The Newspeak Programming System

The Ministry of Truth

July 11, 2025

1 Introduction

The Newspeak programming system consists of the Newspeak programming language and allied tools and libraries. This paper gives an overview of some of the platform's key features and reviews its current status. Much of the content is derived, often verbatim, from prior writings of the Newspeak team including [Bra25, Mir, Byk08].

Newspeak is an object-capability language in the Smalltalk [GR83] tradition. It differs from Smalltalk in a number of important ways[1] though the basic syntax is quite similar. See [Mac] for a quick tutorial on some basic syntax, or [Bra25] for the full grammar. The current implementation of Newspeak runs in the web browser.

The most important property of the Newspeak language is that it is a *message based* programming language. This means that all computation — even an object's own access to its internal structure — is performed by sending messages to objects. This implies that everything in Newspeak is an object, from elementary data such as numbers, booleans and characters up to functions, classes and modules.

Below is a simple example of a Newspeak class named Point. The class defines two slots x and y. Slots are declared between vertical bars. Slots are similar to instance variables, except that they are never accessed directly. Slots are accessed only through automatically generated getters and setters. If p is a Point, p x and p y denote the values stored in p's x and y slots respectively. Note that there is no dot between p and x; since method invocation is the only operation in Newspeak, it can be recognized implicitly by the compiler. Comments are demarcated between (* and *), and may be nested. Strings appear between matching single quotes.

[1] The key differences are: there are no global, class or pool variables; variables always have implicit accessors; no assignment operator; classes may nest; there is a top level program/class syntax.

```
class Point x: i  y: j  =  (* This section is the instance initializer *) (
  (* declare slots *)
  |
  public x ::= i.  (* ::= denotes mutable slot initialization *)
  public y ::= j.
  | )
( (* instance side *)
  public printString  = (
    ^ 'x = ', x printString, ' y = ', y printString
  )
)
```

The first pair of parentheses in the class declaration delimits the *instance initializer* where slots are declared and initialized. Slots that are not explicitly initialized are set to **nil**.[2]

Setter methods are denoted by the slot name followed by a colon, so that p x: 91 sets the x coordinate of p to 91. This is an example of Newspeak's keyword syntax. In general, the header of an N-ary method, $N > 0$, is declared id1: p1 id2: p2 ... idN: pN, where id1:id2:...idN: is the name of the method and $p_i, 1 \leq i \leq N$ are the formal parameters; each id_i: is a *keyword*. An invocation of such a method is written id1: e1 id2: e2 ... idN: eN, where the e_i are expressions denoting the actual arguments. The order of the keywords is significant and cannot be changed.

In addition to keyword methods, one may define methods whose name is composed of a sequence of special symbols such as +, &, | etc. These methods always take one argument and are written using infix notation, e.g., exponentiation in c ** 2. Here c is the receiver, and 2 is the argument. These methods are known as *binary methods*. Our example shows the use of such a binary method: the "," method of String which implements string concatenation.

Within the body of Point, the names x, y, x: and y: are in scope and can be used directly, as shown in the method printString (note that the caret (^) is used to indicate that an expression should be returned from the method, just like the **return** keyword in conventional languages). However, x and y denote calls to the getter methods, not references to variables.

Newspeak programs enjoy the property of *representation independence* — one can change the layout of objects without any need to make further source changes anywhere in the program. For example, if we chose to modify Point so that it uses polar coordinates, no modification to the printString method would be needed, as long as we preserved the interface of Point by providing methods x and y that compute the cartesian coordinates.

The class declaration evaluates to a *class object*. Instances may only be created by invoking a factory method on Point. Every class has a single *primary factory,* in this case x:y:. If no factory name is given, it defaults to new. The primary factory method's header is declared immediately after the class name.

[2] **nil** is also an object of course.

The formal parameters of the primary factory are in scope in the instance initializer. The slot declarations above include an *initialization clause* of the form ::= e where e is an arbitrary expression. For example x is initialized to the value of the formal parameter i using the syntax x ::= i.

The declaration of the primary factory automatically generates a corresponding method on the class object. When invoked, this method will allocate a fresh instance of the class, o, ensure that the instance initializer of the class is executed with **self** = o, and return the initialized instance o. To create a fully initialized instance of Point write, e.g.: Point x: 42 y: 91.

The factory method is somewhat similar to a traditional constructor. However, it has a significant advantage: its usage is indistinguishable from an ordinary method invocation. This allows us to substitute factory objects for classes (or one class for another) without modifying instance creation code. Instance creation is always performed via a late bound procedural interface. This eliminates the primary motivation for dependency injection frameworks [Jav].

1.1 Nested Classes

Newspeak class declarations can be nested within one another to arbitrary depth. Thus, a Newspeak class can have three kinds of members: slots, methods and classes. All references to names are always treated as method invocations, so any member declaration within a class can be overridden in a subclass. It is possible not only to override methods with methods, but to override slots, classes and methods with each other. For example, one can decide that in a particular subclass, a slot value should in fact be computed by a function, and simply override the slot with a method.

An example of class nesting is ShapeLibrary, a class library for manipulating geometric shapes, sketched below:

```
class ShapeLibrary usingPlatform: platform = (
  | (* We use = to define immutable slots *)
   private List = platform collections List.
   private Point = platform graphics Point.
  |
)
(
  public class Shape = (...)(...)
  public class Circle = Shape (...)(...)
  public class Rectangle = Shape (...)(...)
)
```

ShapeLibrary has a number of nested classes within it. Several of these are outlined in the figure. We elide the details of some class declarations, replacing them with ellipses.

Shapes are organized in a class hierarchy rooted in class Shape. The details of this code are unimportant to us here, except for one point: the classes refer

to each other. For example both Circle and Rectangle inherit from Shape (the name of the superclass follows the equal sign in the class declaration). Recall, however, that a name such as Shape cannot refer to a class directly. Rather, it is a method invocation that will return the desired class. This method is defined implicitly in class ShapeLibrary by the declaration of the nested class Shape. The method is available to all code nested within ShapeLibrary. This is how an enclosing class provides a namespace for its nested classes.

Top level classes are also known as as *module definitions*, and their instances are often referred to as *modules*. The next few sections discuss Newspeak modularity in more detail.

1.2 Imports

Code within a module must often make use of code defined by other modules. For example, ShapeLibrary requires utility classes such as List, defined by the standard collections library. In the absence of a global namespace, there is no way to refer to a class such as List directly. Instead, we have defined a slot named List inside ShapeLibrary.

The slot declarations used in ShapeLibrary differ slightly from our earlier examples. Here, slot initialization uses = rather than ::=. The use of = signifies that these are *immutable slots*, that will not be changed after they are initialized. No setter methods are generated for immutable slots, thus enforcing immutability.

When ShapeLibrary is instantiated, it expects an object representing the underlying platform as an argument to its factory method usingPlatform:. This object will be the value of the factory method's formal parameter platform. During the initialization of the module, the slot List will be initialized via the expression platform collections List. This sends the message collections to platform, presumably returning an object representing an instance of the platform's collection library. This object then responds to the message List, returning the desired class. The class is stored in the slot, and is available to code within the module definition via the slot's getter method.

The slot definition of List fills the role of an import statement, as does that of Point. Note that the parameters to the factory method are only in scope within the instance initializer. The programmer must take explicit action to make (parts of) them available to the rest of the module. The preferred idiom is to extract individual classes and store them in slots, as shown here. It is then possible to determine the module's external dependencies at a glance, by looking at the instance initializer. Encouraging this idiom is the prime motivation for restricting the scope of the factory arguments to the initializer.

1.3 Access Control

Readers will have noted modifiers such as **public** and **private** on members in our code examples. It is important to understand their exact meaning, as it differs somewhat from mainstream programming languages.

Members of a class may optionally be declared with an *access modifier*, which may be one of **public**, **protected** or **private**. If no access modifier is declared, the default is **protected**. Only **public** and **protected** members are accessible to subclasses.

The access rules depend on the syntactic form of the message send used to access the member. Each form has its own lookup rules.

Ordinary sends have the form $e\ m$ where e denotes the object receiving the message m. A message m identifies a member by name, and also contains any arguments. Ordinary sends can only access **public** members.

Self sends have the form **self** m. They can access all members declared in the receiver's class, as well as any **public** or **protected** members of superclasses.

Super sends have the form **super** m. They can access any **public** or **protected** members of the superclasses of the enclosing class.

Implicit receiver sends (sometimes referred to as *here sends*) have the form m. They can access any members declared in the surrounding lexical scope, and any **public** or **protected** members of the receiver's superclasses.

Outer sends have the form **outer** $C\ m$, where C is the name of a lexically enclosing class. They can access any members declared in C, and any **public** or **protected** members of C's superclasses.

One of the most common uses of **private** is for imports - slots used to hold classes passed in during object initialization. What imports a class uses should not be a concern of any other code, not even its subclasses. The implementation details of the class may change; it might use lists internally at one point, and thus have a slot like List = platform collections List. Later it might be changed so it uses maps; the List slot will be removed and replaced with Map = platform collections Map. The implementor should be free to make that change without impacting any other code, and so it is advisable to declare import slots **private**.

1.4 Modularity

Since the entire library is embedded within a single class, it is possible to create multiple instances of the library and use them simultaneously (side-by-side deployment). Different module instances do not interfere with each other, because there is no static state. Module definitions are therefore re-entrant.

It is also possible to provide *different* implementations of a library. For example, we could instantiate ShapeLibrary with different platform objects, which could provide different implementations of List with different characteristics (e.g., varying speed and memory requirements, logging capabilities etc.). Likewise, we could have multiple different implementations of ShapeLibrary itself. Since a library is accessed strictly via a procedural interface, functionally equivalent implementations may be used transparently to clients.

In all of the above scenarios, one can switch between libraries dynamically, store libraries in data structures, pass them as parameters etc., because libraries are represented as first class objects.

The use of nested classes to encapsulate entire libraries forms the basis of Newspeak's module system.

1.5 Platform Objects

Above, we saw that the ShapeLibrary factory expected a platform parameter. This is a common idiom. Newspeak allows different platform objects to be tailored to different needs. Examples might include:

- Platforms for different operating environments, such as Newspeak running on the web or on top of Smalltalk, or natively on a particular operating system.
- A platform that is specialized for command-line use, that does not include a GUI.
- Platforms that are restricted due to security concerns, such as a platform that does not support reflection or foreign function calls.

The platform object one commonly uses in the current system is an instance of the class Platform defined in the module RuntimeForHopscotchForHTML. It provides accessors for modules that provide basic library support (like kernel and collections), reflection (mirrors), the underlying web browser (js), the UI (hopscotch) and so on.

When Newspeak ran on top of Squeak Smalltalk, we used a different platform. We have variants of the web platform that run on Wasm (the current system) or JavaScript (RuntimeForJS, an earlier, incomplete variant that may be resuscitated at some point). We also have a version that supports live on-line collaboration (RuntimeForCroquet).

The use of platform objects makes it easy to run modules against a shared interface that may be common across several implementations of the system, without limiting these to a least common denominator. It also fits well with Newspeak's object capability security model described in section 2.

1.6 Class Hierarchy Inheritance

The dynamic binding of class names applies to a class' declared superclass as well, effectively making each class a mixin [BC90]. This ability to override classes has substantial repercussions, especially when coupled with Newspeak's support for nested classes. One can define an entire class library or framework as a set of classes nested within an outer class, and then modify the library via inheritance, overriding classes defined within it. This is known as *class hierarchy inheritance* [OH92].

We now extend our example to show how class hierarchy inheritance works in Newspeak.

```
class ExtendShapes withShapes: shapes = (
  | private ShapeLibrary = shapes. |
)(
    public class ColorShapeLibrary usingPlatform: platform  =
        ShapeLibrary usingPlatform: platform (
    )(
      public class Shape =  super Shape ( | color | )(...)
  )
)
```

The factory method for ExtendShapes, withShapes:, takes a single argument, shapes which should be a shape library class such as ShapeLibrary. The instance initializer imports this library under the name ShapeLibrary.

Nested within ExtendShapes is class ColorShapeLibrary, which inherits from ShapeLibrary. This shows that it is possible to subclass an imported class. The name of the superclass, ShapeLibrary, is followed by the message usingPlatform: platform. We do this in order to determine what parameters will be made available to the superclass' initializer. Before a subclass' instance initializer is executed, control passes to the instance initializer of its superclass in much the same way as constructors are chained in mainstream languages. Here we indicate that platform will be passed on to the superclass' instance initializer.

ColorShapeLibrary overrides Shape. It defines its own class Shape that subclasses the superclass' class of the same name. The overriding class Shape has added a slot, color. Since Shape is the superclass of all other classes in ShapeLibrary, they all inherit the new slot. Let us see how this occurs.

When the class declaration for Shape is overridden in ColorShapeLibrary, the automatically generated accessor method is overridden as well. Hence every attempt to access Shape on an instance of ColorShapeLibrary will produce the overridden class rather than the original.

Classes are generated lazily. The first time a class is referenced, its accessor manufactures the class and caches the result. Subsequent accesses always return the same class object.

As an example, consider the class Circle. This class is declared in ShapeLibrary and inherited unchanged in ColorShapeLibrary. It declares its superclass to be Shape. The first attempt to use Circle on an instance of ColorShapeLibrary will cause the class to be generated; this will require accessing the superclass. The superclass clause denotes a method invocation, not a direct reference to a class, just like every other name in a Newspeak program. Therefore, the new class will be a subclass of the overridden version of Shape, as intended. The same holds for all other subclasses of Shape in the library, as one would expect.

1.7 Application Assembly and Deployment

In the absence of a global namespace, it may not be obvious how one combines separately developed top level classes into a single application. Here we discuss some options; other variations are possible.

An application is typically constructed by instantiating a top level class T representing the application as a whole. T will likely depend on a number of separately compiled module definitions; its factory method should take these, and only these, as arguments. The Newspeak IDE provides us with an extra-linguistic namespace containing all classes used in development. It in this namespace that we will instantiate T. Use of the IDE namespace is analogous to how tools like make reference the components of an application utilizing the file system as a namespace.

Class T should have a method main:args: as its entry point. The method takes an object representing the underlying platform, and an array of command line arguments. The code in main:args: will instantiate the various module definitions imported by T, linking them together as required and then start up the application.

Below is an example of an application. Our application is a trivial one, an interactive counter.

```
class CounterApp packageUsing: manifest = (
  | private CounterUI = manifest CounterUI. |
) (
  public main: platform args: args = (
    |   ui = CounterUI usingPlatform: platform.  |
    platform hopscotch HopscotchWindow openSubject:
      (ui CounterSubject onModel: ui Counter new).
  )
)
```

By convention, the factory of an application is named packageUsing:. It takes a manifest, an object with accessor methods for every object the application depends on. The application uses the manifest to import its dependencies. In our example, there is only one dependency, CounterUI, which is the actual implementation of the counter widget. In main:args: the application creates an instance, ui, of CounterUI and uses ui's API to feed hopscotch, the platform GUI, with the necessary information to display the counter.

The IDE recognizes classes with a packageUsing: factory as applications, and will provide UI support for running and deploying them.

One deployment option is to use serialized objects as our binary format. We can serialize an instance of T, and later run it via a tool that deserializes it and invokes its main:args: method. This is similar to the classic C convention of invoking an application via a distinguished function main(). T is analogous to the C program, its main:args: method is analogous to the main() function, and the serialized instance is analogous to a binary file. Deserialization is the equivalent of linking and loading.

1.8 Additional Language Features

1.8.1 Actors and Eventual Sends

Newspeak supports asynchronous message passing via *eventual sends*.

remoteView $<-$: display

Asynchronous messages immediately return a *promise* as a result. A promise is initially in the unresolved state. When the asynchronous computation completes, the promise's state will change to either resolved, if it succeeded, or broken, if it failed. Unlike promises in other languages, you can keep sending asynchronous messages to a promise without having to check if it has resolved. These messages will be queued up , and when the promise resolves, invoked on the result. In other words, you can and should freely chain eventual sends. This is known as *promise pipelining.*

At some point, you will need to get an actual result out a promise - say, if you need to print something, or do arithmetic. To deal with such situations, promises have a method whenFulfilled:whenBroken: that takes two closures - one to be run in the case when the promise is resolved, and one in the case when it's broken.

Eventual sends allow you to proceed with your work while some other activity, such as I/O, proceeds concurrently. Let's examine how concurency is managed in Newspeak.

Newspeak uses an actor model of concurrency [Agh86] that is heavily influenced by the E programming language [Mil06]. Actors are objects with their own thread of control and their own heap. They share no state with other actors; they communicate exclusively via asynchronous messages. Actors are non-blocking, race-and-deadlock free, and scalable. Actors are exposed via the module platform actors. [3] This module has a nested class PromiseFactories which is useful for managing promises.

When an actor is created, it is *seeded* with an initial value - a deeply immutable object. The result is a *far reference* to a copy of the value in the actor's heap. [4] Far references serve as handles for objects that are not available in the heap of the currently executing actor. Far references only accept eventual sends.

If you send an object to another actor, the actor receives a *far reference* to that object. Similarly, an eventual send to another actor will resolve to a far reference to an object in that actor's heap.

Normally, the value used to seed an actor should be a mixin, which can then be applied to a superclass to produce a class that you then instantiate. Once you have this instance (or, rather, a far reference to it) you perform the desired computation using additional eventual sends.

For an extensive discussion of the Newspeak actor model, see [Bot12].[5]

1.8.2 Lazy Slots

It may be that certain data may never be required, so it is wasteful to allocate it before we know that we need it. For example, perhaps we have a need to

[3] In the current implementation, this is an instance of ActorsForPrimordialSoup.

[4] in reality, there is no need to copy the value unless the actor is running in a different Newspeak engine, but that is just an optimization.

[5] Note that this work uses an earlier version of Newspeak, with slightly different syntax and APIs. Beyond syntax and API discrepancies, some examples may not work due to bugs in the current implementation.

display an object that includes a large list of sub-objects . Often, the user will not want to view the object is such overwhelming detail, and so we may collapse the list. We don't want to allocate UI objects for each element in the list until they actually needs to be displayed. Yet we don't want to recompute the list unnecessarily either.

We could deal with this by defining a slot, elementsSlot to hold the widgets, and defining a method elements that populates the slot lazily.

```
elements = (
  elementsSlot isNil ifTrue:  [elementsSlot:: computeElements].
  ^elementsSlot.
)
```

Such situations are common enough that Newspeak supports them via *lazy slots.* We can write

```
lazy elements = computeElements.
```

This behaves much like the longer code above it, but avoids boiler-plate. It is more concise, leaves less room for errors, and spares the programmer the trouble of coming up with two names, one for the slot and one for the accessor. If we still wanted the option of explicitly setting the slot, we could write

```
lazy elements ::= computeElements.
```

As usual, the use of ::= in the slot declaration will produce a setter method (in this case elements:).

1.8.3 Mutually Recursive Slots

Modules are often mutually recursive. Newspeak facilitates this using *simultaneous slot declarations.* These are delimited by double vertical bars, rather than single ones as used for ordinary slots. A slot inside a simultanous slot declaration is set to a future for its initalization expression. The futures are pipelined so that any well founded mutual recursion between simultaneous slots will resolve properly.

1.9 Style Guidelines

At this point, we should make a few points about coding style. Generous use of whitespace is encouraged. In particular, a single space between keywords and formal parameters, and before and after type annotations is recommended. Likewise a space between keywords and actual arguments. For single argument keyword sends, if the argument is itself a keyword message, it is best to use the special setter form using *::*. So, instead of foo: (x bar: e), one should write
foo:: x bar: e.

Comments should precede the construct they are describing. This is important for the sytem to interpret metadata (see section 6.1 below) in particular.

This concludes our introduction to the Newspeak language. In the following two sections, we describe Newspeak's approach to security and reflection. In sections 4 through 5 we shift to a discussion of the Newspeak system, going

through the GUI and IDE, and the approach to interoperability with other languages.

2 Security

Newspeak's security model is founded on the object-capability model [Mil06]. In this model, the authority to perform an operation (which may have potential security implications) is provided exclusively through objects that act as *capabilities.* This places several requirements on the programming language. These include:

1. Objects must provide true data abstraction; they must be able to hide their internals from other objects - even other objects of the same class.

2. There must be no static state (e.g., static or global variables). Such state can be accessed by code that was not explicitly authorized to do so, providing *ambient authority.*

With respect to point 1, Newspeak supports object-level encapsulation. Object members that are **private** or **protected** can only be referenced within the scope of the object (see section 1.3). This is not the case in mainstream object-oriented languages such as C++, Java or C#.

Now consider point 2. A typical example of ambient authority might be a class File with a constructor that takes a file name and returns an instance that can access a file in the local file system. This is a standard design, but in a situation where file system access must be restricted, requires authorization checks on every access.

As we have already noted, there is no static state in Newspeak, addressing point 2. In Newspeak, each module runs in its own sandbox, created explicitly by providing the module with the desired capabilities (i.e., objects provided as parameters to the factory when the object was constructed). There is usually no need for explicit (and costly) security checks on individual operations to ensure that the caller has the appropriate authority to invoke them. The fact that the caller holds a reference to an object that can perform the operation conveys the necessary authority.

We illustrate the use of the object-capability model in Newpeak in our subsequent discussion of reflection (section 3) and foreign function calls (section 5).

Notwithstanding all of the above, the current Newspeak prototype does not provide any security guarantees. The language provides a foundation for security, but a secure system requires a lot more: careful API design, security audits, a secure binary format etc. These problems are topics for future work.

3 Reflection

Newspeak supports both *introspection* - the ability of a running program to examine its own structure - and *hotswapping*, which allows a program to modify itself while running. There is, however, a natural tension between reflective access and security. Newspeak uses a mirror based reflective architecture [BU04] to resolve this tension.

Whereas mainstream languages provide a uniform level of reflective access to every object via a standard method such as getClass(), in a mirror based system, reflection is mediated by objects known as *mirrors*. Mirrors serve as capabilities for reflection. Only code that is in possession of a mirror for an object can reflect on it. Different mirrors can provide varying levels of reflective access - for example, some mirrors might only allow introspection, others might restrict even introspection to the public API of a class, and others might provide unlimited access.

Mirrors can be obtained from the platform's reflection module. To reflect on a class c, you might write

```
cm: (platform mirrors ClassMirror reflecting: c)
```

The code above returns a class mirror on the class c, and sets the slot cm to hold it. A reflection module can choose to perform any additional authorization checks it may require, and return a mirror object that conveys the desired reflective capabilities. In the extreme case, one could provide a module that refused to return mirrors at all, effectively disabling reflection.

We can ask the class mirror for a mirror on its mixin:

```
mx: cm mixin
```

we can then ask the mixin mirror for its methods, slots or nested classes:

```
mtds: mx methods.
slts: mx slots.
clsss: mx classes.
```

Each of these operations returns a MirrorGroup representing the set of elements in question. Note that it is easy for a given mirror implementation to return a group that consists of some subset of the elements - for example, only the public members. The mirror group may be immutable, in which case it can be queried for specific elements, but not modified:

```
mtds mirrorNamed: #foo
```

The mirrorNamed: method looks up a method whose name matches the symbol #foo in mtds and returns a *method mirror* on it. This provides for introspection but prevents modifications to the running system. On the other hand, a mirror group may be mutable, allowing for changes, e.g.:

```
mtds addFromSource: 'twice: x = (^2*x')
```

It is clear from the above examples that the mirror API can be refined to provide fine grained control over what reflective operations are permitted. The design of a suitable API that provides the necessary degree of functionality and control is non-trivial and ongoing.

4 GUI and IDE

4.1 GUI

Newspeak's UI, Hopscotch [Byk08], is a general purpose reactive GUI framework. Interaction with Hopscotch combines desirable features of web browser style navigation with support for hierarchical decomposition.

Hopscotch is built upon the notion of *fragments*, which are resuable interactive widgets that may be composed via *fragment combinators*. Hopscotch provides a number of built-in fragments like links, buttons etc. It also supports *presenters*, which are user defined fragments designed to present a specific kind of data. These are analogous to views in most MVC UI frameworks.

Hopscotch imposes a navigational paradigm that is similar to the one used in web browsers. This allows for easy navigation of graphs of presenters. Presenters can be hierarchical, so Hopscotch applications are well suited to displaying hierarchical data (e.g., tree structures such as file systems or program code). The combination of navigation and hierarchy imposed by Hopscotch naturally handles the display of any hypergraph of user data.

The largest Hopscotch application is the IDE, described below.

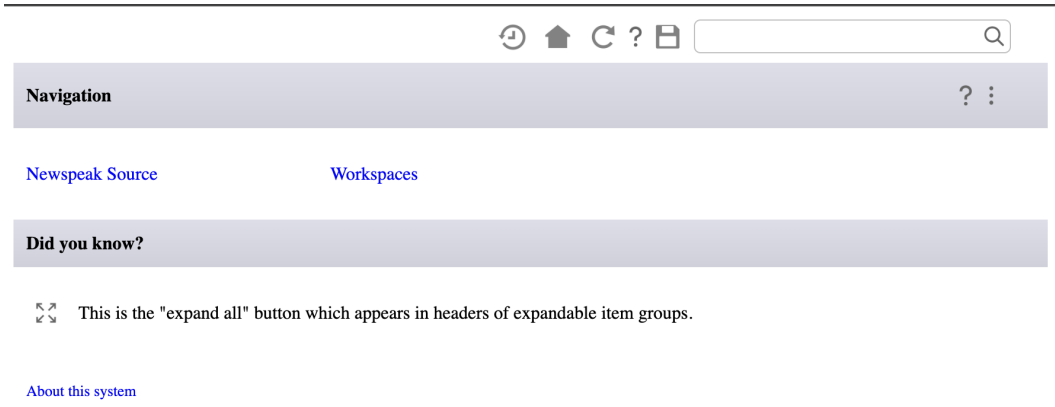
4.2 IDE

The IDE supports editing and browsing of code and documents, object inspection, interactive evaluation and debugging, unit testing and application deployment.

Unlike conventional IDEs, the Newspeak IDE is not centered around files. Files serve only for persistent storage of code. Browsing and editing are done using *class presenters,* which provide a structured view that supports live, incremental development. Changes to code take effect immediately, without a need for a costly build cycle. In fact, there is no such thing as a build. You edit individual methods, with no need to compile an entire class when a change is made.

To further facilitate live development, the system supports the use of *exemplars*: example instances of classes, or example activations of methods, which you can interact with while coding. The system will endeavor to provide you with concrete instance of a class when you browse that class, so you get a better sense of what data its instances hold, and can examine it interactively to see how it behaves. Likewise, when browsing a method, it will try to make actual activations of the method available. Exemplars are described in some detail in [Bra17, Bra21].

The best way to learn how to use the IDE is to work with it. Below is a screen shot of the Web IDE's home page, which is what you see when you open it for the first time.



Look for the ? icon, which opens up context specific help throughout the UI. When you open the IDE, you'll find the help button for the IDE as a whole at the top of the screen, a bit left of center. It will provide an explanation of the tool bar in which it is situated, which is always available. Below it you'll find another toolbar, with a light gray background. It has its own help button which, when clicked, will tell you about the IDE's home page. Based on these instructions, you can follow links to other pages, which will have their own help buttons (often several of them, for different parts of the page). If you go thru these systematically, you will find all the IDE's functionality clearly documented.

5 Interoperability and FFI

Newspeak includes facilities for calling out to other programming languages, and being called by them. The generic term for such facilities is a *foreign function interface* (FFI). Newspeak does not have a language feature (such as native methods or **extern** functions) supporting an FFI. Instead, foreign calls are mediated via objects known as *aliens* [Mir, Bra25]. Aliens are objects that provide access to foreign code. They are object-capabilities for accessing non-Newspeak code.

The exact same discipline of modularity and security used throughout the Newspeak system applies to foreign function calls. One cannot access foreign code unless one has access to alien objects. Newspeak's design makes it impossible to do otherwise.

The exact formulation of aliens depends on the host environment Newspeak is running in. In a native implementation, one typically needs to call out to C in order to interact with the host system. The current Newspeak implementation runs in the web browser, and one needs to run JavaScript to access the host system and interact with the outside world. Hence the need for JavaScript

aliens.

Newspeak code accesses JavaScript objects via instances of Alien, defined within JSForPrimordialSoup.

You may pass Newspeak objects as arguments when sending messages to aliens. The system will represent them to JavaScript as JavaScript objects; these are known as *expats*. In general, any JavaScript object coming into Newspeak (say, a result returned by a method, or arguments passed into a callback) is converted into an alien, and any Newspeak object passed into JavaScript is converted to an expat. When expats return to Newspeak they revert to their original Newspeak representation. Likewise, when aliens are passed back into JavaScript, they will also be transformed back into the underlying JavaScript object.

Some datatypes are handled specially by the alien system. Numbers get converted to floats when passed to JavaScript. Strings are converted back and forth between their native Newspeak and JavaScript representations. Newspeak arrays may not be passed into JavaScript at the moment, so you will need to create JavaScript arrays and copy their contents when sending array data into JavaScript. JavaScript arrays support the messages at: and at:put: for indexing, as well as their normal JavaScript methods and properties. Newspeak closures sent to JavaScript are represented via wrapper closures that convert their arguments and results recursively as needed. Closures are a common and important case, because of their use as callbacks.

The conventions for accessing a JavaScript object via an alien are as follows:

To access JavaScript, use platform js, which is an instance of JSForPrimordialSoup. The JavaScript global namespace is available via

platform js global

Below, we assume that this alien is available via a slot named global that has been imported. Through global, we can reach any JavaScript object.

1. To access properties of a JavaScript object, you use at: and put: to read and write the property respectively, so one writes alien.a as alien at: 'a' and alien.a = b as alien at: 'a' put: b. Examples:

   oldVisual.firstChild in JavaScript would be oldVisual at: #firstChild and node.contentEditable = "false" is written in Newspeak as

   node at: #contentEditable put: 'false' .

2. To invoke a method on a JavaScript object:

   If there are no arguments, use the method name alone, so alien.m() becomes alien m. Since you are not in the JavaScript global scope, methods that are global must be accessed via an import (as noted above, we use global). Example: window.getSelection() in JavaScript would turn into global window getSelection. Or, we might import window itself, and then write window getSelection. The latter is usually better style.

   If there is one argument, append a colon and the argument to the method name. The general pattern is that alien.m(e) becomes alien m: e . Examples:

setTimeout(1000) becomes global setTimeout: 1000.

document.createTextNode(label) becomes document createTextNode: label, (assuming document was imported).

If the method uses $n > 1$ arguments, you use a keyword message of the form $m: e_1\ kw_2: e_2 \ldots kw_n: e_n$, where m is the method name and $kw_2 \ldots kw_n$ are arbitrarily chosen keywords. Example: you would write oldVisual.replaceChild(newNode, oldNode) as

oldVisual replaceChild : newNode oldChild: oldNode.

The second and later keywords don't matter; choose names that are clear and useful to you. We could have written

oldVisual replaceChild: newNode insteadOf: oldNode

with the same effect. As a matter of style, one should always use the same name for a given multi-argument JavaScript method.

3. The rules for invoking constructors are similar, except that one uses $m = new$. So, to invoke a constructor with no arguments, send the message new. Example: JSObject new, (assuming we imported JavaScript's Object under the name JSObject to avoid confusion with Newspeak's Object). To invoke a constructor of one argument, use new:. Example:

   JSArray new: fs size (again, we've chosen to import JavaScript's Array under a distinct name).

   For more than one argument, use $new: e_1\ kw_2: e_2 \ldots kw_n: e_n$, where $kw_2 \ldots kw_n$ are arbitrarily chosen keywords. Example:

   Blob new: data options: iterableOptions.

6 Additional Features

In this section, we'll go over features of the Newspeak system that have not been discussed above.

6.1 Metadata

Newspeak code may include *metadata comments* that provide useful information to various tools. For example, metadata is used to define exemplars for the IDE. Here is the header for a class BankAccount [Tea21] that uses metadata for this purpose.

```
class BankAccount balance: b (* :exemplar: BankAccount balance: 100 *) = (
|
        private balance_slot  ::= b.
|
)
```

Note the comment on the first line.

The comment begins with a colon-delimited identifier :exemplar:. The use of a colon-delimited identifier is what identifies it as a metadata comment. The identifier is used to determine how to interpret the rest of the comment. Newspeak's reflection system provides access to metadata, and the Newspeak IDE interprets metadata comments whose identifier begins with exemplar as data indicating how to generate exemplar data for a give abstraction.

The remainder of the comment in our example is an expression that creates an instance of BankAccount. The IDE can use this information to create an exemplar instance of BankAccount. However, there is a question as to what scope should this expression be evaluated in.

In the case of top level classes, we evaluate the instantiation expression with respect to the IDE's root namespace. The same is true for class methods of top level classes.

A similar mechanism is used for methods. BankAccount has a method withdraw:, shown below:

```
public withdraw: amount
(* :exemplar: withdraw: 100 *) = (
  amount > balance ifTrue: [
    Error signal: 'Overdraft not allowed. Withdrawal amount ',
                  amount printString,
                  ' exceeds balance ',
                  balance printString
  ].
  balance_slot:: balance - amount
)
```

Here we see that the exemplar provides a sample invocation of withdraw:. The method invocation is evaluated in the scope of the enclosing instance - that is, the instance of BankAccount we derived from the metadata for the classes' factory above.

6.2 Types

Newspeak supports type annotations but does not enforce them. Eventually, the system will include a typechecker that will analyze your programs. It will be strictly optional, like a linter, and will never prevent your code from running. In the meantime, types are used strictly as comments.

Type annotations are delimited by angle brackets. and can appear in three places:

- After slot declarations (both the slots of a class, and those of a method). For example, we might define the slots x and y in Point via code such as | **public** x <Integer>::= 0. **public** y <Integer>::= 0. | .

- After formal parameter declarations in method headers, as in
  withdraw: amount <Integer> or
  class Point x: i <Integer> y: <Integer>.

- After method headers. e.g.,
  printString ^ <String> = (^ 'x = ', x printString, ' y = ', y printString)
  Note that in this case, we use the carat as part of the type annotation to make clear that it denotes the type of the object returned by the method. If no return type is specified, the system assumes the return type is Self which denotes the type of the receiver.

You can use the name of a class or mixin as a type. You can also write generic types, like List[Point] or Map[String, Object]. If the generic arguments are elided, they default to Object. There are a few special types predefined, such as Self, or Instance. The latter can be used on the class side to denote the type of instances of the class. The types of tuples and closures are written in a manner analogous to the literal expressions for these constructs.

Hence [:Class :Integer | String] denotes the type of a closure that takes a class and an integer and returns a string, [String] is the type of a closure that takes no arguments but returns a string. If the return type is elided, it defaults to Object.

Similarly, {} is the type of an empty tuple and { Integer. String. Float} is the type of a 3-tuple consisting of an integer, a string and a float in that order.

6.3 Minitest: Newspeak's Unit Testing System

Newspeaks unit testing framework is called Minitest. Support for Minitest is integrated into the IDE, so that it recognizes test classes and automatically instruments them with a UI to allow you to run tests and examine their results.

In Minitest, you define a testing module, which is designed to test a particular interface (not a particular implementation). To run tests, one needs to feed the testing module with the particular implementation(s) that one wishes to test. A test configuration module does just that. Newspeak naturally enforces this separation of interface and implementation.

Here is a testing module ListTesting. It is a very simplistic set of tests for lists. ListTesting's factory method takes 3 arguments: platform (the Newspeak platform, from which all kinds of generally useful libraries might be obtained), minitest (an instance of Minitest, naturally) and listClass, a factory that will produce lists for us to test. This is typical; the first two arguments to a test module factory are almost always a platform object and an instance of Minitest, while the third is the object under test.

```
class ListTesting usingPlatform: platform minitest: minitest listClass: listClass = (
  |
  private TestContext = minitest TestContext.
  private List = listClass.
  |
) (
  public class ListTests = TestContext (
    | private list |
  ) (
  public test_addition = (...)
  public test_removal = (...)
  ) : (
  TEST_CONTEXT = ()
  )
)
```

Nested within the module is the class ListTests, which includes the actual tests. Test methods are identified by the convention that their names begin with test. Each test will be executed in a test context; that is, for each test method being run, Minitest will instantiate a fresh ListTests object. That is why ListTests is called a test context - it provides a context for a single test.

It is common to define test context classes like ListTests as subclasses of the class TestContext defined by the Minitest framework. One reason why having a Minitest factory argument is useful is so we can import TestContext. TestContext provides useful methods like deny:, so it is convenient to use it. However, inheriting from TestContext is not essential. What identifies ListTests as a test context is the marker class method TEST_CONTEXT, not inheriting from TestContext.

Minitest will do its work by examining the nested classes of the test module and seeing which are test contexts (that is, those which have a class method named TEST_CONTEXT). For each test context *tc*, Minitest will list all its test methods (the ones with names beginning with test) and for each of those, it will instantiate *tc* and call the selected method on it, gathering data on success or failure.

We need a *test configuration* to run the tests, as the test module definition is always parametric with respect to any implementation that we would actually test.

A test configuration module is defined by a top level class with the factory method

packageTestsUsing: namespace

The factory takes a namespace object that should provide access to the testing module declaration and to any concrete classes or objects we want to test. This arrangement is very similar to how we package applications from within the IDE.

We show a single test configuration ListTestingConfiguration, but you can define as many you like.

The method testModulesUsingPlatform:minitest: must be provided by the configuration. It will be called by Minitest to produce a set of testing modules, each of which will be processed by the framework as outlined above (i.e., searched for test contexts to be run).

```
class ListTestingConfiguration packageTestsUsing: namespace = (
  | private ListTesting = namespace ListTesting. |
) (
public testModulesUsingPlatform: platform minitest: minitest = (
  ^{ListTesting
      usingPlatform: platform
      minitest: minitest
      listClass: platform collections List
    }
)
)
```

The IDE recognizes test configurations based on the name of the factory method - that is, a class with a class method packageTestsUsing: is considered a test configuration, and the IDE will provide a run tests link in the class presenter's upper right hand corner, as you can here:

6.3.1 More about Minitest

If you are used to SUnit (or any of the many unit testing frameworks it has inspired, like JUnit etc.), it may be worth noting some of the differences.

Minitest does away with concepts like TestResource that are typically used to hold data for tests.

In the simple case above, the data for the test gets created by the instance initializer of ListTests. However, what if the data for the test needs to be shared among multiple tests (say, because it is expensive to create)?

As an example, suppose we want to test a compiler, and setting up the compiler is relatively costly.

```
class MyCompilerTesting usingPlatform: platform
                        minitest: minitest
                        compilerClass: compilerClass = (
 | private Compiler = compilerClass. |
)
(
  class CompilerHolder = (
   | compiler = Compiler configuredInAParticularWay. |
  )(
    class StatementsTests () (): ( TEST_CONTEXT = ())
  )
)
```

Minitest leverages Newspeak's nested structure in these cases. A test context (StatementTests above) does not have to be a direct nested class of the test module. Instead, we can nest it more deeply inside another nested class (CompilerHolder). That nested class will serve to hold any state that we want to share among multiple tests - in our case, an instance of the compiler, which it will create and store as part of its initialization.

As you can see there is no need for a special setUp method or a test resource class. Newspeak's nesting structure and built-in instance initializers take care of all that. If the shared resource is just an object in memory, then it will also be disposed of via garbage collection after the test is run. Of course, some resources cannot be just garbage collected. In that case, one should define a method named cleanUp in the test context class.

Minitest cleanly breaks down the multiple roles an SUnit TestCase has. The definition of a set of tests is done by a test context. The actual configuration is done in a test configuration. And the actual command to run a specific test (the thing that should be called TestCase) is not the users concern anymore - the test framework handles it but need not expose it.

6.4 Documents

Ampleforth [Bra22] is an editor for rich text documents with embedded media, including arbitrary interactive user interface elements. These may themselves be Ampleforth editors.

The editor is written in the Newspeak programming language, and is integrated with the Newspeak IDE. Ampleforth documents are Newspeak objects. Each document has its own unique class. In the IDE, one can view a document in an object presenter, and edit code in the class with the document as the exemplar. It is thus easy to write code in the scope of the document and evaluate it live at any point.

A document contains an HTML markup program that defines the document's structure. The system allows for either WYSIWYG or markup editing (or a mix of both), and maintains a live bidirectional relation between the two.

Live UI widgets inside a document are known as *amplets*. To insert an amplet into the document, type Newspeak code into the WYSIWYG view, select it and click on the Make it an Amplet button. The code should evaluate to a Hopscotch fragment. The system will evaluate the code in the scope of the document and insert the resulting fragment.

Amplets are represented in the markup via HTML nodes of class ampleforth that include the Newspeak code that creates the UI in their name attribute. Amplets are cached and managed by the editor.

References

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computing in Distributed Systems*. MIT Press, Cambridge, Massachusetts, 1986.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming*, October 1990.

[Bot12] Nikolay Botev. Actor-based concurrency in Newspeak 4, 2012.

[Bra17] Gilad Bracha. Newspeak exemplar-mode demo from live literate programming talk at Programming 17. Video, May 2017. https://youtu.be/Yv7yX27Tx4U.

[Bra21] Gilad Bracha. Exemplars in the Newspeak web IDE. Video, August 2021. https://youtu.be/qKWPSvcF0zA.

[Bra22] Gilad Bracha. Ampleforth: A live literate editor. In *Live 22: The Eighth Workshop on Live Programming*, 2022. Available at https://blog.bracha.org/Ampleforth-Live22/out/primordialsoup.html?snapshot=Live22Submission.vfuel. Recorded talk at https://www.youtube.com/watch?v=gfbzC_90fJE.

[Bra25] Gilad Bracha. The Newspeak programming language specification, 2025. https://newspeaklanguage.org/spec/newspeak-spec.pdf.

[BU04] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, October 2004.

[Byk08] Vassili Bykov. Hopscotch: Towards user interface composition, July 2008. ECOOP 2008 International Workshop on Advanced Software Development Tools and Techniques (WASDeTT).

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.

[Jav] Java on Guice: Guice user's guide. Available at http://code.google.com/p/google-guice/.

[Mac] Ryan Macnak. Newspeak by example. https://newspeaklanguage.org/samples/Literate/literate.html.

[Mil06] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[Mir] Eliot Miranda. Newspeak FFI internal documentation. Available at http://wiki.squeak.org/squeak/uploads/6100/Alien%20FFI.pdf.

[OH92] Harold Ossher and William Harrison. Combination of inheritance hierarchies. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 25–40, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

[Tea21] Newspeak Team. Class BankAccount source code, 2021.