

GBBopen Tutorial

Version 1.5

Dan Corkill

The GBBopen Project

<http://GBBopen.org>

April 10, 2010

10:52 EDT

This tutorial is under construction;
additional exercises will be added.

Copyright © 2005–2010 by Daniel D. Corkill for the GBBopen Project.

This tutorial may be reproduced and distributed in whole or in part, subject to the following conditions:

- The copyright notice above and this permission notice must be preserved complete on all complete or partial copies.
- Any translation or derivative work of this tutorial must be approved by the copyright holder in writing before distribution.
- If you distribute this tutorial in part, instructions and a means for obtaining a complete version of this tutorial must be included.
- Small portions may be reproduced as illustrations for reviews or quotes in other works without this permission notice if proper citation is given.
- Distribution of this work or a derivative of this work in any standard (hard copy) book form is prohibited without prior written permission from the copyright holder.

All source code examples in this work are placed under and covered by the GBBopen software license that accompanies each GBBopen distribution and is also available at <http://GBBopen.org/svn/GBBopen/trunk/LICENSE>.

This work is licensed and provided “as is” without warranty of any kind, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose or a warranty of non-infringement. GBBopen software and the information in this tutorial are subject to change without notice.

Please help improve this tutorial by reporting any errors, inaccuracies, bugs, misleading or confusing statements, missing or unhelpful index entries, and typographical errors that you find. E-mail bug reports, comments, and suggestions to bugs@GBBopen.org. Your help is greatly appreciated and will be acknowledged.

GBBopen is a trademark of the GBBopen Project.

Any other brand or product names are trademarks or registered trademarks of their respective holders.

The GBBopen Project
181 Pondview Drive
Amherst, Massachusetts 01002

GBBopen@GBBopen.org
<http://GBBopen.org>

This tutorial was produced using \LaTeX and PDF \LaTeX .

Contents

Acknowledgments	iv
Introduction	1
1 Starting GBBopen	3
2 Creating a Unit Instance	11
3 Creating a Space Instance	15
4 Deleting Instances	21
5 Enhancing Your Development Environment	29
6 Working Within a File	35
7 Adding Dimensions	37
8 Using a Control Shell	45
9 Application Startup and Event Functions	53
10 Add Another KS	59
11 Making Connections	69
12 Creating a GBBopen Application	79
13 Multiple Walkers	95
14 A Dimensional Detour	97
15 More to come...	99
16 The Completed Application	101
Index	103

Acknowledgments

Many people have contributed comments, suggestions, design ideas, questions (and answers), bug reports, and code to GBBopen, and we appreciate their time and effort. Acknowledgment of some of their contributions here does not necessarily imply that any individual or the organizations with which they are affiliated endorse GBBopen or this documentation. Disclaimers aside, GBBopen users thank each of you!

Douglas Crosher ported GBBopen to [Sciener CL](#). [Gary King](#) worked on the initial [Digitool MCL](#) and [OpenMCL](#) porting efforts. Christian Lynbech performed the initial CMUCL port. [Sam Steingold](#) initiated the [CLISP](#) port. Vladimir Tzankov provided Portable Threads support for [CLISP/MT](#).

Questioners, bug reporters, capability demanders, contributors, and great idea suggesters include: [Pascal Costanza](#), Matthew Danish, Michael Hannemann, Susan Lander, Attila Lendvai, Wendall Marvel, [Clayton Morrison](#), Beryl Nelson, Eric O'Connor, [Zack Rubinstein](#), Bill St. Clair, Earl Wagner, Paul Werkowski, and [Huzaifa Zafar](#).

Organizations [Franz Inc.](#) and [LispWorks Ltd.](#) provided (and continue to provide) Common Lisp licenses and technical support to the Project. [BBTech Corporation](#) provided hardware and Internet resources.

Some early design work for GBBopen was supported by DARPA's Information Exploitation Office ([IXO](#)) under contract MDA-972-02-C-0028 to [Information Extraction & Transport, Inc.](#)

Other efforts using GBBopen that have indirectly led to contributed GBBopen improvements and enhancements include: research supported by the "Fusion Based Knowledge for the Future Force" ATO program and the "Advanced REsearch Solutions - Fused Intelligence with Speed and Trust" program at the U.S. Army RDECOM CERDEC Intelligence and Information Warfare Directorate, Fort Monmouth, NJ, under contract W15P7T-05-C-P621; research on "Command & Control and Data Fusion Architectures" supported by DND Canada under contract W7701-4-2118; work on "Knowledgable Dynamic-Process Modeling and Execution" supported by Boeing and Infosys Technologies Limited; research on "A Multi-Agent Approach for Heterogeneous Persistent Surveillance" supported by Raytheon Intelligence & Information Systems; work on "Massive-Scale Representation and Reasoning" supported by GHX; and research supported by the AFRL "Advanced Computing Architecture" program, under contract FA8750-05-1-0039.

Legacy Contributors GBBopen builds upon concepts and ideas that were explored and refined in the UMass Generic Blackboard system and the commercial GBB product. The following people made significant contributions to those systems:

UMass Generic Blackboard System	GBB Product
Dan Corkill	Tony Carrico
Kevin Gallagher	Dan Corkill
Philip Johnson	Raymond de Lacaze
Kelly Murray	Kevin Gallagher
	Susan Lander
	Zack Rubinstein
	Suzanne Tromara

The UMass Generic Blackboard Project received research support from The National Science Foundation, the Defense Advanced Research Projects Agency, the Office of Naval Research, and Texas Instruments, Inc.

The random-walk example used in this tutorial is adapted from *Getting Started with GBB*, written by

Dan Corkill and Suzanne Tromara.

Introduction

GBBopen is a high-performance, open source blackboard-system framework. This tutorial shows you how to get started using GBBopen through a series of exercises that cover GBBopen's concepts and features in a step-by-step sequence. The exercises guide you in creating a simple "random walk" application that simulates taking a sequence of straight-line excursions, each of random length and direction. Although the application is simple, it involves many of GBBopen's features, from very basic to fairly advanced.

GBBopen and Common Lisp

GBBopen is an extension of [Common Lisp](#) and uses CLOS (the Common Lisp Object System) and the [Metaobject Protocol](#) (MOP) to provide blackboard-specific object capabilities. The blending of GBBopen with Common Lisp transfers all the advantages of a rich, dynamic, reflective, and extensible programming language to blackboard-application developers. Thus, GBBopen's "programming language" includes all of Common Lisp in addition to the blackboard-system extensions documented in the *GBBopen Reference* manual.

This tutorial does not attempt to teach Common Lisp programming, and an understanding of basic Common Lisp and CLOS concepts is assumed. Although it is possible to read through the tutorial exercises without Common Lisp expertise, a much deeper understanding of GBBopen's potential is gained by understanding how GBBopen and Common Lisp work smoothly together. Two frequently recommended Common Lisp books are Peter Seibel's [Practical Common Lisp](#) and Paul Graham's [On Lisp](#). Both books are available on line, as well in traditional book form. A less programmer-oriented introduction to Common Lisp is David Touretzky's [Common Lisp: A Gentle Introduction to Symbolic Computation](#), also available on line.

Ken Pitman's [Common Lisp HyperSpec](#), an easily navigable HTML document derived from the ANSI Common Lisp standard, is the customary programmer's reference for Common Lisp. A [downloadable archive](#) of the HyperSpec is also available, which is very convenient when working without a continuous connection to the Internet. We will show how to make the HyperSpec and the GBBopen Reference HyperDoc directly accessible in your Common Lisp development environment in [Exercise 5](#).

Using a Common Lisp/GBBopen development environment

The tutorial exercises build on one another, and they are intended to be performed sequentially. The initial exercise involves installing GBBopen and preparing it for use in your environment. This is followed by several exercises where you interact with GBBopen by entering forms into the "Lisp Listener" (also called the read-eval-print-loop or simply the REPL) that is provided by your Common Lisp implementation. As the scope of the random-walk application grows, however, it is important to set up a working environment where your work is done in a file. So, after these initial GBBopen exercises, we will spend an exercise setting up your environment to provide you power-user productivity for the remainder of the tutorial (and for future GBBopen activities). This diversion exercise will be worth your time!

1 Starting GBBopen

This initial exercise requires that you:

- Obtain and install Common Lisp
 - Obtain and install GBBopen
 - Subscribe to the GBBopen mailing lists
 - Interact with Common Lisp’s read-eval-print loop (REPL)
 - Recover from errors
 - Prepare to use GBBopen in the next exercises by loading the `:gbbopen-user` module and changing to the `:gbbopen-user` package
-

Step 1: Obtain and install Common Lisp

GBBopen requires a supported Common Lisp implementation. If one has already been installed on your system, then this step is finished. (That was easy!) If not, then you will need to choose, obtain, and install a Common Lisp implementation before moving on to the next step.

The list of Common Lisp implementations, with version numbers, on which GBBopen is supported is maintained on the “[Current ports](#)” page of the GBBopen Project [web site](#). The list includes commercial products and open-source implementations made available under varying license arrangements. The vendors of commercial Common Lisp products offer no-cost “Trial” or “Personal” editions that can support the tutorial exercises and allow you to become familiar with their products before making a purchase decision.

Choosing a particular implementation is a subjective decision. Some Common Lisp implementations run on only a single platform. Some implementations do not provide multiprocessing (thread) support on some or all of the platforms that they run on. Some implementations come with their own integrated development environment (IDE), interactive graphics facilities, and supported libraries and extensions. When it comes to selecting an implementation, there is no “best” answer (but there is also no wrong answer, if the implementation meets your current needs). All these Common Lisp implementations strive to conform to the ANSI (American National Standards Institute) standard for the Common Lisp language. By writing GBBopen applications to remain consistent with the ANSI standard (including portable extensions to the standard that are provided by GBBopen), we can easily run our code on any Common Lisp implementation that provides similar capabilities (such as threads, for example).

If you would rather not explore the space of Common Lisp implementations for your platform, Peter Seibel (the author of *Practical Common Lisp*) provides [Lispbox](#). Lispbox (“Lisp in a Box”) offers an easy to install, no-cost, Common Lisp environment for Linux/86, Macintosh OS X, and Windows. Lispbox packages together a Common Lisp implementation, [Emacs](#), and the [SLIME](#) Common Lisp development environment for Emacs. For the less adventurous, installing Lispbox can be an appealing way to get started.

However you choose to obtain Common Lisp, you must have an installed and operating implementation for your system before proceeding to the next step.

Step 2: Obtain and install GBBopen

GBBopen is available in source form from <http://GBBopen.org/>. A snapshot archive of the GBBopen source-code repository can be downloaded from <http://gbbopen.org/downloads/GBBopen.tar.gz>. Extract the archive into a directory of your choosing, and follow the “Compiling/Loading GBBopen” instructions that are contained in the `README` file of the installation.

Alternatively, if you are familiar with [Subversion](#) and have a Subversion client installed on your computer, you can checkout the latest files directly from the GBBopen repository. For example, the shell command:

```
[~]$ svn checkout http://GBBopen.org/svn/GBBopen/trunk/ gbbopen
```

will create a GBBopen repository tree rooted at the directory named `gbbopen` in your current working directory. As above, follow the “Compiling/Loading GBBopen” instructions that are contained in the `README` file of the installation.

GBBopen development is ongoing, and you should update your GBBopen installation regularly in order to obtain the latest capabilities and enhancements. Using Subversion is the easiest way to keep current, and you are strongly encouraged to install a Subversion client and use it perform frequent updates. Simply issue the command:

```
[~]$ svn update
```

from the root directory of your GBBopen repository tree. The [TortoiseSVN](#) Subversion client is highly recommended for Windows users. TortoiseSVN is smoothly integrated with the Windows Shell (Explorer) and is as easy to use as [TortoiseCVS](#) (also highly recommended if aren't already using it as your CVS client on Windows).

Subversion `.svn` administrative directories are included in the GBBopen snapshot archive, so a Subversion `update` command can be used to freshen a GBBopen installation that was originally installed from a snapshot archive.

Step 3: Subscribe to the GBBopen mailing lists

I strongly recommend that you subscribe to the GBBopen Announcements (`gbbopen-announce`) and the GBBopen Users (`gbbopen-list`) mailing lists. To subscribe, go to the “Mailing Lists” page on the [GBBopen web site](#) and click on the appropriate `subscribe` links. The traffic is low, and the messages and advice will save you time and frustration and get you writing quality GBBopen applications easier and sooner. (You might even want to browse the archived messages that are accessible from the “Mailing Lists” page.)

If you need help or advice, the GBBopen Users list is the place to ask (should this tutorial and archived messages from the GBBopen Users list come up short).

Step 4: Interact with the REPL

This tutorial assumes that you have a basic understanding of Common Lisp and, in particular, how to interact with the Common Lisp implementation that you are using. Thus, you should be able to start up your Common Lisp system and enter forms into the “Lisp Listener” (also called the read-eval-print loop or simply the REPL) that it provides.

When you start your Common Lisp system, it may first display some informational messages and then you should see a prompt for input that looks something like:

>

The specific format of the prompt differs depending on your Common Lisp implementation. The prompt character may vary, such as `*`, `:`, or `?`. The prompt might include an entry number that is incremented each time an expression is entered, such as:

```
[1]>
```

or it might also include the name of the current package (symbol namespace) being used by the REPL:

```
cl-user(1)>
```

Throughout this tutorial, we will include the package name in our example prompts to make it clear what package should be current:

```
cl-user>
```

The REPL prompt indicates that it is awaiting a Common Lisp expression to evaluate and then display the evaluation results. For example, enter the expression:

```
cl-user> (+ 1 2)
3
cl-user>
```

Note that what you need to enter is shown in black and other items, such as the REPL prompt and displayed result, `3`, are shown in gray. We will follow this convention throughout the tutorial to help make it clear what you must provide in the context of other information.

You may find it easier and faster to cut and paste text from this tutorial (if you are working with it on-line) rather than typing what is requested. On the other hand, some feel that they learn faster (and improve retention) through the action of typing. Either text-entry approach, however, is preferable to simply reading through the exercises. This tutorial is about *learning by doing!*

While we are on the subject of what is displayed, enter the following expression:

```
cl-user> 'symbol
symbol
cl-user>
```

The displayed result that you see might be in lower case, as shown, or upper case, or even capitalized. In this tutorial, results are shown with Common Lisp symbols displayed in lower case, which I prefer as being slightly easier to read. If you wish, you can duplicate this behavior in your REPL by entering the following form:

```
cl-user> (setf *print-case* ' :downcase)
:downcase
cl-user>
```

Your Common Lisp implementation may also differ slightly on how it displays multiple returned values. For example:

```
cl-user> (values 1 2 3)
1
2
3
cl-user>
```

In this tutorial, we show multiple returned values as displayed on separate lines. Your Common Lisp implementation may show them differently, such as with semicolon value-separator characters:

```
cl-user> (values 1 2 3)
1 ;
2 ;
3
cl-user>
```

Step 5: Recover from an error

Even you are a very careful typist, sooner or later you will enter a Common Lisp expression that signals an error. Therefore, it is important that you know how to get back on track when the inevitable occurs. Let's intentionally generate an error. Enter:

```
cl-user> (/ 1 0)
Error: Attempt to divide 1 by zero.
cl-user>>
```

The behavior of your Common Lisp environment should look similar, and you should now be in your implementation's debugger or "break loop." The debugger prompt may differ from the standard REPL prompt to help remind you that you are in the break loop. Alternatively, your implementation may open another window or buffer in response to the error. Your implementation may allow you to continue entering Common Lisp expressions at the break prompt that are evaluated and the results displayed just as if you were in the normal REPL (opening the possibility of causing another error and triggering another instance of the debugger). You can use the debugger to inspect the nesting, or "backtrace," of function calls that led to the error, view and edit local variable bindings, and so forth, and you can often correct the cause of the error and resume the broken evaluation directly from the debugger.

For tutorial purposes, you only need to know how to abort out of the evaluation and return back to the REPL if you trigger an error. The details for aborting the computation and exiting the debugger are implementation dependent, so you may need to investigate how to abort out of an error on your Common Lisp implementation. For example, it might be as easy as entering an abort debugger command:

```
cl-user> (/ 1 0)
Error: Attempt to divide 1 by zero.
cl-user>> :a
cl-user>
```

Consult your Common Lisp documentation or a knowledgeable friend if you need assistance with the debugger in your Common Lisp implementation (:a or q are typical abort commands).

Breaking a computation

There are other mistakes that don't signal an error. Suppose I foolishly evaluate this expression:

```
cl-user> (loop (print "This is repetitive...") (sleep 1))

"This is repetitive."
"This is repetitive."
...
```

Once I grow tired of watching this phrase repeat, it would be good to terminate the evaluation without having to kill the entire Common Lisp program.

Again, the specifics depend on your Common Lisp environment, but all that is needed is to interrupt, or “break,” the computation. Often this is associated with typing one or more Control-c (^c) characters:

```
cl-user> (loop (print "This is repetitive...") (sleep 1))

"This is repetitive."
"This is repetitive."
...
"This is repetitive."
"This is repetitive."
"This is rep^C
Error: Received keyboard interrupt ^C
cl-user>>
```

Typically a keyboard interrupt invokes the debugger where the computation can be resumed or aborted:

```
cl-user> (loop (print "This is repetitive...") (sleep 1))

"This is repetitive."
"This is repetitive."
...
"This is repetitive."
"This is repetitive."
"This is rep^C
Error: Received keyboard interrupt ^C
cl-user>> :a
cl-user>
```

Help! My REPL is broken!

Here is one last difficulty. Suppose I enter:

```
cl-user> (list "This" "is" "also" "a" "problem!")
cl-user>
cl-user>
cl-user>
```

My REPL seems dead: no result is displayed and I continue to be prompted again and again for input. I suppose I should try the entering the expression again:

```
cl-user> (list "This" "is" "also" "a" "problem!")
cl-user>
cl-user>
cl-user>
cl-user> (list "This" "is" "also" "a" "problem!")
cl-user>
cl-user>
```

Nope, my REPL is still broken.

Of course the cause of the problem is that I forgot the closing double-quote character on "is" when I entered the first expression. The REPL is still patiently waiting for me to finish that first Common Lisp expression. (I didn't break it after all!)

So how do I get out of this? I could try entering a sequence of double-quote and close parentheses and hope I that I get lucky:

```
cl-user> (list "This" "is "also" "a" "problem!")
cl-user>
cl-user>
cl-user>
cl-user> (list "This" "is" "also" "a" "problem!")
cl-user>
cl-user>
cl-user> "
cl-user> )
Error: Attempt to take the value of the unbound variable 'also'.
cl-user>>
```

or I could interrupt the REPL read operation, just as I did with the infinite loop situation above, and abort back from the debugger to the REPL and try again:

```
cl-user> (list "This" "is "also" "a" "problem!")
cl-user>
cl-user>
cl-user>
cl-user> (list "This" "is" "also" "a" "problem!")
cl-user>
cl-user>
cl-user> ^C
Error: Received keyboard interrupt ^C
cl-user>> :a
cl-user> (list "This" "is" "also" "a" "problem!")
("This" "is" "also" "a" "problem!")
cl-user>
```

You may discover even more creative ways to get into problems, but these example situations should give you enough experience to get through the rest of the tutorial.

Step 6: Load the :gbbopen-user module

GBBopen is packaged with its own module system that supports compiling and loading GBBopen components.

To compile and load the :gbbopen-user module and all the GBBopen modules it requires, you need to evaluate the following forms within your Common Lisp environment. First, load GBBopen's <install-dir>/startup.lisp file from wherever GBBopen has been installed:

```
cl-user> (load "<install-dir>/startup.lisp")
;; Loading <install-dir>/startup.lisp
;; GBBopen is installed in <install-dir>
;; Your "home" directory is <homedir>
;; Loading <install-dir>/source/module-manager/module-manager-loader.lisp
```

```

;; Loading <install-dir>/source/module-manager/module-manager.lisp
;; Loading <install-dir>/source/module-manager/module-manager-user.lisp
;; Loading <install-dir>/source/modules.lisp
;; No shared module definitions were found in <install-dir>/shared-gbbopen-modules/.
;; No personal module definitions were found in <homedir>/gbbopen-modules/.
t
cl-user>

```

Then compile and load the `:gbbopen-user` module and all the GBBopen modules it requires by evaluating:

```

cl-user> (module-manager:compile-module :gbbopen-user :propagate :create-dirs)
;; Compiling <install-dir>/source/module-manager/module-manager.lisp
;; Loading <install-dir>/<platform-dir>/module-manager/module-manager.fasl
...
;; Compiling <install-dir>/source/gbbopen/gbbopen-user.lisp
;; Loading <install-dir>/<platform-dir>/gbbopen/gbbopen-user.fasl
cl-user>

```

GBBopen should compile (if necessary) and load all the files needed for the next exercise without error. The output on your Common Lisp implementation may vary somewhat from that shown above. For example, the file extension for compiled files, shown as `.fasl` throughout this tutorial, is dependent on your Common Lisp implementation and platform.

File protection problems?

If one or more GBBopen files needs to be compiled, you must have write permission for those files and directories on your file system. If you do not have write permission and someone else is maintaining your GBBopen installation, they must start up Common Lisp and then evaluate the following two expressions:

```

cl-user> (load "<install-dir>/initiate.lisp")
;; Loading <install-dir>/initiate.lisp
;; GBBopen is installed in <install-dir>
;; Your "home" directory is <homedir>
;; Loading <install-dir>/extended-repl.lisp
;; Loading <install-dir>/commands.lisp
;; Loading <install-dir>/gbbopen-modules-directory.lisp
;; No shared module command definitions were found in <install-dir>/gbbopen-modules/.
;; No personal module command definitions were found in <homedir>/gbbopen-modules/.
t
cl-user> (compile-gbbopen)
;; Loading <install-dir>/startup.lisp
...
;;; GBBopen module compilation completed.
[~]$

```

every time they update GBBopen to ensure that all GBBopen modules have been compiled.

Beyond `<install-dir>/startup.lisp`

Although loading GBBopen's `<install-dir>/startup.lisp` file is a very easy way to get started with GBBopen, setting up your computing environment for serious GBBopen development requires only a few moments that will be recouped almost immediately. The steps needed to set everything up will be detailed soon in the Enhancing Your Development Environment exercise (see page 29).

Step 7: Change to the `:gbbopen-user` package

Now that the `:gbbopen-user` module is loaded, change the current package to the `:gbbopen-user` package:

```
cl-user> (in-package :gbbopen-user)
#<package GBBOPEN-USER>
gbbopen-user>
```

In Common Lisp, a *package* is a namespace that maps names to symbols. One package is always designated as the “current” package, and it is this package that is used by default for creating and finding symbols by their names. Initially, the current package is set to the `:common-lisp-user` package. The `:gbbopen-user` module creates a new package, named `:gbbopen-user`, that includes GBBopen Tools and GBBopen Core symbols in addition to the standard Common Lisp symbols. If you do not set the current package to the `:gbbopen-user` package, references to GBBopen functions and variables will not map to the proper symbols.

2 Creating a Unit Instance

Blackboard objects in GBBopen are called *unit instances*. Each unit instance is a member of a *unit class*. The unit class defines the structure of all instances of the unit class, such as the slots in each unit instance. At a more precise Common Lisp level, every GBBopen unit instance is member of a (possibly non-direct) subclass of GBBopen's unit class, `standard-unit-instance`, which is itself a subclass of Common Lisp's `standard-object` class. Even more technically, the metaclass of each GBBopen unit class is an instance of the GBBopen metaclass, `standard-unit-class`, which is a subclass of Common Lisp's `standard-class` metaclass. In other words, GBBopen classes and unit instances fit naturally into the CLOS (the Common Lisp Object System) and MOP ([Metaobject Protocol](#)) hierarchies. Fortunately, understanding such details is not required to put GBBopen to use.

The word “unit” is used in GBBopen because it has a very neutral meaning that is unlikely to be confused with programming terminology, such as “object,” or with application-domain concepts, such as “location” or “goal”. When someone refers to “unit instances,” it is clear that they are talking about GBBopen's blackboard objects. Thus, “unit instance” always refers to an instance of a particular unit class, and “unit class” refers to a class of unit instances.

So let's start using them...

This exercise shows you how to:

- Define a unit class
 - Create a unit instance
 - Display a description of a unit instance
 - Find a unit instance by its name
 - Read and write slot values
-

Prerequisites

If you ended the Common Lisp session used in the last exercise, begin a new session and evaluate the following forms:

```
cl-user> (load "<install-dir>/startup.lisp")
...
cl-user> (module-manager:compile-module :gbbopen-user :propagate
:create-dirs)
...
cl-user> (in-package :gbbopen-user)
#<package GBBOPEN-USER>
gbbopen-user>
```

Step 1: Define the location unit class

We begin by defining a unit class `location` that has two slots, named `x` and `y`.

```
gbbopen-user> (define-unit-class location ()
              (x y))
#<location>
gbbopen-user>
```

This unit-class definition instructs GBBopen to:

- Define initialization argument `:x` for the `x` slot and `:y` for the `y` slot. You will use these initialization arguments in the next step, to specify initial slot values for an instance.
- Define reader and writer methods for the `x-of` and `y-of` generic slot-accessor functions, used to reading and modifying the `x` and `y` slot values.

Step 2: Create a unit instance

Next, you can create a *unit instance* for the `location` unit class. To simplify access to the unit instance during subsequent activities, first define a global variable called `ui` by entering the following form:

```
gbbopen-user> (defvar ui)
ui
gbbopen-user>
```

Now, create a unit instance and assign it to the variable `ui` by entering the form:

```
gbbopen-user> (setf ui (make-instance 'location :x 40 :y 60))
#<location 1>
gbbopen-user>
```

This creates a unit instance of the `location` class, initializes the instance's `x` and `y` slots with the values specified by the `:x` and `:y` initialization arguments. The created unit instance is then assigned to the variable `ui`.

Step 3: Display a description of the unit instance

Now, display a description of the unit instance by using GBBopen's **describe-instance** function. Enter the following form:

```
gbbopen-user> (describe-instance ui)
Location #<location 1>
  Instance name: 1
  Space instances: None
  Dimensional values: None
  Non-link slots:
    x: 40
    y: 60
  Link slots: None
gbbopen-user>
```

Note that you didn't specify a name for the unit instance when you created it. In fact, there is often no natural reason to name each unit instance you create. By default, GBBopen names unit instances by giving it a sequentially increasing number. (For example, the unit instance you just created is named 1.) GBBopen requires that each instance of a unit class is uniquely named within the class and the accessor **instance-name-of** can be used to access this name:

```
gbbopen-user> (instance-name-of ui)
1
gbbopen-user>
```

Step 4: Find a unit instance by its name

You can look up a particular unit instance by its name. For example, to find the `location 1` unit instance by name, enter the form:

```
gbbopen-user> (find-instance-by-name 1 'location)
#<location 1>
gbbopen-user>
```

Of course, we assigned the `location` unit instance to the global variable `ui`, but it is nice to know that we can always find our unit instance using its name.

Step 5: Change the `x` slot value

Use the `x-of` reader and writer methods `GBBopen` defined for the `x` slot (in Step 3 above) to get the value of the `x` slot of the unit instance:

```
gbbopen-user> (x-of ui)
40
gbbopen-user>
```

change the slot value to 50:

```
gbbopen-user> (setf (x-of ui) 50)
50
gbbopen-user>
```

and then get the (new) value of the `x` slot:

```
gbbopen-user> (x-of ui)
50
gbbopen-user>
```

Display the description of the `location` unit instance again, observing the changed `x` slot value:

```
gbbopen-user> (describe-instance ui)
Location #<location 1>
  Instance name: 1
  Space instances: None
  Dimensional values: None
  Non-link slots:
    x: 50
    y: 60
  Link slots: None
gbbopen-user>
```


3 Creating a Space Instance

In GBBopen, *space instances* serve as containers for unit instances. As we saw in the last exercise, unit instances need not be placed in a space instance. They are perfectly useful on their own. In this exercise we will create a space instance and add and remove unit instances from it. A unit instance can also be contained in multiple space instances at the same time. Although containment “in” a space instance is a more technically correct phrase, historically developers talk of unit instances being “on” a space instance or of adding a unit instance “to” a space instance. Whichever preposition is used, the meaning is the same.

As with unit instances, each space instance is a member of a *space class*. The space class defines the structure of all instances of the space class, such as any slots associated with each space instance. Unlike unit instances, however, the standard GBBopen space class, `standard-space-instance`, is often sufficient for most applications. Therefore, space-instance subclasses rarely need to be defined.

Space instances can be organized into hierarchical structures, much like the directories in a hierarchical file system. A useful, but not exact, analogy is to think of unit instances as being similar to files and space instances as similar to directories. Stretching this analogy one step further, adding a unit instance to a space instance is akin to creating a symbolic link to a file in a directory. Removing the unit instance from the space instance is like removing the symbolic link: the unit instance itself, like the file, is not deleted by the removal.

This exercise shows you how to:

- Create a space instance
 - Find a space instance by its path
 - Add and remove a unit instance from the space instance
 - Display a description of the blackboard repository
 - Find unit instances on a space instance
-

Prerequisites

If you ended the Common Lisp session used in the last exercise, begin a new session and evaluate the following forms:

```
cl-user> (load "<install-dir>/startup.lisp")
...
cl-user> (module-manager:compile-module :gbbopen-user :propagate
:create-dirs)
...
cl-user> (in-package :gbbopen-user)
#<package GBBOPEN-USER>
gbbopen-user> (define-unit-class location ()
                (x y))
#<location>
gbbopen-user> (defparameter ui (make-instance 'location :x 40 :y 60))
ui
gbbopen-user>
```

Step 1: Create a space instance

Create a *space instance* named `known-world`. To simplify access to the space instance during subsequent activities, first define a global variable called `si` by entering the following form:

```
gbbopen-user> (defvar si)
si
gbbopen-user>
```

Now, create the space instance and assign it to the variable `si` by entering the form:

```
gbbopen-user> (setf si (make-space-instance '(known-world)))
#<standard-space-instance (known-world)>
gbbopen-user>
```

The argument to the **make-space-instance** is the “path” to be used for the created space instance. A *space-instance path* is the complete list of space-instance names, starting with the name of the most distant indirect parent, that uniquely identifies a space instance in the blackboard repository. Our `known-world` space instance does not have a parent, so its path is simply `(known-world)`.

Step 2: Display a description of the blackboard repository

Display a description of the blackboard repository, which now contains the `known-world` space instance:

```
gbbopen-user> (describe-blackboard-repository)

Space Instance          Contents
-----
known-world            Empty

Unit Class              Instances
-----
location                1
standard-space-instance 1
-----
                        2 instances

gbbopen-user>
```

The description indicates that:

- One space instance, `known-world`, exists in the repository
- The `known-world` space instance is empty; there are no unit instances stored in it
- The `location` unit instance exists, but it does not reside on any space instance
- There is one `standard-space-instance`, our `known-world` space instance, which also does not reside on any space instance

Step 3: Find a space instance by its path

You can look up a particular space instance by its space-instance path:

```

gbbopen-user> (find-space-instance-by-path '(known-world))
#<standard-space-instance (known-world)>
gbbopen-user>

```

We assigned the `known-world` space instance to the global variable `si`, but it is nice to know that we can always find it again using its path.

Step 4: Add the unit instance to the space instance

Now, add the `location` unit instance to the space instance. Enter the following form:

```

gbbopen-user> (add-instance-to-space-instance ui si)
#<location 1>
gbbopen-user>

```

Step 5: Again, display the blackboard-repository description

Display the description of the blackboard repository again:

```

gbbopen-user> (describe-blackboard-repository)

Space Instance                                Contents
-----
known-world                                  1 instance (1 location)

Unit Class                                    Instances
-----
location                                     1
standard-space-instance                       1
-----
                                             2 instances

gbbopen-user>

```

This time the description indicates that the `known-world` space instance has one instance of the `location` unit class stored on it.

Step 6: Display the description of the unit instance

Display the description of the `location` unit instance once again, observing the change in the space instances from the last exercise:

```

gbbopen-user> (describe-instance ui)
Location #<location 1>
  Instance name: 1
  Space instances: ((known-world))
  Dimensional values: None
  Non-link slots:
    x: 50
    y: 60
  Link slots: None
gbbopen-user>

```

Step 7: Find the unit instance on the space instance

Now that the `location` unit instance is on the `known-world` space instance, we can find it on the space instance. The form is as follows, where `:all` is a very basic retrieval pattern specifying that all unit instances on the `known-world` space instance are to be returned:

```
gbbopen-user> (find-instances 'location ' (known-world) :all)
(#<location 1>)
gbbopen-user>
```

The list of found unit instances is returned. (In this case, there is only one in the list.)

This is a very simple retrieval; however, GBBopen can perform extremely complex searches as well. We will use more complex retrieval patterns in upcoming exercises.

Step 8: Add another unit instance to the space instance

Create a second instance of the `location` unit class, with `x` and `y` slot values 80 and 90, respectively, and add it to the `known-world` space instance:

```
gbbopen-user> (add-instance-to-space-instance
               (make-instance 'location :x 80 :y 90)
               si)
#<location 2>
gbbopen-user>
```

This time we did not assign the new `location` unit instance to a global variable but, as before, we can find the unit instance by its name:

```
gbbopen-user> (find-instance-by-name 2 'location)
#<location 2>
gbbopen-user>
```

As you would expect, we can retrieve both `location` unit instances from the `known-world` space instance using **find-instances**:

```
gbbopen-user> (find-instances 'location ' (known-world) :all)
(#<location 2> #<location 1>)
gbbopen-user>
```

Step 9: Again, display the blackboard-repository description

Display the description of the blackboard repository again:

```
gbbopen-user> (describe-blackboard-repository)

Space Instance          Contents
-----
known-world            2 instances (2 location)

Unit Class             Instances
-----
location              2
standard-space-instance 1
```

```
-----  
3 instances  
gbbopen-user>
```

This time the description indicates that the `known-world` space instance has both instances of the `location` unit class stored on it.

Step 10: Remove a unit instance from the space instance

Now, remove the first `location` unit instance from the `known-world` space instance. Enter the following form:

```
gbbopen-user> (remove-instance-from-space-instance ui si)  
#<location 1>  
gbbopen-user>
```

As you would expect, only the second `location` unit instance remains on the `known-world` space instance:

```
gbbopen-user> (find-instances 'location ' (known-world) :all)  
(#<location 2>)  
gbbopen-user>
```

and describing the `location` unit instance confirms this:

```
gbbopen-user> (describe-instance ui)  
Location #<location 1>  
  Instance name: 1  
  Space instances: None  
  Dimensional values: None  
  Non-link slots:  
    x: 50  
    y: 60  
  Link slots: None  
gbbopen-user>
```


4 Deleting Instances

GBBopen retains created unit and space instances until they are explicitly deleted. This behavior is important to blackboard applications, where shared information in the form of unit instances remains available until a decision is made to remove them.

In this exercise, we explore some implications of deleting unit and space instances.

This exercise shows you how to:

- Delete a unit instance
 - Create a space-instance hierarchy
 - Delete a space instance
 - Delete all unit and space instances from the blackboard repository
-

Prerequisites

If you ended the Common Lisp session used in the last exercise, begin a new session and evaluate the following forms:

```
cl-user> (load "<install-dir>/startup.lisp")
...
cl-user> (module-manager:compile-module :gbbopen-user :propagate
:create-dirs)
...
cl-user> (in-package :gbbopen-user)
#<package GBBOPEN-USER>
gbbopen-user> (define-unit-class location ()
              (x y))
#<standard-unit-class location>
gbbopen-user> (defparameter ui (make-instance 'location :x 50 :y 60))
ui
gbbopen-user> (defparameter si (make-space-instance '(known-world)))
si
gbbopen-user> (add-instance-to-space-instance
              (make-instance 'location :x 80 :y 90)
              si)
#<location 2>
gbbopen-user>
```

Step 1: Create a few more unit instances

Just to review, we created two `location` unit instances. The unit instance named `1` is no longer on the `known-world` space instance, but it is still assigned to the global variable `ui`:

```
gbbopen-user> ui
#<location 1>
gbbopen-user>
```

The `location` unit instance named 2 is the only `location` unit instance on the `known-world` space instance:

```
gbbopen-user> (find-instances 'location '(known-world) :all)
(#<location 2>)
gbbopen-user>
```

Now, let's create five more `location` unit instances. Enter:

```
gbbopen-user> (dotimes (i 5) (make-instance 'location))
nil
gbbopen-user>
```

Note that we did not specify `x` and `y` slot values for these new unit instances. We will explore the implications of this shortly.

Step 2: Apply a function to all instances of a unit class

We did not assign the new `location` unit instances to global variables or add them to the `known-world` space instance. Can we still reference them? If we know the names of the new unit instances, we can use the **find-instance-by-name** function that we learned earlier. For example:

```
gbbopen-user> (find-instance-by-name 5 'location)
#<location 5>
gbbopen-user>
```

It is often useful to perform some action on all instances of a unit class. GBBopen provides a *mapping function*, or “iterator,” that repeatedly calls a function with each instance of a unit class as the argument to the function. For example:

```
gbbopen-user> (map-instances-of-class #'print 'location)

#<location 6>
#<location 3>
#<location 7>
#<location 1>
#<location 2>
#<location 4>
#<location 5>
nil
gbbopen-user>
```

displays each of our `location` unit instances. Note that the exact order that `location` unit instances are supplied to the `print` function may differ from the above example in your Common Lisp implementation.

Currently, there is only one `location` unit instance on the `known-world` space instance:

```
gbbopen-user> (find-instances 'location '(known-world) :all)
(#<location 2>)
gbbopen-user>
```

Let's use **map-instances-of-class** to add all the `location` unit instances to the `known-world`:

```

gbbopen-user> (map-instances-of-class
                #'(lambda (instance)
                    (add-instance-to-space-instance instance si)
                    'location))
Warning: In add-instance-to-space-instance: #<location 2> is already on
space instance #<standard-space-instance (known-world)>.
nil
gbbopen-user>

```

GBBopen warns us that the `location 2` unit instance is already on the `known-world` and adds all the other `location` instances. We can verify this by using **find-instances:**

```

gbbopen-user> (find-instances 'location ' (known-world) :all)
(#<location 7> #<location 6> #<location 5> #<location 4> #<location 3>
 #<location 2> #<location 1>)
gbbopen-user>

```

For those who prefer a more iterative programming style, GBBopen provides a `dolist`-style macro, **do-instances-of-class**, as an alternative to **map-instances-of-class**. So, to add all the `location` unit instances to the `known-world`, we could have chosen to use the form:

```

(do-instances-of-class (instance 'location)
  (add-instance-to-space-instance instance si))

```

instead of the **map-instances-of-class** version. As with things stylistic, the choice is yours.

Step 3: Delete a unit instance

Let's delete the first `location` unit instance that we created:

```

gbbopen-user> (delete-instance ui)
#<deleted-unit-instance location 1>
gbbopen-user>

```

Note that the displayed representation indicates that the unit instance has been deleted.

We can no longer find the deleted unit instance by its name:

```

gbbopen-user> (find-instance-by-name 1 'location)
nil
gbbopen-user>

```

and it is no longer included in a **map-instances-of-class** iteration:

```

gbbopen-user> (map-instances-of-class #'print 'location)

#<location 6>
#<location 3>
#<location 7>
#<location 2>
#<location 4>
#<location 5>
nil
gbbopen-user>

```

and it is no longer on the known-world space instance:

```
gbbopen-user> (find-instances 'location '(known-world) :all)
(#<location 7> #<location 6> #<location 5> #<location 4> #<location 3>
 #<location 2>)
gbbopen-user>
```

However, the deleted location unit instance is still assigned to the ui global variable:

```
gbbopen-user> ui
#<deleted-unit-class location 1>
gbbopen-user>
```

and that can lead to problems. Let's try to place the deleted location unit instance on the known-world space instance:

```
gbbopen-user> (add-instance-to-space-instance ui si)
Error: No methods applicable for generic function
      #<standard-generic-function add-instance-to-space-instance> with args
      (#<deleted-unit-instance location 1> #<standard-space-instance
(known-world)>)
      of classes (deleted-unit-instance standard-space-instance)
gbbopen-user>> :a
gbbopen-user>
```

Most of GBBopen's operations signal an error if they are given a deleted unit instance. This is because **delete-instance** changes the class of the deleted unit instance to `deleted-unit-instance` which, despite its name, is not a `standard-unit-instance`. We cannot use **describe-instance** to verify this, because the deleted unit instance is no longer a unit instance, but we can use Common Lisp's `describe` function:

```
gbbopen-user> (describe ui)
#<deleted-unit-instance location 1> is an instance of
  #<standard-class deleted-unit-instance>:
The following slots have :instance allocation:
  instance-name    1
  original-class   #<standard-unit-class location>
gbbopen-user>
```

Note that the slots that we defined for location unit instances are not present in the `deleted-unit-instance` object. A deleted unit instance has only two slots, `instance-name` and `original-class`, which are set when the class of the deleted unit instance is changed to `deleted-unit-instance`.

Several GBBopen operations on a deleted unit instance do not signal errors. In particular:

```
gbbopen-user> (instance-name-of ui)
1
gbbopen-user>
```

The value returned by **instance-name-of**, along with that by **original-class-of**, can be used to identify the unit instance that, when deleted, became the `deleted-unit-instance`.

Typically, blackboard applications obtain unit instances from the blackboard repository (or as we will see in Exercise 8, as an event argument) rather than maintaining references to them in variables. This limits the possibility of retaining a deleted unit instance and performing GBBopen operations on it. The deletion status of a unit instance can be determined using the **instance-deleted-p** predicate:

```
gbbopen-user> (instance-deleted-p ui)
t
gbbopen-user>
```

Step 4: Create a simple space-instance hierarchy

In the last exercise, we noted that space instances can be organized in hierarchical structures. To illustrate this, let's create a few more space instances:

```
gbbopen-user> (make-space-instance '(known-world my-town))
#<standard-space-instance (known-world my-town)>
gbbopen-user> (make-space-instance '(known-world my-town east-side))
#<standard-space-instance (known-world my-town east-side)>
gbbopen-user> (make-space-instance '(known-world my-town west-side))
#<standard-space-instance (known-world my-town west-side)>
gbbopen-user>
```

Recall that the *space-instance-path* argument to **make-space-instance** is the complete list of space-instance names, starting with the name of the most distant indirect parent. So, the `my-town` space instance has the `known-world` as its parent and the `east-side` and `west-side` as children. We can use **describe-blackboard-repository** to see this organization:

```
gbbopen-user> (describe-blackboard-repository)

Space Instance                                Contents
-----
known-world                                  6 instances (6 location)
  my-town                                    Empty
    east-side                                Empty
    west-side                                Empty

Unit Class                                    Instances
-----
location                                     6
standard-space-instance                       4
-----
                                           10 instances

gbbopen-user>
```

Observe from the above description that child space instances that we created are not placed on their parent. Unlike the directories in a file system in the analogy that we presented in the last exercise, in GBBopen a space-instance hierarchy is orthogonal to containment. In fact, space instances are actually unit instances (of the class **standard-space-instance**, by default) and a space instance can be placed on other space instances—or even on itself. Typical blackboard applications do not involve using space instances as unit instances, but this property of space instances is very powerful when it is needed.

The functions **parent-of** and **children-of** are provided to traverse the space-instance hierarchy. For example:

```
gbbopen-user> (children-of
                (find-space-instance-by-path '(known-world my-town)))
(#<standard-space-instance (known-world my-town west-side)>
 #<standard-space-instance (known-world my-town east-side)>)
gbbopen-user>
```

Step 5: Add a unit instance to multiple space instances

Now, let's add location 2 to the my-town and east-side space instances:

```
gbbopen-user> (let ((location-2 (find-instance-by-name 2 'location)))
                (add-instance-to-space-instance location-2 '(known-world
my-town))
                (add-instance-to-space-instance location-2 '(known-world
my-town east-side)))
#<location 2>
gbbopen-user>
```

As we expect, location 2 is now on three space instances:

```
gbbopen-user> (describe-instance (find-instance-by-name 2 'location))
Location #<location 2>
  Instance name: 2
  Space instances: ((known-world my-town east-side)
                   (known-world my-town)
                   (known-world))
  Dimensional values: None
  Non-link slots:
    x: 80
    y: 90
  Link slots: None
gbbopen-user>
```

and the description of the blackboard repository is:

```
gbbopen-user> (describe-blackboard-repository)

Space Instance          Contents
-----
known-world            6 instances (6 location)
  my-town              1 instance (1 location)
    east-side          1 instance (1 location)
    west-side          Empty

Unit Class             Instances
-----
location                6
standard-space-instance 4
-----
                        10 instances

gbbopen-user>
```

Placing a unit instance on multiple space instances is useful when each space instance represents a different view of unit instances. In this case, location 2 is in the known-world, in my-town, and on the east-side:

```
gbbopen-user> (find-instances 'location '(known-world my-town) :all)
(#<location 2>)
gbbopen-user> (find-instances 'location '(known-world my-town east-side)
:all)
(#<location 2>)
gbbopen-user>
```

Step 6: Delete a space instance

Now let's delete some of what we just created. Let's delete the `my-town` space instance:

```
gbbopen-user> (delete-space-instance '(known-world my-town))
#<deleted-unit-instance standard-space-instance (known-world my-town)>
gbbopen-user>
```

and describe the blackboard repository:

```
gbbopen-user> (describe-blackboard-repository)

Space Instance          Contents
-----
known-world            6 instances (6 location)

Unit Class             Instances
-----
location                6
standard-space-instance 1
-----
                        7 instances

gbbopen-user>
```

Notice that GBBopen has also deleted all the child space instances of `my-town`: both `east-side` and `west-side` were also deleted. So, just as with our file-system directory analogy, space-instance deletion is recursive over children and should be performed with caution.

Also note that deleting space instances did not delete any of the unit instances that were stored on them. If we want to delete a space instance *and* all the unit instances that are stored on it, we must explicitly delete the unit instances before deleting the space instance. For example, we could do:

```
gbbopen-user> (map-instances-on-space-instances #'delete-instance
              'location '(known-world my-town))
nil
gbbopen-user>
```

to delete the unit instances in `my-town`. However, we must keep in mind that a unit instance can reside on multiple space instances; so deleting a unit instance that is on other space instances might not be the desired action.

As is the case with **map-instances-of-class**, GBBopen provides a **do-instances-on-space-instances** macro as an alternative to **map-instances-on-space-instances**. So we could have chosen to use the form:

```
(do-instances-on-space-instances (instance 'location '(known-world my-town))
  (delete-instance instance))
```

to explicitly delete the `location` unit instances in `my-town`.

Step 7: Delete the Blackboard Repository

If we really want to get rid of all our unit and space instances, we can use the **delete-blackboard-repository** function:

```
gbbopen-user> (delete-blackboard-repository)
t
gbbopen-user> (describe-blackboard-repository)
There are no space instances in the blackboard repository.
gbbopen-user> (map-instances-of-class #'print location)
nil
gbbopen-user>
```

Calling **delete-blackboard-repository** has deleted every space and unit instance, but it has not eliminated our `location` unit class definition. Let's create a `location` unit instance once again:

```
gbbopen-user> (make-instance 'location)
#<location 1>
gbbopen-user>
```

Note that deleting the blackboard repository also reset the counter for `location` instance names, so the created unit instance is again named 1.

5 Enhancing Your Development Environment

Now that you are experienced creating and deleting unit and space instances in GBBopen, we will take a short break before working further on our random-walk application. In the exercises thus far, we have been working directly in Common Lisp's REPL. As our application develops, we want to save our code in files. In this exercise, we will provide recommendations for making your GBBopen and Common Lisp environment more productive. Even if you have already customized your Common Lisp setup, I recommend surveying this exercise for useful GBBopen tips.

This exercise shows you how to:

- Add GBBopen keyword commands to your Common Lisp implementation
 - Customize your Common Lisp initialization file
 - Set up GBBopen HyperDoc and Common Lisp HyperSpec access
-

Step 1: Autoloading GBBopen

Thus far, we have entered the forms:

```
cl-user> (load "<install-dir>/startup.lisp")
...
cl-user> (module-manager:compile-module :gbbopen-user :propagate
:create-dirs)
...
cl-user> (in-package :gbbopen-user)
#<package GBBOPEN-USER>
gbbopen-user>
```

to compile and load needed GBBopen components and to set the current package to `:gbbopen-user`. We can set up our Common Lisp environment so that we can do this (and eventually compile and load our random-walk application) by issuing only a single command.

The file `<install-dir>/initiate.lisp` is an alternative for `<install-dir>/startup.lisp` that adds handy GBBopen keyword commands to the top-level REPL listener for [Allegro CL](#), [CLISP](#), [Clozure CL](#), [CMUCL](#), [ECL](#), [LispWorks](#), [SBCL](#) and [Sciener CL](#) users. GBBopen keyword commands are also supported in [SLIME](#)'s Emacs-based REPL. Other interaction interfaces that use their own REPL, rather than the one provided by Common Lisp, may not support keyword REPL command processing (see below).

To make these GBBopen keyword commands available, simply load the file `<install-dir>/initiate.lisp` rather than `<install-dir>/startup.lisp`. Let's try it. Start up a fresh Common Lisp session and enter the following form in the REPL:

```
cl-user> (load "<install-dir>/initiate.lisp")
;; Loading <install-dir>/initiate.lisp
;; GBBopen is installed in <install-dir>
;; Your "home" directory is <homedir>
;;   Loading <install-dir>/extended-repl.lisp
;;   Loading <install-dir>/commands.lisp
;;   Loading <install-dir>/gbbopen-modules-directory.lisp
;; No shared module command definitions were found in <install-dir>/gbbopen-modules/.
;; No personal module command definitions were found in <homedir>/gbbopen-modules/.
```

```
t
cl-user>
```

Note that the Module Manager Facility and GBBopen module definitions have not been loaded. Now enter the REPL command, `:gbbopen-user`:

```
cl-user> :gbbopen-user
;; Loading <install-dir>/startup.lisp
...
;; Loading <install-dir>/<platform-dir>/gbbopen/gbbopen-user.fasl
gbbopen-user>
```

The `:gbbopen-user` module (and supporting modules) are loaded, and the current package is set to `:gbbopen-user`:

```
gbbopen-user> *package*
#<package GBBOPEN-USER>
gbbopen-user>
```

Module-compilation REPL commands, such as `:gbbopen-user`, always include the `:propagate` option when they call the Module Manager's **compile-module** function. We don't need to specify it as a command option.

If it didn't work...

If you are running GBBopen in a Common Lisp environment that doesn't support REPL commands and the `:gbbopen-user` REPL command didn't work, all is not lost. Loading `<install-dir>/initiate.lisp` also defines functions in the `:common-lisp-user` package with the same name as the REPL command:

```
cl-user> (gbbopen-user)
;; Loading <install-dir>/startup.lisp
...
;; Loading <install-dir>/<platform-dir>/gbbopen/gbbopen-user.fasl
gbbopen-user>
```

REPL command syntax

GBBopen's REPL commands are defined to mimic the syntax of existing REPL commands in your Common Lisp environment. On many Common Lisp implementations, or if you are using the [SLIME](#) interface, REPL commands with arguments can be specified in either list or "spread" notation. For example:

```
cl-user> (:gbbopen-user :create-dirs)
;; Loading <install-dir>/startup.lisp
...
;; Loading <install-dir>/<platform-dir>/gbbopen/gbbopen-user.fasl
gbbopen-user>
```

or

```
cl-user> :gbbopen-user :create-dirs
;; Loading <install-dir>/startup.lisp
...
;; Loading <install-dir>/<platform-dir>/gbbopen/gbbopen-user.fasl
gbbopen-user>
```

However, [Allegro CL](#), [CLISP](#), and [LispWorks](#) do not support the list notation. CLISP versions prior to 2.45 and [ECL](#) only support the spread notation—but without arguments. So, a user running on these Common Lisp implementations must use the `:common-lisp-user` functions rather than a REPL command when the command requires arguments:

```
cl-user> (cl-user:gbopen-user :create-dirs)
;; Loading <install-dir>/startup.lisp
...
;; Loading <install-dir>/<platform-dir>/gbopen/gbopen-user.fasl
gbopen-user>
```

or use the [SLIME](#) REPL interface.

We will show the spread command syntax (see page 30) throughout the rest of the Tutorial. If you are running a [CLISP](#) version prior to 2.45, remember to use the `:common-lisp-user` function rather than the REPL command when the command requires arguments. (Equivalent functions in the `:common-lisp-user` package are always defined for each REPL keyword command, and these functions can be used when REPL keyword processing is not fully supported.)

Step 2: Customizing your Common Lisp initialization file

I'm lazy and would rather not have to explicitly load the `<install-dir>/initiate.lisp` file each time I start a new Common Lisp session. So, I have my Common Lisp's initialization file load it for me. To make it easy to similarly customize different Common Lisp implementations, I use the following strategy:

1. I create a file, `shared-init.lisp`, in my home directory, containing the following:

```
(in-package :common-lisp-user)

;; My personal preferences. Note: Allegro CL requires
;; tpl:setq-default during initialization to retain changes
;; to these global variables (done in .clinit.cl):
(setf *print-case* ' :downcase)
(setf *compile-verbose* 't)
(setf *load-verbose* 't)

(let ((defaults *load-truename*))
  (load (make-pathname
        ;; where GBOpen is installed:
        :directory ' (:absolute "usr" "local" "gbopen")
        :name "initiate"
        :type "lisp"
        :defaults defaults)))
```

The `:directory` argument to `make-pathname` is a Common Lisp *absolute pathname* that is portable across all operating systems. I installed my GBOpen in `/usr/local/gbopen/`, and I specified this with the "usr", "local", and "gbopen" elements in the `:directory` argument. Change these elements as appropriate for the location of your GBOpen installation.

2. For each Common Lisp implementation that I use, I create a personal initialization file that performs any implementation-specific initializations and then loads `shared-init.lisp`. For example, here is the `.clisprc` initialization file that I use for [CLISP](#):

```
(in-package :common-lisp-user)

;; enable maximum ANSI compliance:
(setf custom:*ansi* 't)

(let ((defaults *load-truename*))
  (load (make-pathname
         :name "shared-init"
         :type "lisp"
         :defaults defaults)))
```

SBCL has a very strict interpretation of `*load-truename*` semantics, so my `.sbclrc` initialization file is:

```
(in-package :common-lisp-user)

;; *load-truename* returns nil when used in SBCL's .sbclrc
;; initialization file, so use (user-homedir-pathname):
(let ((defaults (user-homedir-pathname)))
  (load (make-pathname
         :name "shared-init"
         :type "lisp"
         :defaults defaults)))
```

Here is my `.clinit.cl` initialization file for **Allegro CL**:

```
(in-package :common-lisp-user)

;; Allegro CL requires tpl:setq-default during initialization
;; to retain changes to these global variables:
(tpl:setq-default *print-case* ' :downcase)
(tpl:setq-default *compile-verbose* 't)
(tpl:setq-default *load-verbose* 't)

(let ((defaults *load-truename*))
  (load (make-pathname
         :name "shared-init"
         :type "lisp"
         :defaults defaults)))
```

The initialization file names for supported GBBopen ports are:

Allegro CL	<code>.clinit.cl</code>
CLISP	<code>.clisprc</code>
Clozure CL	<code>ccl-init.lisp</code>
CMUCL	<code>init.lisp</code>
Digitool MCL	(see below)
ECL	<code>.eclrc</code>
LispWorks	<code>.lispworks</code>
SBCL	<code>.sbclrc</code>
Scieneer CL	<code>init.lisp</code>
XCL	<code>.xclrc</code>

Digitool MCL does not look for an initialization file in the user's home directory. Instead it loads the file `init.lisp` in its installation directory. One approach is to have that file load a personal initialization file, say `mcl-init.lisp`, from the user's home directory, if a `mcl-init.lisp` file is present.

The Personal Edition of [LispWorks](#) does not load initialization files, requiring you to manually load your `.lispworks` file each time you start up [LispWorks](#).

The interpretation of where a user's "home" directory is located is inconsistent on Windows. Ideally, the "home directory" location used for the Common Lisp implementation's initialization file and the result of the Common Lisp function `user-homedir-pathname` should be consistent. Then, by using the `shared-init.lisp` scheme, you only need to determine where the initialization file should be for each Common Lisp implementation that you use, and then you can have those implementation-specific initialization files load your `shared-init.lisp` file from whichever directory you deem as your "home" directory.

Here is a quick way to have Common Lisp tell you where it thinks your "home" directory is located:

```
cl-user> (not (princ (truename (user-homedir-pathname))))
C:\Documents and Settings\corkill\
nil
cl-user>
```

Alternatively, loading either GBBopen's `<install-dir>/startup.lisp` file or `<install-dir>/initiate.lisp` will display the "home" directory.

We will refer to this directory from now on as your "homedir" directory.

3. With this setup in place, all that is needed to use GBBopen is to start up a Common Lisp and type a GBBopen command, such as `:gbbopen-user`. My fingers thank me!

Step 3: Set up HyperDoc and HyperSpec access

If you are using [Emacs](#) in your Common Lisp development environment, you can make it easy to bring up appropriate Common Lisp and GBBopen documentation in your browser. The file `<install-dir>/browse-hyperdoc.el` defines an interactive Emacs command named `browse-hyperdoc` and binds it to `META-?` (on most keyboards, `META-?` means pressing both `Alt` and `?` keys at the same time). To enable this Emacs command, add a command to load `<install-dir>/browse-hyperdoc.el` in your `.emacs` initialization file:

```
;; GBBopen hyperdoc (where GBBopen was installed):
(load "<install-dir>/browse-hyperdoc")
```

If there is no `.emacs` file present in your home directory, simply create one containing the above command. Once again, Windows users need to worry about where Emacs looks for their "home" directory.

While you are editing your `.emacs` file, you might also want to add a command to load GBBopen's Emacs indentation customizations:

```
;; GBBopen indentations (where GBBopen was installed):
(load "<install-dir>/gbbopen-indent")
```

If you are using [Lispbox](#), you will have to enable loading of the `.emacs` initialization file, as it is disabled in the Lispbox startup file. Edit the Lispbox startup file (`lispbox.bat` on Windows, `lispbox.sh` on Unix, and `/Applications/Lispbox/Emacs.app/Contents/MacOS/lispbox.sh` on Mac OS) and remove the `--no-init-file` option from the Emacs invocation line.

Adding the Common Lisp HyperSpec

The `hyperspec.el` utility is included in the [SLIME](#) and [ILISP](#) distributions. However, if `hyperspec.el` is not already part of your Emacs, you can download it and explicitly load it from your `.emacs` initialization file. Once `hyperspec.el` is present, GBBopen's `browse-hyperdoc` Emacs command will automatically defer to the [Common Lisp HyperSpec](#) when given a non-GBBopen entity.

I prefer to download a local copy of the Common Lisp HyperSpec using the [down-loadable archive](#) provided by [LispWorks](#), LTD. This allows me to quickly reference the HyperSpec without a network connection. I set the value of `common-lisp-hyperspec-root` in my `.emacs` initialization file to a URL that points to my local copy of the HyperSpec:

```
(setf common-lisp-hyperspec-root "file:/usr/local/CLHS/")
```

Non-Emacs access

If you are not using an Emacs-based environment, GBBopen provides a Common Lisp function, **`browse-hyperdoc`**, that can be used to access GBBopen HyperDoc pages from Common Lisp. GBBopen's `:os-interface` module must be loaded to make **`browse-hyperdoc`** available. For example:

```
cl-user> :os-interface
;; Loading <install-dir>/startup.lisp
...
;; Loading <install-dir>/<platform-dir>/tools/os-interface.fasl
gbbopen-tools> (browse-hyperdoc 'define-unit-class)
t
gbbopen-tools>
```

will display the GBBopen HyperDoc page for **`define-unit-class`** in your browser. Note that the `:gbbopen-user` module requires (and therefore loads) the `:os-interface` module, so if you have loaded `:gbbopen-user`, you do not need to do anything further in order to call **`browse-hyperdoc`**.

6 Working Within a File

Now that we've enhanced our GBBopen and Common Lisp development environment, let's begin developing the random-walk application in earnest.

This exercise shows you how to:

- Begin working with files for application development
 - Compile and load an application file using the SLIME or ELI environments
-

Step 1: Create the tutorial-example directories

Create a directory to hold the random-walk application. I'm calling mine `tutorial`. Next, create a subdirectory in that directory named `source`. The reason for doing this will become clear in an upcoming exercise (see page 79). Here are the shell commands that I used to create my directories:

```
[~]$ mkdir tutorial
[~]$ cd tutorial
[~/tutorial]$ mkdir source
[~/tutorial]$
```

Step 2: Create the tutorial-example file

Start up a fresh Common Lisp session and load the `:gbbopen-user` module, using the REPL command we set up in the last exercise:

```
cl-user> :gbbopen-user
;; Loading <install-dir>/startup.lisp
...
;; Loading <install-dir>/<platform-dir>/gbbopen/gbbopen-user.fasl
gbbopen-user>
```

Next, begin editing a new file named `tutorial-example.lisp` in the `source` subdirectory that you just created. Even if you are more comfortable using another editor, use the editing facilities that are provided by your Common Lisp environment. The development features of a quality Common Lisp environment are well worth the price of learning a new editor. In an Emacs-based environment, such as [SLIME](#) or [Allegro CL's ELI](#), typing `C-x C-f` will prompt you for the name of a file to editor or create. (We will use Emacs key-binding notation, where `C-x C-f` means typing `^c` followed by `^f`.)

Type the following two forms into the `tutorial-example.lisp` file buffer:

```
(in-package :gbbopen-user)

(define-unit-class location ()
  (x y))
```

The `in-package` form specifies the Common Lisp package that is made current when the file is compiled or loaded. The first form in every Common Lisp source file that you create should begin with an `in-package` form. This form is also used by most Common Lisp editing environments to set the package associated with development operations.

The **define-unit-class** definition is the same one we used in Exercise 2.

Now, save the file.

Step 3: Compile and load the tutorial-example file

At this point, we have been using Common Lisp's development environment, but we have not loaded the forms in our file into Common Lisp:

```
gbbopen-user> (make-instance 'location)
Error: No class named: location.
gbbopen-user>> :a
gbbopen-user>
```

We could use Common Lisp's `compile-file` to compile the file and then load to load the resulting compiled file. For example:

```
gbbopen-user> (load (compile-file "~/tutorial/source/tutorial-example.lisp"))
;; Compiling file ~/tutorial/source/tutorial-example.lisp
;; Loading ~/tutorial/source/tutorial-example.fasl
t
gbbopen-user>
```

but you should be able to compile and load the file directly from the editor buffer. In SLIME, the command `C-c C-k` will compile and load the file currently being edited. Allegro's ELI interface `compile-and-load` command is `C-c C-b`. Identify and use the `compile-and-load-file` command in your editing environment.

Verify that all is well by creating a `location` unit instance in the REPL:

```
gbbopen-user> (make-instance 'location)
#<location 1>
gbbopen-user>
```

Then delete the blackboard repository in preparation for the next exercise:

```
gbbopen-user> (delete-blackboard-repository)
t
gbbopen-user>
```

7 Adding Dimensions

A central concept in GBBopen is *dimensionality*. Dimensional abstraction of space instances, unit instances, and proximity-based retrieval patterns is used to provide a semantically meaningful separation of blackboard-repository storage mechanisms from system and application code. This separation provides flexibility in developing and evolving complex blackboard applications and allows GBBopen to change storage and search strategies and optimizations dynamically.

Each space instance can be created as a conceptual hyper-dimensional volume. Unit instances occupy multidimensional extent based on their attributes. The location of a unit instance within the space instance is determined by the intersection of the space instance's dimensionality and the unit instance's dimension values.

GBBopen supports three types of dimensions:

- *ordered*: a real-number line
- *boolean*: true and false values
- *enumerated*: a set of named elements (the set can be either closed or infinite)

Determining the dimensionality of space and unit instances is an important part of designing a blackboard application.

In this exercise, we will redefine the `location` unit class to have two ordered dimensions, `x` and `y`, that represent Euclidean positions on a two-dimensional plane. Then we will create a two-dimensional `known-world` space instance, create some `location` unit instances on the `known-world`, and retrieve the instances based on their two-dimensional positions.

This exercise shows you how to:

- Add dimensional values to a unit-class definition
 - Create a multidimensional space instance
 - Retrieve unit instances from a space instance based on their dimensional values
 - Compile and load individual forms directly from an Emacs file buffer using the SLIME or ELI environments
-

Prerequisites

- The `tutorial-example.lisp` file created in the last exercise, containing:

```
(in-package :gbbopen-user)

(define-unit-class location ()
  (x y))
```
- The `:gbbopen-user` module is loaded

Step 1: Add dimensions to the `location` unit class

Edit your `tutorial-example.lisp` file, and change the `location` unit-class definition as follows:

```
(define-unit-class location ()
  (x y)
  (:dimensional-values
   (x :point x)
   (y :point y)))
```

In this tutorial, we will highlight code additions and changes using a black font.

The `:dimensional-values` unit-class option specifies that `location` unit instances have two ordered dimensions, `x` and `y`, and that the value of each dimension will be a single numeric value obtained from the slots `x` and `y`, respectively.

Because we chose to use the same name for each dimension and its associated slot value, our `:dimensional-values` option might appear to be double-talk. We could have defined our class as:

```
(define-unit-class location ()
  (x-slot y-slot)
  (:dimensional-values
   (x :point x-slot)
   (y :point y-slot)))
```

which clarifies the semantics of the `:dimensional-values` option. Often, however, it is most convenient to use the same name for a slot and the dimension associated with the slot's value, so we will stick with our original definition.

Step 2: Compile and load the new definition

We could compile and load the entire `tutorial-example.lisp` file just as we did in the last exercise. However, as we develop our application it can be convenient to compile and load (and debug) each form as we write it. Your Common Lisp development environment should provide this capability. In SLIME, the command to compile the current top-level form is `C-c C-c`. In Allegro's ELI, the command is `C-c C-x`. Try compiling and loading just the new `location` unit-class definition.

Step 3: Make a location unit instance

Let's test our new `location` unit class definition by making an instance. Enter the following form in the REPL:

```
gbbopen-user> (defparameter ui (make-instance 'location :x 40 :y 60))
ui
gbbopen-user>
```

and display its description:

```
gbbopen-user> (describe-instance ui)
Location #<location 1>
  Instance name: 1
  Space instances: None
  Dimensional values:
    x: 40
    y: 60
  Non-link slots:
    x: 40
```

```
      y: 60
      Link slots: None
gbbopen-user>
```

Note the dimensional values for the `x` and `y` dimensions.

Step 4: Make the known-world space instance

Create the `known-world` space instance by evaluating:

```
gbbopen-user> (defparameter si (make-space-instance '(known-world)))
si
gbbopen-user>
```

Step 5: Add the unit instance to the space instance

Now, add the `location` unit instance to the space instance:

```
gbbopen-user> (add-instance-to-space-instance ui si)
Warning: In add-instance-to-space-instance: #<location 1>
         does not share any dimensions with space instance
         #<standard-space-instance (known-world)>.
#<location 1>
gbbopen-user>
```

GBBopen has warned us that our `location` unit instance does not have any dimensions in common with the `known-world` space instance (because we didn't specify any dimensions for the `known-world`). GBBopen dutifully added the unit instance, as shown by **describe-blackboard-repository**:

```
gbbopen-user> (describe-blackboard-repository)

Space Instance          Contents
-----
known-world            1 instance (1 location)

Unit Class             Instances
-----
location                1
standard-space-instance 1
-----
                        2 instances

gbbopen-user>
```

but we cannot perform dimension-based retrieval of our `location` unit instance on the `known-world`.

Step 6: Create a dimensioned known-world

Let's delete the `known-world` and create another one—this time with `x` and `y` dimensions:

```

gbbopen-user> (delete-space-instance si)
#<deleted-unit-instance standard-space-instance (known-world)>
gbbopen-user> (setf si (make-space-instance '(known-world)
      :dimensions '((x :ordered) (y :ordered))))
#<standard-space-instance (known-world)>
gbbopen-user>

```

We have specified `x` and `y` as ordered dimensions, making the `known-world` a two-dimensional Euclidean plane. (Flatlanders would be proud!)

Verify the dimensionality of the `known-world` space instance by evaluating:

```

gbbopen-user> (describe-space-instance si)
Standard-space-instance #<standard-space-instance (known-world)>
  Allowed unit classes: t
  Dimensions:
    (x :ordered)
    (y :ordered)
gbbopen-user>

```

Step 7: Add the location unit instance to the new known-world

Add the `location` unit instance to the new `known-world` space instance:

```

gbbopen-user> (add-instance-to-space-instance ui si)
#<location 1>
gbbopen-user>

```

The dimension warning is gone.

Step 8: Adding every location to the known-world automatically

Up to this point, we have used **`add-instance-to-space-instance`** to add each `location` unit instance to the `known-world` space instance. We can tell GBBopen to automatically add new unit instances to one or more space instances by using the `:initial-space-instances` class option in **`define-unit-class`**.

Add the following `:initial-space-instances` class option to the `location` unit-class definition in your `tutorial-example.lisp` file:

```

(define-unit-class location ()
  (x y)
  (:dimensional-values
   (x :point x)
   (y :point y))
  (:initial-space-instances (known-world)))

```

Compile and load the new `location` unit-class definition.

Step 9: Create more location unit instances

Let's test our new location unit class definition by making another instance. Enter the following form in the REPL:

```
gbbopen-user> (make-instance 'location :x 70 :y 30)
#<location 2>
gbbopen-user>
```

and confirm that the new location is on the known-world:

```
gbbopen-user> :dsbb

Space Instance          Contents
-----
known-world            2 instances (2 location)

Unit Class              Instances
-----
location                2
standard-space-instance 1
-----
                        3 instances

gbbopen-user>
```

Here we used GBBopen's `:dsbb` REPL command, which is equivalent to evaluating `(describe-blackboard-repository)`. Describing the repository is a useful check that our unit instances and space instances are being created and deleted as intended.

Now, let's populate the known-world with a few more locations:

```
gbbopen-user> (make-instance 'location :x 20 :y 20)
#<location 3>
gbbopen-user> (make-instance 'location :x 25 :y 25)
#<location 4>
gbbopen-user> (make-instance 'location :x 20 :y 30)
#<location 5>
gbbopen-user>
```

and verify that they are all on the known-world:

```
gbbopen-user> :dsbb

Space Instance          Contents
-----
known-world            5 instances (5 location)

Unit Class              Instances
-----
location                5
standard-space-instance 1
-----
                        6 instances

gbbopen-user>
```

Step 10: Dimensional retrieval

We have seen how we can use **find-instances** to retrieve all `location` unit instances from the `known-world`:

```
gbbopen-user> (find-instances 'location '(known-world) :all)
(#<location 5> #<location 4> #<location 3> #<location 2> #<location 1>)
gbbopen-user>
```

Now that we have added `x` and `y` dimensions to `location` unit instances and to the `known-world`, we can retrieve `location` unit instances using dimensional patterns. For example, let's retrieve the unit instance positioned at (20,20):

```
gbbopen-user> (find-instances 'location '(known-world)
                '(and (= x 20) (= y 20)))
(#<location 3>)
gbbopen-user>
```

We can use **describe-instance** to verify that we found the desired `location` unit instance:

```
gbbopen-user> (find-instances 'location '(known-world)
                '(and (= x 20) (= y 20)))
(#<location 3>)
gbbopen-user> (describe-instance (first *))
Location #<location 3>
Instance name: 3
Space instances: ((known-world))
Dimensional values:
  x: 20
  y: 20
Non-link slots:
  x: 20
  y: 20
Link slots: None
gbbopen-user>
```

Note that we used Common Lisp's REPL `*` variable that is always set to the value returned by evaluating the last REPL expression (in this case, the result of **find-instances**).

Step 11: Customize the display of location unit instances

It would be convenient if we could easily see the coordinates of `location` unit instances without having to describe them. Fortunately, GBBopen makes this is easy to do by providing the **print-instance-slots** generic function that allows us to extend how Common Lisp's `print-object` displays `location` unit instances.

Add the following **print-instance-slots** method after the `location` unit-class definition in your `tutorial-example.lisp` file:

```
(define-unit-class location ()
  (x y)
  (:dimensional-values
   (x :point x)
   (y :point y))
```

```

(:initial-space-instances (known-world)))

(defmethod print-instance-slots ((location location) stream)
  (call-next-method)
  (when (and (slot-boundp location 'x)
             (slot-boundp location 'y))
    (format stream " (~s ~s)"
              (x-of location)
              (y-of location))))

```

The method first performs a `(call-next-method)` to produce the initial printed representation of the `location` unit instance. It then checks that both `x` and `y` slots are bound and, if so, writes the location's coordinates to the output stream. Checking that the slots are bound is necessary to avoid generating an error in the **print-instance-slots** if the slots have not been given a value. You should always perform this safety check in any **print-instance-slots** methods. (GBBopen provides the generic function **print-instance-slot-value** for use in safely displaying a slot value in **print-instance-slots** methods. We did not use **print-instance-slot-value** here, as our **print-instance-slots** method presents two slots, `x` and `y`, formatted together.)

Now, compile and load the **print-instance-slots** method and try it out:

```

gbbopen-user> (find-instances 'location ' (known-world)
                  ' (= (x y) (20 20)))
(#<location 3 (20 20)>)
gbbopen-user>

```

Note that this time we used a two-dimensional (`x,y`) retrieval pattern rather than the conjunction of two one-dimensional patterns that we used previously. The two patterns are equivalent, but often a higher-dimensional pattern may be more convenient than a conjunction. Also note that we can see immediately that we retrieved the desired `location`.

Step 12: More dimensional retrievals

Let's try some additional dimensional retrievals. First, find all `location` unit instances with an `x` position of 20:

```

gbbopen-user> (find-instances 'location ' (known-world)
                  ' (= x 20))
(#<location 5 (20 30)> #<location 3 (20 20)>)
gbbopen-user>

```

Find all `location` unit instances whose `x` and `y` coordinates are less than or equal to 25:

```

gbbopen-user> (find-instances 'location ' (known-world)
                  ' (<= (x y) (25 25)))
(#<location 4 (25 25)> #<location 3 (20 20)>)
gbbopen-user>

```

Find all `location` unit instances whose `x` coordinates are between 0 and 40 (inclusive) and whose `y` coordinates are between 60 and 100 (inclusive):

```

gbbopen-user> (find-instances 'location ' (known-world)
                  ' (within (x y) ((0 40) (60 100))))
(#<location 1 (40 60)>)
gbbopen-user>

```

Find all `location` unit instances whose coordinates are not within the above region:

```
gbbopen-user> (find-instances 'location '(known-world)
                '(not (within (x y) ((0 40) (60 100)))))
(#<location 5 (20 30)> #<location 4 (25 25)> #<location 3 (20 20)>
 #<location 2 (70 30)>)
gbbopen-user>
```

Step 13: Change a dimensional value

Recall that we assigned `location 1` to the global variable `ui`:

```
gbbopen-user> ui
#<location 1 (40 60)>
gbbopen-user>
```

and we can retrieve it by its `x` and `y` coordinates:

```
gbbopen-user> (find-instances 'location '(known-world)
                '(= (x y) (40 60)))
(#<location 1 (40 60)>)
gbbopen-user>
```

Let's change its `x` position to 80 and try retrieving it again:

```
gbbopen-user> (setf (x-of ui) 80)
80
gbbopen-user> (find-instances 'location '(known-world)
                '(= (x y) (40 60)))
nil
gbbopen-user>
```

It has moved on the `known-world`. As expected, the `location 1` unit instance is now at (80, 60):

```
gbbopen-user> (find-instances 'location '(known-world)
                '(= (x y) (80 60)))
(#<location 1 (80 60)>)
gbbopen-user>
```

Also note that the textual representation of `location 1` shows the new `x` slot value.

8 Using a Control Shell

A control shell is one of the three major components of a blackboard system (along with KSs and the blackboard). The control shell directs the problem-solving process by managing how KSs respond to contributions that are placed on the blackboard by an executing KS and to other events that may be triggered by the application or received from external sources. In this exercise we will use a control shell called the “Agenda Shell.”

GBBopen’s Agenda Shell is a generalization of the priority-based scheduling approach that was used in the original Hearsay-II blackboard architecture. The Agenda Shell manages KS definitions, and it initiates and terminates KS activities by:

- Triggering KSs in response to events
- Deciding which triggered KSs should be activated and their priority rating
- Maintaining a rating-based queue of pending KS activations (KSAs)
- Executing the top-rated KSAs, one at a time

The Agenda Shell is highly customizable and extensible, and it can be used as the foundation for implementing advanced control mechanisms. We will use only the most basic Agenda Shell capabilities in this Tutorial.

This exercise shows you how to:

- Load GBBopen’s Agenda Shell control shell
 - Start the Agenda Shell executing
 - Define a KS
 - Display control-shell activities (control-shell events)
 - Use control-shell stepping
-

Prerequisites

The `tutorial-example.lisp` file as modified thus far:

```
(in-package :gbbopen-user)

(define-unit-class location ()
  (x y)
  (:dimensional-values
   (x :point x)
   (y :point y))
  (:initial-space-instances (known-world)))

(defmethod print-instance-slots ((location location) stream)
  (call-next-method)
  (when (and (slot-boundp location 'x)
            (slot-boundp location 'y))
    (format stream " (~s ~s)"
            (x-of location)
            (y-of location))))
```

Step 1: Load the Agenda Shell

Start up a fresh Common Lisp session and load the `:agenda-shell-user` module, using the `:agenda-shell-user` REPL command:

```
cl-user> :agenda-shell-user
;; Loading <install-dir>/startup.lisp
...
;; Loading <install-dir>/<platform-dir>/gbbopen/
;;           control-shells/agenda-shell-user.fasl
gbbopen-user>
```

This loads everything that we've been loading with the `:gbbopen-user` module and, additionally, GBBopen's Agenda Shell control shell. As with the `:gbbopen-user` REPL command, the current package in the REPL is set to the `:gbbopen-user` package.

Step 2: Run the control shell

Now, start the Agenda Shell:

```
gbbopen-user> (start-control-shell)
;; Control shell 1 started
;; No executable KSAs remain, exiting control shell
;; Control shell 1 exited: 2 cycles completed
;; Run time: 0 seconds
;; Elapsed time: 0 seconds
:quiescence
gbbopen-user>
```

What just happened? The Agenda Shell began executing and looked for something to do. However, we have not yet defined any knowledge sources (KSs), so the Agenda Shell indicates that it did not find any executable KSAs in its initial KS-execution cycle. This situation is called *quiescence* and, by default, the Agenda Shell signals that quiescence has occurred and then continues for an additional KS-execution cycle in case any executable KSAs resulted from the quiescence signal. Again, no executable KSAs were found in cycle 2, so the Agenda Shell exits due to `:quiescence`.

Note that the Agenda Shell requires that the idle-loop process has been started on [CMUCL](#) and that multiprocessing has been started on [LispWorks](#). (An error message will instruct you on what to do if this is not the case.)

Step 3: Define a KS

So let's define a KS!

Edit your `tutorial-example.lisp` file and add the following function and KS definition to the end of the `tutorial-example.lisp` file:

```
(define-unit-class location ()
  (x y)
  (:dimensional-values
   (x :point x)
   (y :point y))
  (:initial-space-instances (known-world)))
```

```

(defmethod print-instance-slots ((location location) stream)
  (call-next-method)
  (when (and (slot-boundp location 'x)
             (slot-boundp location 'y))
    (format stream " (~s ~s)"
             (x-of location)
             (y-of location))))

;;; =====
;;; Startup KS

(defun startup-ks-function (ksa)
  (declare (ignore ksa))
  ;; Create an initial location unit instance at (0,0):
  (make-instance 'location :x 0 :y 0))

(define-ks startup-ks
  :trigger-events ((start-control-shell-event))
  :execution-function 'startup-ks-function)

```

The function `startup-ks-function` implements the KS. The Agenda Shell always calls a KS's execution function with a single argument, a `ksa` unit instance that represents the activation of the KS. For the present, we will ignore the `ksa` argument. Our simple `startup-ks-function` creates a location unit instance at the center of the known-world, coordinate (0,0).

The **define-ks** form defines a KS named `startup-ks` to the Agenda Shell. The `:trigger-events` value indicates that the KS should be triggered when an event called `start-control-shell-event` is signaled. The Agenda Shell signals `start-control-shell-event` once, when the control shell is started. The `:execution-function` names the function (that we just defined) that implements the KS.

Compile and load the entire `tutorial-example.lisp` file directly from the editor buffer (using `C-c C-k` in SLIME; `C-c C-b` in ELI).

The Agenda Shell creates a `ks` unit instance for each KS that we define:

```

gbbopen-user> (map-instances-of-class #'print 'ks)
#<ks startup-ks>
nil
gbbopen-user>

```

Note that the name of the `ks` unit instance is the name of the KS.

Step 4: Make the known-world

Before we can run our KS, we must make the `known-world` space instance:

```

gbbopen-user> (make-space-instance '(known-world)
                                   :dimensions '((x :ordered) (y :ordered)))
#<standard-space-instance (known-world)>
gbbopen-user>

```

Step 5: Start the control shell

Start the Agenda Shell again:

```
gbbopen-user> (start-control-shell)
;; Control shell 1 started
;; No executable KSAs remain, exiting control shell
;; Control shell 1 exited: 3 cycles completed
;; Run time: 0 seconds
;; Elapsed time: 0 seconds
:quiescence
gbbopen-user>
```

Did it work? Let's describe the blackboard repository:

```
gbbopen-user> :dsbb

Space Instance                Contents
-----
known-world                   1 instances (1 location)

Unit Class                    Instances
-----
control-shell                 1 *
ks                             1 +
ksa-queue                     2 +
location                      1
ordered-ksa-queue             1 +
standard-space-instance       1
-----
                                7 instances

gbbopen-user>
```

The initial `location` unit instance is there!

Note that some other unit instances have been created by the control shell. There is one `ks` unit instance, our `startup-ks` KS, as well as two `ksa-queue` and one `ordered-ksa-queue` unit instances. The `ordered-ksa-queue` is the rating-based queue of pending KSAs (no pending KSAs remain on it) and the two `ksa-queue` queues are the Agenda Shell's executed KSAs and obviated KSAs queues (both empty). We will discuss these queues in a later exercise.

The `ks`, `ksa-queue`, `ordered-ksa-queue` unit-instance counts are followed by plus signs (+). This indicates that these unit classes have been defined to be *retained*, meaning that their instances are not deleted by our call to **delete-blackboard-repository**. The plus sign indicates that the retention attribute will be propagated to all subclasses of those unit classes; retained, but not propagated, would be shown by an asterisk (*).

Let's verify this behavior by calling **delete-blackboard-repository** and then describe the blackboard repository:

```
gbbopen-user> (delete-blackboard-repository)
t
gbbopen-user> :dsbb
There are no space instances in the blackboard repository.
```

```
Unit Class                    Instances
```

```

-----
control-shell          1 *
ks                    1 +
ksa-queue             2 +
ordered-ksa-queue     1 +
gbbopen-user>

```

Step 6: Display control shell activities

With all but the retained unit instances deleted from the blackboard repository, let's rerun the control shell. However, this time we will ask GBBopen to display more of what the control shell is doing. First, enable display of all control-shell and instance-creation events by evaluating:

```

gbbopen-user> (enable-event-printing '(control-shell-event :plus-subevents))
nil
gbbopen-user> (enable-event-printing 'create-instance-event)
nil
gbbopen-user>

```

or the shorthand equivalent:

```

gbbopen-user> (enable-event-printing '(control-shell-event +))
nil
gbbopen-user> (enable-event-printing 'create-instance-event)
nil
gbbopen-user>

```

We will cover events, event printing, and event functions in greater detail in a later exercise.

Start the Agenda Shell once again:

```

gbbopen-user> (start-control-shell)
;; Control shell 1 started
=> Start-control-shell-event
=> Control-shell-cycle-event
    :cycle 1
=> Create-instance-event
    :instance #<ksa 1 startup-ks 1>
=> Ksa-activation-event
    :instance #<ksa 1 startup-ks 1>
    :cycle 1
=> Ksa-execution-event
    :instance #<ksa 1 startup-ks 1>
    :cycle 1
=> Create-instance-event
    :instance #<location 1 (0 0)>
=> Control-shell-cycle-event
    :cycle 2
=> Quiescence-event
=> Control-shell-cycle-event
    :cycle 3
;; No executable KSAs remain, exiting control shell
;; Control shell 1 exited: 3 cycles completed
;; Run time: 0 seconds

```

```
;; Elapsed time: 0 seconds
:quiescence
gbbopen-user>
```

Step 7: Control-shell stepping

Let's rerun the control shell once again, but this time we will enable control-shell stepping. Again, delete the blackboard repository:

```
gbbopen-user> (delete-blackboard-repository)
t
gbbopen-user>
```

and disable the event printing that we enabled in the last step:

```
gbbopen-user> (disable-event-printing)
nil
gbbopen-user>
```

Now start the Agenda Shell, this time with stepping enabled:

```
gbbopen-user> (start-control-shell :stepping 't)
;; Control shell 1 started
>> CS Step (cycle 1):
  About to process event #<start-control-shell-event>... [? entered]
Stepping commands (follow with <Return>):
  d      Disable this kind of stepping (:process-event)
  e      Enable another kind of stepping
  f      Evaluate a form
  h or ? Help (this text)
  q      Quit (disable all stepping and continue)
  s      Show enabled stepping kinds
  x      Exit control shell
  =      Describe the object of interest (bound to ==)
  +      Enable all stepping
  -      Disable all stepping
  <Space> Continue (resume processing)
>> CS Step (cycle 1):
  About to process event #<start-control-shell-event>... [<Return> entered]
>> CS Step (cycle 1):
  About to activate KS startup-ks on
    start-control-shell-event... [<Return> entered]
>> CS Step (cycle 1):
  About to execute KSA #<ksa 1 startup-ks 1>... [<Return> entered]
<< KSA 1 returned: (#<location 1 (0 0)>)
>> CS Step (cycle 2):
  About to signal quiescence... [<Return> entered]
>> CS Step (cycle 3):
  About to signal quiescence... [<Return> entered]
;; No executable KSAs remain, exiting control shell
;; Control shell 1 exited: 3 cycles completed
;; Run time: 0 seconds
;; Elapsed time: 1 minute, 6 seconds
```

```
:quiescence  
gbbopen-user>
```


9 Application Startup and Event Functions

In the last exercise, we needed to create the `known-world` space instance, with `x` and `y` dimensions, before we started the Agenda Shell. We also needed to call **`delete-blackboard-repository`** before calling **`start-control-shell`** to execute another run of our developing application. It is convenient to have these activities performed automatically whenever the control shell is started, so we'll do so in this exercise.

This exercise shows you how to:

- Define a function to perform all application-specific initialization and re-execution activities
 - Automatically invoke the initialization/re-execution function at control-shell startup using GBBopen's event-function capabilities
 - Restrict the classes of unit instances that can be stored on a space instance
 - Specify the dimensionality of a space instance relative to the dimensional specifications of a unit class
-

Prerequisites

- The `tutorial-example.lisp` file as modified thus far:

```
(in-package :gbbopen-user)

(define-unit-class location ()
  (x y)
  (:dimensional-values
   (x :point x)
   (y :point y))
  (:initial-space-instances (known-world)))

(defmethod print-instance-slots ((location location) stream)
  (call-next-method)
  (when (and (slot-boundp location 'x)
             (slot-boundp location 'y))
    (format stream " (~s ~s) "
             (x-of location)
             (y-of location))))

;;; =====
;;; Startup KS

(defun startup-ks-function (ksa)
  (declare (ignore ksa))
  ; Create an initial location unit instance at (0,0):
  (make-instance 'location :x 0 :y 0))

(define-ks startup-ks
  :trigger-events ((start-control-shell-event))
  :execution-function 'startup-ks-function)
```



```
gbbopen-user> (enable-event-printing '(control-shell-event +))
nil
gbbopen-user>
```

Add the following form at the end of your `tutorial-example.lisp` file:

```
(add-event-function 'initializations 'start-control-shell-event
                    ;; Initializations should be done first!
                    :priority 100)
```

(We'll place the **add-event-function** form immediately after the `initializations` function definition in our file, but this choice of location is purely a code organizational style preference—the form could be placed anywhere relative to the function definition.)

Step 3: Run the application

Start a fresh Common Lisp session, compile and load the `tutorial-example.lisp` file directly from the editor buffer (using `C-c C-k` in SLIME; `C-c C-b` in ELI) and start the Agenda Shell again:

```
gbbopen-user> (start-control-shell)
;; Control shell 1 started
;; No executable KSAs remain, exiting control shell
;; Control shell 1 exited: 3 cycles completed
;; Run time: 0 seconds
;; Elapsed time: 0 seconds
:quiescence
gbbopen-user>
```

Note that our developing application performs the same as it did in the last exercise, but now our `initializations` event function is taking care of all the details of starting up our application. We no longer have to remember to create the `known-world` space instance or to delete the blackboard repository before running the application another time.

Step 4: Run it Again

Let's verify that we can re-run our application. Without doing anything else, start the Agenda Shell again:

```
gbbopen-user> (start-control-shell)
;; Control shell 1 started
;; No executable KSAs remain, exiting control shell
;; Control shell 1 exited: 3 cycles completed
;; Run time: 0 seconds
;; Elapsed time: 0 seconds
:quiescence
gbbopen-user>
```

As before, our `initializations` event function took care of all the details of starting up our application.

Step 5: It's a new world...

GBBopen allows us to restrict the classes of unit instances that can be stored on a space instance. For example, we can limit the `known-world` to `location` unit instances by specifying an `:allowed-unit-classes` value to **make-space-instance**:

```
(defun initializations (event-name &key &allow-other-keys)
  (declare (ignore event-name))
  ;; Clean up any previous run:
  (delete-blackboard-repository)
  ;; Make a new known-world space instance:
  (make-space-instance
   '(known-world)
   :allowed-unit-classes '(location)
   :dimensions '((x :ordered) (y :ordered))))
```

Attempting to add any unit-instance that is not a `location` to `known-world` will now generate an error.

It is often convenient to specify the dimensions of a space-instance relative to those of one or more unit classes. Edit the definition of `initializations`, removing the `x` and `y` dimensions specification:

```
(defun initializations (event-name &key &allow-other-keys)
  (declare (ignore event-name))
  ;; Clean up any previous run:
  (delete-blackboard-repository)
  ;; Make a new known-world space instance:
  (make-space-instance
   '(known-world)
   :allowed-unit-classes '(location)
   :dimensions '((x :ordered) (y :ordered))))
```

and replacing it with a call of **dimensions-of** to obtain the dimensions associated with instances of the `location` unit class:

```
(defun initializations (event-name &key &allow-other-keys)
  (declare (ignore event-name))
  ;; Clean up any previous run:
  (delete-blackboard-repository)
  ;; Make a new known-world space instance:
  (make-space-instance
   '(known-world)
   :allowed-unit-classes '(location)
   :dimensions (dimensions-of 'location))
```

Step 6: Run the application again

Compile and load the `tutorial-example.lisp` file directly from the editor buffer (using `C-c C-k` in SLIME; `C-c C-b` in ELI) and start the Agenda Shell again:

```
gbbopen-user> (start-control-shell)
;; Control shell 1 started
;; No executable KSAs remain, exiting control shell
;; Control shell 1 exited: 3 cycles completed
```

```
;; Run time: 0 seconds
;; Elapsed time: 0 seconds
:quiescence
gbbopen-user>
```

Verify the dimensionality of the `known-world` space instance by evaluating:

```
gbbopen-user> (describe-space-instance '(known-world))
Standard-space-instance #<standard-space-instance (known-world)>
  Allowed unit classes: t
  Dimensions:
    (x :ordered)
    (y :ordered)
gbbopen-user>
```


10 Add Another KS

The last exercise made it easy to initialize and run our application repeatedly by simply starting the Agenda Shell. We also specified the dimensionality of our `known-world` space instance relative to the dimensional specifications of the `location` unit class. With these niceties in place, its time to move beyond our initial `location` unit instance.

This exercise shows you how to:

- Add an additional dimension to a unit class
 - Define a KS that obtains its execution-context information from its triggering unit instance
 - Extend the random-walk application to do some walking
 - Explore the resulting random walk
-

Prerequisites

- The `tutorial-example.lisp` file as modified thus far:

```
(in-package :gbbopen-user)

(define-unit-class location ()
  (x y)
  (:dimensional-values
   (x :point x)
   (y :point y))
  (:initial-space-instances (known-world)))

(defmethod print-instance-slots ((location location) stream)
  (call-next-method)
  (when (and (slot-boundp location 'x)
            (slot-boundp location 'y))
    (format stream " (~s ~s)"
            (x-of location)
            (y-of location))))

;;; =====
;;; Startup KS

(defun startup-ks-function (ksa)
  (declare (ignore ksa))
  ;; Create an initial location unit instance at (0,0):
  (make-instance 'location :x 0 :y 0))

(define-ks startup-ks
  :trigger-events ((start-control-shell-event))
  :execution-function 'startup-ks-function)

;;; =====
;;; Initializations (run at Agenda Shell startup)
```

```
(defun initializations (event-name &key &allow-other-keys)
  (declare (ignore event-name))
  ;; Clean up any previous run:
  (delete-blackboard-repository)
  ;; Make a new known-world space instance:
  (make-space-instance
   '(known-world)
   :dimensions (dimensions-of 'location))

  (add-event-function 'initializations 'start-control-shell-event
    ;; Initializations should be done first!
    :priority 100))
```

- The `:agenda-shell-user` module is loaded

Step 1: Add another dimension

It's time we introduce the notion of time to our application. Edit the `location` unit-class definition in `tutorial-example.lisp`, adding a new slot, `time`, to the `location` unit class definition and a corresponding `time` dimensional value:

```
(define-unit-class location ()
  (time
   x y)
  (:dimensional-values
   (time :point time)
   (x :point x)
   (y :point y))
  (:initial-space-instances (known-world)))
```

Recall that we specified that the dimensions of the `known-world` space instance that is created by our `initializations` function relative to the dimensions of the `location` unit class:

```
(make-space-instance
 '(known-world)
 :dimensions (dimensions-of 'location))
```

Therefore, we don't need to modify our call to `make-space-instance` in order to add `time` as a dimension of `known-world`.

Next, modify `startup-ks-function` in `tutorial-example.lisp` so that it creates the initial `location` unit instance at time 0:

```
(defun startup-ks-function (ksa)
  (declare (ignore ksa))
  ;; Create an initial location unit instance at (0,0) at time 0:
  (make-instance 'location :time 0 :x 0 :y 0))
```

Step 2: A test of time

Let's verify our work. Compile and load the `tutorial-example.lisp` file directly from the editor buffer (using `C-c C-k` in `SLIME`; `C-c C-b` in `ELI`) and start the Agenda Shell:

```

gbbopen-user> (start-control-shell)
;; Control shell 1 started
;; No executable KSAs remain, exiting control shell
;; Control shell 1 exited: 3 cycles completed
;; Run time: 0 seconds
;; Elapsed time: 0 seconds
:quiescence
gbbopen-user>

```

Check that the initial `location` unit instance is at time zero:

```

gbbopen-user> (describe-instance (find-instance-by-name 1 'location))
Location #<location 1 (0 0)>
  Instance name: 1
  Space instances: ((known-world))
  Dimensional values:
    time: 0
    x: 40
    y: 60
  Non-link slots:
    time: 0
    x: 40
    y : 60
  Link slots: None
gbbopen-user>

```

Step 3: Define another KS

Define a KS called `random-walk-ks` that:

- Is triggered when a `location` unit instance is created
- Has a constant KSA rating of 100
- Has an execution function called `random-walk-ks-function`, which:
 - Checks if we've already walked for 75 locations and prints a message if we have.
 - Otherwise:
 - * Determines a random location for which the `x` and `y` values are within 10 of the `x` and `y` values of the triggering unit instance (that is, the `location` instance whose creation triggered the `random-walk-ks` KS)
 - * If both of the `x` and `y` values for the new random location are between -50 and 50, creates a `location` unit instance at the random location; otherwise, prints a message indicating that we've walked off the world

Step 3a: Define a utility function

Begin implementing the `random-walk-ks` by adding the following utility function to the end of your `tutorial-example.lisp` file:

```

;;; =====
;;; Random-walk KS

(defun add-linear-variance (value max-variance)

```

```

;;; Returns a new random value in the interval
;;; [(- value max-variance), (+ value max-variance)]
(+ value (- (random (1+ (* max-variance 2))) max-variance))

```

Then compile the definition (using C-c C-c in SLIME or C-c C-x in ELI) and evaluate the following test in the REPL:

```

gbbopen-user> (dotimes (i 15) (printv (add-linear-variance 0 10)))
;; (add-linear-variance 0 10) => 8
;; (add-linear-variance 0 10) => 9
;; (add-linear-variance 0 10) => 4
;; (add-linear-variance 0 10) => 3
;; (add-linear-variance 0 10) => -4
;; (add-linear-variance 0 10) => -10
;; (add-linear-variance 0 10) => -1
;; (add-linear-variance 0 10) => 0
;; (add-linear-variance 0 10) => 4
;; (add-linear-variance 0 10) => 5
;; (add-linear-variance 0 10) => 8
;; (add-linear-variance 0 10) => -5
;; (add-linear-variance 0 10) => -3
;; (add-linear-variance 0 10) => 7
;; (add-linear-variance 0 10) => 6
nil
gbbopen-user>

```

Because `add-linear-variance` is stochastic, your results will be similar but not identical. Note that we used GBBopen's **printv** macro to display the result of each generated value. **Printv** can greatly assist debugging by printing forms and the results of evaluating them. **Printv** can be transparently wrapped around any form in a complex function definition, as it evaluates and displays all the forms in its body and returns the values resulting from evaluating the last form:

```

gbbopen-user> (printv "Some multiple values" (values 1 2) "Some more"
(values 3 4 5))
;; Some multiple values
;; (values 1 2) => 1; 2
;; Some more
;; (values 3 4 5) => 3; 4; 5
4
5
6
gbbopen-user>

```

Step 3b: Define the `random-walk-ks` execution function

Next add the following KS-execution function to the end of your `tutorial-example.lisp` file:

```

(defun random-walk-ks-function (ksa)
  ;;; Move to the next (random) location in the world
  (let* ((trigger-instance (sole-trigger-instance-of ksa))
         ;;; The new time is one greater than the stimulus's time:
         (time (1+ (time-of trigger-instance))))
    (cond

```

```

;; If the maximum time value (75) is reached, tell the user we've
;; walked too long:
(>= time 75) (format t "~2&Walked too long.~%" )
(t ;; The new location is +/- 10 of the stimulus's location:
  (let ((x (add-linear-variance (x-of trigger-instance) 10))
        (y (add-linear-variance (y-of trigger-instance) 10)))
    (cond
     ;; Check that the new location is within the known-world
     ;; boundaries. If so, create the new location instance:
     ((and (<= -50 x 50) (<= -50 y 50))
      (make-instance 'location
                    :time time
                    :x x
                    :y y))
     ;; Otherwise, tell the user that we've walked too far away:
     (t (format t "~2&Walked off the world: (~d, ~d).~%" x y))))))

```

Unlike the KS-execution functions that we have defined previously, `random-walk-ks-function` does not ignore its `ksa` argument. Instead, it calls **sole-trigger-instance-of** with the `ksa` unit-instance argument in order to obtain the `location` unit instance whose creation triggered the KSA. This pattern of obtaining the unit instance that triggered a KSA and then using that triggering unit instance as the context for the KS execution is typical of many KSs.

Step 3c: Add the `random-walk-ks` definition

Finally, add this `define-ks` form to the end of your `tutorial-example.lisp` file to complete the `random-walk-ks` definition:

```

(define-ks random-walk-ks
  :trigger-events ((create-instance-event location))
  :rating 100
  :execution-function 'random-walk-ks-function)

```

Step 4: Run the application

Compile and load the `random-walk-ks` forms, and then start the Agenda Shell:

```

gbbopen-user> (start-control-shell)
;; Control shell 1 started

Walked off the world: (23, 55).
;; No executable KSAs remain, exiting control shell
;; Control shell 1 exited: 64 cycles completed
;; Run time: 0.01 seconds
;; Elapsed time: 0 seconds
:quiescence
gbbopen-user>

```

It looks like something happened! (Again, because `add-linear-variance` is stochastic, your results will be similar but not identical.) Let's look at the blackboard repository and see how many `location` unit instances were created:

```

gbbopen-user> :dsbb

Space Instance          Contents
-----
known-world            61 instances (61 location)

Unit Class             Instances
-----
control-shell         1 *
ks                    1 +
ksa-queue             2 +
location              61
ordered-ksa-queue    1 +
standard-space-instance 1
-----
                        67 instances

gbbopen-user>

```

The 61 location instances makes sense. Previously, it required 3 control-shell cycles to create the initial location unit instance (one to execute the initial-ks KSA followed by two additional cycles of quiescence before the Agenda Shell exits). We now create one additional location unit instance with every execution of random-walk-ks, so we always create 3 fewer location instances than the total number of control-shell cycles.

Step 5: Where have we been?

It would be interesting to see where our random walk has taken us. We could use GBBopen's **map-instances-of-class** iterator to print each of the location unit instances:

```

gbbopen-user> (map-instances-of-class #'print 'location)

#<location 58 (5 31)>
#<location 13 (-7 10)>
#<location 26 (-40 35)>
#<location 39 (-4 3)>
#<location 52 (2 23)>
#<location 7 (3 17)>
#<location 20 (2 27)>
#<location 33 (-25 6)>
#<location 46 (-2 32)>
...
#<location 31 (-22 18)>
#<location 44 (-7 14)>
#<location 57 (2 41)>
#<location 12 (-15 15)>
#<location 25 (-32 38)>
#<location 38 (-10 -4)>
#<location 51 (-2 16)>
#<location 6 (10 27)>
#<location 19 (-1 17)>
#<location 32 (-25 12)>
#<location 45 (-7 23)>

```

```
nil
gbbopen-user>
```

Unfortunately, the order that unit instances are supplied to the `print` function is not controllable. Our walk would be much clearer if we printed the `location` unit instances in time order.

We might consider taking advantage of the instance names that GBBopen assigns to unit instances. We could do something like the following:

```
gbbopen-user> (dotimes (i 76)
                (let ((location (find-instance-by-name i 'location)))
                    (when location
                      (print location))))

#<location 1 (0 0)>
#<location 2 (10 4)>
#<location 3 (19 10)>
#<location 4 (14 9)>
#<location 5 (14 18)>
#<location 6 (10 27)>
#<location 7 (3 17)>
#<location 8 (-6 20)>
#<location 9 (4 15)>
#<location 10 (-5 14)>
...
#<location 50 (5 26)>
#<location 51 (-2 16)>
#<location 52 (2 23)>
#<location 53 (9 33)>
#<location 54 (7 43)>
#<location 55 (-2 36)>
#<location 56 (0 46)>
#<location 57 (2 41)>
#<location 58 (5 31)>
#<location 59 (13 39)>
#<location 60 (17 41)>
#<location 61 (21 50)>
nil
gbbopen-user>
```

This is a bad idea for several reasons. First, we are looking up every `location` unit instance by its instance name, which is less efficient than operating on `location` instances directly. While this isn't an significant issue in expressions that we evaluate in the REPL to investigate our application, we should seek to avoid such inefficiencies in application code. More importantly, however, the `location` instance name just happens to mirror the sequencing that we really want to display: the `time` value of the locations. We should find a way to sequence `location` printing that relies on the `time` values directly.

GBBopen provides a variant of **map-instances-of-class**, called **map-sorted-instances-of-class**, that sorts the unit instances based on a comparison predicate and an optional `:key` accessor function that suits our needs:

```
gbbopen-user> (map-sorted-instances-of-class #'print 'location #'<
              :key #'time-of)
```

```

#<location 1 (0 0)>
#<location 2 (10 4)>
#<location 3 (19 10)>
#<location 4 (14 9)>
#<location 5 (14 18)>
#<location 6 (10 27)>
#<location 7 (3 17)>
#<location 8 (-6 20)>
#<location 9 (4 15)>
#<location 10 (-5 14)>
...
#<location 50 (5 26)>
#<location 51 (-2 16)>
#<location 52 (2 23)>
#<location 53 (9 33)>
#<location 54 (7 43)>
#<location 55 (-2 36)>
#<location 56 (0 46)>
#<location 57 (2 41)>
#<location 58 (5 31)>
#<location 59 (13 39)>
#<location 60 (17 41)>
#<location 61 (21 50)>
nil
gbbopen-user>

```

Using **map-sorted-instances-of-class** involves a sorting operation, so this approach still has some efficiency concerns for use in application code. However, it suits our REPL-exploration needs just fine. (There is a **do-sorted-instances-of-class** macro, if an iterative style is preferred over a mapper.) We will explore a more efficient approach to displaying the random walk in the next exercise.

Step 6: Run the application a few more times

If we run the application a few more times, we eventually encounter a case where we create the allotted 75 location unit instances without walking off the known-world:

```

gbbopen-user> (start-control-shell)
;; Control shell 1 started

Walked too long.
;; No executable KSAs remain, exiting control shell
;; Control shell 1 exited: 78 cycles completed
;; Run time: 0.04 seconds
;; Elapsed time: 0 seconds
:quiescence
gbbopen-user>

```

Here is one such random walk:

```
gbbopen-user> (map-sorted-instances-of-class #'print 'location #'<
                :key #'time-of)

#<location 1 (0 0)>
#<location 2 (2 7)>
#<location 3 (-1 5)>
#<location 4 (-1 0)>
#<location 5 (3 -2)>
#<location 6 (13 -7)>
#<location 7 (8 -5)>
#<location 8 (1 2)>
#<location 9 (8 0)>
#<location 10 (5 8)>
...
#<location 70 (-13 -11)>
#<location 71 (-13 -6)>
#<location 72 (-9 -6)>
#<location 73 (1 -4)>
#<location 74 (-8 -11)>
#<location 75 (-13 -15)>
nil
gbbopen-user>
```


11 Making Connections

We finally did some walking in the last exercise and learned how to display the `location` unit instances in our walk from the REPL. In this exercise, we learn how to use GBBopen's link capabilities to represent relationships among unit instances. Links are an important aspect of almost every GBBopen application, so it's time that we started taking advantage of them.

This exercise shows you how to:

- Add link slots to a unit class
 - Use link slots to traverse and display the resulting random walk
-

Prerequisites

- The `tutorial-example.lisp` file as modified thus far:

```
(in-package :gbbopen-user)

(define-unit-class location ()
  (time
   x y)
  (:dimensional-values
   (time :point time)
   (x :point x)
   (y :point y))
  (:initial-space-instances (known-world)))

(defmethod print-instance-slots ((location location) stream)
  (call-next-method)
  (when (and (slot-boundp location 'x)
             (slot-boundp location 'y))
    (format stream " (~s ~s)"
             (x-of location)
             (y-of location))))

;;; =====
;;;   Startup KS

(defun startup-ks-function (ksa)
  (declare (ignore ksa))
  ;; Create an initial location unit instance at (0,0):
  (make-instance 'location :time 0 :x 0 :y 0))

(define-ks startup-ks
  :trigger-events ((start-control-shell-event))
  :execution-function 'startup-ks-function)

;;; =====
;;;   Initializations (run at Agenda Shell startup)
```

```

(defun initializations (event-name &key &allow-other-keys)
  (declare (ignore event-name))
  ;; Clean up any previous run:
  (delete-blackboard-repository)
  ;; Make a new known-world space instance:
  (make-space-instance
   '(known-world)
   :dimensions (dimensions-of 'location)))

(add-event-function 'initializations 'start-control-shell-event
  ;; Initializations should be done first!
  :priority 100)

;;; =====
;;; Random-walk KS

(defun add-linear-variance (value max-variance)
  ;; Returns a new random value in the interval
  ;; [(- value max-variance), (+ value max-variance)]
  (+ value (- (random (1+ (* max-variance 2))) max-variance)))

(defun random-walk-ks-function (ksa)
  ;; Move to the next (random) location in the world
  (let* ((trigger-instance (sole-trigger-instance-of ksa))
        ;; The new time is one greater than the stimulus's time:
        (time (1+ (time-of trigger-instance))))
    (cond
     ;; If the maximum time value (75) is reached, tell the user we've
     ;; walked too long:
     ((>= time 75) (format t "~2&Walked too long.~%" t))
     (t ;; The new location is +/- 10 of the stimulus's location:
        (let ((x (add-linear-variance (x-of trigger-instance) 10))
              (y (add-linear-variance (y-of trigger-instance) 10)))
          (cond
           ;; Check that the new location is within the known-world
           ;; boundaries. If so, create the new location instance:
           ((and (<= -50 x 50) (<= -50 y 50))
            (make-instance 'location
                          :time time
                          :x x
                          :y y))
           ;; Otherwise, tell the user that we've walked too far away:
           (t (format t "~2&Walked off the world: (~d, ~d).~%" x y)))))))

(define-ks random-walk-ks
  :trigger-events ((create-instance-event location))
  :rating 100
  :execution-function 'random-walk-ks-function)

```

- The `:agenda-shell-user` module is loaded

Step 1: Add a link

In the last exercise, we used **map-sorted-instances-of-class** to display the random walk. Another way that we could represent the walk is by connect each newly created `location` unit instance to the `location` unit instance that preceded it in the walk. We'll use GBBopen's link capabilities to do this.

A *link* is a bidirectional relationship between two unit instances that is implemented by two pointers. From the perspective of a particular unit instance, each link consists of an outgoing, or *direct*, pointer to another unit instance and an incoming, or *inverse*, pointer that is stored in unit instance pointed to by the direct pointer. GBBopen automatically maintains the bidirectional-link consistency of these pointers when creating new links, deleting existing links, or deleting unit instances. Links remove the possibility of “one-sided” relationships or “dangling” pointers to deleted unit instances.

Edit the `location` unit-class definition in your `tutorial-example.lisp` file, adding two link slots, `next-location` and `previous-location`, to the `location` unit class definition:

```
(define-unit-class location ()
  (time
   x y
   (next-location
    :link (location previous-location :singular t)
    :singular t)
   (previous-location
    :link (location next-location :singular t)
    :singular t))
  (:dimensional-values
   (time :point time)
   (x :point x)
   (y :point y))
  (:initial-space-instances (known-world)))
```

Each link-slot specification is a list whose first element is the name of the link slot. This is followed by the link slot option `:link` and a concise specification of the inverse link slot associated with that link slot. In this case, the `next-location/previous-location` link is between instances of the same (`location`) unit class, but often links are between instances of different unit classes.

Links can be many-to-many, many-to-one, one-to-many, or one-to-one. In this case, the `next-location/previous-location` link is one-to-one, which is specified by including the `:singular t` slot option in the link-slot definition (and the corresponding `:singular t` specification in the concise inverse-link-slot specification). To help clarify the specification of link slot arity, let's temporarily assume that we want a `location` instance that can have many next locations, but only a single previous location. This link relation would be specified as follows:

```
...
(next-locations
 :link (location previous-location :singular t))
(previous-location
 :link (location next-locations)
 :singular t)
...
```

We've followed the natural GBBopen convention of giving singular link slots a singular name (such as `previous-location`) and link slots that can contain multiple links a plural name (such as `next-locations`). Note that the `:singular` option is associated with the `previous-location` link slot as both a slot option in the `previous-location` link-slot definition and in the concise inverse-link-slot specification for `previous-location` in the `next-locations` link-slot definition.

Step 2: Break some links

The concise inverse-link-slot specification supplied by the `:link` slot option provides a “double entry” redundancy that is useful when links are between instances of different unit classes, as the link can be understood by viewing either class definition. The redundancy also helps GBBopen recognize inconsistencies in link specifications. The function **check-link-definitions** asks GBBopen to validate that all link definitions are consistent. Let’s try it on our current random-walk application. Compile and load the latest changes in your `tutorial-example.lisp` file (including the new `next-location` and `previous-location` link slots). Then check link consistency:

```
gbbopen-user> (check-link-definitions)
;; All link definitions are consistent.
t
gbbopen-user>
```

GBBopen reports that all link definitions are consistent.

Suppose that we had forgotten to add the `previous-location` end of the link in our `location` unit-class definition. Edit the `location` unit-class definition in your `tutorial-example.lisp` file, adding the line `#+ignore` immediately before the `previous-location` link-slot definition:

```
(define-unit-class location ()
  (time
   x y
   (next-location
    :link (location previous-location :singular t)
    :singular t)
   #+ignore
   (previous-location
    :link (location next-location :singular t)
    :singular t))
  (:dimensional-values
   (time :point time)
   (x :point x)
   (y :point y))
  (:initial-space-instances (known-world)))
```

The `#+ignore` read-time conditionalization tells Common Lisp to skip over the next form if `ignore` is not an element of the feature list `*features*`. By convention, `ignore` is never added to `*features*`, so `nobr#+ignore` is a handy mechanism for temporarily “commenting out” a single form.

Compile the now-defective definition (using `C-c C-c` in SLIME or `C-c C-x` in ELI) and then recheck link consistency:

```
gbbopen-user> (check-link-definitions)
Warning: The inverse of link slot next-location in unit class location
        refers to link slot previous-location which is not present in
        unit class location.
nil
gbbopen-user>
```

As expected, GBBopen alerts us to the problem.

Remove the `#+ignore` that we just added and comment out the `:singular t` portion of the inverse link-slot specification in `next-location`:

```
(define-unit-class location ()
  (time
   x y
   (next-location
    :link (location previous-location) ; :singular t)
    :singular t)
   #+ignore
   (previous-location
    :link (location next-location :singular t)
    :singular t))
  (:dimensional-values
   (time :point time)
   (x :point x)
   (y :point y))
  (:initial-space-instances (known-world)))
```

Compile the again-defective definition (using C-c C-c in SLIME or C-c C-x in ELI) and then recheck link consistency:

```
gbbopen-user> (check-link-definitions)
Warning: Link slot next-location in unit class location incorrectly
         declares its inverse link slot previous-location in unit
         class location as not singular.

nil
gbbopen-user>
```

Once again, GBBopen has alerted us to the problem.

Restore the `:singular t` portion of the inverse link-slot specification in `next-location` that we just commented out:

```
(define-unit-class location ()
  (time
   x y
   (next-location
    :link (location previous-location) ; :singular t)
    :singular t)
   (previous-location
    :link (location next-location :singular t)
    :singular t))
  (:dimensional-values
   (time :point time)
   (x :point x)
   (y :point y))
  (:initial-space-instances (known-world)))
```

Then recompile and recheck link consistency:

```
gbbopen-user> (check-link-definitions)
;; All link definitions are consistent.
t
gbbopen-user>
```

Step 3: Create some links

Let's use our newly defined `next-location/previous-location` link to connect our location unit instances. Edit the `random-walk-ks-function` definition in your `tutorial-example.lisp` file, adding the trigger instance as a new `:previous-location` argument to **make-instance**:

```
(defun random-walk-ks-function (ksa)
  ;;; Move to the next (random) location in the world
  (let* ((trigger-instance (sole-trigger-instance-of ksa))
        ;; The new time is one greater than the stimulus's time:
        (time (1+ (time-of trigger-instance))))
    (cond
     ;; If the maximum time value (75) is reached, tell the user we've
     ;; walked too long:
     ((>= time 75) (format t "~2&Walked too long.~%"))
     (t ;; The new location is +/- 10 of the stimulus's location:
        (let ((x (add-linear-variance (x-of trigger-instance) 10))
              (y (add-linear-variance (y-of trigger-instance) 10)))
          (cond
           ;; Check that the new location is within the known-world
           ;; boundaries. If so, create the new location instance:
           ((and (<= -50 x 50) (<= -50 y 50))
            (make-instance 'location
                          :time time
                          :x x
                          :y y
                          :previous-location trigger-instance))
           ;; Otherwise, tell the user that we've walked too far away:
           (t (format t "~2&Walked off the world: (~d, ~d).~%" x y))))))))))
```

Compile the `random-walk-ks-function` (using `C-c C-c` in SLIME or `C-c C-x` in ELI) and then run the application:

```
gbbopen-user> (start-control-shell)
;; Control shell 1 started

Walked off the world: (55, 35).
;; No executable KSAs remain, exiting control shell
;; Control shell 1 exited: 66 cycles completed
;; Run time: 0.01 seconds
;; Elapsed time: 0 seconds
:quiescence
gbbopen-user>
```

Let's describe a couple of location unit instances to check our work. First, the initial location unit instance:

```
gbbopen-user> (describe-instance (find-instance-by-name 1 'location))
Location #<location 1 (0 0)>
Instance name: 1
Space instances: ((known-world))
Dimensional values:
  time: 0
  x: 40
```

```

    y: 60
  Non-link slots:
    time: 0
    x: 40
    y: 60
  Link slots:
    next-location: #<location 2 (-10 10)>
    previous-location: nil
gbbopen-user>

```

Note that the next-location link slot points to the next location unit instance in our random walk. Let's describe that unit instance:

```

gbbopen-user> (describe-instance (find-instance-by-name 2 'location))
Location #<location 2 (-10 10)>
  Instance name: 2
  Space instances: ((known-world))
  Dimensional values:
    time: 1
    x: -10
    y: 10
  Non-link slots:
    time: 1
    x: -10
    y: 10
  Link slots:
    next-location: #<location 3 (-6 19)>
    previous-location: #<location 1 (0 0)>
gbbopen-user>

```

The next-location link slot in location 2 points to the third location unit instance in our random walk and its previous-location link slot points back to the initial location unit instance.

We can now follow the links to display the random walk:

```

gbbopen-user> (loop with location = (find-instance-by-name 1 'location)
  do (print location)
  while (setf location (next-location-of location)))

#<location 1 (0 0)>
#<location 2 (-10 10)>
#<location 3 (-6 19)>
#<location 4 (0 14)>
#<location 5 (-1 14)>
#<location 6 (8 10)>
#<location 7 (17 3)>
#<location 8 (7 -6)>
#<location 9 (10 4)>
#<location 10 (5 -5)>
...
#<location 60 (29 17)>
#<location 61 (31 21)>
#<location 62 (40 23)>

```

```
#<location 63 (45 28)>
nil
gbbopen-user>
```

Step 4: Define a “print walk” KS

Let’s add a new KS, `print-walk-ks`, that displays the random walk once it is completed. Add the following KS to the end of your `tutorial-example.lisp` file:

```
;;; =====
;;;   Print-walk KS

(defun print-walk-ks-function (ksa)
  ;;; Starting with the initial location instance, print the instance
  ;;; name and location of the walk
  (declare (ignore ksa))
  (format t "~2&The random walk:~%" )
  (let ((instance (find-instance-by-name 1 'location)))
    (while instance
      (format t "~s (~s ~s)~%"
              (instance-name-of instance)
              (x-of instance)
              (y-of instance))
      (setf instance (next-location-of instance))))
  ;; Tell the Agenda Shell to exit:
  ':stop)

(define-ks print-walk-ks
  :trigger-events ((quiescence-event))
  :rating 100
  :execution-function 'print-walk-ks-function)
```

The `print-walk-ks` is triggered by a `quiescence-event`. Recall that the Agenda Shell signals that quiescence has occurred when no executable KSAs are available to be executed and then it continues for an additional KS-execution cycle in case any executable KSAs resulted from the quiescence event. So, `print-walk-ks` will be triggered once no `random-walk-ks` KSAs are triggered by newly created `location` unit instances.

The `print-walk-ks-function` follows the `next-location/previous-location` link to display the walk. More importantly, the function returns the keyword symbol `:stop`. The Agenda Shell checks the value returned by a KS execution function for this special indicator and, if it is returned, the control shell is exited. If we did not return `:stop`, the `print-walk-ks` KS would be triggered and activated on the first `quiescence-event`, the KSA would execute, then the Agenda Shell would detect another quiescence condition, signal a new `quiescence-event`, and our application would print the random walk over and over again.

Let’s compile our latest changes and then run our application with the new `print-walk-ks` KS in place:

```
gbbopen-user> (start-control-shell)
;; Control shell 1 started

Walked off the world: (54, 15).
```

```
The random walk:
1 (0 0)
2 (-6 9)
3 (-14 8)
4 (-5 6)
5 (-13 5)
6 (-11 13)
7 (-11 4)
8 (-17 8)
9 (-21 15)
10 (-12 14)
    ...
35 (40 28)
36 (50 22)
37 (49 12)
38 (47 10)
;; Explicit :stop issued by KS print-walk-ks
;; Control shell 1 exited: 41 cycles completed
;; Run time: 0.01 seconds
;; Elapsed time: 0 seconds
:stop
gbbopen-user>
```


12 Creating a GBBopen Application

GBBopen's Module Manager Facility provides mechanisms that make it easy to define and use your own GBBopen applications.

This exercise shows you how to:

- Structure an application using the Module Manager Facility
 - Define a REPL command for your application
 - Compile and load your application using your REPL command
 - Create and use an application-specific package
 - Add an “autorun” action
-

Prerequisites

- The `tutorial-example.lisp` file as modified thus far:

```
(in-package :gbbopen-user)

(define-unit-class location ()
  (time
   x y
   (next-location
    :link (location previous-location :singular t)
    :singular t)
   (previous-location
    :link (location next-location :singular t)
    :singular t))
  (:dimensional-values
   (time :point time)
   (x :point x)
   (y :point y))
  (:initial-space-instances (known-world)))

(defmethod print-instance-slots ((location location) stream)
  (call-next-method)
  (when (and (slot-boundp location 'x)
            (slot-boundp location 'y))
    (format stream " (~s ~s)"
            (x-of location)
            (y-of location))))

;;; =====
;;; Startup KS

(defun startup-ks-function (ksa)
  (declare (ignore ksa))
  ;; Create an initial location unit instance at (0,0):
  (make-instance 'location :time 0 :x 0 :y 0))
```

```

(define-ks startup-ks
  :trigger-events ((start-control-shell-event))
  :execution-function 'startup-ks-function)

;;; =====
;;;   Initializations (run at Agenda Shell startup)

(defun initializations (event-name &key &allow-other-keys)
  (declare (ignore event-name))
  ;; Clean up any previous run:
  (delete-blackboard-repository)
  ;; Make a new known-world space instance:
  (make-space-instance
   '(known-world)
   :dimensions (dimensions-of 'location)))

(add-event-function 'initializations 'start-control-shell-event
  ;; Initializations should be done first!
  :priority 100)

;;; =====
;;;   Random-walk KS

(defun add-linear-variance (value max-variance)
  ;; Returns a new random value in the interval
  ;; [(- value max-variance), (+ value max-variance)]
  (+ value (- (random (1+ (* max-variance 2))) max-variance)))

(defun random-walk-ks-function (ksa)
  ;; Move to the next (random) location in the world
  (let* ((trigger-instance (sole-trigger-instance-of ksa))
        ;; The new time is one greater than the stimulus's time:
        (time (1+ (time-of trigger-instance))))
    (cond
     ;; If the maximum time value (75) is reached, tell the user we've
     ;; walked too long:
     ((>= time 75) (format t "~2&Walked too long.~%" ))
     (t ;; The new location is +/- 10 of the stimulus's location:
        (let ((x (add-linear-variance (x-of trigger-instance) 10))
              (y (add-linear-variance (y-of trigger-instance) 10)))
          (cond
           ;; Check that the new location is within the known-world
           ;; boundaries.  If so, create the new location instance:
           ((and (<= -50 x 50) (<= -50 y 50))
            (make-instance 'location
                          :time time
                          :x x
                          :y y
                          :previous-location trigger-instance))
           ;; Otherwise, tell the user that we've walked too far away:
           (t (format t "~2&Walked off the world: (~d, ~d).~%" x y)))))))

```

```

(define-ks random-walk-ks
  :trigger-events ((create-instance-event location))
  :rating 100
  :execution-function 'random-walk-ks-function)

;;; =====
;;; Print-walk KS

(defun print-walk-ks-function (ksa)
  ;;; Starting with the initial location instance, print the instance
  ;;; name and location of the walk
  (declare (ignore ksa))
  (format t "~2&The random walk:~%"
    (let ((instance (find-instance-by-name 1 'location)))
      (while instance
        (format t "~s (~s ~s)~%"
          (instance-name-of instance)
          (x-of instance)
          (y-of instance))
        (setf instance (next-location-of instance))))))
  ;; Tell the Agenda Shell to exit:
  ':stop)

(define-ks print-walk-ks
  :trigger-events ((quiescence-event))
  :rating 100
  :execution-function 'print-walk-ks-function)

```

- The GBBopen environment setup using `<install-dir>/initiate.lisp` as described in Steps 1 and 2 of the Enhancing Your Development Environment exercise (see page 29)

Step 1: Create your personal gbbopen-modules directory

Create a directory named `gbbopen-modules` in your “homedir” directory (see page 33). For example:

```

[~]$ mkdir gbbopen-modules
[~]$

```

This is a special directory that is read by used by GBBopen to find applications when GBBopen is started using `<install-dir>/initiate.lisp`, as described in Steps 1 and 2 of the Enhancing Your Development Environment exercise (see page 29).

Step 2: Create a module-definition file for the random-walk application

Recall that you created a directory to hold the random-walk application in Step 1 of Working Within a File exercise (see page 35). I used these shell commands to create my directories:

```

[~]$ mkdir tutorial
[~]$ cd tutorial
[~/tutorial]$ mkdir source
[~/tutorial]$

```

Then you created the `tutorial-example.lisp` file in this source subdirectory. We said that we would explain why we created the `source` directory in a later exercise. Well, later has arrived.

Each GBBopen application is packaged in a directory that contains:

- a `modules.lisp` file that contains module definitions (loaded after the personal `codegbbopen-init.lisp` file if there is one in the user's "homedir")
- a directory named `source` containing all the source files for the module or application
- an optional `commands.lisp` file that specifies REPL commands for the module (loaded after the personal `gbbopen-commands.lisp` file if there is one in the user's "homedir")

You already have the `source` directory and the `tutorial-example.lisp` source file. Next, we create the `modules.lisp` file for the application. (We will create a `commands.lisp` file for the random-walk application in Step 5.)

Use your Common Lisp editor to create a new file named `modules.lisp` in the `tutorial` directory (just as you created the `tutorial-example.lisp` file in Step 2 of Working Within a File exercise (see page 35)). Note that this file is *not* in the `source` subdirectory, but in the `tutorial` directory that contains the `source` subdirectory.

Type the following two forms into the new `modules.lisp` file:

```
(in-package :module-manager-user)

(define-module :tutorial
  (:requires :agenda-shell-user)
  (:files "tutorial-example"))
```

and then save the file.

Recall that the `in-package` form specifies the Common Lisp package that is made current when the file is compiled or loaded. A `modules.lisp` file should always specify the `:module-manager-user` package as the first form in the file.

The second form defines our application module, which we will name `:tutorial`. The `:requires` subform specifies that the `:agenda-shell-user` module must be compiled (if necessary) and then loaded before our `:tutorial` module. The `:files` subform specified the files that comprise the module. In our case, there is one file: `tutorial-example.lisp`. We leave off the `.lisp` file extension, as the Module Manager will add the appropriate source or compiled file extension for us.

Step 3: Add the random-walk application to your personal `gbbopen-modules` directory

The `gbbopen-modules` directory in your "homedir" is expected to consist of directories each containing an individual GBBopen application. We could place the random-walk application directly in the `gbbopen-modules` directory by moving the `tutorial` directory there. However, it is generally more convenient to use a symbolic link to point to the actual application directory. For example, an application can be provided to a number of users by creating a symbolic link to the application directory in each user's `gbbopen-modules` directory.

Unless you are running Windows, add the random-walk application to your `gbbopen-applications` by creating a symbolic link. For example:

```
[~]$ cd ~/gbbopen-modules/
[~]$ ln -s ~/tutorial .
[~]$
```

Windows users

Instead of creating a symbolic link, GBBopen also supports a special “pseudo symbolic-link” file that can be used with Windows. This is simply a text file with the file extension `.sym` that contains the target directory path as the sole line in the file. For example, you could create the file `tutorial.sym` in your `gbbopen-modules` directory with:

```
C:\tutorial\
```

as the sole line in the file.

Step 4: Try the `:tutorial` module definition

Let’s try out our module definition. Exit Common Lisp and start a fresh Common Lisp session. If you have set up your environment according to the Enhancing Your Development Environment exercise (see page 29), the following files should be loaded:

```
...
;; Loading <homedir>/shared-init.lisp
;; Loading <install-dir>/initiate.lisp
;; GBBopen is installed in <install-dir>
;; Your "home" directory is <homedir>
;; Loading <install-dir>/extended-repl.lisp
;; Loading <install-dir>/commands.lisp
;; Loading <install-dir>/gbbopen-modules-directory.lisp
;; No shared module command definitions were found in <install-dir>/gbbopen-modules/.
;; No personal module command definitions were found in <homedir>/gbbopen-modules/.
cl-user>
```

Note that some basic GBBopen initialization files have been loaded for us as well as GBBopen’s command definitions and any command definitions for applications linked from our `<homedir>/gbbopen-modules/` directory. No module definitions have been defined yet, and GBBopen itself (or even the Module Manager Facility) were not loaded by `<install-dir>/initiate.lisp`.

Now, instead of loading the `:agenda-shell-user` module, let’s load only the `:module-manager-user` module:

```
cl-user> :module-manager-user
;; Loading <install-dir>/startup.lisp
...
;; Loading <install-dir>/<platform-dir>/module-manager/module-manager-user.fasl
;; Loading <install-dir>/modules.lisp
;; Loading module definitions from <homedir>/gbbopen-modules/...
;; Loading <homedir>/tutorial/modules.lisp
module-manager-user>
```

Note that when the Module Manager was loaded as part of loading the `:module-manager-user` module, the module definitions for our personal GBBopen modules were loaded automatically. (In this case, the `<homedir>/tutorial/modules.lisp` file.)

Now that we have defined our `:tutorial` module, we can use the **compile-module** REPL command, `:cm`, to compile (if needed) and load it:

```
module-manager-user> :cm :tutorial
;; Loading <install-dir>/<platform-dir>/tools/portable-threads.fasl
...
```

```
;; Loading ../gbbopen/control-shells/agenda-shell-user.fasl

Error: Directory <homedir>/tutorial/<platform-dir>/
      in module :tutorial doesn't exist.

Restart actions (select using :c n):
  0: Create this directory.
  1: Create this directory and any future missing directories.
module-manager-user>>
```

The `:requires` in our `:tutorial` module definition causes the `:agenda-shell-user` module (and its required modules) to be loaded for us. Then the Module Manager stops with a continuable error, telling us that the directory to hold the compiled application files does not exist. The compiled files are put in a Common Lisp and platform-specific subdirectory, `<platform-dir>`, in our `tutorial` directory that mirrors the `source` directory. This organization makes it easy to use the application with a number of Common Lisp implementations and on a file system shared with a number of different hosts and operating systems.

We could have avoided this continuable error by providing the `:create-dirs` option to the `:cm` command:

```
module-manager-user> :cm :tutorial :create-dirs
```

to allow the Module Manager to create the `<platform-dir>` subdirectory automatically for us. Since we did not do this, we can still continue from the error:

```
Restart actions (select using :c n):
  0: Create this directory.
  1: Create this directory and any future missing directories.
module-manager-user>> :c 0
;; Compiling file <homedir>/tutorial/source/tutorial-example.lisp
;; Loading <homedir>/tutorial/<platform-dir>/tutorial-example.fasl
module-manager-user>
```

At this point, we've compiled and loaded our `:tutorial` application module.

Step 5: Create a command-definition file for the random-walk application

It is convenient to define a REPL command to compile and load you application (and any required GBBopen modules).

Use your Common Lisp editor to create a new file named `commands.lisp` in the `tutorial` directory. Type the following two forms into the new `commands.lisp` file:

```
(in-package :common-lisp-user)

(define-repl-command :tutorial (&rest options)
  "Compile and load the Random-Walk Tutorial Application Module"
  (startup-module :tutorial options :gbbopen-user))
```

and then save the file. A `commands.lisp` file should always specify the `:common-lisp-user` package as the first form in the file.

The **define-repl-command** form adds a REPL command, named `:tutorial`, to the set of handy REPL commands. The **startup-module** call does all the work associated with executing the

command. The first argument to **startup-module** specifies that the `:tutorial` module will be compiled (if necessary) and then loaded by the Module Manager when the `:tutorial` command is issued. The second, `options`, argument passes any options given with the command to a **compile-module** call that is performed by **startup-module**. The third argument, `:gbbopen-user` is optional and specifies that the REPL's current package should be changed to `:gbbopen-user` after the `:tutorial` module is loaded.

Step 6: Try the `:tutorial` command

Let's try our command definition. Exit Common Lisp and start a fresh Common Lisp session. If you have set up your environment according to the Enhancing Your Development Environment exercise (see page 29), the following files should be loaded:

```
...
;; Loading <homedir>/shared-init.lisp
;; Loading <install-dir>/initiate.lisp
;; GBBopen is installed in <install-dir>
;; Your "home" directory is <homedir>
;; Loading <install-dir>/extended-repl.lisp
;; Loading <install-dir>/commands.lisp
;; Loading <install-dir>/gbbopen-modules-directory.lisp
;; No shared module command definitions were found in <install-dir>/gbbopen-modules/.
;; Loading personal module command definitions from <homedir>/gbbopen-modules/...
;; Loading <homedir>/gbbopen-modules/tutorial/commands.lisp
cl-user>
```

Note that the `commands.lisp` file from the tutorial directory has been loaded by `<install-dir>/initiate.lisp`.

Now, we can compile and load the `:tutorial` module by simply issuing the `:tutorial` REPL command:

```
cl-user> :tutorial
;; Loading <install-dir>/startup.lisp
...
;; Loading <homedir>/tutorial/<platform-dir>/tutorial-example.fasl
gbbopen-user>
```

With the command definition in place, we are able to compile and load our random-walk application by issuing a single command, `:tutorial`.

Note a potential continuable error due to a missing `<platform-dir>` subdirectory can always be avoided by providing the `:create-dirs` option to the `:tutorial` command:

```
gbbopen-user> :tutorial :create-dirs
```

to allow the Module Manager to create the `<platform-dir>` subdirectory automatically for us. However, since we created `<platform-dir>` in Step 4, we did not need to specify the `:create-dirs` option again in this step.

Step 7: Create missing subdirectories automatically

If you prefer, you can specify that the Module Manager should always create missing `<platform-dir>` directories and subdirectories instead of signaling an continuable error if one is missing (and `:create-dirs` was not specified). This behavior is controlled by the value of the symbol

automatically-create-missing-directories, which is in the `:common-lisp-user` package and is set to `nil` by default. If you wish, add the following form to your `shared-init.lisp` file (in your “homedir” directory):

```
(defparameter *automatically-create-missing-directories* 't)
```

I actually prefer to have the Module Manager generate the continuable error if it has to create a `<platform-dir>` directory and I didn’t specify `:create-dirs`, so I leave ***automatically-create-missing-directories*** set to the default `nil` value.

Step 8: Create and use an application-specific package

We have been developing our random-walk application in GBBopen’s `:gbbopen-user` package. The `:gbbopen-user` package is convenient, and we could continue using it. However, if we develop multiple GBBopen applications in the `:gbbopen-user` package and load several of them at the same time, symbol-name clashes could occur.

To eliminate this possibility, we can create our own package for the random-walk application. First, let’s determine what packages are being used by GBBopen’s `:gbbopen-user` package. Evaluate the following:

```
gbbopen-user> (package-use-list :gbbopen-user)
(#<package PORTABLE-THREADS> #<package AGENDA-SHELL>
 #<package MODULE-MANAGER> #<package COMMON-LISP>
 #<package GBBOPEN-TOOLS> #<package GBBOPEN>)
gbbopen-user>
```

Change `tutorial-example.lisp`

We want our new `:tutorial` package to use the same packages that the `:agenda-shell-user` package used. Edit your `tutorial-example.lisp` file and replace the `:gbbopen-user` package specification:

```
(in-package :gbbopen-user)
```

with the following:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (unless (find-package :tutorial)
    (defpackage :tutorial
      (:use :common-lisp :module-manager :gbbopen-tools :gbbopen
           :portable-threads :agenda-shell))))

(in-package :tutorial)
```

and save the file.

Note the use of `eval-when` in the first form above. Normally, top-level forms in a file are not evaluated at compile time. In this case, however, we want to define the `:tutorial` package when needed, whether the file is being compiled or loaded. The `eval-when` special operator with the three situations `(:compile-toplevel, :load-toplevel, and :execute)` provides this behavior to the forms that it contains. Such `eval-when` forms are a standard Common Lisp idiom for compile-time and load-time evaluation.

An application feature

In my applications, I also add a feature to Common Lisp's `*features*` list to indicate that the application has been fully loaded. To do this, add the following at the end of your `tutorial-example.lisp` file:

```
(pushnew :tutorial *features*)
```

and save the file.

Change `commands.lisp`

Next, edit your `commands.lisp` file and delete the `:gbbopen-user` package-name argument to **startup-module**:

```
(define-repl-command :tutorial (&rest options)
  "Compile and load the Random-Walk Tutorial Application Module"
  (startup-module :tutorial options :gbbopen-user))
```

and add the package-name `:tutorial` in its place:

```
(define-repl-command :tutorial (&rest options)
  "Compile and load the Random-Walk Tutorial Application Module"
  (startup-module :tutorial options :tutorial))
```

Save the file.

Change `modules.lisp`

Finally, we no longer need the `:gbbopen-user` package that is created by requiring the `:agenda-shell-user` module. Let's take a closer look at the `:tutorial` module that we defined:

```
gbbopen-user> (describe-module :tutorial)
Module :tutorial (loaded)
  Requires: (:agenda-shell-user)
  Fully expanded requires: (:module-manager :module-manager-user
:portable-threads
                           :gbbopen-tools :gbbopen-core :polling-functions
                           :queue :agenda-shell :os-interface :gbbopen-user
                           :agenda-shell-user)
  Source directory: <homedir>/tutorial/source/
  Compiled directory: <homedir>/<platform-dir>/
  Forces recompile date: None
  Files:   Mar 24 06:02 tutorial-example
gbbopen-user>
```

Although we only specified that the `:agenda-shell-user` module was required, our `:tutorial` module implicitly requires a number of packages that are required by the `:agenda-shell-user` module and its implicitly required packages. These are shown as the “Fully expanded requires” value.

If we look at the details of the `:agenda-shell-user` module we see:

```

gbbopen-user> (describe-module :agenda-shell-user)
Module :agenda-shell-user (loaded)
  Requires: (:agenda-shell :gbbopen-user)
  Fully expanded requires: (:module-manager :module-manager-user
:portable-threads
                                :gbbopen-tools :gbbopen-core :polling-functions
                                :queue :agenda-shell :os-interface
:gbbopen-user)
  Source directory: <install-dir>/source/gbbopen/control-shells/
  Compiled directory: <install-dir>/<platform-dir>/gbbopen/control-shells/
  Forces recompile date: None
  Files:   Mar 23 12:27  agenda-shell-user
gbbopen-user>

```

Note that the `:agenda-shell-user` module requires two modules: `:agenda-shell` and `:gbbopen-user`. We can eliminate the loading of the `:gbbopen-user` module by editing our `modules.lisp` file and delete `:agenda-shell-user` in the `:requires` option in our `:tutorial` module definition:

```

(in-package :module-manager-user)

(define-module :tutorial
  (:requires :agenda-shell-user)
  (:files "tutorial-example"))

```

and replace it with `:agenda-shell`:

```

(in-package :module-manager-user)

(define-module :tutorial
  (:requires :agenda-shell)
  (:files "tutorial-example"))

```

Save the file.

Step 9: Verify your changes

Let's make sure that everything is still working. Exit Common Lisp and start a fresh Common Lisp session. Next enter the `:tutorial` REPL command:

```

cl-user> :tutorial
;; Loading <install-dir>/startup.lisp
...
;; Loading <homedir>/tutorial/<platform-dir>/tutorial-example.fasl
tutorial>

```

Note that we are now in our newly defined `:tutorial` package. We should still be able to run the `random-walk` application:

```

tutorial> (start-control-shell)
;; Control shell 1 started

Walked off the world: (56, 38).

```

```

The random walk:
1 (0 0)
2 (-1 -1)
3 (-8 -10)
4 (0 -2)
5 (-5 2)
6 (3 11)
7 (8 5)
8 (12 2)
9 (3 12)
10 (10 4)
    ...
55 (50 40)
56 (42 47)
57 (47 41)
;; Explicit :stop issued by KS print-walk-ks
;; Control shell 1 exited: 60 cycles completed
;; Run time: 0.01 seconds
;; Elapsed time: 0 seconds
:stop
gbbopen-user>

```

Installation-wide sharing

GBBopen also has an `<install-directory>/shared-gbbopen-modules` directory. As with our personal `gbbopen-modules` directory, this `shared-gbbopen-modules` directory is assumed to contain symbolic links (or “pseudo-symbolic-link” files on Windows) to individual GBBopen module directory trees.

This is the recommended mechanism for installation-wide managing and sharing of modules and applications, and if we wanted to share our random-walk application to everyone using our GBBopen installation, we could create our symbolic link (or “pseudo-symbolic-link” file) in the `shared-gbbopen-modules` directory.

Step 10: Add an “autorun” action

Suppose we want the random-walk application to run automatically when it is loaded. You could simply add:

```
(start-control-shell)
```

as a top-level form at the end of your `tutorial-example.lisp` file. The problem with this is that sometimes you may want to compile and load the application without running it.

GBBopen’s Module Manager Facility supports a convention that makes it easy to conditionalize load-time action execution via the value of `*autorun-modules*`. Normally, `*autorun-modules*` will be true, but it can be set to `nil` when a module is loaded with the `:noautorun` option.

Add the following at the end of your `tutorial-example.lisp` file:

```
(when *autorun-modules*
  (format t "~{~&~s~%~}" (multiple-value-list (start-control-shell))))
```

and save the file. We could have simply called **start-control-shell** when `*autorun-modules*` is true, but then we would not be able to see what values are returned by the Agenda Shell. The format form above prints each returned value on a separate output line.

By convention, the “autorun” form is placed at the very end of the file, immediately after the form to add `:tutorial` to Common Lisp’s `*features*`. This is so that the `:tutorial` feature will be present during the “autorun” execution and thereafter—even if an error occurs when executing the “autorun” form.

Step 11: Try it out

Enter the `:tutorial` REPL command. The modified `tutorial-example.lisp` file should compile and load, followed by a random walk:

```
tutorial> :tutorial
;; Compiling <homedir>/tutorial/source/tutorial-example.lisp
;; Loading <homedir>/tutorial/<platform-dir>/tutorial-example.fasl
;; Control shell 1 started

Walked off the world: (11, -51).

The random walk:
1 (0 0)
2 (6 7)
3 (6 3)
4 (3 -4)
5 (10 -13)
...
21 (-5 -46)
22 (-9 -39)
23 (1 -33)
24 (8 -41)
;; Explicit :stop issued by KS print-walk-ks
;; Control shell 1 exited: 27 cycles completed
;; Run time: 0.01 seconds
;; Elapsed time: 0 seconds
:stop
tutorial>
```

Let’s try it again:

```
tutorial> :tutorial
tutorial>
```

This time, nothing happened. Why?

Since no source files were modified, the Module Manager knows that the latest compiled files for the `:tutorial` module and its required modules have all been loaded. So, because the `tutorial-example` file is not loaded, its “autorun” conditional form is not evaluated.

We can tell the Module Manager to always reload the `tutorial-example` file by editing our `modules.lisp` file and adding the `:reload file` option to the `:tutorial` module definition:

```
(define-module :tutorial
  (:requires :agenda-shell)
  (:files ("tutorial-example" :reload)))
```

Note that once a file has one or more options, the file name and its options are enclosed in parentheses.

Save the modified `modules.lisp` file. Now, if you try the `:tutorial` command, the `tutorial-example` file will always be loaded and its “autorun” form evaluated:

```
tutorial> :tutorial
;; Loading <homedir>/tutorial/modules.lisp
;; Loading <homedir>/tutorial/<platform-dir>/tutorial-example.fasl
;; Control shell 1 started
...
tutorial>
```

The Module Manager noticed our updated `modules.lisp` file and loaded it, redefining the `:tutorial` module definition, and then followed our `:reload` specification.

Let’s try the `:tutorial` command one more time, just to be certain that `:reload` is happening when no files have been updated:

```
tutorial> :tutorial
;; Loading <homedir>/tutorial/<platform-dir>/tutorial-example.fasl
;; Control shell 1 started
...
tutorial>
```

Module Manager propagation

The Module Manager’s **compile-module** function also accepts a `:reload` option, so we might be tempted to simply add that option when we specify our `:tutorial` command:

```
tutorial> :tutorial :reload
;; Loading <install-dir>/<platform-dir>/module-manager/module-manager.fasl
;; Loading
<install-dir>/<platform-dir>/module-manager/module-manager-user.fasl
...
;; Control shell 1 started
...
tutorial>
```

however, this would also reload all the files of every module required by the `:tutorial` module, as the **startup-module** function that we used in defining our `:tutorial` REPL command always adds a `:propagate` option to the options that we provide to the command. We could override (cancel) this propagation behavior by adding the `:nopropagate` option when we specify our `:tutorial` command:

```
tutorial> :tutorial :reload :nopropagate
;; Loading <homedir>/tutorial/<platform-dir>/tutorial-example.fasl
;; Control shell 1 started
...
tutorial>
```

which would eliminate reloading of all but the files in the `:tutorial` module. Since we only have one file (at the moment) specified in our `:tutorial` module definition, this behavior is equivalent, but less convenient, than specifying the `:reload file` option to our `tutorial-example` file in the module definition. If we had a number of files, however, we would probably only want the last one reloaded every time.

Module Manager memories

The Module Manager also remembers the last module (and any provided options, such as `:create-dirs`) that was specified to a `:cm` (`textbfcompile-module`) or `:lm` (**load-module**) command. The **startup-module** function that we used in defining our `:tutorial` REPL command performs an implicit `:cm` command for us, so we could have alternatively typed the `:cm` REPL command rather than `:tutorial` once we have issued the first `:tutorial` REPL command:

```
tutorial> :cm
;; :cm :tutorial :propagate
;; Loading <homedir>/tutorial/<platform-dir>/tutorial-example.fasl
;; Control shell 1 started
...
tutorial>
```

Just the thing for lazy typists like me! Note that the `:cm` command echos the full (implicitly completed) command with the remembered module name and any remembered options.

Step 12: Add a second source file

Reloading the entire `tutorial-application` file in order to evaluate our “autorun” form is still a bit heavy handed. There are two ways to improve this situation, and both involve placing the “autorun” form in a separate file. Use your Common Lisp editor to create a new file named `autorun.lisp` in the source subdirectory of the `tutorial` directory. Type (or copy) the following two forms into the new `autorun.lisp` file:

```
(in-package :tutorial)

(when *autorun-modules*
  (format t "~{&~s~%~}" (multiple-value-list (start-control-shell))))
```

and save the file.

Next, edit the `tutorial-example.lisp` file and remove the “autorun” form from the end of the file:

```
(when *autorun-modules*
  (format t "~{&~s~%~}" (multiple-value-list (start-control-shell))))
```

Save the file.

We could define a second random-walk application module, say `:run-tutorial`, in our `modules.lisp` file that contains the new `autorun.lisp` file and requires our current `:tutorial` module. This definition would look like:

```
(define-module :run-tutorial
  (:requires :tutorial
   (:files ("autorun" :reload))))
```

However, we can avoid creating a new module by simply adding the `autorun` file to our current `:tutorial` module definition.

Edit the `modules.lisp` file in the `tutorial` directory and remove the `:reload` file option from the `:tutorial-example` file specification:

```
(define-module :tutorial
  (:requires :agenda-shell)
  (:files ("tutorial-example" :reload)))
```

and then add a new line for the autorun file with the `:reload` file option:

```
(define-module :tutorial
  (:requires :agenda-shell)
  (:files "tutorial-example"
    ("autorun" :reload)))
```

and save the file.

Step 13: One last check

Let's double-check that everything is working:

```
tutorial> :cm
;; :cm :tutorial :propagate
;; Loading <homedir>/tutorial/modules.lisp
;; Compiling <homedir>/tutorial/source/tutorial-example.lisp
;; Loading <homedir>/tutorial/<platform-dir>/tutorial-example.fasl
;; Compiling <homedir>/tutorial/source/autorun.lisp
;; Loading <homedir>/tutorial/<platform-dir>/autorun.fasl
;; Control shell 1 started
...
tutorial>
```

The modified files are compiled and loaded and the Agenda Shell is invoked.

Let's try it one last time, just to be sure that the application runs when no files have been modified:

```
tutorial> :cm
;; :cm :tutorial :propagate
;; Loading <homedir>/tutorial/<platform-dir>/autorun.fasl
;; Control shell 1 started
...
tutorial>
```

Congratulations! It's time to move to the next exercise.

13 Multiple Walkers

Coming later ...

Please continue with the next exercise.

14 A Dimensional Detour

Coming later, exploration of unbound dimensional values, intersection of unit instance, space instance, and retrieval dimensionality, ...

Please continue with the next exercise.

15 More to come...

Additional exercises will be added soon.

16 The Completed Application

The complete, finished code for the random walk application is in the `tutorial.lisp` file in the `source/gbbopen/examples/` directory in the GBBopen distribution and at <http://gbbopen.org/svn/GBBopen/trunk/source/gbbopen/examples/tutorial.lisp>.

Index

Page references are shown in **bold** when they refer to the definition or main source of information on the entry. A page reference that is given in *green italics* indicates an instructive example of the use of that entity.

(setf x-of) slot writer method, [13](#)
(setf x-of), [13](#)
automatically-create-missing-directories, [85](#)
print-case, [5](#)
:cm REPL command, [83](#)
:dsbb REPL command, [41](#), [41](#), [48](#), [63](#)
:gbbopen-user REPL command, [30](#)
:stop, [76](#)

acknowledgments, [iv](#)
add-instance-to-space-instance, [17](#), [18](#), [24](#), [26](#), [39](#), [40](#)
applying a function to all instances of a class, [22](#)

blackboard repository, describing, [16](#)
browse-hyperdoc, [34](#)
browse-hyperdoc.el, [33](#)

children-of, [25](#)
Common Lisp
 debugger, [6](#)
 installing, [3](#)
 REPL, *see* REPL
compile-gbbopen, [9](#)
compile-module, [9](#), [11](#), [15](#), [21](#)
compiling
 all of GBBopen, [9](#)

define-ks, [46](#), [63](#), [76](#)
define-repl-command, [84](#)
define-unit-class, [11](#), [40](#)
defparameter, [15](#), [21](#), [38](#), [41](#)
defvar, [12](#), [16](#)
delete-blackboard-repository, [27](#), [36](#), [50](#)
delete-instance, [23](#), [27](#)
delete-space-instance, [27](#), [39](#)
describe-blackboard-repository, [16](#), [17](#), [18](#), [25–27](#), [39](#)
describe-instance, [12](#), [13](#), [17](#), [38](#), [42](#), [61](#)
describe-module, [87](#)
describe-space-instance, [40](#), [57](#)
do-instances-on-space-instances, [27](#)
do-sorted-instances-of-class, [66](#)
dotimes, [22](#)

errors
 file protection, [9](#)
 recovering from, [6](#)

find-instance-by-name, [13](#), [18](#), [22](#), [23](#), [26](#), [61](#), [76](#)

find-instances, [18](#), [18](#), [19](#), [22](#), [42–44](#)
find-space-instance-by-path, [16](#)

GBBopen
 installing, [4](#)
 mailing lists, [4](#)
 updating, [4](#)

in-package, [10](#), [11](#), [15](#), [21](#)
installing GBBopen, [4](#)
instance-deleted-p, [24](#)
instance-name-of, [12](#), [24](#)

knowledge-source activation, [46](#)
knowledge source, [46](#)
KS, *see* knowledge source
KSA, *see* knowledge-source activation

make-instance, [12](#), [18](#), [22](#), [28](#), [38](#), [41](#)
make-space-instance, [16](#), [25](#), [39](#), [56](#)
map-instances-of-class, [22](#), [22](#), [23](#), [27](#), [64](#)
map-instances-on-space-instances, [27](#)
map-sorted-instances-of-class, [65](#), [66](#)

original-class-of, [24](#)

package, [10](#)
parent-of, [25](#)
print, [21–23](#)
print-instance-slot-value, [43](#)
print-instance-slots, [42](#)
printv, [62](#)

quiescence, [46](#)
quiescence-event, [46](#), [76](#)

read-eval-print loop, *see* REPL
remove-instance-from-space-instance, [19](#)
REPL, [4](#)
REPL command
 :cm, [83](#)
 :dsbb, [41](#), [41](#), [48](#), [63](#)
 :gbbopen-user, [30](#)

shared-gbbopen-modules, installation-wide
 application sharing, [89](#)

slot
 getting value of, [13](#)
 setting value of, [13](#)
slot, link, *see* link slot
sole-trigger-instance-of, [63](#)
space instance
 creating, [16](#)
 deleting, [27](#)
 finding by path, [16](#)
 hierarchy, creating, [25](#)
 is a unit instance, [25](#)
standard-space-instance, [25](#)

start-control-shell, [46](#), [48–50](#), [55](#), [63](#), [66](#)

Subversion client, [4](#)

TortoiseCVS, [4](#)

TortoiseSVN, [4](#)

unit class

 applying a function to all instances of, [22](#)

 defining, [11](#)

unit instance

 adding to a space instance, [17](#)

 applying a function to all instances of a class, [22](#)

 creating, [12](#)

 deleting, [23](#)

 displaying a description of, [12](#)

 finding

 by name, [13](#)

 on a space instance, [18](#)

 removing from a space instance, [19](#)

 slot

 getting value of, [13](#)

 setting value of, [13](#)

updating GBBopen, [4](#)

values, [4](#), [5](#)

x-of slot reader method, [13](#)