom of the class:

private double realPart\_, imaginaryPart\_;

You may notice that the declaration is prefixed with the word private. This is called an *access modifier*. It controls what scripts can interact with realPart\_ and imaginaryPart\_. By making these two identifiers private we're telling **trygve** that their objects should be accessible only to scripts inside class Complex itself. By the same token you notice that most of the scripts are declared as public, which means that the scripts can be enacted by any stakeholder that is using the Complex class.

Back to the constructor — we stored away these parameters by saying:

this.realPart\_ = realPart;

this.imaginaryPart\_ = imaginaryPart

What this means is: "take the object named by realPart and also bind it to the realPart\_ identifier that is inside the current object, this", and then to do the analogous thing for the imaginaryPart. The identifier this is always bound to the object to which the current script belongs. When referring to members of the current object we can usually omit it; that is, the above lines could just as well have been written:

realPart\_ = realPart;

imaginaryPart\_ = imaginaryPart

You'll also notice that these identifiers end with an underscore ("\_") character. This is a convention used when naming object declarations inside a class; again, it is up to you, and not up to the compiler, whether you want to do this.

Next, we find this script:

public Complex(double realPart) {

realPart\_ = realPart; imaginaryPart\_ = 0.0

}

It looks a little like the other one but takes only one parameter, which is the real part of the complex number. It is another constructor. This is used if we want to create a complex number whose imaginary part is zero. We can say

Complex myComplexNumber = new Complex(2.0)

Of course, this is identical to writing:

Complex myComplexNumber = new Complex(2.0, 0.0)

and either is acceptable. And there is a third constructor:

public Complex() { realPart\_ = imaginaryPart\_ = 0.0 }

that just initializes the new complex object with both its real and imaginary parts zeroed out.

### Teaching the Compiler about New Types

The next script is a bit more interesting:

public Complex +(Complex other) {

double resultingRealPart = other.realPart() + realPart\_;

double resultingImaginaryPart = other.imaginaryPart() + imaginaryPart\_;

Complex retval =

new Complex(resultingRealPart, resultingImaginaryPart);

return retval

}

This script defines what **trygve** should do if we ask a Complex object to enact the "+" script. That looks like this:

Complex c1 = new Complex(1.0), c2 = new Complex(2.0);

Complex c3 = c1.+(c2)

That would cause the realPart\_ identifier in c3 (a new object, different from either c1 or c2) to be bound to a double object with the value 3.0. Of course, this syntax looks pretty strange and computer-like, and while it works, **trygve** recongises this equivalent way of saying the same thing:

Complex c1 = new Complex(1.0), c2 = new Complex(2.0);

Complex c3 = c1 + c2

which is just how you learned it in algebra class.

There is something fundamental going on here, and it's worth making it explicit. When we're writing code like this, we're teaching the compiler how to recognize new types. The **trygve** environment already recognizes a "+" operator (as well as "-", "\*", "/", and a few others) on int and double types. The compiler-writers baked that into the compiler when they wrote it. Now, **trygve** is giving you a chance to effectively extend the definition of the language. You are teaching the compiler how to work with a new type: Complex. It's a type you created. Of course, all the scripts of the Complex class do this in varying degree, but it's all the more obvious here with the ability to redefine the semantics of an ordinary operator. You can't do this with all operators.

You can also change the say that the relational operators (like "<" and "==") work. These operators are special in that there is some redundancy across them. That is, if you know "<" and "==" you can write the code for ">" in terms of those two (if neither "<" nor "==" is true then ">" must be true) and you of course can trivially write the code for "<=". It can be tedious and error-prone to make all of these work consistently — and the consequences of getting it wrong can be very confusing or even disastrous.

So **trygve** takes a simpler approach, inspired by Java. If you have a class for whose objects you want to support these operators, you just give That Class the following script:

public int compareTo(ThatClass other) { . . . . }

It is a very simple function: it must return an int object and it must take one parameter, which is usually of the same type as the class in which we install it as a script. If a is less than b and we call a.compareTo(b), it should return -1. If a equals b then it should return 0. And if a is greater than b then it should return 1. The trygve environment will take care of the rest, and will arrange to call compareTo on your behalf when you use any of the built-in relational operators. You can of course enact the compareTo script directly. But it is not allowed to define a method whose name is any of the relational operators. We teach the compiler about comparison with the addition of this one simple function: the compiler will guarantee consistency across all the relational operators.

### Accessors: Everyday scripts

Last, we find some pedestrian scripts that just answer our questions about the values of the real and imaginary parts of the object, or which render the object as a String:

public String toString() const { . . . . }

public double realPart() const { return realPart\_ }

public double imaginaryPart() const { return imaginaryPart\_ }

There is nothing mysterious or special about these, except they allow us to access the information hidden away in the objects attached to the private identifiers. They are sometimes called *accessors*. You notice that the modifier "const" follows the first part of the declaration. This tells **trygve** that these scripts promise not to modify any of the object state when they execute. Any script may be made const but **trygve** will enforce it. You should strive to make as many of your scripts const as possible.

You should also make it a habit to include a toString script for most classes you write. It comes in handy when you are displaying them for human consumption. Most built-in types have a toString script built-in.

## An Alternative Implementation of Complex, and Interfaces

Here is another class:

class Complex {

public Complex(double realPart, double imaginaryPart) {

theta\_ = Math.asin(imaginaryPart / realPart);

r\_ = Math.sqrt(realPart \*\* 2 + imaginaryPart \*\* 2)

}

public Complex(double realPart) {

r\_ = realPart; theta\_ = 0.0

}

public Complex() { r\_ = theta\_ = 0.0 }

public Complex +(Complex other) {

double resultingRealPart = other.realPart() + realPart\_;

double resultingImaginaryPart = other.imaginaryPart() + imaginaryPart\_;

Complex retval =

new Complex(resultingRealPart, resultingImaginaryPart);

return retval

}

. . . .

public String toString() const { . . . . }

public double realPart() const { return r\_ \* Math.asin(theta\_) }

public double imaginaryPart() const { return r\_ \* Math.acos(theta\_) }

private double r\_, theta\_;

}

It also implements complex numbers! The first one is often called "Cartesian complex numbers" and this one "polar complex numbers"? Would we ever want to keep both of them around in our library of classes? Maybe. Maybe one or the other would be particularly suitable to some set of algorithms because its approach would prove more efficient than the other. Yet they both faithfully represent complex numbers. From a logical point of view they are completely interchangeable. We say that they are alternative implementations of the type Complex. Remember: a class is an *implementation* of a type.

Let's say that someone wanted to write some generic code that abstracted away the difference between these two implementations. We make that code dependent on the common interface to these two classes. To that end **trygve** offers a construct called an *interface*, which is a kind of declaration. It looks the public bits of the class with all the implementation parts thrown away:

interface Complex {

public Complex(double realPart, double imaginaryPart);

public Complex(double realPart);

public Complex();

public Complex +(Complex other);

. . . .

public double realPart();

public double imaginaryPart() ;

}

The interface will bear the name Complex: it represents the essence of the concept. It is a pure abstract data type. It compiles, but it's really empty — there are no real scripts or declarations here. It is a kind of a contracts that shapes how the implementations must be written.

Given that the name Complex is now taken, we have to rename our two previous attempts. (We were already in trouble because we can't have two classes with the same name at the same level in the same system.) Let's call them CartesianComplex and PolarComplex. Each one will be rewritten as a new declaration with an explicit allegiance to Complex:

class CartesianComplex implements Complex {

public Complex(double realPart, double imaginaryPart) {

realPart\_ = realPart; imaginaryPart\_ = imaginaryPart

}

. . . .

}

class PolarComplex implements Complex {

public Complex(double realPart, double imaginaryPart) {

theta\_ = Math.asin(imaginaryPart / realPart);

r\_ = Math.sqrt(realPart \*\* 2 + imaginaryPart \*\* 2)

}

. . . .

}

Somewhere in our code we'll find a client of Complex numbers — in some script of some class. Maybe that method receives a Complex number from one of *its* clients . We might write the script like this:

class SomeClass {

public Complex someScript(Complex oneNumber, Complex anotherNumber) {

Complex retval = oneNumber \* anotherNumber + 3.13 \* oneNumber;

return retval;

}

}

The client of this class can choose to work with it in terms of a CartesianComplex or a PolarComplex, or both, at its whim:

CartesianComplex one = new CartesianComplex(1.0), two = new PolarComplex(2.0)

SomeClass someObject = new SomeClass();

someObject.someScript(one, two)

Alternatively, we could have written:

Complex one = new CartesianComplex(1.0), two = new PolarComplex(2.0)

SomeClass someObject = new SomeClass();

someObject.someScript(one, two)

So we see that Complex works as a generic type. Classes can be used to create types that behave just like the built-in types int and String, and we can use a class name just about everywhere we can use the name of a "regular" type like String. Interfaces can be used to group together alternative class implementations that may be used interchangeably. We can also use interface names just about everywhere that an ordinary type name can appear, with one exception: we can't of course say new InterfaceTypeName.

If you declare that a class implements an interface, the **trygve** environment will ensure that it has the right to do so. That is, every script published in the interface must also be implemented by the class. And only script declarations may appear in the interface — no object declarations!

It's not common, but a class may implement multiple simultaneous interfaces:

class PolarComplex implements Complex, ConvertibleToString {

. . . .

}

This gives the class the ability not only to be used anywhere a Complex is expected, but also anywhere any other ConvertibleToString is expected. We might have another class which is not a Complex but which is a ConvertibleToString:

class ClassMate implements ConvertibleToString {

. . . .

public String toString() {

String retval = this.firstName() + " \"" +

this.nickName() + "\" " +

this.familyName();

return retval

}

}

So if we had a ClassMate object in hand, and if we add this toString script, we could ask the object to run that script to give us a String representation of the classmate's name. Maybe in some application we want to keep all of our complex classmates in an array together with a bunch of Complex numbers, to collect all of our "complex" things together, so to speak:

ConvertibleToString [] complexThings = new ConvertibleToString [2];

complexThings[0] = new ClassMate("Smith", "John", 1969);

complexThings[1] = new CartesianComplex(5.0);

We could then print out the entire array, which is now a mixed collection of objects of different classes — but all of *type* ConvertibleToString:

for (String printable : complexThings) System.out.println(printable)

### What is abstraction?

By the way, we used two sophisticated words in the last section — *generic* and *abstraction*. These two words are often misunderstood.

Abstraction means to simplify by throwing information away. Throwing information away is always dangerous because, most of the time, we need every bit of information we can get! But sometimes too much information is just too much. If you're a manager and you want to know about your employees' progress on some project you probably should be getting regular updates on the coarse, big picture. To look at every facet of every everyday task is called micromanagement. So good managers abstract away the details. To do that takes trust.

If one piece of software trusts another piece of software, it can make the interaction easier to manage by using abstraction. We demarcate a boundary between the two pieces of software (e.g., between a client method and the class of the object from which it is receiving service) with an abstraction boundary. A good class interface is an abstraction boundary, and that boundary is punctured only at the points of public declarations. Because it is devoid of any implementation, an interface is an almost perfect abstraction boundary.

If we don't have to care about the details of how classes do their work, then the interface can represent a generic interface to one or more implementations. It not only can better hide the implementation of a single class but can make the differences between several alternative classes transparent. We still get to choose between the implementations (e.g., for engineering reasons) but that decision can be decoupled from the way our code interacts with the "provider of choice." It's like buying generic soap: we care what it does without caring too much about how it was made or who the provider is: yet at some point we make a decision to buy a certain kind of soap, and we are free to change our decision next time. This balance between complete interchangeability, and the techniques that make substitutability transparent, is maybe the central theme of class-oriented and object-oriented programming.

### Interface Guidelines

Classes, objects, interfaces — so many concepts! There will be more. And we'll return to this topic later to clarify which tool to use when. But it's important at this point to say something about interfaces. Interfaces live in the world of classes and of the major system building blocks. They do not live in the arena of the system's functionality, of its use cases. We have another language facility to cover that: Roles.

So good interfaces organize together classes without regard to the appearance of their objects in use cases. It is a "purely structural" classification. Taking this common engineering rule to heart will make it easier for us all to understand each others' code, and increases the amount of information conveyed by an interface declaration — because we can assume that the designer is trying to use it to organize the classes, rather than to unify any use case logic.

## Generic Classes

One of the uglier but most celebrated parts of C++ is its template system. What started as a text-substitution facility to stamp out cookie-cutter clones of a common glob of code eventually made it into the compiler as a language feature, Java adopted this features and, first, sadly called it "generics," though it is just one form of generic typing, and second, implemented using an approach that suffers from "type erasure" that takes away most of the clever text-substitution power. It has nonetheless become a staple of how Java programmers deal with collections, and we include it in **trygve** for the sake of familiarity. In some sense, it is a poor man's substitute for manually generated broad, shallow inheritance hierarchies.

Generic classes are related to the above discussion of interface. We can use an interface to group together multiple classes whose implementations are wildly different. The interface declaration stipulates the rallying point of what those classes must have in common.

Generic classes are much the same except that the source code for all of the classes is exactly the same. How they differ is in the types of objects they interact with. There are always business needs that arise where we want to capitalize on very low-level commonality between classes, but in a way that would be inconvenient for the programmer to highlight in the code. For example, the List generic may care only that its objects all have a compareTo operation, yet there may be no guarantee that all such classes adhere to a single industry-standard interface called Comparable that ties them together. (Such an interface doesn't historically exist at all in C++ and though the Java library provides such an interface, there is no guarantee that all classes with comparable behaviour pay it homage). So generic classes were created. They allow the compiler to actually analyze the code of the classes that the generic is interacting with and to bind the generic to its client objects at the points where scripts like compareTo exist. This made it possible for a single generic class to use classes from several different, uncooperative sources according to their adherence to common conventions, in the absence of the engineering discipline to create type groupings at an industry level or even a project level. Without that discipline it is impossible to leverage interfaces, but generic classes come to the rescue for low-level concerns.

That said, you *can* use generic classes to express high-level groupings among the classes of a cooperative bunch of good people. However, in that case it's almost always better to use interfaces instead.

So, down to nuts and bolts. The **trygve** environment comes with some built-in generic types, one of which is List. One declares and creates Lists like this:

List<int> intList = new List<int>()

Read as a single concept, List<int> is a class. As a class it behaves like any other class. There may be another class List<double>. However, these two classes have no more to do with each other than class Apple and class Orange. But the phrase List<int> may be used anywhere a class name may be used.

Though they can't interact in any powerful way at run time, all List classes offer exactly the same set of facilities to the programmer. I can add items to a List:

intList.add(1); intList.add(2)

or I can ask its size:

int size = intList.size()

or ask for the index of where a given item appears in the List:

int theIndex = intList.indexOf(2)

A List is very much like a vector, except it grows at run-time according to need. You can find a more complete description of List and other generic collections in the index. What's important to note here that, as a generic class, List can broaden to handle a wide range of types. So by analogy to the above, we can also write:

List<String> stringList = new List<String>();

stringList.add("one"); stringList.add("two");

int size = intList.size();

int theIndex = intList.indexOf("two")

### Defining generics

NOTE: We strongly discourage you from creating your own generics.

You define a generic class like this:

class MyGeneric <Parameter1, Parameter2> {

. . . .

}

The parameters are placeholders for type names that the client will provide. These parameter names can be used within the generic declaration just as other type names may be used:

class MyGeneric <Parameter1, Parameter2> {

Parameter1 myFunction(Parameter2 foo) {

Parameter1 retval = new Parameter1(foo.aNumber());

return retval

}

}

Try doing that in Java.

## Inheritance

Inheritance is a relationship between classes. In short, we start with a *base class* that approximates the result that we want. We create another class called a *derived class* that describes a delta of additions to the base class to get us where we are going, and we use inheritance to put them together to create a new class (named after the second class) that incorporates an increment of functionality over the original. The syntax looks like this:

class DerivedClass extends BaseClass {

// the definition of DerivedClass declarations and scripts

}

One original purpose of inheritance was to realize code reuse: the derived class "reuses" the code of the base class. However, experience over the years has dampened enthusiasm about inheritance as a reuse feature. First, it's not clear that it's any more powerful as a code reuse mechanism than one procedure reusing the code of another. Second, reuse is first an economic phenomenon and second a technical approach, so something has to be useable (and used) before it can be reused. Good reuse may be a natural consequence of good domain analysis and design, and inheritance plays a very small role in it, if any. Third, inheritance obfuscates code. Even they it did aid reuse, derived classes end buried deep in inheritance hierarchies. Inheritance hierarchies have a habit of becoming unmanageably deep and messy beyond the point of convenient utility. In a nutshell, the preferred approach to code reuse is just to encapsulate an object of one existing class inside of the new one we're building. Generics also had their heyday as a harbinger of code reuse but they jury is out on this. The famous and well-designed Standard Template Library (STL) of C++ has come under attack for not being suitable to the different performance profiles that different applications require.

Third, inheritance was originally interwoven with genericity in early OO programming languages. Smalltalk in particular used inheritance as a way of documenting the fact that one object could be generic with respect to another. Base class objects were more generic in that they were less refined, carried less information, and were more general than their derived class counterparts. A derived class object could do anything a base class object could do and, if designed well, could be used anywhere a base class object was expected. So class Rectangle might be a generic base class useable by any programmer in the general public, with derived classes Square and GoldenRectangle for use by ice-cube makers and artists, respectively. This notion of genericity is tied up with how we classify things in the real world, which we want to express in our programs.

Things got out of hand when languages attempted to extend this idea into complex design. Inheritance nominally leads to hierarchical structures, with a root base class at the top with a small collection of derived classes, each of which may (or may not) have its own derived classes, and so forth. A hierarchical organization of classes is equivalent to a Venn diagram with no overlap between circles: only total enclosure of one circle by another. Within an enclosing circle, enclosed circles are disjoint. The problem is that real-world classification is more complex than can be expressed that way, particularly as regards how we organize objects into classes according to their use — i.e., their place in use cases. So if we have the set of Men and set of Women and set of Other, as well as the set of drivers and non-drivers, we cannot reasonably use classes and inheritance to capture our mental model of this system.

The canonical failure mode can be demonstrated with a naive use of inheritance to describe the concepts of Vehicle, Boat, Plane, and SeaPlane:

Boat

Seaplane

Plane

Vehicle

Let's say that a vehicle must have a registration. How many registrations does a Seaplane have? Well, it might have two: one with the aviation authority and the other with the marine authority. Let's say that a Plane has a seat and a Boat has a seat. How many seats does a Seaplane have? For reasons of this problem (called the "diamond problem" after the rough shape of the graph) and other related problems, multiple inheritance has not thrived. The problem is that each of the parents talks about the inherent properties of the object by which we classify it, rather than talking about the behaviours we can expect from each of these entities which form a strong base for how we classify them. (Children, by the way, are apt to classify things by their properties at an early age. If a child learns that something with four legs is a horse, and the four legs are what make an impression on the child, then the child is apt to also call a dog a horse when seeing it for the first time.)

However, the world *is* more complex than hierarchy alone can describe. However, we are trying to view the word in terms of the objects and the external services they provide, rather than attempting to manage our understanding of them in terms of understanding the details of their private innards. We already have a way to organize classes into multiple hierarchies: interfaces. We do not use inheritance for that sort of organization.

What is inheritance good for, then? It's a good question. The **trygve** environment makes limited use of inheritance itself, and it's for the sake of convenient code reuse. There are a few utility scripts like assert that live in class Object. Every **trygve** class is implicitly derived from class Object. This is sometimes useful if we want to manage a collection of totally unrelated things: we can put anything into an object declared as Set<Object>.

# Object-Oriented Programming

Object-oriented programming might best be defined as the use of a network of objects working together to solve some problem or produce some result. Earlier definitions of object-orientation were more nerd-based and talked about the paradigm in terms of encapsulation or polymorphism — the main tenets of the technology development of that era. But that is a nerd's perspective, and we want to put more programming power back in the hands of people who use computers to amplify their mental muscles. And if we go back to the earliest foundations of object-orientation in Alan Kay's work, we again see this focus on the human element and on connecting the human mind with the surrounding world.

Even more commonly, people confuse class-oriented programming with object-oriented programming. Class–oriented programming is basically just abstract data types broadened beyond the traditional type realm of languages based on arithmetic and mathematical building blocks. Object-oriented programming is about understanding, modeling, and dovetailing with real-world phenomenæ such as managing books in a library, controlling an airplane flight, running a telephone switch, or optimizing fuel consumption in a car's fuel injection system. These are characteristic of the areas of the surrounding world into which we wish to extend the human mind. These are complex systems that cannot be appreciated or understood in terms of the properties on classes or the operations, such as inheritance, used to combine classes together. We deal with classes in the dozens; we deal with objects in the thousands.

Let's look at those first words again: "a network of objects working together to solve some problem." (I had originally written "community of objects" which is socially more apt to the sense of the term, but phrase "network of objects" offers better insight into what we need to build to make this happen). It is this network that is the program. Where do you go in your source code to understand the structure of this network? Where do you go in a class to understand the nature of the interaction between its objects and other objects in this network? You don't find this in Java or in C++ or in C# except to the very limited degree that inheritance or templates or generics say something about the composition of minuscule collections of classes or objects working together. And even then, those language constructs rarely answer the interest questions about program execution, about "what will happen next in the program?"

And in most object-oriented languages it is in fact almost impossible to reason, from the source code, about such interactions. The reason is that objects interact with each other with an interaction mechanism that has a built-in wildcarding transfer of control commonly called *polymorphism*. (To be technically correct, polymorphism is much broader than this, and this mechanism is the popular way of implementing one form of polymorphism called *inclusion polymorphism* by Peter Wegner). If we have a reference to some object, our object can ask that other object to enact a script of a certain name that takes a certain set of arguments. But we do not, from the source code, know for sure what the class of that object will be at run time. We can at best reason one or two steps ahead in execution. What we lose is the *structure of the flow of execution over time*. We know the structure of individual classes, but we cannot reason about the logic of the business-level services that a network of cooperating objects offers to its constituency.

In a nutshell, we don't understand our programs any more. We understand only their parts. In object-oriented programming we need to understand the whole. Our goal with **trygve** is to give a home to the expression of this network of connection between objects and to raise the level of the business logic to the level of that network, instead of burying it in the classes.

The analysis people understood this very early on. UML came on the scene in the 1990s, and Ivan Jacobssen had already developed a vision for use cases. Use cases are what we deliver. People who buy our software rarely buy it according to the classes it contains, but according to the use cases it supports. Modern programming languages haven't caught up. Antique programming languages like FORTRAN took expression of the work flow as their primary expression of structure all along, but the object-oriented pendulum slammed away from this direction in the 1980s, and it kind of stuck over on one side.

The trick is to do so without slicing objects in half — that part that relates to what they *are*, and that part that relates to what they *do*. When we hire an actor to play Hamlet, the actor has certain physical characteristics that owe to his DNA and which *intrinsically* have nothing in particular to do with being able to play Hamelt. But of course he can't play Hamlet without being able to talk or to hold a skull in his hands, and as an actor, he does this as an integrated being, Indeed, much of the success of good theatre is to support a willful suspension of disbelief so the audience, and even the actor to some degree, believes that the actor actually *is* Hamlet. Of course, at another level, we know that the Hamlet-ness of the actor lies in a script and his handsome demeanor and booming voice owe to his genes. But when he acts there is a single Hamelt on the stage: not a person with a Hamlet part and a separate I-was-hired-as-the-actor part.

So, on one hand, we have two separable concerns: the DNA of the actor's humanness on one hand, and Shakespear's script on the other. The publishing company has likely extracted all of Hamlet's lines and put them together in a separate little book, together with the cues (lines from other Roles, or happenings on the stage) that will call Hamlet into action. These are called cue scripts and actors find them useful for memorizing their lines.

And so we will do it in **trygve**. We have scripts for wonderful plays, such as making a fllght reservation, or transferring money between bank accounts, or planning a release schedule. Each of these scripts is written in terms of Roles that characterize the objects in the network of interacting objects. We have a whole variety of actors suitable to playing those Roles. For the flight reservation system there may be many different sizes and speeds of airplane, and each one may be associated (in its network) with a set of flight schedules that affect its own departures and also those of other airplanes. We have different Roles of people and things who will be on that airplane: pilots, attendants, first class passengers and coach passengers, their luggage, and commercial content that firms are shipping via the common carrier. The Roles, of course, are scripted. Each Role may have one or more scripts. The actors give those scripts life, combine with their own, more humble local scripts for moving their lips and limbs. It is by enabling actors with the appropriate Role scripts that we bring the network of objects to life, and then we say: Lights, Camera, Action!

## A caveat, and not throwing the baby out with the bath water

Sometimes class-oriented programming has served us well over the past 30 years; sometimes not. There can be a million differentiating factors, but one of them has to do with the difference between two kinds of business interactions. In one kind of interaction, an end-user poke on a screen causes some very short burst of activity which both the end user and programmer view as atomic. If I'm using a presentation editor, changing the color of a circle I have drawn is an example of such a computation. Writing a whole use case for it is probably overkill: it's simple, understandable, and atomic. Let's in fact call these atomic interactions. They may involve a small number of closely interacting objects.

The other kind of interaction unfolds over time and may entail multiple success computations, decisions, and branches in direction without intervening human interaction. The epitome of such a computation is running a batch program. While we tend to equate batch processing with card readers and the approaches of the 1960s, some major business programs still do account reconciliation as a nightly batch run instead of doing things on-the-fly. But even in-the-small, some computations entail rather complex algorithms by which a network of objects cooperate to give some result. Simulated annealing is one example. Perhaps making an airline seat reservation is another. These are use cases, and these are where

## Some new programming constructs

If we go back to our "network of objects working together to solve some problem," we find analysis and design concepts that usually cannot be expressed in many programming languages. We give each of these perspectives an articulation in a **trygve** program.

* Use Cases in a **trygve** program are given expression in Contexts. There is usually one Context declaration per use case. A Context realizes a use case through interactions between objects, in terms of the *Roles* those objects play. A Context implements a special kind of type and in that sense it is in the same category as Classes are. We declare a Context with the context keyword in **trygve**.
* Roles in **trygve** both serve as the name of an object (called the *Role-player*) which plays the role at run-time and describe the scripts for a given Role. A Role exists only as text in the program; they don't exist at run-time. You will find a Role only as a declaration inside a Context. We declare a Role with the role keyword in **trygve**. Roles also look a little like classes except they may contain only script declarations (no objects), just as there are no real people or princes or castles in a *script* of Hamlet. However, any object can dynamically ingest all the scripts of a Role at run time for the sake of participating in some network of objects in a Context. It is Roles, and they way they interact, that give a Context its structure and meaning. When the Context goes away (the use case comes to an end) the objects forget the scripts from the Roles they were playing; they may come back to play those Roles again, or they may move on to play another Role in another Context.

### Roles: the heart of what-the-system-does

Roles are just special names for objects, with the understanding the names have the power to convey meaning. Roles are part of how our mental models make sense of the world around us, and they tend to be slightly more general characterizations of objects than we'd find in any single, specific scenario. (Remember, a use case is a *collection* of *related* scenarios all compressed together. We'll discuss the *genericity* of Roles below).

Let's explore a bit further the concept that Roles are a way to name obects. Meaningfully calling someone a Server develops some expectations of what that person is expected to be able to do in a certain Context:

context TakeFoodOrder {

role Server {

void explainIngredients() { . . . . }

void registerOrderForCustomer(MenuItem food, Customer customer) {

. . . .

}

void describeDishOfTheDay() { . . . . }

}

}

This is *my* mental model of how a Server behaves in the *Context* of taking a food order. It is a perception; it is not truth. There is no single best way to represent either Roles or Contexts. I happened to write this according to my mental model. In a good software product, the software people will spend a lot of time building consensus in the mental models across all the stakeholders with dialogue and feedback.

Here Is another Context in the same domain:

context ServeFood {

role Server {

void unfoldAndPlaceServiettes() { . . . . }

void putFoodOnTable() { . . . . }

void removeCoversFromFood() { . . . . }

}

}

Notice that again we have the role Server. But this is not the same Server role as was described in the TakeFoodOrder Context. It can be confusing because it's likely *that both Roles are played by the same Role-player* (let's call him James). But the two Roles are totally unrelated. They do not interact with each other. They make no assumptions about each other.

## An example

## Design tips

### What makes a good Context?

A Context expresses a use case in code. A use case is an encapsulation of several related scenarios, all of which strive to attain some business goal. Use cases are a widely misunderstood formalism, but we can summarise them here. The essential parts of a use case are:

* **The name**. The name should come from everyday business terminology rather than computerese. The name should be short, crisp, and concise. Example names might be: Find Shortest Path (as for Dijkstra's algorithm); MoneyTransfer (in a financial system); Call Forwarding (in a telecom system). The Context name usually follows the use case name.
* **The goal**. A use case strives to attain a goal. Find Shortest Path has, as its goal, to find the shortest path. Money Transfer has, as its goal, to decrease the balance on one account and increase it on another. Call forwarding has, as its goal, to re-route a call to a third party. Note that sometimes the use case will not attain the goal because of extenuating circumstances. Money Transfer may fail because there is not enough money in the Source Account. Call Forwarding might fail because the line is busy on the number to which the call is to be transferred. In Dijksta's algorithm, there may be no path at all between two given nodes, let alone a shortest one. Handling each of these situations is a mini-scenario within the use case. But it's important to recognize that a use case *strives* to achieve the goal, though it may not always reach it.
* **Pre-conditions**. What must be true to even consider trying to run this use case? For Find Shortest Path we need an already set-up network of interconnected nodes. For Call Forwarding we need a call in progress, in a stable state, and a designated third-party number to which to forward the call. Whether the number has already been validated is a business question, and it may or may not find its way into the preconditions depending on the business rules of the real case-at-hand.
* **The main scenario**. This is also called the "success scenario." It describes the sequence of script enactments and their corresponding Roles, ordered according to their business dependency. We usually number the steps in a sequence: 1, 2, 3, 4… In a use case (the analysis artefact) the ordering expresses logical business dependencies, but not time. Usually, if *a* depends on *b*, we'll evaluate a before b in time. However, we leave the time decision to the programmer. The rationale behind this is that both *a* and *c* may depend on *b*, and from a business perspective there is no other dependency between *a* and *c*. The use case does not constrain the design by adding any ordering constraint on *a* and *c*. Because **trygve** enactments are sequenced along a single time thread, the programmer will have to make a decision about which one to evaluate first. The programmer has complete freedom to evaluate either one first since the use case provides no business constraint on the ordering, and the programmer may find that one or the other leads to a cleaner design. However, this does point out a fundamental difference between use cases and Contexts. Use cases leave ordering decisions open; Context bind them to a concrete ordering based on implementation insight or, in the worse case, an arbitrary decision.
* **Variations**. These are the alternative scenarios, and we describe them as increments on the main scenario. We usually describe them in terms of the main scenario step that is modified. So we maybe describe a step 2.1 as an alternative to the step 1 of the main scenario. There may also be a step 2.2. In Call Forwarding we may need to take an alternative branch through the use case if the phone of the third party is busy; in Dijkstra's algorithm, we may need to deal with the situation that no path can be found. Notice that such "error conditions" are not exceptions: we don't deal with them by throwing an exception and expecting someone else to handle the problem. We handle the problem in an orchestrated fashion, close to the source of the problem. If the problem must be escalated to a higher level, the use case specifies what Roles will handle the exception, and how, through the appropriate sequence of enactment. This is why there is no exception handling in **trygve**: we want to be able to understand what will happen by reading the code, rather than just guess or to defer the decision.
* **Post-conditions**: What must be true when all the enactments have completed successfully? What, at least, must be true even if the goal was not reached? Testers love these, and they help tie up what are otherwise unanswered design questions.
* **Dependencies**: What other use cases does this use case depend on? Call Forwarding certainly depends on Two-Party Call being in place and working. Dijkstra's algorithm may depend on no other pre-existing use case.

A good Context embodies all the scenarios for a use case: that is, it handles all the variations and weaves together the scenarios into one body of code. A use case separates out extensions as physically separate sections. We can do that in a Context but it almost always makes more sense to weave them together. So if we're implementing the use case Phone Call Origination and the main scenario says that the call becomes connected to the called party (pronounced "call Ed party", with the emphasis on "Ed"), there must be an extension that describes what to do if the called party is busy. Instead of writing two separate scripts, we'll have a single script, in which we'll probably find an if statement that leads the script to do one thing if the called party is available and another if the party is busy. We of course in this case can put the "busy handling" code in one script in its own right, but one should be careful about making every alternative into a script. That turns the problem of understanding a large script into one of being able to find the right script pertinent to the situation at hand.

### What makes a good Role?

**State**

Good Roles are stateless. The trygve language enforces against you declaring any objects inside of a Role. Of course, you can cheat and move some of the Role's state to the surrounding Context. While the language can't protect you from yourself here, it's not a good idea.

Why?

**Genericity**

The thing that people most misunderstand about Roles Is that they are a focal point for genericity in a **trygve** design. Most plays describe roles that can be played by a wide variety of actors; we can find many suitable actors and actresses, respectively, to play Hamlet and Ophelia. But in the movie *The Holiday* we find a role for Dustin Hoffman, which, in the cinematic milieu, is unlikely to be played by anyone but Dustin Hoffman.

This metaphor is in line with a one-time discussion in the community about the Role / object contract. It was suggested that the Role Account state in its requires section that it required a double. That is like the Dustin Hoffman role. We can think of Dustin Hoffman as a (singleton) object of class Dustin Hoffman, and to make him a Role confuses the concept of Role with the concept of class. If the code requires an object of class double, there is not much to be gained by making it a Role, except maybe to give it a more meaningful name within the domain. That might be fine except that, in fact the requirements for an Account are both more than (e.g., a currency attribute) and less than (no square root) those for a double. Thinking this way probably demonstrates a failure to let go of class-oriented thinking.

Another example can be found in an earlier attempt to implement Towers of Hanoi using **trygve**. The obvious roles are Disk and Stick. However, the only object that can play the Role of Stick is a Stick, and similarly for Disk. There is no genericity in the expression of the Role: it maps directly onto a class.

Good Roles accommodate unforeseen types of objects as Role-players. As such, they represent archetypes in our mental models rather than individuals or complete classifications. And by "archetype" here we imply that there will come many different individuals patterned after the original. Much of the power of Roles comes from the ability to accommodate such a wide variety of kinds of objects, without regard to their class.

# 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. Date(2016, 12, 25) is Christmas 2016. | |

**Method Summary**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Modifier and Type** | **Method and Description** | | **int** | **getDate()**  Get the day of the month, 1-indexed from January | | **int** | **getMonth()**  Get the month of the year, 1-indexed | | **int** | **getYear()**  Get the Gregorian year | | **int** | **getDay()**  Get the day of the week, 1-indexed from Sunday | | **void** | **setDate(int date)**  Set the day of the month | | **void** | **setMonth(int month)**  Set the month of the year, 1-indexed from January | | **void** | **setYear(int year)**  Set the Gregorian year | | **void** | **setDay(int day)**  Set the day of the week, 1-indexed from Sunday | | **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 | | **boolean** | **remove(T element)**  Remove the item that equal-compares with element. Return true if the element was found and removed and false otherwise. | | **T** | **remove(int theIndex)**  Remove the item at index theIndex and return it as the return value. | |

**Class Set<T>**

All implemented interfaces:

None

Direct known subclasses:

None

public class template **Set<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** | | **Set()**  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 set | | **int** | **indexOf(T element)**  Get the (zero-based) index of the first appearance of the indicated element in the Set | | **boolean** | **contains(T element)**  Return true or false according to whether the Set 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 | | **boolean** | **remove(T element)**  Remove the item that equal-compares with element. Return true if the element was found and removed 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.

# Design Decisions

Den 04/01/2016 kl. 01.18 skrev Andreas <notifications@github.com>:

This may be a little naive example, but I was considering an everyday

reasoning like "if I have money and the fridge is empty, go shopping."

Would you check the fridge even if you discover that your wallet is empty?

Why do you stipulate which one must be done first?

A cardinal rule of writing use cases is: Don’t introduce sequence unnecessarily. This is a perfect example. So the use case articulation would be in terms of both conditions, and if we weren’t limited by linear text, it would be communicated in a way that neither comes before the other.

The practical reason for this analysis rule is that introducing sequence constrains the programmer from what might otherwise be a more efficient implementation.

I see DCI as closer to the use case world; we have said so many times. Writing the code according to a sequence stipulates unnecessary conditions on the business process. It introduces invisible business rules. One implicit rule in your example is that my wallet is the only place with money. Maybe there is some in the safe. Maybe there is a use case extension whereby I stop by the ATM to get money on the way. Maybe there is an extension on the extension that I discover that my ATM card is missing. Or that I can’t find my wallet.

If you are really a control freak and want to control the order in which your users do things (insist that they check their wallet before going to the fridge) make it explicit in the code:

if (moneyInWallet()} {

if (fridgeNeedsSomething()) {

. . . .

}

}

The statement

if (moneyInWallet() && fridgeNeedsSomething()) {

should not, in my opinion, stipulate any ordering by virtue of any fact that there might be some obscure language rule that stipulates left-to-right evaluation. In any case, it is as important to be able to express that we *don’t* care about the order as that we *do* care. That’s a human concern and is really fundamental to the mental model. Rune’s argument is closer to a computing efficiency concern. I want the language to express concepts in human terms. Stipulating an ordering not only robs me of being able to say some things about a mental model, but gives the impression that the software is controlling the person’s behaviour (in arbitrarily mandating the order in which they do things). That’s in general considered a bad idea.

The Pascal designers recognized this and, apart from the blocked if approach above, actually made it possible to distinguish between these two situations within the logical operators. I consider the other languages to be either 1. sloppy in their thinking or 2. overly focused on implementation effiiciency for our concerns here.

Ada is the same, and we find the following accompanying advice: “Shortcut operators are used to make the evaluation of parts of boolean expressions conditional: *and then*, *or else*. This should never be done to speed up the evaluation (with modern optimizing compilers, it will possibly not have that effect). The correct use is to prevent the evaluation of expressions known to raise an exception.”

And all these languages have exceptions. Note that the following logical operators are \*not\* evaluated left-to-right in any language I know of:

a == c || d != e && f

The order of evaluation is ==, !=, ||, &&.

Yeah, try explaining that to a 7-year-old, while at the same time conveying the importance of partial evaluation.

# Adding a New Library to the Framework

1. Create a new class in info.fulloo.trygve.add\_ons. It must have a setup method which adds the type and class to the global scope:

globalScope = StaticScope.globalScope();

globalScope.declareType(*scannerType\_*);

globalScope.declareClass(classDecl);

1. In info.fulloo.trygve.code\_generation, open InterpretiveCodeGenerator.java. Find the implementation of the method @Override public void compile(). Add a block of code for your new type analogous to the other blocks of code:

typeDeclarationList = ScannerClass.*typeDeclarationList*(); // "Scanner"

compileDeclarations(typeDeclarationList);

1. While you're in InterpretiveCodeGenerator, find compileMethodInScope. Add an entry for your new type:

} **else** **if** (typeDeclaration.name().equals("Scanner")) {

processScannerMethodDefinition(methodDeclaration, typeDeclaration);

**return**;

Also add the corresponding procedure.

1. In info.fulloo.trygve.runtime, create a class for the run-time object representing an instance. Usually this class will go in its own new file.
2. In RTExpression.java, find class RTNew. In the method run add an entry to create the initial instance.

} **else** **if** (classType\_.name().equals("Scanner")) {

newlyCreatedObject = **new** RTScannerObject(rTType\_);

1. In StaticScope.java, at the end of initializeBuiltIns, add a call to the setup method of your new class:

**ScannerClass.setup()**

# Adding a new method to a library

1. Add the code for the method in the run-time class for the method. For example, adding join to String would add this code. (String happens to be in RTObjectCommon.java. Other libraries can be found in the modules in info.fulloo.trygve.add-ons and in StaticScope.java)

RTStringObject join(**final** RTObject string, **final** RTListObject listOfStrings) {

// NOTE: Pseudo-static method; "this" is unused

**final** RTStringObject rTInsertingString = (RTStringObject)string;

**final** String insertingString = rTInsertingString.stringValue();

**final** List<String> listCopy = **new** ArrayList<String>();

**final** **int** listSize = listOfStrings.size();

**for** (**int** i = 0; i < listSize; i++) {

**final** RTStringObject aString = (RTStringObject)listOfStrings.get(i);

listCopy.add(aString.stringValue());

}

**final** String sRetval = String.*join*(insertingString, listCopy);

**final** RTStringObject retval = **new** RTStringObject(sRetval);

**return** retval;

}

1. In RTClass,find the run-time class for your type (in this case, RTStringClass). Add a class for the new code you are writing:

**public** **static** **class** RTJoinCode **extends** RTStringCommon {

**public** RTJoinCode(**final** StaticScope methodEnclosedScope) {

**super**("String", "join", *asList*("delimiter", "elements"),

*asList*("String", "List<String>"),

methodEnclosedScope,

StaticScope.*globalScope*().lookupTypeDeclaration("String"));

}

@Override **public** RTCode runDetails(**final** RTObject myEnclosedScope) {

**assert** myEnclosedScope **instanceof** RTDynamicScope;

**final** RTDynamicScope dynamicScope = (RTDynamicScope)myEnclosedScope;

**final** RTStackable delimeterObject = dynamicScope.getObject("delimiter");

**final** RTStringObject delimeter = (RTStringObject)delimeterObject;

**final** RTStackable elementsObjects = dynamicScope.getObject("elements");

**final** RTListObject elements = (RTListObject)elementsObjects;

**final** List<String> listCopy = **new** ArrayList<String>();

**final** **int** listSize = elements.size();

**for** (**int** i = 0; i < listSize; i++) {

**final** RTStringObject aString = (RTStringObject)elements.get(i);

listCopy.add(aString.stringValue());

}

**final** String sResult = String.*join*(delimeter.stringValue(), listCopy);

**final** RTStringObject result = **new** RTStringObject(sResult);

addRetvalTo(dynamicScope);

dynamicScope.setObject("ret$val", result);

**return** **super**.nextCode();

}

}

1. In InterpretiveCodeGenerator find the "processXMethodAddition" for your type. Write code suitable to add the above behaviour to the type.

**private** **void** processStringMethodDefinition(

**final** MethodDeclaration methodDeclaration,

**final** TypeDeclaration typeDeclaration) {

. . . .

} else if (3 == formalParameterList.count()) {

. . . .

} **else** **if** (methodDeclaration.name().equals("join")) {

code.add(**new** RTStringClass.RTJoinCode(methodDeclaration.enclosedScope()));

retvalType = RetvalTypes.***usingString***;

}

1. In StaticScope.java, add the code to link the method into the type:

private static void reinitializeString(final Type intType, final Type booleanType) {

. . . .

**final** Type listType = StaticScope.*globalScope*().lookupTypeDeclaration("List");

**assert** **null** != listType;

*addStringMethod*(stringType, "join", stringType, *asList*("delimeter, elements"),

*asList*(stringType, listType));

}