

# Newspeak Language Presentation TUG

## **Newspeak, November 2014. Presentation at the Toronto Smalltalk Users Group (TUG)**

Presenter: Milan Zimmermann

How did it happen? Chatting at the end of our September TUG session, I mentioned I have been playing with Newspeak. Bob Nemeč said something like “would you like to present it on our next meeting?”, and I said “yes”. Surely I am not the best qualified to present Newspeak, but I enjoyed immensely over the years reading Gilad Bracha’s [Room 101](#) blog and more recently playing with Newspeak.

A Note: This is a long document. I prepared it as a background for my talk, a few days before the presentation realizing it is way longer than what I can do it a 90 minute talk. The presentation was a rather small subset of this document.

Thanks TUG for giving me the opportunity to discuss Newspeak.

## **Credits: Gilad Bracha, all Newspeak authors and contributors**

Newspeak Authors

- Gilad Bracha
- Vassili Bykov
- Yaron Kashai
- Eliot Miranda
- Ryan Macnak
- contributors

All misunderstandings, errors and unintended misrepresentations are mine.

See [References] for full references and credits.

Material of this presentation is based on ideas and sometimes copied from material in the footnotes section.

## Newspeak: Where to start learning it

If I was to point out two links to start, they would be:

1. Gilad Bracha's blog site, [Room 101](#), is always a great education.
2. Newspeak starting page is <http://www.newspeaklanguage.org/>

## Newspeak: How to install

We will not go into much detail here, but provide two links

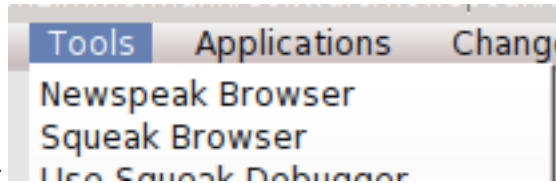
1. Direct easy install of the latest stable version (2013-09-14): <http://www.newspeaklanguage.org/downloads>. This is the much easier way. Steps to install :
  - Download zip for your OS
  - Unzip file to directory \$NEWSPEAKDIR,
  - run in shell or as appropriate: \$NEWSPEAKDIR/nsvmlinux/nsvm \$NEWSPEAKDIR/ns-101.image
  - (replace nsvmlinux with directory name for your OS)
2. Create your own latest boot image: Follow instructions at [https://bitbucket.org/newspeaklanguage/nsboot\\_bleeding\\_edge](https://bitbucket.org/newspeaklanguage/nsboot_bleeding_edge)

## Using code from this presentation

As Newspeak changes, if you want to run code from this document unchanged, please use the Newspeak version linked in the previous section (version from 2013-09-14).

**If you are interested in code, not so much descriptions, explanations etc, just scan this document for sections that look like code, and copy / paste it to the Newspeak IDE**

You can paste code such as class definitions directly from this text into the Newspeak Browser using following steps:



- Tools -> Newspeak Browser
- Categories -> click AATUG (this category must be created first)

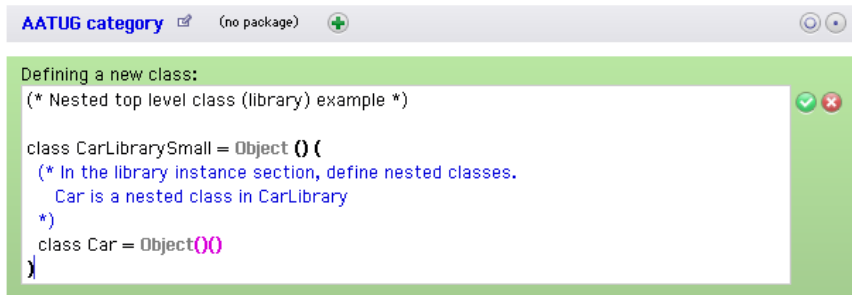
## Categories

AATUG  
 BrazilForMorphic-CustomMorphs  
 Collections-Abstract  
 Hopscotch-Core

- Click the + beside the AATUG will bring the new class definer section



- In the above, select, delete, and paste in a class definition from this document, for example



## Newspeak: Behind the name - George Orwell's 1984

Background to Newspeak name, settings, and references from George Orwell's 1984. The name Newspeak, the name of Gilad Bracha's blog site, [Room 101](#),

and references to The Ministry of Truth.

So direct quotes from Newspeak specs or articles in this presentation are sometimes introduced by “The Ministry of Truth claims”.

Links to sources are always provided at the beginning of chapters or inline.

## The Case for Newspeak

I do not know when the idea of Newspeak first came, but, from my understanding of reading Gilad Bracha’s blogs and articles is that his overall goal when creating Newspeak was to create a language that is in the tradition of Self and Smalltalk while being <sup>1</sup>:

- Purely object oriented and message based
- Dynamic and reflective
- Modular and secure
- “Shrinkable”
- Utilizing sound results of CS research in the last 30+ years (actors, mirrors, etc - this is not stated but feels that way)

Gilad Bracha has built up a **case for features included in Newspeak** (with arguments and discussion) in his blog **Room 101** between 2007 and 2010 (blog still continues).

- The blog is at [Room 101](#) and is an excellent reading for everyone interested in software languages and computing in general.

## Who is Newspeak for?

- In <sup>2</sup>, the Ministry of Truth says:
  - “One of the things that has surprised me working with Newspeak is how easy it is to convert Smalltalk code to Newspeak.”
  - “Still, if you are (or were, in some happier time) a Smalltalker and want to move into the future rather than dwelling on the glorious past, I assert that Newspeak is for you. If you are using an open source Smalltalk, it is likely you could do better using Newspeak.”

---

<sup>1</sup>[The Newspeak Programming Language](#)

<sup>2</sup>[Converting Smalltalk to Newspeak](#)

- “Newspeak explicitly addresses Smalltalk’s weaknesses: modularity, security, interoperability. Of course, some people aren’t bothered by these weaknesses. ”
- “Newspeak should interest those who appreciate the power of Smalltalk but want to move forward.”
- “Of course, you have to be an early adopter by nature. Things will evolve and change under your feet. The syntax will become less Smalltalk-ish over time ... (in the end) Your code will be much more maintainable and better structured.”
- As to “The syntax will become less Smalltalk-ish over time” - the changes that are discussed include: braces instead of brackets, accessibility control, replacing ^ with return: - note the column - etc

So it seems Smalltalk users were/are the intended primary target group for Newspeak.

To the presenter, Newspeak is a very interesting project that stimulates curiosity and gives a chance to study a language designed by the best.

### Migration from Smalltalk to Newspeak

- There is a document <http://bracha.org/Smalltalk2NS2.pdf> describing how to convert Smalltalk code to Squeak.
- I do not have enough Newspeak to prove this on any sufficiently sized program, but, according to The Ministry of Truth in , code converted from Smalltalk to Newspeak:
  - The converted code is better than the original.
  - It becomes clear:
    - \* what the code’s external dependencies are
    - \* what the module boundaries should be
    - \* who is responsible for initialization
  - There is no longer any static state.
  - Easier to tie libraries together (or tear them apart),
  - Easier libraries independent testing
  - Smalltalkers can migrate to Newspeak relatively easily.

I will do my best to show some of these points at least briefly on an example in the modularity section.

## Newspeak - As Advertised - Highlights from the Specs

This section and subsections is a “jot-dotted” summary of Newspeak highlights in the Newspeak specs <sup>3</sup>

High level ideas and goals for Newspeak. Newspeak is:

### Newspeak is: Network Serviced (Supported by *partially implemented synchronization*)

- Newspeak applications can be updated over the internet while running.
- The language supports orthogonal synchronization, making it straightforward to:
  - synchronize persistent data with a remote server
  - Support backup
  - Share and collaborate.
- *The synchronization features are in their early design stages, and only partially implemented.*

### Newspeak is: Class based

### Newspeak is: Message Based (and purely OO as a result)

- All computation - even an object’s own access to its internal structure - is performed by sending messages to objects.
- The only run time operation is a message send (no assignments).
- Hence, everything in Newspeak is an object, from elementary data such as numbers and booleans up to function, classes and modules.

Sidenote: It is an interesting exercise to think through how a purely message based system supports the principles generally associated with Object Oriented Languages and environment:

- Encapsulation
- Abstraction
- Polymorphism
- Inheritance

---

<sup>3</sup>[Newspeak Programming Language Draft Specification Version 0.091](#)

## **Newspeak is: Secure (Supported by encapsulation, no static state)**

- Newspeak objects encapsulate their representation, and Newspeak programs have no static state.
- These properties provide a sound basis for an object-capability security model <sup>4</sup>.
- *An essential component of this vision is dynamically enforced access control, which is not yet implemented.*

## **Newspeak is: Reflective (Mirror based reflection)**

- Newspeak programs are causally connected to their executable representation via a reflective API.
- Reflection in Newspeak is mirror based, with mirrors acting as capabilities - see Mirrors: Design principles for meta-level facilities of object-oriented programming languages, <http://bracha.org/mirrors.pdf>
- Given access to the appropriate mirrors (and only given such access), a running program can both introspect and modify itself.

## **Newspeak is: Modular (Independent, immutable, parametric namespaces)**

- Newspeak module definitions are independent, immutable, self-contained parametric namespaces.
- They can be instantiated into modules which may be stateful and mutually recursive.
- These modules are inherently re-entrant, because there is no static state in Newspeak.
- All inter-module dependencies are explicit.
- Modules and their definitions are first class objects that can be manipulated at run time.

---

<sup>4</sup>[Towards a Unified Approach to Access Control and Concurrency Control](#)

## Newspeak is: Concurrent (Actor based concurrency)

- Concurrency in Newspeak is based on actors.
- Actors:
  - Are objects with their own thread of control.
  - Share no state with other actors; they communicate exclusively via asynchronous message passing.
  - Are non-blocking, race-and-deadlock free, and scalable.
- Only a partial prototype has been implemented.
- Also note that the FFI (8.5) can undermine actor isolation as C can take state passed from one actor, store it globally, and return it to another actor. Non-blockingness also requires care, as a callback passed in by one actor can be invoked when C is called by another. Must ensure that said callback acts as a future, or fails (the former, to allow event processing).
- In an ideal world, one would only communicate with foreign languages running in a distinct actor. This would be more secure, and require less special handling; this was part of the original vision of Smalltalk. Newspeak is pragmatic in this regard; it remains idealistic, but only to an extent.

## Newspeak is: Optionally typed *Unimplemented*

- Newspeak supports pluggable types - see <sup>5</sup> .
- This allows the language to be extended with arbitrary type systems. These type systems are necessarily optional, and never affect run-time semantics. They utilize Newspeak's metadata facility (4.3), which allows annotations to be attached to any node in a program's abstract syntax tree.

## Newspeak - A few core principles

These two items are critical to allow some of the features described in the previous section:

1. **The only runtime operation is virtual method invocation** (message send in Smalltalk terminology)
  - So there are no variable assignments

---

<sup>5</sup>[Pluggable type systems](#)



- So even each object's access to it's internal structure uses method invocation
2. All names are late bound (also follows from 1)
  3. **There is no global namespace**
  4. There is no static state (follows from 3)

For a better description and more details, see <sup>6</sup>

## Newspeak - Below the Surface - Details, discussion, examples

Sections and statements of this paragraph are directly used from (or at least inspired by) and other items in references.

### Newspeak 101: How to define a class - basics

- In Newspeak, **there is a standard text representation of class declaration**. It can be pasted in to create a class, or can be saved from existing class. But let us first go to Newspeak.

- Open Newspeak, click Tools -> Newspeak Browser
- Go to an existing category, create a category AATUG (by clicking +)
- Select AATUG, click on the + to add class, paste the class definitions below. Repeat for each class.
- Newspeak simplest class declaration

```
(* Simplest possible class declaration. Note two sets of parenthesis *)
class Simple = () ()
```

```
(* Equivalent is: *)
class Simple1 = Object () ()
```

```
(* Another Equivalent is: *)
class Simple2 = Object () () : ()
```

```
(* Another Equivalent is: *)
class Simple3 = Object (||) () : ()
```

---

<sup>6</sup>[Modules as Objects in Newspeak](#)

- ```

(* What the above means: *)
class Simple4 = Object (instance initializer) (instance method definitions and nested

```
- ```

(* What the above means in detail - pseudocode: *)
class Simple5 = Object (|slotDefinitions|) (instance method definitions and nested

```
- ```

(* Again equivalent to (pseudocode, just indenting): *)
class Simple6 = Object (
  |slotDefinitions|
) (
  (* instance methods and nested classes definitions *)
) : (
  (* class method definitions *)
)

```
- Unlike Smalltalk, Newspeak has a text representation of code - you can paste the above code to the Newspeak IDE.
  - ==> Paste the above to the Newspeak Browser - explain and store classes.
    - \* put all as top level class, quick nesting note regarding top level classes
    - \* Newspeak class declarations can be nested
    - \* In the first bracket, slots are defined
    - \* In the second bracket nested classes and instance methods are defined
    - \* So Newspeak has three kinds of members: slots, methods, and (nested) classes.
    - \* It is possible to override slots, classes and methods with each other.
  - ==> Open a workspace, highlight a code section and click "Evaluate" the following expressions
    - \* Simple 'Evaluate'
    - \* Simple new 'Evaluate'
  - ==> inspect evaluation results

## Newspeak 101: The Newspeak Workspace

- ==> In workspace:

```

|x|
x == 20.

```

- Make sure always select everything you need to evaluate
- Use Evaluate, not ^D

## Newspeak 101: How to define a nested class (outer and inner class)

- In Newspeak, classes can be nested hierarchically.
- In fact, nested classes are a cornerstone of Newspeak modularity.
- Nested classes enable the mantra “everything is an object”: In Newspeak, **all applications, and modules (libraries) are just classes - top level classes.**
- All class examples we have shown so far was a top level class, but in practice **almost everything you do in Newspeak lives in a deeper level class- only applications and libraries(modules) are top level classes.**


(\* Nested top level class (library) example \*)

```
class CarLibrarySmall = Object () (  
  (* In the library instance section, define nested classes.  
   Car is a nested class in CarLibrary  
  *)  
  class Car = Object()()  
)
```

## Newspeak 101: Hello Brave new world (in Transcript) - An example of Newspeak application

This section describes how to create a Newspeak application “Hello brave new World”, following <sup>7</sup>

- ==> Start in the Newspeak browser AATUG category, and click the +

icon: 

Add this code:

```
class HelloBraveNewWorld usingPlatform: platform = Object (  
  |Transcript = platform blackMarket Transcript.|  
  Transcript open show: 'Hello, Oh Brave new world'.  
)(  
)
```

---

<sup>7</sup>[Newspeak on Squeak - a guide to the perplexed](#)

```
Defining a new class:
class HelloBraveNewWorld usingPlatform: platform = Object (
  ITranscript = platform blackMarket Transcript.I
  Transcript open show: 'Hello, Oh Brave new world'.
)(
)
```

Click on the checkmark -

Several comments about the code:

- The code in `()` is initializer
- `platform` in the `platform = (etc)` is a parameter to the initializer object. `platform` object encapsulates the underlying platform
- `blackMarket` is a message to `platform` object. Black Market is a temporary escape to the IDE's global namespace - and provide access to things like `Squeak Transcript` to which there are no Newspeak alternatives.
- The code in `Transcript open show: 'Hello, Oh Brave new world'.` will show the string in the Transcript. Because this code is in the initializer of `HelloBraveNewWorld`, it will be **executed when `HelloBraveNewWorld` is created.**
- There is one slot, named `Transcript`, initialized from the `platform` object.
- Declaring `Transcript` in the initializer is , as the dependence on `Transcript` is clearly localized to one point of declaration of the `Transcript` slot.
- Reader of the code can see all external dependencies of the `HelloBraveNewWorld` module in one place.
- This use of slots (of creating slots from external dependencies) is effectively **code import**, and allows to rename imported elements where it makes sense.
- How to run the Brave new World?
  - ==> Open workspace and type in `HelloBraveNewWorld usingPlatform: platform`, then evaluate.
  - Transcript will open, showing the message 'Hello, Oh Brave new world'
  - NOTE: if we did declare the application class as `class HelloBraveNewWorld = Object (Transcript open show: 'Hello, Oh Brave new world'.)`, running it would get a `doesNotUnderstand`, as there is no way to access any system state (`Transcript`, output stream, etc) without the system state being passed a parameter when the module is initialized.

- TODO How is the *platform* object, when running in workspace, created?

## Newspeak 101: How to define a more complex class - class Thing and it's subclass Car, both living in CarLibrary module.

```
class CarLibrary = Object()(
```

```
  class Thing = Object (
  )(
    (* instance methods, starting with a category string *)
    'category misc'
    printMe      = (^ 'I am a thing'.)
    'category test'
    testThis     = (^ 'Testing a thing'.)
  ):
    'category on the class side'
    aClassMethod = (^ 'I am a class method'.)
  )
)
```

```
class CarLibrary = Object()(
```

```
  (*
  - This Car class introduces message pattern as part of the class definition.
  - The message pattern /Car color: aColor/ is the *primary constructor for the class*.
  - /aColor/ is a formal parameter, which is in scope in the class body
  - The result of sending this message /color: aColor/ (to class Car) results in:
    - executes the instance initializer code /color := aColor./
    - creates a fresh car instance.
  - The slot /color/ is accessed only through automatically generated getter (/color/) and
  - Client example:
    - Car color: 'blue'.
  *)
```

```
  class Car color: aColor = Thing (
    (* initializer - section between vertical bars. *)
    |
    (* color is a slot. Slots are similar to instance variables, but they are never accessed
    only through automatically generated getters and setters. The getter name and usage is
    the setter name and usage is "color: newColor".
    slots setters/getters exist (among others) to enforce "the only runtime operation is
    *)
```

```

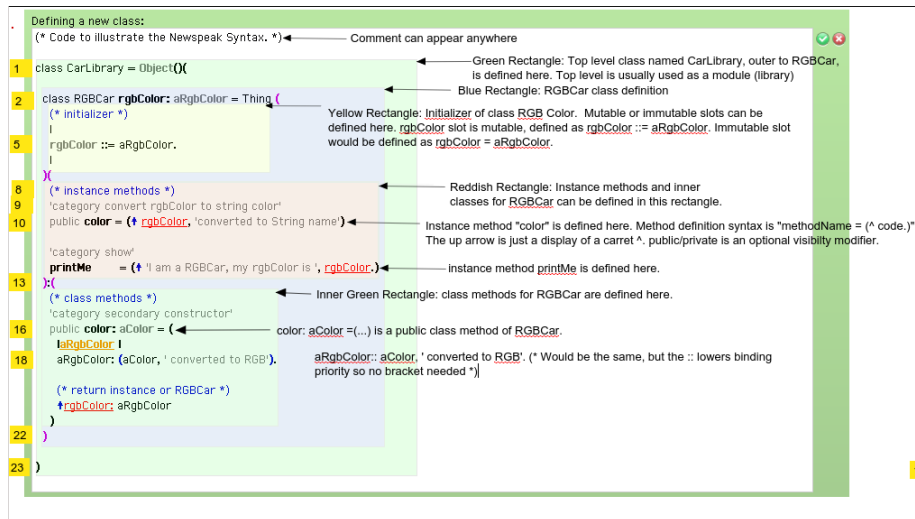
    color ::= aColor. (* ::= defines a mutable slot. If we used /color = aColor/, then color
    |
  )(
    (* instance methods *)
    'category misc'
    printMe      = (^ 'I am a Car, my color is ', color.)
    printColor   = (^ 'My color is ', color.)
  )
)

```

- Unlike Smalltalk, Newspeak has a text representation of code - you can paste the above code to the Newspeak IDE.
- Paste Car and Thing under TugPresentation
- Note that after pasting the above to the Newspeak IDE, some things are underlined, for example method names. This looks like a syntax error but it is not - underline shows message names that may not be known and top level class names.
- Also note that in the latest stable Newspeak, unlike various examples on the web, the category string above method ('category misc') is required for the textual representation to work.

## Newspeak 101: Newspeak Syntax in a nutshell

Below is an annotated and (over) colored example of a Newspeak class declaration. See <http://lively-kernel.org/repository/webwerkstatt/users/mzimmerm/Projects/Newspeak/NewspeakSyntax.xhtml?forceInvalidateCache=1414792071362> for more details.



### Syntactic Elements in the CarLibrary example above:

#### 1. Class Declaration (line 2) [sec-11-6-1-1]

Let us ignore the outer class CarLibrary starting at line 1 and closing at line 23.

On line 2, class RGBCar is declared. As part of the declaration, a constructor “rgbColor: aRgbColor” is declared. This would be used in client code as

```
| car |
car: RGBCar rgbColor: '#0000FF'. (* this code creates a new RGBCar and uses the auto-generated
car printMe. (* would print 'I am a RGBCar, my rgbColor is #0000FF*)
```

If the section *rgbColor: aRgbColor* was cut out from the default constructor on line 2, the class would define a default constructor “new”. That would be used in client code as

```
| car |
car: RGBCar new. (* this code creates a new RGBCar with no color and uses the auto-generated
car printMe. (* would produce 'I am a RGBCar, my rgbColor is null*)
```

RGBCar Extending Thing: Line 2 shows how RGBCar can extend class Thing (declared elsewhere as part of CarLibrary). The string Thing can be replaced with Object or nothing (which is equivalent to Object)

#### 2. Initializer (lines 3-6): [sec-11-6-1-2]

On lines 3 to 6, initializer defines a slot *rgbColor* between vertical bars. Slots are similar to instance variables, but they are, nowhere inside or outside of the class, accessed directly. Setters and getters are automatically generating for slots. Inside the class body, lines 8-12, *rgbColor* value can be obtained by using it as part of expression such as on line 10 (*rgbColor* slot getter) , or set using *rgbColor* slot setter *rgbColor:* such as

```
rgbColor: '#00FF00'.
```

3. Instance Body: Instance method and inner classes declarations (lines 8-12) [sec-11-6-1-3]

In our example, only methods are defined: *color* and *printMe*

4. Class method declarations (class method can be considered an alternative constructor) (lines 14-22) [sec-11-6-1-4]

One class method, *RgbCar color:* is defined. This could be used in client code as

```
|car|
car: RGBCar color: 'blue'. (* this code creates a new RGBCar and uses the auto-generated
car printMe. (* would print 'I am a RGBCar, my rgbColor is blue converted to RGB*)
```

5. Constructor (line 2) [sec-11-6-1-5]

The class declaration evaluates to a class object. Instances may only be created by invoking a factory method on *RgbCar*. Every class has a single primary factory, in this case *rgbColor:*. If no factory name is given, it defaults to *new*. The **primary factory method's header** is declared immediately after the class name. The formal parameters of the primary factory are in scope in the instance initializer. In lines 3-6, the slot declarations include an initialization clause of the form *::= e* where *e* can be an arbitrary expression. In our example, the *rgbColor* slot is initialized to the value of the formal parameter *aRgbColor* (*rgbColor ::= aRgbColor*).

## Newspeak 101: Opinion - Is the “=” character overused?

The = character can appear in the following syntactic context:

1. In the class declaration: See for example, line 1:

```
class CarLibrary = Object(...)(...)
```

2. In the method declaration: See for example, line 10:



```
public color = (^...)
```

3. In the initializer, to define a mutable slot: See line 5:

```
rgbColor ::= aRgbColor.
```

4. In the initializer, to define an immutable slot: No usage in the above example, but it could be a line inserted after line 5:

```
numWheels = 4.
```

5. As an equality symbol =: No usage in the above example, but it could be a line inserted after line 18:

```
(aRgbColor = aColor) ifTrue: [Transcript open show: 'ERR'] ifFalse: [Transcript open sh
```

6. As part of the object identity symbol, = - no example here but could be used similar to the above substitute = for =

## Newspeak 101: Representation independence

Newspeak objects are independent of their representation. We have changed the layout of `Car` to `RgbCar` with two additions.

The `RgbCar` class has the same API as the `Car` class, because:

- a) While `RgbCar` stores `color` internally as `rgbColor`, we provided the ability to also create `RgbCar` from `color`, by making `color: constructor` - a class method
- b) We preserved the `Car` interface by providing instance method “`color`” which converts `RgbCar`’s `rgbColor` to `color`.

***RgbCar* should be now be renamed to *Car*, because *RgbCar* provides a representation independent API with respect to `color` / `rgbColor`**

## Newspeak 101: Mutable vs. Immutable Slots

`rgbColor` slot on line 5 is mutable, defined as `rgbColor ::= aRgbColor`. Immutable slot would be defined as `rgbColor = aRgbColor`.

## Newspeak 101: Newspeak differences from Smalltalk

- Newspeak fields (slots) automatically define access methods
  - So the only way to set or get a slot value is by invoking a method.
  - And if a class changes and replaces the slot with a method that does something more than access the slot, client code will not be affected
    - code is representation independent.

### From Modules as Objects in Newspeak (dot-jotted, emphasis added):

- Newspeak is a direct descendant of Smalltalk.
- Unlike Smalltalk Newspeak has an **intentional, syntactic representation** of classes; this is crucial in **supporting nested classes**, which are not present in Smalltalk.
- Smalltalk has a **global namespace** and abundant **static state**. Most fundamentally, Smalltalk **distinguishes between method invocation and variable access** it is **not a purely message based language**.
- These differences lead to a different semantics of method lookup (cont).

### From Message Based Programming (emphasis added)

(see <http://gbracha.blogspot.ca/2007/05/message-based-programming.html>)

- Smalltalk terminology refers to method invocations as message sends. Message passing is often associated with asynchrony, but it doesn't have to be. Smalltalk message sends are synchronous. As such, they seem indistinguishable from virtual method invocations. However, the terminology matters.
- **Insisting that objects communicate exclusively via message sends rules out aberrations such as static methods, non-virtual methods, constructors and public fields.** More than that: It means that one cannot access another object's internals - we have to send the object a message. So when we say that an object encapsulates its data, encapsulation can't be interpreted as just bundling - it means data abstraction. Two objects that respond the same way to all messages are indistinguishable, regardless of their implementation details.
  - We can nevertheless ask: is Smalltalk a message based programming language? I think not. I would take message-based programming to have an even stronger requirement: all computation is done via message passing. That includes the computation done within a single object as well. Whereas Smalltalk objects can access variables and assign them, message based programming would require that an object use messages internally as well. This is exactly what happens in Self, as I discussed in an earlier post about representation independent code.

### Newspeak Syntax notes (as different from Smalltalk)

- There are no assignment operator in Newspeak

- ::= (initializer only), vs = (2 roles, function and class declaration, equality) vs :: vs == (identity) - these were already discussed in a section above.
- More differences from Smalltalk:
  - Refer to slides 11-20 (accent: slide 15) from <sup>8</sup> and discuss
  - No Global variables, no assignment, no static (global) state.
  - The only runtime operation is message send. In the example on slide 15:
    - \* *t::* looks like a variable but is a setter, automatically generated (no assignment, all slot access in Newspeak is replaced with a message send)
    - \* Array must be implicitly passed to the application (no static)
  - Class categories are a Smalltalk legacy that will likely be dropped in the future
  - Packages are likely to be removed as well.

### Migration from Smalltalk to Newspeak

- There is a document <http://bracha.org/Smalltalk2NS2.pdf> describing how to convert Smalltalk code to Squeak.
- I do not have enough Newspeak to prove this on any sufficiently sized program, but, according to The Ministry of Truth in , code converted from Smalltalk to Newspeak:
  - The converted code is better than the original.
  - It becomes clear:
    - \* what the code's external dependencies are
    - \* what the module boundaries should be
    - \* who is responsible for initialization
  - There is no longer any static state.
  - Easier to tie libraries together (or tear them apart),
  - Easier libraries independent testing
  - Smalltalkers can migrate to Newspeak relatively easily.

I will do my best to show some of these points on an example in the modularity section.

---

<sup>8</sup>Newspeak: Evolving Smalltalk for the age of the Net

## Gotchas

1. Some syntax is evolving. I think I saw on the web some examples using  $a := b$ , replace that with  $a:: b$  (setter send). Note the  $::$  must not be separated by space.
2. See the notes on  $::=$  vs  $=$  vs  $::$  vs  $=()$  in this presentation in syntax notes above

## Newspeak - As Advertised - Expanding on the Specs Highlights section

### TODO Newspeak is: Network Serviced (Supported by *partially implemented synchronization*)

Gilad Bracha describes his vision for objects as services (Serviced Objects = SOBs), replacing the static-y web services with objects. Great reading:

<sup>9</sup> and <http://bracha.org/objectsAsSoftwareServices.pdf>

### How Newspeak solves Problems with constructors, and removes any static state

- Having introduced Newspeak Basics, let us get back to:
  - Newspeak highlights in detail,
  - Showing some issues with Java and Smalltalk, concentrating on:
    - \* Static global state (variables) issues
    - \* Constructor issues

We will show how Newspeak resolves the issues elegantly

### Newspeak is: Class Based (with constructor issues removed)

In this section we concentrate on describing constructor and instance creation deficiencies in Java and similar languages, but also Smalltalk, and show which improvements were applied to Newspeak.

But first a few notes on why Newspeak is class based not prototype based:

---

<sup>9</sup>SOBs

- Classes must be part of language; It was shown that JavaScript implementation of classes as libraries leads to fragmentation
- Briefly Describe Classes, Constructors, Objects, and their Definition.

### Constructors and Instance Creations: Issues in Existing Languages (Java, Smalltalk)

There is an important improvement in Newspeak regarding constructors.

In his Room 101 blog, Gilad Bracha describes:

- Weaknesses of and deficiencies of constructors in languages such as Java.
- Weaknesses of object creation in Smalltalk

Newspeak's constructors (and also Dart to a degree) resolve the discussed constructor issues. - Most comments here are again inspired by and used from Gilad Bracha's Room 101 blog

- [Constructors Considered Harmful](#) <sup>10</sup>
- [Object Initialization and Construction Revisited](#) <sup>11</sup>

### Constructor Example in pseudo-Java (similar to Groovy, C#, PHP) - First hint at problems

Let us take a look at this pseudo-Java code:

```
class Thing {
    // implicit extends Object,
    // constructor can be implicit
    Thing() {
    }
}

// class declaration
class Car extends Thing {
    // constructor
    Car(String color) {
        super(); // implicit
        this.color = color.
    }
}
```

---

<sup>10</sup>[Constructors Considered Harmful](#)

<sup>11</sup>[Object Initialization and Construction Revisited](#)

```

    }
    int countWheels() {
        return 4;
    }
}

// client code uses constructor this way:
Car c = new Car('blue'); // This is not Object Oriented - no receiver

// client code sends a message this way:
int wheels = c.countWheels(); // This is Object Oriented: receiver.message()

```

Here we see a few issues:

- is *new* a method? what object is it called on?
- is *Car('blue')* a method invocation? what object is it called on?

Answer:

- *new* is not a message to any object
- *Car('blue')* is not a message to any object either
- Because there is no receiver
- Rather the “constructor construct” *new Car('blue')* is wired in as a special case, in a way that does not match the message send OO pattern.
- So when we say Java is lacking because not everything is an object, it is true, but the problem goes deeper - non uniformity and non-object orientedness / receiver.message() syntax of some core constructs.

### Constructors - more hints at problems

In Java (and similar languages), there are constructor issues like:

- Constructor cannot be overridden like instance methods (no target object, so no dynamic dispatch).
- Constructor *new Car('blue')* cannot return an instance of another object or cached or proxy object.
- All constructors need to call another constructor, or a superclass constructor etc.
- Mixins are hard to implement in a language with constructors.

- Constructors are a major cause of need of dependency injection.
- Constructors are a major issue for testability: (`new Car('blue')` cannot return a mock of a car).

### Typical (Java) Solutions to constructor problems

- Use a static method `makeCar('blue')` on the same class on another class (aka “factory method”):

`CarFactory.makeCar(String color)` { can return instance of `Car`, it's subclass, a proxy or ...

– But static methods in Java has similar problems:

- \* Static method has no runtime target object - are wired at compile time => no abstraction via interface, no dynamic binding or overriding
- \* see the static state section
- \* also see <http://stackoverflow.com/questions/2223386/why-doesnt-java-allow-overriding-of-static-methods>

- Ok, so what to use if not static methods? We can define a factory class and make instances of it - this is OO

– But to create the factory class instance we need a constructor:

```
new CarFactory().makeCar('blue')
```

– so we are in a problem loop!

- A better solution is to use Dependency Injection (DI framework)
  - That is reasonable, but requires an extra-lingual framework and adds a dependency
  - DI frameworks are workarounds for the lack of support in the underlying language

**Smalltalk: Has a better approach to constructors - does not have constructors in the above sense, but factory objects for instances.**

- 30+years old, Smalltalk
- There are **no constructors in Smalltalk**, instead, there are **factory objects for instances (instance creators)**.
  - This is a solution we were trying to show in the above Java-like examples using factories.

- In Smalltalk abstraction is preserved (we have an object as a target for new).
- But inheritance instance initialization is not guaranteed - has to be worked on, does not come for free, see the next heading.
- Saving the following Car class declaration creates the class object (=the factory object for instances).

"Defining and saving Car"

```
Object subclass: #Car
  instanceVariableNames: 'color'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'AAATUG'
```

"Unlike Java `new Car('blue')`, or even `CarFactory` in `/CarFactory.makeCar('blue')`/, "  
 " Smalltalk Car class is an object, not just a static global name"  
 " Below, `>>new` is invoked on a target which is an object, instance of Car class, not  
`myCar := Car new`.

- The fact that Car class is an object allows abstraction and method lookup etc.

### **Smalltalk: But there is another problem: no absolute guarantee that *myCar* instance is initialized**

In the Smalltalk Car definition, we defined *instanceVariableNames: color* to illustrate this point.

- We want the Smalltalk car to be created initialized with it's color.
- We do not want clients to call *Car new*.

```
myCar := Car new.
myCar printMe. "myCar's color not initialized"
```

- To guarantee clients will not call *Car new*, we must override *Behavior>>new* or *Behavior>>initialize* and throw exception in the override
- Otherwise, if *Car new* is used by clients, *myCar color* value is not initialized
- The core issue here is that:



- the instance of *Car class* (the **factory for car instances**) is a subclass of *Object class* (the **factory for object instances**)
  - along with method inheritance (*new* is inherited in the above case)
  - lead to the **unintended** ability to partially initialize *myCar* with *color* null if called as *myCar := Car new*.
- If this was not the case, *Car class instance* would not have access to *Object class instance new* message!
  - Solving the above issue, is the core to better constructors (instance creators) in Newspeak, removing both the Java-ism issues (no object target) and Smalltalk-ism issues (incomplete initialization must be manually prevented).

**Smalltalk:** There is also no guarantee object is not initialized twice

Find an Example.

**How Newspeak solves constructor deficiencies described above - an example**

- Newspeak combines Scala constructors with Smalltalk factory for instances.
- But unlike Smalltalk, one cannot call a superclass's constructors on a subclass. This prevents clients from partially instantiating an object, say by writing:  
*Car new.* (\* not visible and illegal as long as Car defines Car color: \*)

Let us create a simple class in Java, Smalltalk and Newspeak: *Thing and Car*:

- In Smalltalk: *Thing's subclass Car*

```
Thing: subclass Car
  instanceVariableNames: 'color'
  ..etc..
```

'and class side method to create instances initialized with color'

```
Car>>initWith: aColor
|car|
car := Car new. "or self new; not self initialize"
car setColor: aColor.

^car.
```

- Smalltalk Problem: Unless special care is taken (overriding >>new), client code can still call *myCar = Car new* and leave uninitialized state (instance variable color)

- In Java: *Car extends Thing*

```
class Car extends Thing {
    String color;
    Car(String aColor) {
        this.color = aColor;
    }
}
```

- Java Problem(s):

- \* Java client call such as *myCar = new Car('blue')* will always be wired to the concrete type Car causing issues with testability, leading to need of Dependency Injection frameworkd.
- \* *myCar = new Car('blue')* is a wired-in non object oriented syntax
  - *new* is not a message to a receiver object, cannot

- In Newspeak: *class Car = Thing ()()*

```
(*
- TODO - Repeat Car constructor from previous section here:
*)
```

- How is the Newspeak constructor different from the constructors in Java and instance creators in Smalltalk, and how does it solve the issues discussed? A summary from :

- Because **the Newspeak instance is created by sending a message to an object**, and not by a special construct like a constructor invocation *Java: car = new Car('blue')*, we can:

- \* Replace the receiver of that message with any object that responds to that message.
- \* The receiver can be another class, or it can be a factory object.

- But Newspeak also solves, without need to extra care in defining Car, the **Smalltalk initialization issues** (non-initialization, multiple utilization's)

- \* Newspeak client code:

- *Thing new printMe. #'I am a thing.'*
- BUT: *Car new printMe =>* Does not understand *Car>>new!* which is **good** - Newspeak hid the ability to call "new"
- SO: client use of *Car color: 'blue'*. is enforced instead of *Car new:*  
*(Car color: 'blue') printMe. 'I am a Car, my color is blue'*

## Newspeak methods that take class-factory as a parameter

Let us assume a factory method to make cars:

```
makeCar: carFactory = (
  ^carFactory color: 'blue'.
)

(* Can be called as *)
makeCar: Car

(* But also as *)

makeCar: RGBCar

(* Where *)
class CarLibrary = Object()(

  class RGBCar rgbColor: aRgbColor = Thing (
    (* initializer *)
    |
    rgbColor ::= aRgbColor.
    |
  )(
    (* instance methods *)
    'category convert rgbColor to string color'
    public color = (^ rgbColor, 'converted to String name') (* call some converter From RgbC...

    'category show'
    printMe          = (^ 'I am a RGBCar, my rgbColor is ', rgbColor.)
  ): (
    (* class method which provides, for the RGBCar
       an EQUIVALENT OF the /Car color:/ constructor,
       But this method returns RGBCar.
    *)
    'secondary constructor for RGBCar'
    public color: aColor = (
      |aRgbColor |
      aRgbColor: (aColor, ' converted to RGB'). (* hack - should call a converter stringCol...

    (* return instance or RGBCar *)
    ^rgbColor: aRgbColor
  )
)
)
```

Clients may call the RGBCar using either one of the color: rgbColor constructors, as follows:

```
(RGBCar rgbColor: '#FFCCCC') printMe -> 'I am a RGBCar, my rgbColor is #FFCCCC'
```

or

```
(RGBCar color: 'blue') printMe. -> 'I am a RGBCar, my rgbColor is blue converted to RGB'.
```

The ability to take the Car class and create a different version of it, named RGBCar is also an example of *Representation Independence* in Newspeak - we can now delete Car, and rename RGBCar to Car, and no clients will notice, because:

1. Clients can still create the Car the same way as before (no need to change the client code):

```
(renamed from RGB)Car color: 'blue'.
```

Because a “legacy” color: constructor was provided as a class method.

2. Clients can still send *color* to Car:

```
(renamed from RGB)Car color: 'blue' color.
```

- This is because we provided, in RGBCar, the “legacy” *color* method

Although note that we only pseudo-implemented the *color* method, the result in our case is not 100% the same, but making it the same is just a matter of mechanics of a converter between color and RGB col

### **Nested classes and Inner classes: Significance of nested classes and examples**

- We have shown an example of a nested class with more comments in the section [Newspeak-101:-How-to-define-a-nested-class-outer-and-inner-class]
- Nested classes are used (and needed) to implement plumbing often “solved” by static state.

From <sup>12</sup>

---

<sup>12</sup>[On the Interaction of Method Lookup and Scope with Inheritance and Nesting](#)

- “The only widely used language that supports such nesting is Java. Java nested classes are used in very restricted, stylized ways. They are often used simply for packaging; the nested classes are static and the scope of the enclosing class is inaccessible to them so the issue does not arise. The situation in Python is similar: nested classes have no access to the scope of their enclosing class. Aggressive use of class nesting offers considerable possibilities. In addition to the classic techniques using nested and especially virtual classes [MMP89] demonstrated by the Beta community, **nested classes can enable powerful features such as mixins, class hierarchy inheritance and modules.**”
- (But the actual mechanism how this happens is hard for me to understand, see the reference above if you are interested)

### Newspeak’s Inheritance is implemented using Mixins

We will only make a few notes without much reasoning, referring to once again:

1. In Newspeak, all names, including class names, are late bound
  - So at runtime, there can be more than one instance of a class for a class name (classes are virtual - this is different from other OO languages such as Java or Smalltalk).
  - Because of class declarations (and hence superclass declarations) are virtual, all classes act as mixins.
  - Because a module is just a top level class, also module definitions are mixins
2. All nested classes are virtual
  - So, also entire libraries/frameworks can be inherited, mixed-in, overridden

### Newspeak is: Message Based (and purely OO as a result)

We call Newspeak “message based”, because the only runtime operation in Newspeak is virtual method invocation.

Discuss an interesting thought experiment why a purely message-based language is also purely OO

A few summary notes on this subject:

- Newspeak is a Smalltalk successor: Everything is an Object
- Newspeak has no primitive types

- Newspeak eliminates special cases
- All names are late bound; every name is a dynamically dispatched method invocation, even inside objects
- Everything is an object follows from “Everything is a virtual method invocation”

## **Newspeak is: Secure (Supported by encapsulation, no static state)**

In this section we discuss unwelcomed consequences of static state, among them, how static state affects security.

We define “static state” as /“presence of variables having global accessibility and lifetime”/.

References for this section are from Room 101 static state entry <sup>13</sup> and other links below.

### **Static State (Variables) has unwelcomed consequences**

Static state (variables) have known issues, most of those mentioned here are directly from

Static variables are:

- Bad for security: If your code is attacked, the attacker has access to everything your code does, including static state. Attacker can do things like:
  - Smalltalk at: `#Transcript put: TranscriptWhichForwardsToAttacker. 'Smalltalk is static and holds other static state'`
  - And if your code logs credit card numbers, or social security etc , the attacker can read them. (Assume attacker code can reach out of your network)
- (Mutable static variables) are bad for re-entrancy - see also [http://en.wikipedia.org/wiki/Reentrancy\\_%28computing%29](http://en.wikipedia.org/wiki/Reentrancy_%28computing%29)
- Bad for concurrency (ability to run on multiple threads/cpus) - see the above link as well
- Complicates memory management / garbage collection

---

<sup>13</sup>[Cutting out Static](#)

- Bad for startup time - I think this applies to static methods as well, code using static methods must load the class etc.
- Bad for distributed systems, need to be at one place or constantly synced
- Bad for testability

### **The Ministry of Truth on Static State (Variables)**

Quotes from Room 101 on static state :

"It may seem like you need static state, somewhere to start things off, but you don't. You start off by creating an object, and you keep your state in that object and in objects it references. In Newspeak, those objects are modules.

Newspeak isn't the only language to eliminate static state. E has also done so, out of concern for security. And so has Scala, though its close cohabitation with Java means Scala's purity is easily violated. The bottom line, though, should be clear. Static state will disappear from modern programming languages, and should be eliminated from modern programming practice."

### **Functional programming eradicates all state (static or local) not just static state as Newspeak does.**

Eradicating all state is good but outside the scope of this discussion (as they say to not enter wars)

But the world needs persistence, so need to pass state through some kind of IO for persistence read/write. Proponents of "no state" rarely discuss this need.

It seems to me there always need to be a boundary where state needs to be conveyed *from* and *to* a calculation (IO).

### **Note on Static Methods (not variables): Java (and other languages) static Methods also have issues in common to constructor issues, that makes both static methods and constructors not Object Oriented**

- Why?
  - In both cases, there is no runtime object that is a target of the operation
  - No runtime object, so no interface that can be used to describe the operation (abstraction)
  - No runtime object, so no dynamic binding

## Newspeak is: Reflective (Mirror based reflection)

We will only refer to the Newspeak specs for details

## Newspeak is: Modular (Independent, immutable, parametric namespaces) - Notes and Example Application (CarRace)

Points here are mostly from and

- A *module declaration* is a *class declaration* which is not nested in another class expression
  - Notes:
    - \* The object a module declaration evaluates to is referred to as *module definition*
    - \* Module definitions are instantiated into stateful objects (called *modules*)
- Module = Top level class
- Module has no access to surrounding namespace
- All names locally declared or inherited (from Object?)
- Factory method params are object-capabilities which determine what belongs to the per-module sandbox
- Multiple module instances can be created, with different module parameters
- As everything in Newspeak Modules are objects, accessed via interface:
  - Different implementations of module can coexist
  - Modules cannot step on each other's state
- Modules are re-entrant, because there is no static state. See also [http://en.wikipedia.org/wiki/Reentrancy\\_%28computing%29](http://en.wikipedia.org/wiki/Reentrancy_%28computing%29)

### Modularity example: A simple Newspeak module (*CarRace*) which is using other modules (*DatetimeLibrary*)

To discuss modularity in Newspeak, there are two important concepts: **Nested classes** and **Imports**. We described class nesting before. We will show what is meant by imports in this example.



Let us work out a simple example. Let us say we have a Newspeak module *CarRace*. The module needs a *DatetimeLibrary* to calculate a difference between “finish time” and “start time”.

Let us choose a datetime library (that we wrote) which has a bug in it. Call this *DatetimeLibraryBuggy*.

Upon discovering the bug, we would like to switch to a different datetime library. Call this *DatetimeLibraryCorrect*.

How can this be illustrated in Newspeak?

The buggy datetime library/module is a top level class:

```
(* A datetime library (module) with a bug - elapsed time returns a negative number.
  Illustration only.
*)
class DatetimeLibraryBuggy = Object () (
  class Datetime = Object (
    (* No initializer code, no slots in this illustration example. *)
  )('misc'
    (* Single method elapsedTimeBetween: and: - illustration only.
      Result hardcoded to always return a negative 10 minutes as String
    *)
    elapsedTimeBetween: start and: finish = (^ '-10 minutes'.)
  )
)
```

The correct datetime library/module is a top level class as well:

```
(* A datetime library (module) without a bug - elapsed time returns a positive number.
  Illustration only.
*)
class DatetimeLibraryCorrect = Object () (
  class Datetime = Object (
    (* No initializer code, no slots in this illustration example. *)
  )('misc'
    (* Single method elapsedTimeBetween: and: - illustration only.
      Result hardcoded to always return a positive 10 minutes as String - considered always
    *)
    elapsedTimeBetween: start and: finish = (^ '10 minutes'.)
  )
)
```

The *CarRace* module (which is also a beginning of a Newspeak Application)

```

(* CarRace: a Newspeak module: Illustration of modularity, not a real example.
   parameters platform, carLibrary and dateLibrary are the only way to pass
   any piece of information to the module (no global or static state).
*)
class CarRace usingPlatform: platform usingCarLibrary: carLibrary usingDatetimeLibrary: dateLibrary
|
  (* List (Car, Datetime, etc) each defines a slot.
     List's value (platform collections List) is a List class in the platform.
     The slot definition of List and other slots function as an *import* statement, without
     The platform is only in scope in the initializer - programmer must take action to get it
     All modules' external dependencies can all be gleaned in this initializer section.
  *)
  private List = platform collections List. (* unused *)
  private Car = carLibrary Car.
  private Datetime = datetimeLibrary Datetime.
  private Transcript = platform blackMarket Transcript.
|
)('misc'
  runRace = (
    |blueCar redCar blueStart blueFinish redStart redFinish|
    blueCar: (Car color: 'blue').
    blueStart: '10:25'.
    blueFinish: '10:35'.

    redCar: (Car color: 'red').
    redStart: '10:25'.
    redFinish: '10:35'.

    Transcript open show:
      'Tied race: Car with color ', redCar color, ' took ', (Datetime new elapsedTimeBetween
        ' Car with color ', blueCar color, ' took ', (Datetime new elapsedTimeBetween
  )
)
)

```

To run the CarRace client code using the buggy datetime library, paste the following in workspace:

```

• ==>

|carRace|
carRace:: (CarRace usingPlatform: platform
           usingCarLibrary: CarLibrary new
           usingDatetimeLibrary: DatetimeLibraryBuggy new
           ).
carRace runRace.

```

Migrating client code to use the correct datetime library is a matter of switching the imported module from `DatetimeLibraryBuggy` to `DatetimeLibraryCorrect`

- ==>

```
| carRace |
carRace:: (CarRace usingPlatform: platform
           usingCarLibrary: CarLibrary new
           usingDatetimeLibrary: DatetimeLibraryCorrect new
           ).
carRace runRace.
```

### Modularity Example Continued: Converting the *CarRace* module into a Newspeak Application

- A Newspeak application is an object conforming to a standard API. The application API is defined by the presence of one instance method:

**main:args:**

- A Newspeak application can be **deployed** either as NOF file or as an image. To deploy as a NOF file, the application must also define a class method

**packageUsing:**

- We will glean the parameters passed to **main:args:** and **packageUsing:** from the example.
- Let us create a *CarRaceApp* which *main:args:* method instantiates the *CarRace* object, using the imported modules.

(\* CarRaceApp is a Newspeak application, deployable as NOF file.

- The `/packageUsing: topNamespace/` constructor allows to deploy as a NOF file.

- The `/topNamespace/` is effectively the Newspeak namespace and allows the application packager to wrap classes and objects into the NOF file

- The `/main: platform args: systemArgs/` instance method turns the class into an application. Its presence is picked up by the Newspeak IDE which then adds facilities to run an application. These facilities are the `[deploy]` and `[run]` clickable links on the top right of the application window.

\*)

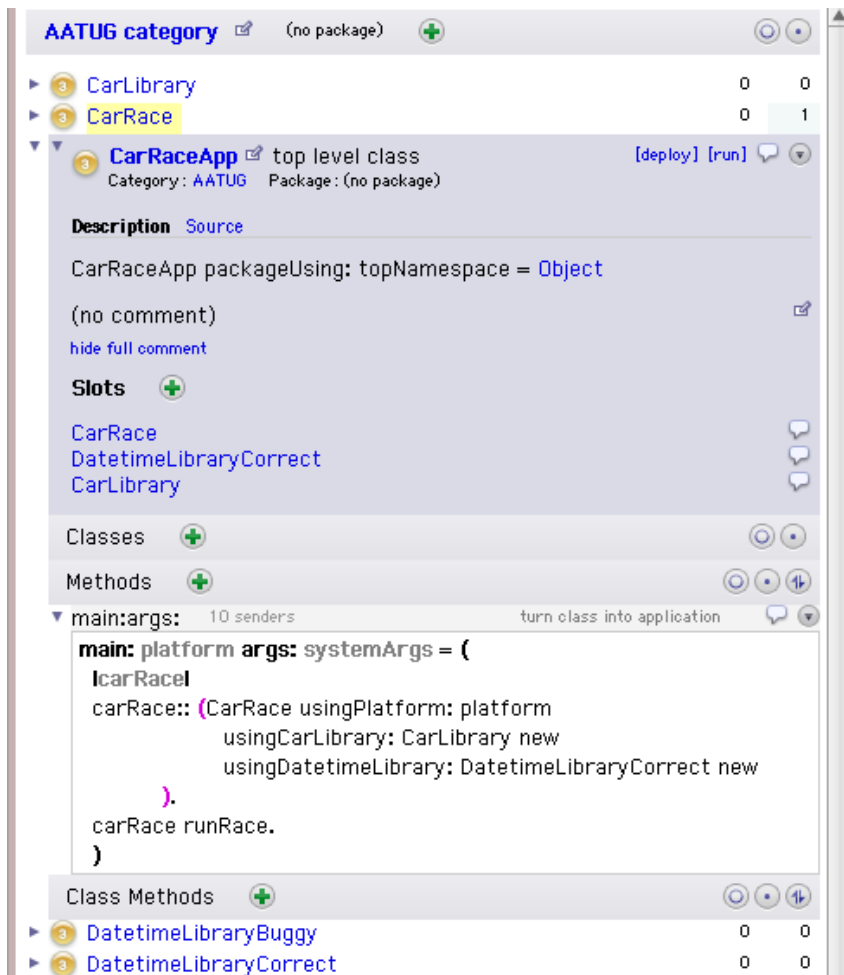
```
class CarRaceApp packageUsing: topNamespace = Object (
|
```

```

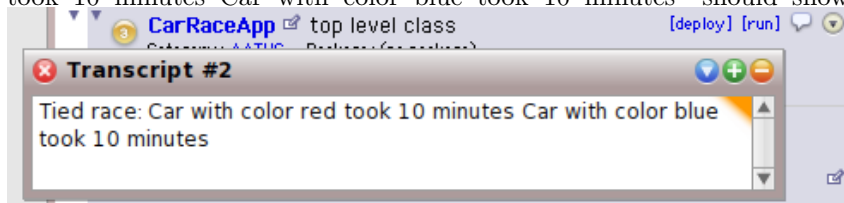
CarRace = topNamespace CarRace.
DatetimeLibraryCorrect = topNamespace DatetimeLibraryCorrect.
CarLibrary = topNamespace CarLibrary.
|
)('turn class into application'
  main: platform args: systemArgs = (
    |carRace|
    carRace:: (CarRace usingPlatform: platform
              usingCarLibrary: CarLibrary new
              usingDatetimeLibrary: DatetimeLibraryCorrect new
              ).
    carRace runRace.
  )
)

```

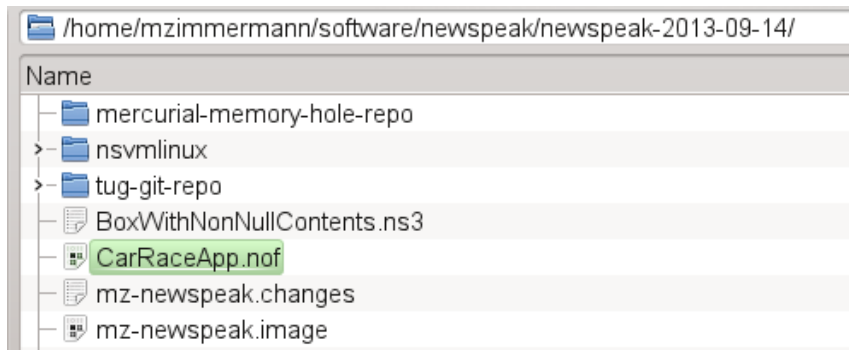
- Paste the above code for the CarRaceApp into a top level class. Notice that on “accept” (clicking the checkmark or do Ctrl-S), two icons appear on top of the class definition: [deploy] and [run]. State after accepting the above code:




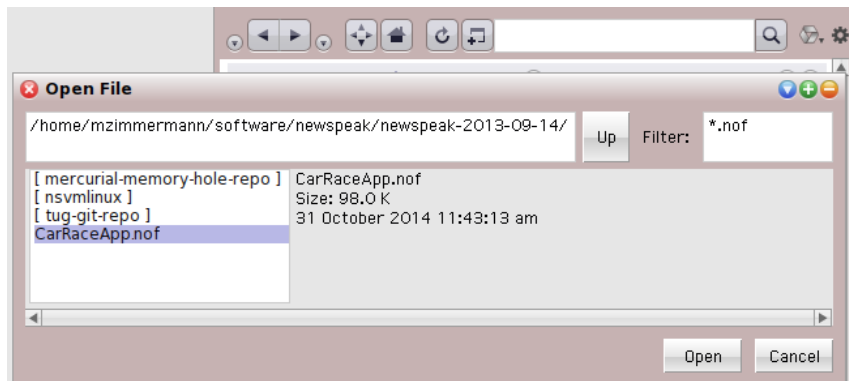
- Click on [run] and a Transcript showing Tied race: "Car with color red took 10 minutes Car with color blue took 10 minutes" should show:



- Click on [deploy] and select "As NOF". Notice that a CarRaceApp.nof file appeared in the directory where Newspeak runs:



- How to Run the Application Saved in the NOF file?
  - Currently, there appears no way to run the NOF from outside the Newspeak IDE. I assume that is intended to be changed?
  - To run the CarRaceApp.nof from inside the IDE, click on the Settings wheel , and select “Run App”. A dialog will appear which allows to select the NOF file.



- Notes:
  - \* On NOF: Currently there seems to be no practical difference between running from the Settings wheel and clicking on the [run] link. I assume that will change and we will be able to run the NOF file from the operating system outside the Newspeak IDE(?).
  - \* On other formats of creating an application. Clicking the [deploy] link allows to choose from:
    - as NOF
    - as Packaged Image
    - as Dart

· as JavaScript

The last two point to an **exciting** prospect of running your application directly from the browser, but at this point there is not enough documentation. I tried to generate the JavaScript application and run from a web browser, but nothing was shown. Need to spend more time on this.

## Newspeak is: Concurrent (Actor based concurrency)

We will only refer to the Newspeak specs for details.

Only partial implementation of Actor system exists at this time

## Newspeak is: Optionally typed *Unimplemented*

There are examples of typed code in Newspeak, but as I understand is not implemented.

## MemoryHole - Source Code Management in Newspeak

All code in Newspeak, is (can) under the covers be managed in MemoryHole (backed by Git or Mercurial)

- Because Newspeak has a code export in text format, users can also ignore the MemoryHole, and save/load classes from files, using any source code control
- To use MemoryHole link, need a Mercurial repo or Local Git repo - created local git at: *home/mzimmermann/software/newspeak/newspeak-2013-09-14/tug-git-repo* but the process does not work and ends up in an exception.

## Footnotes, References and Credits [References]

### Newspeak Authors

- Gilad Bracha
- Vassili Bykov
- Yaron Kashai

- Eliot Miranda
- Ryan Macnak
- contributors

## References

This presentation uses and sometimes quotes directly from the references below. All misunderstandings are mine.

- - Newspeak on Squeak - a guide to the perplexed - <http://bracha.org/newspeak-101.pdf>
- - The Newspeak Programming Language (main page) - <http://www.newspeaklanguage.org/>
- - Modules as Objects in Newspeak - <http://bracha.org/newspeak-modules.pdf>
- - Newspeak Programming Language Draft Specification Version 0.091. <http://bracha.org/newspeak-spec.pdf>.
- - On the Interaction of Method Lookup and Scope with Inheritance and Nesting - <http://bracha.org/dyla.pdf>
- - Pluggable type systems - <http://bracha.org/pluggable-types.pdf>
- <sup>14</sup> - Explorations in Next Generation Web Languages - presentation slides - <https://yow.eventer.com/yow-2013-1080/explorations-in-next-generation-web-languages-by-gilad-bracha>
- Gilad Bracha. Objects as Software Services. A Whitepaper - <http://bracha.org/objectsAsSoftwareServices.pdf>
- Gilad Bracha. Selected Papers -
- - Cutting out Static - <http://gbracha.blogspot.ca/2008/02/cutting-out-static.html>
- <sup>15</sup> - Room 101 - Representation Independent Code <http://gbracha.blogspot.ca/2007/01/representation-independent-code.html>
- - 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. Referenced from

---

<sup>14</sup>Explorations in Next Generation Web Languages - presentation slides

<sup>15</sup>Representation Independent Code



- <sup>16</sup> - Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. Referenced from
- - Room 101 blog: SOBs - Serviced Objects (Objects as Software Services) - <http://gbracha.blogspot.ca/2007/03/sobs.html>
- <sup>17</sup> - Room 101: Message base programming - <http://gbracha.blogspot.ca/2007/05/message-based-programming.html>
- <sup>18</sup> - Room 101 blog (top level) - <http://gbracha.blogspot.ca>
- - Room 101 blog - <http://gbracha.blogspot.ca/2007/06/constructors-considered-harmful.html>
- - Room 101 blog - <http://gbracha.blogspot.ca/2007/08/object-initialization-and-construction.html>
- - Room 101 blog - <http://gbracha.blogspot.ca/2010/07/convertng-smalltalk-to-newspeak.html>
- - Newspeak and Dart Presentation - <http://www.slideshare.net/esug/8-gilad-brachaesug08>
- Newspeak Wiki - <https://bitbucket.org/newspeaklanguage/newspeak/wiki/Home>
- The Newspeak Forum and Mailing list - <https://groups.google.com/forum/#!forum/newspeaklanguage>
- Objects as Modules in Newspeak: Phil Wadler's Blog. - <http://wadler.blogspot.ca/2009/08/objects-as-modules-in-newspeak.html>

## Licence

This file is licensed under Creative Commons Attribution ShareAlike 3.0: <http://creativecommons.org/licenses/by-sa/3.0/>

## How to make the Pdf

C-c C-e l p , then run

```
pandoc -f latex newspeak-TUG-presentation-full-version.tex -o newspeak-TUG-presentation-full-version.pdf
```

---

<sup>16</sup>[Mirrors: Design principles for meta-level facilities of object-oriented programming languages](#)

<sup>17</sup>[Message base programming](#)

<sup>18</sup>[Room 101](#)