

# **Project Darkstar Server Application Tutorial**

Copyright © 2008 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at <http://www.sun.com/patents> and one or more additional patents or pending patent applications in the U.S. and in other countries.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties.

Sun, Sun Microsystems, the Sun logo, Java, Solaris, J2SE and Java RMI are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon architecture developed by Sun Microsystems, Inc.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Products covered by and information contained in this service manual are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright © 2008 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

Sun Microsystems, Inc. détient les droits de propriété intellectuelle relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plus des brevets américains listés à l'adresse <http://www.sun.com/patents> et un ou les brevets supplémentaires ou les applications de brevet en attente aux Etats - Unis et dans les autres pays.

Cette distribution peut comprendre des composants développés par des tierces parties.

Sun, Sun Microsystems, le logo Sun, Java, Solaris, J2SE et Java RMI sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Les produits qui font l'objet de ce manuel d'entretien et les informations qu'il contient sont regis par la legislation americaine en matiere de controle des exportations et peuvent etre soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucleaires, des missiles, des armes biologiques et chimiques ou du nucleaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou reexportations vers des pays sous embargo des Etats-Unis, ou vers des entites figurant sur les listes d'exclusion d'exportation americaines, y compris, mais de maniere non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une facon directe ou indirecte, aux exportations des produits ou des services qui sont regi par la legislation americaine en matiere de controle des exportations et la liste de ressortissants specifiquement designes, sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFACON.

# Contents

---

## Introduction vi

## Coding Project Darkstar Server Applications 1

### Goals and Philosophy 1

### Approach to Execution 1

#### Tasks and Managers 1

#### Task Ordering 2

#### Task Lifetime 2

### Managed Objects and Managed References 3

#### Accessing Managed Objects through Managed References 3

#### Designing Your Managed Objects 4

#### The Player Managed Object 5

#### The AppListener 6

### Locating the Server API Classes 9

#### System Classes and Interfaces 9

#### Task Manager Classes and Interfaces 10

#### Data Manager Classes and Interfaces 10

#### Channel Manager Classes and Interfaces 11

## Lesson One: Hello World! 12

### Coding HelloWorld 12

#### HelloWorld 12

### Running HelloWorld 13

#### Rerunning HelloWorld 14

## Lesson Two: Hello Logger! 17

### Coding HelloLogger 17

#### HelloLogger 17

### The Logging Properties File 18

## Lesson 3: Tasks, Managers, and Hello Timer! 19

### Tasks 19

### Managers 19

### Coding HelloTimer 20

#### HelloTimer 20

## **Lesson 4: Hello Persistence! 23**

**Coding HelloPersistence 23**

    HelloPersistence 23

**Coding HelloPersistence2 25**

    HelloPersistence2 26

    TrivialTimedTask 27

**Coding HelloPersistence3 28**

    HelloPersistence3 29

## **Lesson 5: Hello User! 32**

**Knowing When a User Logs In 32**

    HelloUser 32

**Direct Communication 33**

    HelloUser2 33

    HelloUserSessionListener 34

    HelloEchoSessionListener 36

**Running the Examples 38**

## **Lesson 6: Hello Channels! 39**

**Coding HelloChannels 39**

    HelloChannels 39

    HelloChannelsSessionListener 41

    HelloChannelsChannelListener 43

**Running HelloChannels 44**

## **Conclusion 45**

**Best Practices 45**

**Things Not Covered in This Tutorial 47**

## **Appendix A: SwordWorld Example Code 49**

    Sword World 49

    SwordWorldObject 52

    SwordWorldRoom 54

    SwordWorldPlayer 58

# Introduction

---

Welcome to the Project Darkstar Server (PDS) application tutorial. This document is designed to teach you everything you need to know to start writing game servers that run on top of the Project Darkstar Server. We call such programs *PDS applications*, and you will see that term used in this and other PDS documents.

This tutorial begins with an overview of how to code a PDS application, then steps through the development of a very simple application.

# Coding Project Darkstar Server Applications

---

This chapter presents the fundamental concepts of how to code game server applications in the Project Darkstar Server (PDS) environment. Understanding these concepts is the first step on the path to building massively scalable, reliable, fault-tolerant, and persistent network games.

---

## Goals and Philosophy

---

In order to understand the Project Darkstar Server (PDS) coding model, it is useful to understand the system's goals. The fundamental goals are as follows:

- Make server-side game code reliable, scalable, persistent, and fault-tolerant in a manner that is transparent to the game developer.
- Present a simple single-threaded event-driven programming model to the developer. The developer should never have his or her code fail due to interactions between code handling different events.

Applications coded to run in the PDS environment are called *PDS applications*.

---

## Approach to Execution

---

### Tasks and Managers

From the point of view of the PDS application programmer, PDS applications execute in an apparently monothreaded, event-driven model. The code handling the event appears to the coder to have sole ownership of any data it modifies. Thus, execution is both race-proof and deadlock-proof. Under most conditions there is no need to synchronize application code and, in fact, attempting to use the synchronized keyword in Managed Objects<sup>1</sup> can cause subtle bugs.

In actuality, the system has many threads of control all simultaneously processing their own events. These threads of control are called *tasks*. The system keeps track of what data each task accesses. Should a conflict arise, one task is aborted and scheduled for retry at a later date so that the other task can complete and get out of the way.

Tasks are created by PDS *managers*. a PDS application uses these managers to effect actions in the PDS environment and the outside world.

There are three standard managers in the system. In addition, arbitrary managers may be coded and added to the PDS environment. The standard managers are:

- **Task Manager**  
A PDS application can use the *Task Manager* to queue tasks of its own. Tasks can be queued for immediate execution, delayed execution, or periodic execution. Tasks created from within other tasks are called *child tasks*. The task that queued the child task is called the *parent task*. Multiple child tasks queued by the same parent task are called *sibling tasks*.
- **Data Manager**  
A PDS application can use the *Data Manager* to create and access persistent, distributed Java objects called *Managed Objects*. The PDS application is itself composed of Managed Objects.

---

<sup>1</sup> PDS applications are made out of Managed Objects, which are fully explained in “Managed Objects and Managed References,” below.

- **Channel Manager**

A PDS application can use the *Channel Manager* to create and control publish/subscribe data channels. These data channels are used to communicate between groups of clients and the server.

A PDS application gets access to core PDS functionality through the **AppContext** class, which provides methods to obtain references to the various managers.

## Task Ordering

Tasks that handle events created by a client are guaranteed to execute in order. A task to handle a client event that occurred later in time will not start executing until the tasks to handle all earlier events generated by the same client have finished. A child task is ordered with regard to its parent task, but not with regard to its siblings. This means that execution of a child task will not begin until execution of the parent has completed. There are, however, no guarantees among sibling tasks as to order of execution.

**Important: There are no other order guarantees in the system. In particular, execution of tasks to handle events of different users are not guaranteed to start executing relative to each other in the order they arrived at the server.<sup>2</sup>**

## Task Lifetime

Tasks must be short-lived so that they do not block access for an inordinate amount of time to resources that might be needed by other tasks. The PDS is configured by its operator with a maximum task execution time (the default is 100 milliseconds). Any task that does not finish within that time will be forcibly terminated by the system.

If you have a task that runs too long, there are two approaches to reducing its execution time:

- Split it up into a chain of child tasks, each of which handles one discrete portion of the problem, and then queues the next task in sequence. This is known as *continuation-passing* style.
- Move the time-consuming calculations into a custom manager that queues a result-task when the calculations are complete.

Each approach has its advantages and disadvantages. The first is easier for simple problems that lend themselves to serial decomposition. Care must be taken that each task ends with the data in a sensible and usable state, because there is no guarantee as to exactly when the next step will be executed.

A special case of this approach is where parts of the problem are separable and handleable in parallel. In this case, the time to complete may be reduced by launching parallel chains of tasks. These parallel chains, however, have no guaranteed ordering in relation to each other, so the work they perform must really be independent of each other.

The second approach is easier for problems that don't decompose well into small, discrete components; however, it requires the writing and installation of a custom PDS manager. (Writing custom PDS managers will be covered by a separate document explaining how to extend the PDS environment.)

A particularly important case is code that has to go into system calls that can block for more than the task execution lifetime. These *must* be implemented through a custom manager in order to produce a robust PDS application.

---

<sup>2</sup> This also implies that tasks generated by two different clients will not necessarily get processed in the relative order they were created on the clients. In practice, this is generally not true in online games anyway unless special pains are taken to make it so, since simple variations in latencies in the two clients' net communications to the server can reorder their arrival.



---

## Managed Objects and Managed References

---

The Data Manager maintains a persistent set of Managed Objects stored in a pool of objects called the *Object Store*. Like a normal Java object, each Managed Object contains both data and the methods to act upon that data. In order to be a Managed Object, the object must implement both the **ManagedObject** and **Serializable** interfaces. A Managed Object does not become part of the Object Store's pool until the pool is made aware of the object. This is done by using the Data Manager either to request a *Managed Reference* to the object or to bind a name to the object.<sup>3</sup>

A Managed Reference is a reference object that looks much like the J2SE(tm) reference objects (for example, **SoftReference**, **WeakReference**). Managed Objects must refer to other Managed Objects through Managed References. This is how the Data Manager can tell the difference between a reference to a component object of the Managed Object (for instance, a list) that is part of that Managed Object's state, and a reference to a separate Managed Object with a state of its own.

A name binding associates a string with the Managed Object such that the object may be retrieved from the Object Store by other tasks using the **getBinding** call on the Data Manager.

### Accessing Managed Objects through Managed References

The Managed Reference has two access methods: **get** and **getForUpdate**. Both methods return a task-local copy of the object. The difference between the two methods is:

- **getForUpdate** informs the system that you intend to modify the state of the Managed Object.
- **get** says you intend only to read the state but not write it.

Although all changes to any Managed Object are persistent (even those accessed via **get**), it is more efficient to use **getForUpdate** if you know at that time that you are going to want to modify the Managed Object's state. This allows the system to detect conflicts between tasks and handle them earlier and with greater efficiency.

Conversely, it is better to use **get** if the state of the Managed Object may not be modified. The **get** call can allow for more parallel access to the Managed Object from multiple tasks. If you reach a point later in the execution where you know you are going to modify the object's state, you can upgrade your access from **get** to **getForUpdate** by calling the **markForUpdate** method on the Data Manager. (Multiple calls to mark the same Managed Object for update are harmless.)

Subsequent calls to **get** or **getForUpdate** on equivalent Managed References in the same task will return the same task-local copy.

You can also retrieve an object with a bound name by calling **getBinding**. This is equivalent to a **get** call on a Managed Reference to the object, so, if you intend to modify the object's state, you should call **markForUpdate** after retrieving the object.

*Managed Objects in the Object Store are not garbage-collected.* Once the store is made aware of a Managed Object, it keeps the state of that object until it is explicitly removed from the object store with a call to the **removeObject** call on the Data Manager. It is up to the application to manage the life cycle of Managed Objects and to remove them from the Object Store when they are no longer needed. Failure to do so may result in garbage building up in your Object Store and impacting its performance. Likewise, name bindings are stored until explicitly destroyed with **removeBinding**. A name binding is not removed when the object it refers to is removed.

---

<sup>3</sup> Be aware that this may happen in library code or in the PDS APIs themselves. See the best practices section at the end of this document for more information.

## Designing Your Managed Objects

Managed Objects typically fall into three general types of entity:

- Actual objects in your game's simulated environment, such as a sword, a monster, or a play-space (such as a room).
- Purely logical or data constructs such as a quad-tree for determining player-proximity or a walk-mesh to determine movement paths.
- Proxies for human players in the world of Managed Objects.

Figure 1 below illustrates a very basic world consisting of a single room that contains two players and a sword.

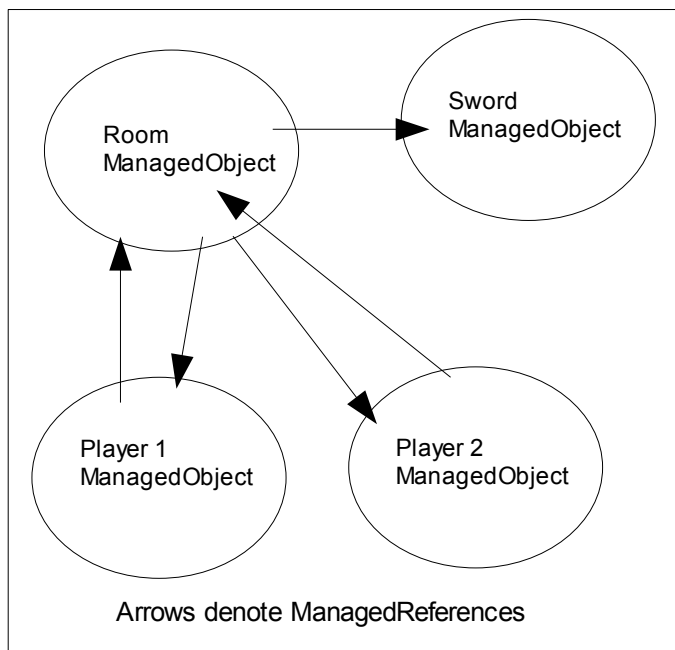


Figure 1: Example of a simple ManagedObject world

When deciding where to break data up into multiple Managed Objects, consider these questions:

- *How big is the data?* The more data a single Managed Object encompasses in its state, the more time it takes to load and save.
- *How closely coupled is the data?* Data that are generally accessed together are more efficiently stored in the same Managed Object. Data that are accessed independently are candidates for separation onto different Managed Objects. Data that have to be modified atomically are best stored in the same Managed Object.
- *How many simultaneous tasks are going to need access to this data?* As explained above, the PDS does its best to execute as many tasks in parallel as it can. **Resolving the conflicts that arise when multiple parallel tasks want to change the state of the same ManagedObject can be expensive.**

It is best to split up data that has to be locked for update from data that can be shared with a **get**. Data that is going to be updated has to be owned by the updating task, whereas data that is just read can be shared by multiple reading tasks. When multiple tasks have to access fields on a Managed Object that is being updated by at least one of them, that Managed Object becomes a potential bottleneck. For best performance, you want as few bottleneck Managed Objects as possible.

Of all these considerations, the third is the most critical to a well-running PDS application.

## The Player Managed Object

Managed Objects register themselves as event handlers with a manager in order to get called when outside events occur. One very important type of Managed Object is the *Player* Managed Object. A Player Managed Object implements the **ClientSessionListener** interface and is returned to the system as the return value from the **loggedIn** callback on the **AppListener**. From then on, it will get called for any incoming data packets and disconnect events from that player.

The Player Managed Object acts as a proxy for the player in the world of Managed Objects. The player sends data packets to the server using the PDS Client API. This causes a **userDataReceived** event in the system, which results in a task that calls the Player Managed Object's **userDataReceived** method. The Player Managed Object should parse the packet to find out what it is supposed to do, and then act on itself and other Managed Objects in order to accomplish the requested task.

Figure 2 shows our simple Managed Object world, with two players connected to the PDS as clients. The Player Managed Objects have a “current room” field, which is a Managed Reference that points to the Room Managed Object. The Room Managed Object has an inventory list, which is a list of Managed References. Currently, there are three items in the list: the two players and a sword. Each is represented by a Managed Object (Player 1 Managed Object, Player 2 Managed Object, and Sword Managed Object).

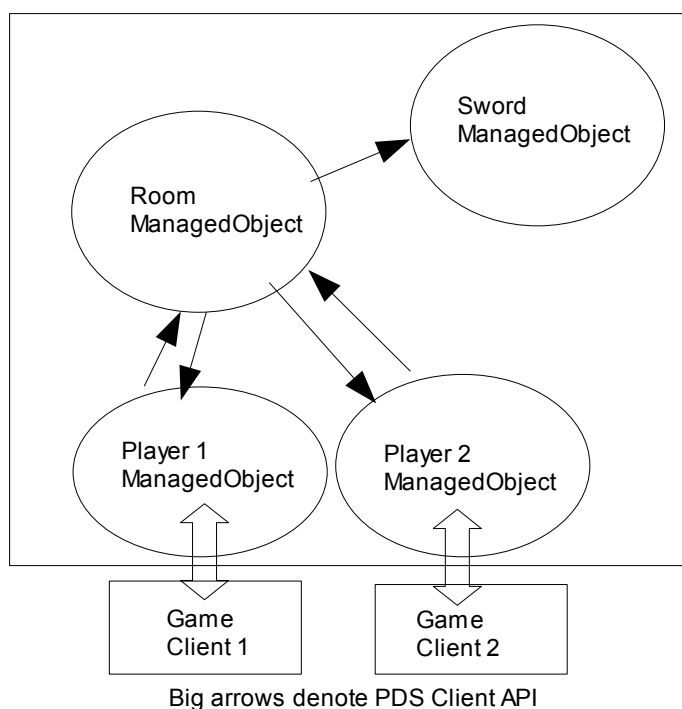


Figure 2: Client connections to the simple ManagedObject world

## The AppListener

Above we had a world of Managed Objects consisting of a Room Managed Object, a Sword Managed Object and a couple of Player Managed Objects. However, when we start the game in the PDS for the first time, the world of Managed Objects doesn't look like that. In fact it looks like this:



Figure 3: World of ManagedObjects at first-time start of game

Which is to say, it is empty.

How then do the Managed Objects get into the Object Store in the first place?

The answer is a special Managed Object called the **AppListener**. There are two special things about the class that defines the **AppListener**:

- It implements the **AppListener** interface. This interface defines two methods:
  - **initialize**
  - **loggedIn**
- It has been specified as the **AppListener** class for this application.

These two properties combine in the following way:

- Upon the boot of the PDS (or the installation of a new application into the PDS), the PDS attempts to locate the **AppListener** for that application in the Object Store.
- If the application has never been booted before, then its Object Store in the PDS is blank (as in Figure 3), and the PDS will fail to find the **AppListener**. In that case, it creates the **AppListener** Managed Object itself, and then starts a task that calls **initialize**.
- If, on the other hand, the application has been booted at least once, the Object Store will contain the **AppListener** Managed Object already. In this case, execution just resumes from where it left off when the system came down, listening for new connections and executing any periodic tasks that were running before.

In the case of our little demo application, the boot method will have a block in it that, in pseudo-code, looks something like this:<sup>4</sup>

---

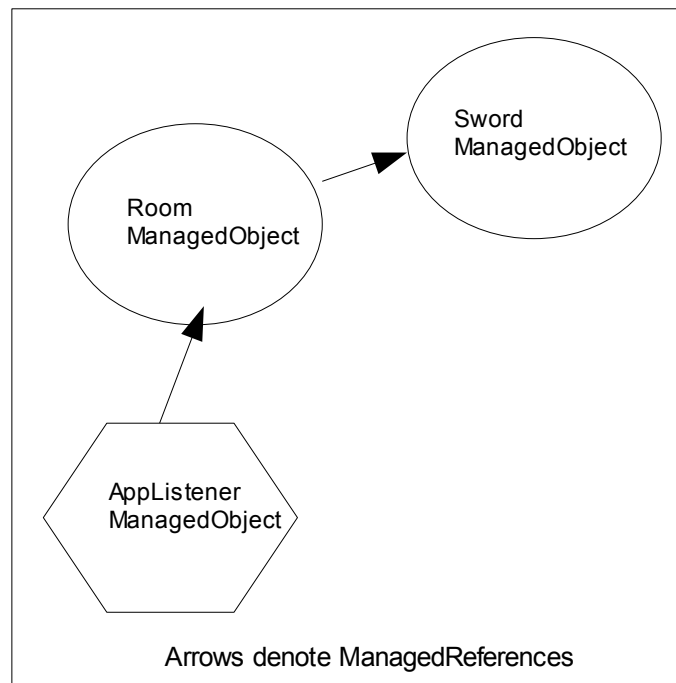
<sup>4</sup> In all pseudo-code in this document, the pseudo-code itself is in all caps, references to variables and methods are in mixed case beginning with a lower case letter, and references to classes and interfaces are in mixed-case beginning with an uppercase letter.

```

initialize {
    CREATE ROOM MANAGED OBJECT
    CREATE SWORD MANAGED OBJECT
    ADD REF TO SWORD MANAGED OBJECT TO ROOM'S INVENTORY
    SAVE A MANAGED OBJECT REF TO ROOM FOR LATER
}

```

In general, it is the responsibility of the **AppListener** to create the initial world of Managed Objects during the first startup.



**Figure 4: AppListener ManagedObject creates initial ManagedObject world**

Now we have something that is beginning to look like our game. We still don't have Player Managed Objects, however. We will create the Player Managed Objects as users join, in much the same way the **AppListener** Managed Object was created. The first time we see a user log in, we create a new Player Managed Object for that user. After that, every time that user logs in, we just reconnect him to his existing Player Managed Object. Thus, the system creates Player Managed Objects as needed, and remembers user information between logins.

So how do we find out when a user has logged in?

The answer is the second callback on our **AppListener**: **loggedIn**. Every time a user logs into a PDS application, a task is started that calls the **loggedIn** method on the application's **AppListener**.

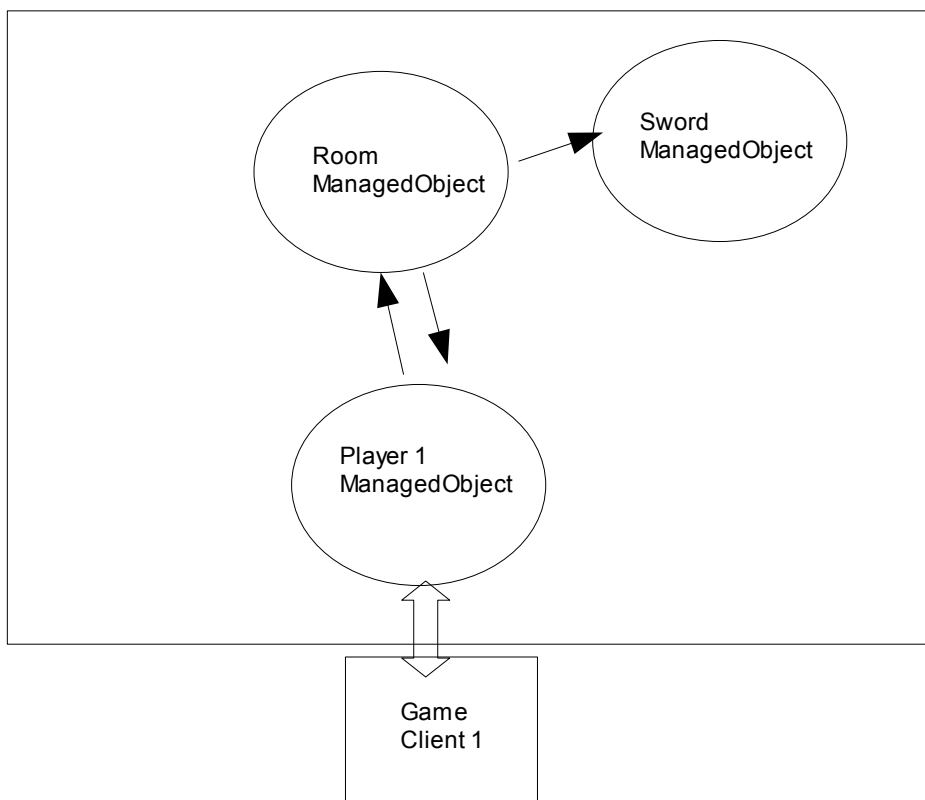
When the **loggedIn** callback is called on our **AppListener**, it executes the following code, presented as pseudo-code.

```

loggedIn {
    managedObject_name = "player_" + SESSION.PLAYER_NAME;
    IF MANAGED OBJECT EXISTS(managedObject_name) {
        FIND MANAGED OBJECT(managedObject_name);
    } ELSE {
        CREATE NAMED PLAYER MANAGED OBJECT(managedObject_name);
    }
    SET currentRoom on PLAYER MANAGED OBJECT
        TO SAVED MANAGED OBJECT REF TO ROOM
    GET ROOM MANAGED OBJECT
    ADD PLAYER REF TO ROOM MANAGED OBJECT'S PLAYERS LIST
    REGISTER PLAYER MANAGED OBJECT AS SessionListener(SESSION);
}

```

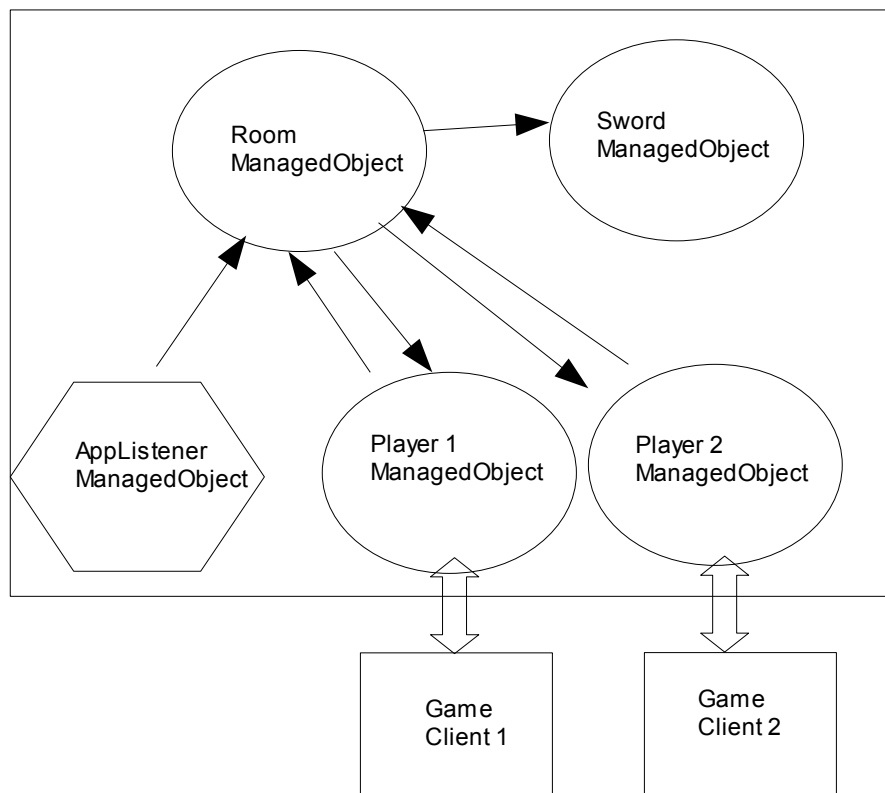
**ClientSessionListener** is another event interface. It defines methods that get called on tasks to respond to actions the client takes with the client API, such as the client sending data to the server for processing and the client logging out.



Big arrows denote PDS Client API

Figure 5 illustrates that our Managed Object world is starting to look the way we want it to. **Figure 5:** Client sends data to server

When a second user logs in, we will be back to our original world. Figure 6 illustrates our world after restarting the game with our previous players:



Big arrows denote PDS Client API

Figure 6: AppListener ManagedObject reestablishes simple world

So far, the logic has been laid out in pseudo-code. The actual code to implement this application is included in Appendix A as the **SwordWorld** application. The actual application code goes a bit further in that it also implements a **look** command, to show you how the Player Managed Object actually handles commands being sent from the client.

## Locating the Server API Classes

All the Project Darkstar Server API classes are in the **com.sun.sgs.app.\*** package.

These are the Project Darkstar Server API classes with brief descriptions:

### System Classes and Interfaces

Class	Description
AppContext	Provides access to facilities available in the current application. Primarily used to find references to managers. This is the starting point for the application code to talk to the system.
AppListener	Interface representing a listener for application-level

<b>Class</b>	<b>Description</b>
	events. This listener is called when the application is started for the first time, and when client sessions log in.
ManagerNotFoundException	Thrown when a requested manager is not found.
ClientSession	Interface representing a single, connected login session between a client and the server.
ClientSessionListener	Listener for messages sent from an associated client session to the server.

## Task Manager Classes and Interfaces

<b>Class</b>	<b>Description</b>
TaskManager	Provides facilities for scheduling tasks.
Task	Defines an application operation that will be run by the Task Manager.
PeriodicTaskHandle	Provides facilities for managing a task scheduled with the Task Manager to run periodically.
TaskRejectedException	Thrown when an attempt to schedule a task fails because the Task Manager refuses to accept the task due to resource limitations.
ExceptionRetryStatus	Implemented by exception classes that want to control whether an operation that throws an exception of that exception should be retried.

## Data Manager Classes and Interfaces

<b>Class</b>	<b>Description</b>
DataManager	Provides facilities for managing access to shared, persistent objects.
ManagedObject	A marker interface implemented by shared, persistent objects managed by the Data Manager.
ManagedReference	Represents a reference to a managed object.
ObjectIOException	Thrown when an operation fails because of an I/O failure when attempting to access a Managed Object.
ObjectNotFoundException	Thrown when an operation fails because it attempted to refer to a Managed Object that was not found.



<b>Class</b>	<b>Description</b>
TransactionAbortedException	Thrown when an operation fails because the system aborted the current transaction during the operation.
TransactionConflictException	Thrown when an operation fails because the system aborted the current transaction when it detected a conflict with another transaction.
TransactionException	Thrown when an operation fails because of a problem with the current transaction.
TransactionNotActiveException	Thrown when an operation fails because there is no current, active transaction.
TransactionTimeoutException	Thrown when an operation fails because the system aborted the current transaction when it exceeded the maximum permitted duration.
NameNotBoundException	Thrown when an operation fails because it referred to a name that was not bound to an object.

## Channel Manager Classes and Interfaces

<b>Class</b>	<b>Description</b>
ChannelManager	Manager for creating and obtaining channels.
Channel	Interface representing a communication group, a channel consisting of multiple client sessions and the server.
ChannelListener	A channel can be created with a <b>ChannelListener</b> , which is notified when any client session sends a message on that channel.
Delivery	Representation for message delivery requirements. A channel is created with a delivery requirement.
NameExistsException	Thrown when an operation fails because it referred to a name that is currently bound to an object.
NameNotBoundException	Thrown if a channel is not bound to a name specified to <b>ChannelManager.getChannel()</b> .

# Lesson One: Hello World!

---

It is traditional in any programming tutorial for the first example to be a simple program that prints “Hello World” to the console. This lets a programmer see the plumbing required to start even a basic application before diving into real-world application logic.

For simplicity, this lesson and the two that follow it print their output on the server console, rather than starting off with client-server networking. Even when a client is connected, server-side log messages are invaluable for debugging and monitoring the application.

---

## Coding HelloWorld

---

All PDS applications start with an **AppListener**. An **AppListener** is the object that handles an application's startup and client login events. An application's **AppListener** is simply a class that implements the **AppListener** interface. Since an **AppListener** is also a Managed Object, it must implement the **Serializable** marker interface as well.

As mentioned above, **AppListener** contains two methods: **initialize** and **loggedIn**. The **initialize** method gets called on the startup of the application *if and only if* the Object Store for this application is empty. The **AppListener** is automatically created in the Object Store by the system the first time the application is started up; in practice, this means that it is created once per application, unless the Object Store for this application is deleted and the system is returned to its pristine “never having run this application” state.<sup>5</sup>

A “Hello World” **AppListener** looks like this:

### HelloWorld

```
/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 *
 * This file is part of Project Darkstar Server.
 *
 * Project Darkstar Server is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation and
 * distributed hereunder to you.
 *
 * Project Darkstar Server is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package com.sun.sgs.tutorial.server.lesson1;

import java.io.Serializable;
import java.util.Properties;

import com.sun.sgs.app.AppListener;
import com.sun.sgs.app.ClientSession;
import com.sun.sgs.app.ClientSessionListener;
```

---

<sup>5</sup> The **SwordWorld** example in the appendix shows how to use this behavior to create your initial world of Managed Objects in the Object Store.

```

/**
 * Hello World example for the Project Darkstar Server.
 * Prints {@code "Hello World!"} to the console the first time it is started.
 */
public class HelloWorld
    implements AppListener, // to get called during application startup.
        Serializable // since all AppListeners are ManagedObjects.
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /**
     * {@inheritDoc}
     * <p>
     * Prints our well-known greeting during application startup.
     */
    public void initialize(Properties props) {
        System.out.println("Hello World!");
    }

    /**
     * {@inheritDoc}
     * <p>
     * Prevents client logins by returning {@code null}.
     */
    public ClientSessionListener loggedIn(ClientSession session) {
        return null;
    }
}

```

---

## Running HelloWorld

---

To run **HelloWorld** you need the following:

- The **Project Darkstar Server** installed on your system.
- A JDK™ 6 installation. The version can be found with:

```
java -version
```

If **java** is not in your default execution path, you will need to set **JAVA\_HOME** to point to the root of its installation on your system.

### Path Conventions

Unix and many Unix-derived systems use a forward slash (/) to show subdirectories in a file path. Win32, however, uses a backslash (\) for this purpose. Throughout this document we use the Unix convention for file paths unless it is in a Windows-specific example.

Please remember that you may have to substitute backslashes for forward slashes in the generic examples if you are working in Windows.

The PDS server is started by using the **sgs-boot.jar** executable jar file in the **bin** folder of the PDS installation on your computer. It requires a configuration file in order to configure the classpath and JVM environment that will be used to boot up the server. By default, this configuration is provided by the **sgs-boot.properties** file in the **conf** folder. However, if a filename is given on the command line, it will be used as the configuration file instead. Booting up the PDS server, therefore, is done in the following manner (assuming the current working directory is the root directory of the PDS installation):

```
java -jar bin/sgs-boot.jar [boot_config_file]
```

Where *boot\_config\_file* is an optional configuration file. If you are interested, you may examine the **sgs-boot.properties** file in the **conf** folder to see what configuration options are available.

In addition to the boot environment properties, a PDS application also needs a set of application properties that are to be passed to the **initialize** method of the **AppListener**. These properties are combined from the following sources. If a property appears in more than one source, the value is taken from the source higher on the list below:

1. Properties specified on the command line with the -D flag.
2. The properties file with the filename **.sgs.properties** in the user's home directory.
3. The properties file specified by name with the **SGS\_DEPLOY** property in the *boot\_config\_file*.
4. The properties file with the resource name **META-INF/app.properties**. This file is typically included in the **META-INF** directory of one of the application jars copied into the **deploy** folder of the PDS installation.

We have provided a **tutorial.jar** file in the **tutorial** folder that contains pre-compiled **.class** files for all the tutorial examples. Additionally, we have included default configuration files for each of the tutorial examples. These files are located in the **tutorial/conf** folder of the PDS installation. In order to run **HelloWorld** as it is shipped to you in the **tutorial** directories, do the following:

1. Open a Unix shell, a Windows command window, or whatever you do to get a command line on your development system.
2. Change your working directory to the directory of your PDS installation. In Unix, the command might be something like this:

```
cd ~/sgs-server-VERSION
```

5. Type the following:

```
java -jar bin/sgs-boot.jar tutorial/conf/HelloWorld.boot
```

You should see the application print out “Hello World!” to standard output (as well as a couple of PDS startup log messages) and then sit doing nothing. At this point you can kill the application; there are two ways of doing this:

1. The preferred method is by using the shipped executable shutdown jar file by issuing the following command:

```
java -jar bin/sgs-stop.jar tutorial/conf/HelloWorld.boot
```

2. Typing **Ctrl-c** in the shell window will also stop the server and get the prompt back.

If you are interested, you can examine the **HelloWorld.boot** and the **HelloWorld.properties** files to see the details of how you set up an application configuration to run in the PDS.

## Rerunning HelloWorld

If you stop the PDS and then run the **HelloWorld** application again, you will notice that you don't get a “Hello World!” output the second time. This is because the **AppListener** already exists in the Object Store from the previous run, and thus the **initialize** method on it is never called.

The location of the Object Store is specified by the *com.sun.sgs.app.root* property in the **HelloWorld.properties** file. If you want to see “Hello World” again, you can do it by clearing the Object Store with the following commands:

- For Unix:

```
rm -r tutorial/data/HelloWorld/dsdb/*
```

- For Windows:

```
del /s tutorial\data\HelloWorld\dsdb\*.*
```



# Lesson Two: Hello Logger!

---

## Coding HelloLogger

---

The PDS supports the standard Java logging mechanisms in the **java.util.logging.\*** package. This is a flexible and configurable logging API used by most servers written in Java. The PDS server itself uses the logger to report various internal states and events. *It is highly recommended that applications use the same logging mechanisms for their reporting.*

Below is a rewrite of **HelloWorld** that sends the “Hello World!” string to the logger rather than to standard out:

### HelloLogger

```
/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 *
 * This file is part of Project Darkstar Server.
 *
 * Project Darkstar Server is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation and
 * distributed hereunder to you.
 *
 * Project Darkstar Server is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package com.sun.sgs.tutorial.server.lesson2;

import java.io.Serializable;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.sun.sgs.app.AppListener;
import com.sun.sgs.app.ClientSession;
import com.sun.sgs.app.ClientSessionListener;

/**
 * Hello World with Logging example for the Project Darkstar Server.
 * It logs {@code "Hello World!"} at level {@link Level#INFO INFO}
 * when first started.
 */
public class HelloLogger
    implements AppListener, // to get called during application startup.
               Serializable // since all AppListeners are ManagedObjects.
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /** The {@link Logger} for this class. */
    private static final Logger logger =
        Logger.getLogger(HelloLogger.class.getName());
```

```

/**
 * {@inheritDoc}
 * <p>
 * Logs our well-known greeting during application startup.
 */
public void initialize(Properties props) {
    logger.log(Level.INFO, "Hello World!");
}

/**
 * {@inheritDoc}
 * <p>
 * Prevents client logins by returning {@code null}.
 */
public ClientSessionListener loggedIn(ClientSession session) {
    return null;
}
}

```

---

## The Logging Properties File

---

The Java logging API has a concept of message severity level. By logging at **Level.INFO**, we are telling the system we want to log this message at the **info** level of severity.

The Java logger's behavior is controlled by a logging properties file. The **sgs-boot.jar** file you use to run your applications uses the file **sgs-logging.properties** that is present in the **conf** directory of your PDS installation. This is accomplished by setting the **java.util.logging.config.file** property on the **java** command line of the PDS process.

By default this file sets the logging level to **info**. This means that logging messages below the level of **info**, such as “fine” debugging messages, will not be printed. You can change this by editing the **sgs-logging.properties** file. For more information on how to edit this file, please see the JDK™ 6 API documentation.



# Lesson 3: Tasks, Managers, and Hello Timer!

---

## Tasks

---

In Lessons 1 and 2, the system automatically invoked the **initialize** method for us in response to an event (in this case, the initial run of a newly installed PDS application). This meant executing code within the PDS environment.

All code run in the PDS environment must be part of a *task*. From the point of view of the application coder, a task is a piece of monothreaded, transactional, event-driven code. This means that the task runs as if it were the only task executing at that moment, and all actions done by the task to change data occur in an all-or-nothing manner.

### The realities of task execution

Each individual task executes in a monothreaded manner. However, if we executed them serially, waiting for each one to finish before the next one started, it would not be possible to get the kind of scaling the PDS provides.

Instead, the PDS executes many of these monothreaded tasks simultaneously and watches for contention on the individual Managed Objects. If two tasks contend for control over a Managed Object, one task will be held up and will wait for the other to finish before it can proceed.

Many tasks can read the state of the same Managed Object at the same time without causing contention. If any of them wants to write to it, however, that can cause contention with tasks that read from or write to the same Managed Object.

**To achieve optimal performance, it is important to design your data structures and game logic with as little potential object-contention as possible.** Be especially wary of places where multiple tasks that are likely to occur simultaneously might have to write to the same Managed Object.

All tasks registered with the PDS scheduler implement the interface **Task**, which has one method on it — **run**.<sup>6</sup> A task may be submitted to the scheduler to be executed either as soon as possible, or after a minimum delay time. A task can be one-shot or repeating. If it is a repeating task, it is also submitted with a period of repeat.

A repeating task is the same thing as a “timer” or “heartbeat” in traditional game systems; it lets you effectively generate an event to be handled every specified number of milliseconds.

---

## Managers

---

All communication between your server application's game logic and the world outside of it is accomplished through *managers*. As described above, there are three standard managers:

- Task Manager
- Data Manager
- Channel Manager

There can also be installation-specific managers; these can be written by the author of the server application and deployed with the application into a PDS back end. The following static calls on the **AppContext** class are used by server application code to get a reference to a manager to talk to:

---

<sup>6</sup> Although the event-handling interfaces like **AppListener** do not implement **Task**, these listeners get called from an internal **Task** just like regular PDS application code.

- `getTaskManager()`
- `getDataManager()`
- `getChannelManager()`
- `getManager(managerClass)`

The first three are covered in this tutorial. The last is a generic call to get an installation-specific manager; it will be covered in the forthcoming *Project Darkstar Server Extension Manual*.

**IMPORTANT:** A Manager Reference is valid only for the life of the task within which the get manager call was invoked. Therefore, you should not try to cache manager references for use outside of that one invocation chain; get them from the **AppContext** instead.

The Task Manager is the part of the PDS that contains the scheduler, and thus what we use to schedule tasks. In order to be scheduled with the Task Manager as a task to be run, an object must be serializable and must implement the **Task** interface.<sup>7</sup> The example below turns our **AppListener** into a task and starts it logging “Hello Timer” messages after a five-second delay at a half-second repeat period.

**IMPORTANT:** While the PDS stack makes a best effort to run tasks on schedule, it may back off execution of repeating tasks under heavy load. In that case, the task will skip execution of this period and reschedule to the next period. Additionally, contention for Managed Objects may cause a timed task to delay its execution.

The requested repeat frequency of a timed task is similar to a target frame rate in a game client, where frames may be dropped if they are taking too long to compute. If your application logic is tied to elapsed time or absolute number of “beats” in a given period of time, you’ll need to check the elapsed time and handle skipped periods in your **run** logic.

---

## Coding HelloTimer

---

The **HelloTimer** application below uses the **TaskManager** to schedule a repeating task. The task will run after a delay of 5000ms at a frequency of once every 500ms.

### HelloTimer

```
/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 *
 * This file is part of Project Darkstar Server.
 *
 * Project Darkstar Server is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation and
 * distributed hereunder to you.
 *
 * Project Darkstar Server is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */
```

---

<sup>7</sup> You might wonder why the **Task** and **ManagedObject** interfaces don’t just extend **Serializable**. The answer has to do with a subtlety of **Serialization** version control on interfaces. To guard against **serialVersionUID** mismatches, any class or interface that extends **Serializable** should declare a private **serialVersionUID** field. But since interfaces can only declare public fields, the best practice is to avoid making interfaces **Serializable** and leave it to the concrete (or abstract) classes to implement **Serializable** instead.

```

package com.sun.sgs.tutorial.server.lesson3;

import java.io.Serializable;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.sun.sgs.app.AppContext;
import com.sun.sgs.app.AppListener;
import com.sun.sgs.app.ClientSession;
import com.sun.sgs.app.ClientSessionListener;
import com.sun.sgs.app.Task;
import com.sun.sgs.app.TaskManager;

/**
 * A simple timed-task example for the Project Darkstar Server.
 * It uses the {@link TaskManager} to schedule itself as a periodic task
 * that logs the current timestamp on each execution.
 */
public class HelloTimer
    implements AppListener, // to get called during application startup.
               Serializable, // since all AppListeners are ManagedObjects.
               Task          // to schedule future calls to our run() method.
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /** The {@link Logger} for this class. */
    private static final Logger logger =
        Logger.getLogger(HelloTimer.class.getName());

    /** The delay before the first run of the task. */
    public static final int DELAY_MS = 5000;

    /** The time to wait before repeating the task. */
    public static final int PERIOD_MS = 500;

    // implement AppListener

    /**
     * {@inheritDoc}
     * <p>
     * Schedules the {@code run()} method to be called periodically.
     * Since SGS tasks are persistent, the scheduling only needs to
     * be done the first time the application is started. When the
     * server is killed and restarted, the scheduled timer task will
     * continue ticking.
     * <p>
     * Runs the task {@code #DELAY_MS} ms from now,
     * repeating every {@code #PERIOD_MS} ms.
     */
    public void initialize(Properties props) {
        TaskManager taskManager = AppContext.getTaskManager();
        taskManager.schedulePeriodicTask(this, DELAY_MS, PERIOD_MS);
    }

    /**
     * {@inheritDoc}
     * <p>
     * Prevents client logins by returning {@code null}.
     */
    public ClientSessionListener loggedIn(ClientSession session) {

```

```

        return null;
    }

    // implement Task

    /**
     * {@inheritDoc}
     * <p>
     * Logs the current timestamp whenever this {@code Task} gets run.
     */
    public void run() throws Exception {
        logger.log(Level.INFO,
            "HelloTimer task: running at timestamp {0,number,#}",
            System.currentTimeMillis());
    }
}

```

Now that we have a repeating event, we have our first application that will do something when stopped and restarted. Task registration is persistent, which is to say, it survives a crash and reboot. Try stopping the server and restarting it again to see this in action.

The periodic task information is stored in the Object Store along with your managed objects, so if you clear the data directory and return it to its pristine, uninitialized state, the periodic tasks will also get cleared.

This behavior allows you to write your code as if the server were always up, with the caveat that you *do* have to check elapsed time in your periodic task's **run** method if a delay between that and the last time it was run has significance to your logic.

How you keep track of the last time **run** was called is the subject of the next lesson.

## Lesson 4: Hello Persistence!

---

Lesson 3 explained that tasks that are run on a delay or repeat don't necessarily happen exactly at the time you asked for. They could happen a bit later if (for example) the system is very loaded, or a lot later if (for example) the entire data center has actually come down and had to be restarted.<sup>8</sup>

To track the last time the **run** task was called and calculate the true time-delta, we need a way of storing the past time value so that it will survive the system going down. This is called *persistent storage*, and in real games it is very important. Imagine how your users would react if your machine went down and they all lost their characters and everything on them!

A Managed Object is an object for which the system tracks state and which the system makes persistent. We mentioned above that **AppListener** interface inherits the Managed Object interface and that your **AppListener** instance is automatically created by the system for you. The system also registers your **AppListener** as a Managed Object with the Data Manager. This means that its state will be preserved by the PDS for you.

---

### Coding HelloPersistence

---

Since our **HelloTimer** task is a Managed Object, all we need to do is add a field to track the last time **run** was called. Below is the code for **HelloPersistence**.

Run **HelloPersistence** as a PDS application. Stop the PDS, wait a minute, and then start it again. You will see that the elapsed time reported includes the down time. This is because **currentTimeMillis** is based on the system clock, and time kept moving forward even when the PDS wasn't running.<sup>9</sup>

**Persistence is that simple and automatic in the PDS. Any non-transient field on a registered Managed Object will be persisted.**<sup>10</sup>

### HelloPersistence

```
/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 *
 * This file is part of Project Darkstar Server.
 *
 * Project Darkstar Server is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation and
 * distributed hereunder to you.
 *
 * Project Darkstar Server is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
```

- 
- <sup>8</sup> A full Project Darkstar Server production environment provides failover mechanisms so that the loss of one server won't bring the game down. In a true disaster, such as loss of power across the entire data center, it is possible the entire back end might go off-line.
- <sup>9</sup> Depending on your operating system, you may see the elapsed time reported by **HelloPersistence** while the PDS is running to be a bit over or a bit under 500ms. This is because **currentTimeMillis** does not necessarily have a 1 ms accuracy. In particular, Windows systems tend to have a lower **currentTimeMillis** accuracy than other Java™ SE platforms.
- <sup>10</sup> A *transient* field is one marked with the **transient** key word. Some values aren't valid beyond the task in which they are used, and thus should be marked **transient** – for example, a field that caches a **Manager** during the current task.

```

* You should have received a copy of the GNU General Public License
* along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

package com.sun.sgs.tutorial.server.lesson4;

import java.io.Serializable;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.sun.sgs.app.AppContext;
import com.sun.sgs.app.AppListener;
import com.sun.sgs.app.ClientSession;
import com.sun.sgs.app.ClientSessionListener;
import com.sun.sgs.app.ManagedObject;
import com.sun.sgs.app.Task;
import com.sun.sgs.app.TaskManager;

/**
 * A simple persistence example for the Project Darkstar Server.
 * As a {@link ManagedObject}, it is able to modify instance fields,
 * demonstrated here by tracking the last timestamp at which a task
 * was run and displaying the time delta.
 */
public class HelloPersistence
    implements AppListener, // to get called during application startup.
               Serializable, // since all AppListeners are ManagedObjects.
               Task          // to schedule future calls to our run() method.
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /** The {@link Logger} for this class. */
    private static final Logger logger =
        Logger.getLogger(HelloPersistence.class.getName());

    /** The delay before the first run of the task. */
    public static final int DELAY_MS = 5000;

    /** The time to wait before repeating the task. */
    public static final int PERIOD_MS = 500;

    /** The timestamp when this task was last run. */
    private long lastTimestamp = System.currentTimeMillis();

    // implement AppListener

    /**
     * {@inheritDoc}
     * <p>
     * Schedules the {@code run()} method to be called periodically.
     * Since SGS tasks are persistent, the scheduling only needs to
     * be done the first time the application is started. When the
     * server is killed and restarted, the scheduled timer task will
     * continue ticking.
     * <p>
     * Runs the task {@code #DELAY_MS} ms from now,
     * repeating every {@code #PERIOD_MS} ms.
     */
    public void initialize(Properties props) {
        TaskManager taskManager = AppContext.getTaskManager();
        taskManager.schedulePeriodicTask(this, DELAY_MS, PERIOD_MS);
    }
}

```

```

    }

    /**
     * {@inheritDoc}
     * <p>
     * Prevents client logins by returning {@code null}.
     */
    public ClientSessionListener loggedIn(ClientSession session) {
        return null;
    }

    // implement Task

    /**
     * {@inheritDoc}
     * <p>
     * Each time this {@code Task} is run, logs the current timestamp and
     * the delta from the timestamp of the previous run.
     */
    public void run() throws Exception {
        long timestamp = System.currentTimeMillis();
        long delta = timestamp - lastTimestamp;

        // Update the field holding the most recent timestamp.
        lastTimestamp = timestamp;

        logger.log(Level.INFO,
            "timestamp = {0,number,#}, delta = {1,number,#}",
            new Object[] { timestamp, delta }
        );
    }
}

```

---

## Coding HelloPersistence2

---

While we could put all the fields of our application on the **AppListener**, there are many good reasons not to do this. As any Managed Object grows larger, it takes more time for the system to store and retrieve it. Also, although PDS task code is written as if it were monothreaded, many tasks are actually executing in parallel at any given time. Should the tasks conflict in what data they have to modify, then one will either have to wait for the other to finish or, in a worst-case situation, actually abandon all the work it had done up to that point and try again later.

For these reasons, an application will want to create other Managed Objects of its own. Luckily, that's easy to do!

All Managed Objects must meet two criteria:

- They must be **Serializable**.
- They must implement the **ManagedObject** marker interface. (**AppListener** actually inherits the **ManagedObject** marker interface for you.)

One good way to break your application up into multiple Managed Objects is by the events they handle. A Managed Object can handle only one event at a time, so you want to separate all event handlers for events that might occur in parallel into separate Managed Objects. Below is the code to **HelloPersistence2**. It creates a separate **TrivialTimedTask** Managed Object from the **AppListener** to handle the timed task.

## HelloPersistence2

```
/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 *
 * This file is part of Project Darkstar Server.
 *
 * Project Darkstar Server is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation and
 * distributed hereunder to you.
 *
 * Project Darkstar Server is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package com.sun.sgs.tutorial.server.lesson4;

import java.io.Serializable;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.sun.sgs.app.AppContext;
import com.sun.sgs.app.AppListener;
import com.sun.sgs.app.ClientSession;
import com.sun.sgs.app.ClientSessionListener;
import com.sun.sgs.app.TaskManager;

/**
 * A simple persistence example for the Project Darkstar Server.
 */
public class HelloPersistence2
    implements AppListener, Serializable
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /** The {@link Logger} for this class. */
    private static final Logger logger =
        Logger.getLogger(HelloPersistence2.class.getName());

    /** The delay before the first run of the task. */
    public static final int DELAY_MS = 5000;

    /** The time to wait before repeating the task. */
    public static final int PERIOD_MS = 500;

    // implement AppListener

    /**
     * {@inheritDoc}
     * <p>
     * Creates a {@link TrivialTimedTask} and schedules its {@code run()}
     * method to be called periodically.
     * <p>
     * Since SGS tasks are persistent, the scheduling only needs to
     * be done the first time the application is started. When the

```



```

    * server is killed and restarted, the scheduled timer task will
    * continue ticking.
    * <p>
    * Runs the task {@value #DELAY_MS} ms from now,
    * repeating every {@value #PERIOD_MS} ms.
    */
    public void initialize(Properties props) {
        TrivialTimedTask task = new TrivialTimedTask();

        logger.log(Level.INFO, "Created task: {0}", task);

        TaskManager taskManager = ApplicationContext.getTaskManager();
        taskManager.schedulePeriodicTask(task, DELAY_MS, PERIOD_MS);
    }

    /**
     * {@inheritDoc}
     * <p>
     * Prevents client logins by returning {@code null}.
     */
    public ClientSessionListener loggedIn(ClientSession session) {
        return null;
    }
}

```

## TrivialTimedTask

This is the Managed Object we are going to have respond to the repeating task.

```

/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 *
 * This file is part of Project Darkstar Server.
 *
 * Project Darkstar Server is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation and
 * distributed hereunder to you.
 *
 * Project Darkstar Server is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package com.sun.sgs.tutorial.server.lesson4;

import com.sun.sgs.app.AppContext;
import java.io.Serializable;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.sun.sgs.app.ManagedObject;
import com.sun.sgs.app.Task;

/**
 * A simple repeating Task that tracks and prints the time since it was
 * last run.
 */

```

```

    */
    public class TrivialTimedTask
        implements Serializable, // for persistence, as required by ManagedObject.
            ManagedObject, // to let the SGS manage our persistence.
            Task // to schedule future calls to our run() method.
    {
        /** The version of the serialized form of this class. */
        private static final long serialVersionUID = 1L;

        /** The {@link Logger} for this class. */
        private static final Logger logger =
            Logger.getLogger(TrivialTimedTask.class.getName());

        /** The timestamp when this task was last run. */
        private long lastTimestamp = System.currentTimeMillis();

        // implement Task

        /**
         * {@inheritDoc}
         * <p>
         * Each time this {@code Task} is run, logs the current timestamp and
         * the delta from the timestamp of the previous run.
         */
        public void run() throws Exception {
            // We will be modifying this object.
            ApplicationContext.getDataManager().markForUpdate(this);
            long timestamp = System.currentTimeMillis();
            long delta = timestamp - lastTimestamp;

            // Update the field holding the most recent timestamp.
            lastTimestamp = timestamp;

            logger.log(Level.INFO,
                "timestamp = {0,number,#}, delta = {1,number,#}",
                new Object[] { timestamp, delta }
            );
        }
    }
}

```

---

## Coding HelloPersistence3

---

A Managed Object does not actually become managed by the Data Manager, and thus persistent, until the Data Manager is made aware of it. The reason **HelloPersistence2** works is because the Task Manager persisted the **TrivialTimedTask** object for us. In order to persist other Managed Objects, though, an application needs to take on the responsibility of informing the Data Manager itself. One way the Data Manager can become aware of a Managed Object is through a request for a Managed Reference.

Managed Objects often need to refer to other Managed Objects. This is done with a Managed Reference. *It is very important that the only fields on one Managed Object that reference another Managed Object be Managed References.* This is how the Data Manager knows that it is a reference to a separate Managed Object. If you store a simple Java reference to the second Managed Object in a field on the first Managed Object, the second object will become part of the first object's state when the first object is stored. The result will be that, the next time the first object tries to access the second, it will get its own local copy and not the real second Managed Object.

**HelloPersistence3** below illustrates this by creating a second persistent object that is called from the **TrivialTimedTask** and that keeps the last-called time as part of its persistent state.

## HelloPersistence3

**HelloPersistence3**, below, is a task that delegates to a sub-task (a **TrivialTimedTask** that is not scheduled to run on its own). The sub-task is stored in a Managed Reference on **HelloPersistence3**.

```
/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 *
 * This file is part of Project Darkstar Server.
 *
 * Project Darkstar Server is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation and
 * distributed hereunder to you.
 *
 * Project Darkstar Server is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package com.sun.sgs.tutorial.server.lesson4;

import java.io.Serializable;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.sun.sgs.app.AppContext;
import com.sun.sgs.app.AppListener;
import com.sun.sgs.app.ClientSession;
import com.sun.sgs.app.ClientSessionListener;
import com.sun.sgs.app.DataManager;
import com.sun.sgs.app.ManagedReference;
import com.sun.sgs.app.Task;
import com.sun.sgs.app.TaskManager;

/**
 * A simple persistence example for the Project Darkstar Server.
 */
public class HelloPersistence3
    implements AppListener, Serializable, Task
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /** The {@link Logger} for this class. */
    private static final Logger logger =
        Logger.getLogger(HelloPersistence3.class.getName());

    /** The delay before the first run of the task. */
    public static final int DELAY_MS = 5000;

    /** The time to wait before repeating the task. */
    public static final int PERIOD_MS = 500;

    /** A reference to our subtask, a {@link TrivialTimedTask}. */
    private ManagedReference<TrivialTimedTask> subTaskRef = null;

    /**
```

```

* Gets the subtask this task delegates to. Dereferences a
* {@link ManagedReference} in this object that holds the subtask.
* <p>
* This null-check idiom is common when getting a ManagedReference.
*
* @return the subtask this task delegates to, or null if none is set
*/
public TrivialTimedTask getSubTask() {
    if (subTaskRef == null)
        return null;

    return subTaskRef.get();
}

/**
* Sets the subtask this task delegates to. Stores the subtask
* as a {@link ManagedReference} in this object.
* <p>
* This null-check idiom is common when setting a ManagedReference,
* since {@link DataManager#createReference createReference} does
* not accept null parameters.
*
* @param subTask the subtask this task should delegate to,
* or null to clear the subtask
*/
public void setSubTask(TrivialTimedTask subTask) {
    if (subTask == null) {
        subTaskRef = null;
        return;
    }
    DataManager dataManager = ApplicationContext.getDataManager();
    subTaskRef = dataManager.createReference(subTask);
}

// implement AppListener

/**
* {@inheritDoc}
* <p>
* Schedules the {@code run()} method of this object to be called
* periodically.
* <p>
* Since SGS tasks are persistent, the scheduling only needs to
* be done the first time the application is started. When the
* server is killed and restarted, the scheduled timer task will
* continue ticking.
* <p>
* Runs the task {@value #DELAY_MS} ms from now,
* repeating every {@value #PERIOD_MS} ms.
*/
public void initialize(Properties props) {
    // Hold onto the task (as a managed reference)
    setSubTask(new TrivialTimedTask());

    TaskManager taskManager = ApplicationContext.getTaskManager();
    taskManager.schedulePeriodicTask(this, DELAY_MS, PERIOD_MS);
}

/**
* {@inheritDoc}
* <p>
* Prevents client logins by returning {@code null}.
*/

```

```

    public ClientSessionListener loggedIn(ClientSession session) {
        return null;
    }

    // implement Task

    /**
     * {@inheritDoc}
     * <p>
     * Calls the run() method of the subtask set on this object.
     */
    public void run() throws Exception {
        // Get the subTask (from the ManagedReference that holds it)
        TrivialTimedTask subTask = getSubTask();

        if (subTask == null) {
            logger.log(Level.WARNING, "subTask is null");
            return;
        }

        // Delegate to the subTask's run() method
        subTask.run();
    }
}

```

Another way to register a Managed Object to the Data Manager is with the **setBinding** call. This call does not return a Managed Reference, but instead binds the Managed Object to the string passed in with it to the call. Once a Managed Object has a name bound to it, the Managed Object may be retrieved by passing the same name to the **getBinding** call. Note that name bindings must be distinct. For each unique string used as a name binding by an application, there can be one and only one Managed Object bound.

Retrieving a Managed Object by its binding has some additional overhead, so it's better to keep Managed References to Managed Objects in the other Managed Objects that need to call them. There are, however, some problems that are best solved with a name-binding convention; one common example is finding the player object for a particular player at the start of his or her session.

There are a number of other interesting methods on the Data Manager. You might want to look at the Javadoc now, but discussion of them will be put off until required by the tutorial applications.

# Lesson 5: Hello User!

---

Up till now the tutorial lessons have focused on getting your logic up and running in the PDS. But there is another side to the online game equation — the users and their computers. This lesson shows how to start communicating between clients and the PDS.

In this tutorial, the server side of that communication will be explained and illustrated using a simple pre-built client. For the client-side coding, please see the *Project Darkstar Client Tutorial*.

---

## Knowing When a User Logs In

---

The first step in communicating with users is knowing who is available to communicate with. The PDS provides a callback method on the **AppListener** for this: **loggedIn**. The **loggedIn** method gets passed an object that describes the user; this object is called a **ClientSession**.<sup>11</sup>

Below is the code for **HelloUser**, a trivial application that logs the login of a user.

### HelloUser

```
/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 *
 * This file is part of Project Darkstar Server.
 *
 * Project Darkstar Server is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation and
 * distributed hereunder to you.
 *
 * Project Darkstar Server is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package com.sun.sgs.tutorial.server.lesson5;

import java.io.Serializable;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.sun.sgs.app.AppListener;
import com.sun.sgs.app.ClientSession;
import com.sun.sgs.app.ClientSessionListener;

/**
 * Simple example of listening for user {@linkplain AppListener#loggedIn login}
 * in the Project Darkstar Server.
 * <p>
 * Logs each time a user logs in, then kicks them off immediately.
 */
```

---

<sup>11</sup> In fact, **ClientSession** describes the new connection session, the user being one of those parameters. This distinction is important, in that you cannot save a **ClientSession** object and expect it to be valid after the session has ended, which is when the user disconnects.

```

    */
public class HelloUser
    implements AppListener, // to get called during startup and login.
               Serializable // since all AppListeners are ManagedObjects.
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /** The {@link Logger} for this class. */
    private static final Logger logger =
        Logger.getLogger(HelloUser.class.getName());

    // implement AppListener

    /** {@inheritDoc} */
    public void initialize(Properties props) {
        // empty
    }

    /**
     * {@inheritDoc}
     * <p>
     * Logs a message each time a new session tries to login, then
     * kicks them out by returning {@code null}.
     */
    public ClientSessionListener loggedIn(ClientSession session) {
        // User has logged in
        logger.log(Level.INFO, "User {0} almost logged in", session.getName());

        // Kick the user out immediately by returning a null listener
        return null;
    }
}

```

---

## Direct Communication

---

You will note that, when you run the server application above and connect to it with a client, the client is immediately logged out. This is because we are returning null from **loggedIn**. The PDS interprets this as our rejecting the user. To accept the user and allow him or her to stay logged in, you need to return a valid **ClientSessionListener**. To be valid, this object must implement both **ClientSessionListener** and **Serializable**. Below is **HelloUser2**, which does this.

## HelloUser2

**HelloUser2** is identical to **HelloUser** except for the **loggedIn** method:

```

/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 *
 * This file is part of Project Darkstar Server.
 *
 * Project Darkstar Server is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation and
 * distributed hereunder to you.
 *
 * Project Darkstar Server is distributed in the hope that it will be useful,

```

```

* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program. If not, see <http://www.gnu.org/licenses/>.
*/

package com.sun.sgs.tutorial.server.lesson5;

import java.io.Serializable;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.sun.sgs.app.AppListener;
import com.sun.sgs.app.ClientSession;
import com.sun.sgs.app.ClientSessionListener;

/**
 * Simple example of listening for user {@linkplain AppListener#loggedIn login}
 * in the Project Darkstar Server.
 * <p>
 * Logs each time a user logs in, and sets their listener to a
 * new {@link HelloUserSessionListener}.
 */
public class HelloUser2
    implements AppListener, // to get called during startup and login.
               Serializable // since all AppListeners are ManagedObjects.
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /** The {@link Logger} for this class. */
    private static final Logger logger =
        Logger.getLogger(HelloUser2.class.getName());

    // implement AppListener

    /** {@inheritDoc} */
    public void initialize(Properties props) {
        // empty
    }

    /**
     * {@inheritDoc}
     * <p>
     * Logs a message each time a new session logs in.
     */
    public ClientSessionListener loggedIn(ClientSession session) {
        // User has logged in
        logger.log(Level.INFO, "User {0} has logged in", session.getName());

        // Return a valid listener
        return new HelloUserSessionListener(session);
    }
}

```

## HelloUserSessionListener

```

/*
 * Copyright 2007-2008 Sun Microsystems, Inc.

```



```

*
* This file is part of Project Darkstar Server.
*
* Project Darkstar Server is free software: you can redistribute it
* and/or modify it under the terms of the GNU General Public License
* version 2 as published by the Free Software Foundation and
* distributed hereunder to you.
*
* Project Darkstar Server is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program. If not, see <http://www.gnu.org/licenses/>.
*/

package com.sun.sgs.tutorial.server.lesson5;

import java.io.Serializable;
import java.nio.ByteBuffer;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.sun.sgs.app.AppContext;
import com.sun.sgs.app.ClientSession;
import com.sun.sgs.app.ClientSessionListener;
import com.sun.sgs.app.ManagedReference;

/**
 * Simple example {@link ClientSessionListener} for the Project Darkstar
 * Server.
 * <p>
 * Logs each time a session receives data or logs out.
 */
class HelloUserSessionListener
    implements Serializable, ClientSessionListener
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /** The {@link Logger} for this class. */
    private static final Logger logger =
        Logger.getLogger(HelloUserSessionListener.class.getName());

    /** The session this {@code ClientSessionListener} is listening to. */
    private final ManagedReference<ClientSession> sessionRef;

    /** The name of the {@code ClientSession} for this listener */
    private final String sessionName;

    /**
     * Creates a new {@code HelloUserSessionListener} for the given session.
     *
     * @param session the session this listener is associated with
     */
    public HelloUserSessionListener(ClientSession session) {
        if (session == null)
            throw new NullPointerException("null session");

        sessionRef = AppContext.getDataManager().createReference(session);
        sessionName = session.getName();
    }
}

```

```

/**
 * Returns the session for this listener.
 *
 * @return the session for this listener
 */
protected ClientSession getSession() {
    // We created the ref with a non-null session, so no need to check it.
    return sessionRef.get();
}

/**
 * {@inheritDoc}
 * <p>
 * Logs when data arrives from the client.
 */
public void receivedMessage(ByteBuffer message) {
    logger.log(Level.INFO, "Message from {0}", sessionName);
}

/**
 * {@inheritDoc}
 * <p>
 * Logs when the client disconnects.
 */
public void disconnected(boolean graceful) {
    String grace = graceful ? "graceful" : "forced";
    logger.log(Level.INFO,
        "User {0} has logged out {1}",
        new Object[] { sessionName, grace }
    );
}
}

```

**HelloUserSessionListener** is a glue object that listens for either data from the user or the disconnect of the user; it allows our server code to respond to these events. So far, all we do is log some information, but in a complete PDS application, these would both be important events to which we would want to respond.

There are two kinds of communication in the PDS:

- Direct Communication
- Channel Communication

*Direct Communication* is built into the core of the system and provides a pipe for the flow of data between a single user client and its PDS application.

*Channel Communication* is provided by a standard manager, the Channel Manager, and provides for publish/subscribe group communications. While there is nothing in the Channel Manager's functionality that could not be implemented on top of the Direct Communication mechanisms, putting the channel functionality in a manager allows for a much more efficient implementation.

The **HelloEcho** PDS application echoes back to the user anything the user sends to the application. Besides the name, there is only one line difference in **HelloEchoSessionListener** from **HelloUserSessionListener**: the addition of a **session.send** call.

## HelloEchoSessionListener

```

/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 */

```

```

* This file is part of Project Darkstar Server.
*
* Project Darkstar Server is free software: you can redistribute it
* and/or modify it under the terms of the GNU General Public License
* version 2 as published by the Free Software Foundation and
* distributed hereunder to you.
*
* Project Darkstar Server is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with this program. If not, see <http://www.gnu.org/licenses/>.
*/

package com.sun.sgs.tutorial.server.lesson5;

import java.io.Serializable;
import java.nio.ByteBuffer;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.sun.sgs.app.AppContext;
import com.sun.sgs.app.ClientSession;
import com.sun.sgs.app.ClientSessionListener;
import com.sun.sgs.app.ManagedReference;

/**
 * Simple example {@link ClientSessionListener} for the Project Darkstar
 * Server.
 * <p>
 * Logs each time a session receives data or logs out, and echoes
 * any data received back to the sender.
 */
class HelloEchoSessionListener
    implements Serializable, ClientSessionListener
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /** The {@link Logger} for this class. */
    private static final Logger logger =
        Logger.getLogger(HelloEchoSessionListener.class.getName());

    /** The session this {@code ClientSessionListener} is listening to. */
    private final ManagedReference<ClientSession> sessionRef;

    /** The name of the {@code ClientSession} for this listener */
    private final String sessionName;

    /**
     * Creates a new {@code HelloEchoSessionListener} for the given session.
     *
     * @param session the session this listener is associated with
     */
    public HelloEchoSessionListener(ClientSession session) {
        if (session == null)
            throw new NullPointerException("null session");

        sessionRef = AppContext.getDataManager().createReference(session);
        sessionName = session.getName();
    }
}

```

```

/**
 * Returns the session for this listener.
 *
 * @return the session for this listener
 */
protected ClientSession getSession() {
    // We created the ref with a non-null session, so no need to check it.
    return sessionRef.get();
}

/**
 * {@inheritDoc}
 * <p>
 * Logs when data arrives from the client, and echoes the message back.
 */
public void receivedMessage(ByteBuffer message) {
    ClientSession session = getSession();

    logger.log(Level.INFO, "Message from {0}", sessionName);

    // Echo message back to sender
    session.send(message);
}

/**
 * {@inheritDoc}
 * <p>
 * Logs when the client disconnects.
 */
public void disconnected(boolean graceful) {
    ClientSession session = getSession();

    String grace = graceful ? "graceful" : "forced";

    logger.log(Level.INFO,
        "User {0} has logged out {1}",
        new Object[] { sessionName, grace }
    );
}
}

```

---

## Running the Examples

---

To try all the examples in this part of the server tutorial, you need a simple client capable of logging in, as well as direct client/server communication. You can find this client as part of Lesson 1 of the *Project Darkstar Client Tutorial* (**com.sun.sgs.tutorial.client.lesson1.HelloUserClient** in the **tutorial-client.jar** file).

## Lesson 6: Hello Channels!

---

The previous lessons have introduced the Task Manager and Data Manager. The final standard manager is the *Channel Manager*. The core of the PDS provides us with basic client/server communications. For simple games, this may be enough. However, for games that organize players into groups, either to isolate game sessions (such as in many casual and fast action games), or to tame the n-squared user-to-user communications scaling issues inherent in massive numbers of simultaneous players, something with lower overhead and more control is required.

The Channel Manager provides publish/subscribe channels. The server application can create these channels and then assign users to one or more of them. Communication between users in a channel does not involve the Task or Data Manager.

---

## Coding HelloChannels

---

The **HelloChannels** Managed Object is similar to our previous **AppListener** implementations with the addition that it opens two reliable channels, **Foo** and **Bar**.

### HelloChannels

```
/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 *
 * This file is part of Project Darkstar Server.
 *
 * Project Darkstar Server is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation and
 * distributed hereunder to you.
 *
 * Project Darkstar Server is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package com.sun.sgs.tutorial.server.lesson6;

import java.io.Serializable;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.sun.sgs.app.AppContext;
import com.sun.sgs.app.AppListener;
import com.sun.sgs.app.Channel;
import com.sun.sgs.app.ChannelManager;
import com.sun.sgs.app.ClientSession;
import com.sun.sgs.app.ClientSessionListener;
import com.sun.sgs.app.Delivery;
import com.sun.sgs.app.ManagedReference;

/**
 * Simple example of channel operations in the Project Darkstar Server.
 */
```

```

* <p>
* Extends the {@code HelloEcho} example by joining clients to two
* channels.
*/
public class HelloChannels
    implements Serializable, AppListener
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /** The {@link Logger} for this class. */
    private static final Logger logger =
        Logger.getLogger(HelloChannels.class.getName());

    /** The name of the first channel {@value #CHANNEL_1_NAME} */
    static final String CHANNEL_1_NAME = "Foo";
    /** The name of the second channel {@value #CHANNEL_2_NAME} */
    static final String CHANNEL_2_NAME = "Bar";

    /**
     * The first {@link Channel}. The second channel is looked up
     * by name.
     */
    private ManagedReference<Channel> channel1 = null;

    /**
     * {@inheritDoc}
     * <p>
     * Creates the channels. Channels persist across server restarts,
     * so they only need to be created here in {@code initialize}.
     */
    public void initialize(Properties props) {
        ChannelManager channelMgr = AppContext.getChannelManager();

        // Create and keep a reference to the first channel.
        Channel c1 = channelMgr.createChannel(CHANNEL_1_NAME,
                                             null,
                                             Delivery.RELIABLE);
        channel1 = AppContext.getDataManager().createReference(c1);

        // We don't keep a reference to the second channel, to demonstrate
        // looking it up by name when needed. Also, this channel uses a
        // {@link ChannelListener} to filter messages.
        channelMgr.createChannel(CHANNEL_2_NAME,
                                new HelloChannelsChannelListener(),
                                Delivery.RELIABLE);
    }

    /**
     * {@inheritDoc}
     * <p>
     * Returns a {@link HelloChannelsSessionListener} for the
     * logged-in session.
     */
    public ClientSessionListener loggedIn(ClientSession session) {
        logger.log(Level.INFO, "User {0} has logged in", session.getName());
        return new HelloChannelsSessionListener(session, channel1);
    }
}

```

The **HelloChannelsSessionListener** is identical to **HelloEchoSessionListener** except for the constructor. When we create the session listener, we also join its session to two channels. One channel is passed in, while the second is looked up by name.

The first **channel.join** is passed null for a **ChannelListener**, so all communication on it is only received by clients. The second channel joined however is given a channel listener. This will be called back whenever a message from this session is posted to that channel. The listener can examine the message and sender and decide to discard the message, send a different message to the channel, or send the original, unmodified message to the channel.

Note that, with this code, each session has its own listener for messages on the second channel. This is preferable to registering a single channel-wide listener, since messages from different clients can be processed in parallel. However, if your design really requires a single listener to all messages sent by any client on a channel, you would declare the listener as the second parameter to the **createChannel** call.

## HelloChannelsSessionListener

```
/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 *
 * This file is part of Project Darkstar Server.
 *
 * Project Darkstar Server is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation and
 * distributed hereunder to you.
 *
 * Project Darkstar Server is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package com.sun.sgs.tutorial.server.lesson6;

import java.io.Serializable;
import java.nio.ByteBuffer;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.sun.sgs.app.AppContext;
import com.sun.sgs.app.Channel;
import com.sun.sgs.app.ChannelManager;
import com.sun.sgs.app.ClientSession;
import com.sun.sgs.app.ClientSessionListener;
import com.sun.sgs.app.DataManager;
import com.sun.sgs.app.ManagedReference;

/**
 * Simple example {@link ClientSessionListener} for the Project Darkstar
 * Server.
 * <p>
 * Logs each time a session receives data or logs out, and echoes
 * any data received back to the sender.
 */
class HelloChannelsSessionListener
    implements Serializable, ClientSessionListener
{
```

```

/** The version of the serialized form of this class. */
private static final long serialVersionUID = 1L;

/** The {@link Logger} for this class. */
private static final Logger logger =
    Logger.getLogger(HelloChannelsSessionListener.class.getName());

/** The session this {@code ClientSessionListener} is listening to. */
private final ManagedReference<ClientSession> sessionRef;

/** The name of the {@code ClientSession} for this listener */
private final String sessionName;

/**
 * Creates a new {@code HelloChannelsSessionListener} for the session.
 *
 * @param session the session this listener is associated with
 * @param channel1 a reference to a channel to join
 */
public HelloChannelsSessionListener(ClientSession session,
    ManagedReference<Channel> channel1)
{
    if (session == null)
        throw new NullPointerException("null session");

    DataManager dataMgr = AppContext.getDataManager();
    sessionRef = dataMgr.createReference(session);
    sessionName = session.getName();

    // Join the session to all channels. We obtain the channel
    // in two different ways, by reference and by name.
    ChannelManager channelMgr = AppContext.getChannelManager();

    // We were passed a reference to the first channel.
    channel1.get().join(session);

    // We look up the second channel by name.
    Channel channel2 = channelMgr.getChannel(HelloChannels.CHANNEL_2_NAME);
    channel2.join(session);
}

/**
 * Returns the session for this listener.
 *
 * @return the session for this listener
 */
protected ClientSession getSession() {
    // We created the ref with a non-null session, so no need to check it.
    return sessionRef.get();
}

/**
 * {@inheritDoc}
 * <p>
 * Logs when data arrives from the client, and echoes the message back.
 */
public void receivedMessage(ByteBuffer message) {
    ClientSession session = getSession();

    if (logger.isLoggable(Level.FINE)) {
        logger.log(Level.FINE, "Message from {0}", sessionName);
    }
    session.send(message);
}

```



```

    }

    /**
     * {@inheritDoc}
     * <p>
     * Logs when the client disconnects.
     */
    public void disconnected(boolean graceful) {
        String grace = graceful ? "graceful" : "forced";
        logger.log(Level.INFO,
            "User {0} has logged out {1}",
            new Object[] { sessionName, grace }
        );
    }
}

```

**HelloChannelsChannelListener** is a simple skeletal listener that just logs what it receives.

## HelloChannelsChannelListener

```

/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 *
 * This file is part of Project Darkstar Server.
 *
 * Project Darkstar Server is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation and
 * distributed hereunder to you.
 *
 * Project Darkstar Server is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package com.sun.sgs.tutorial.server.lesson6;

import java.io.Serializable;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.sun.sgs.app.Channel;
import com.sun.sgs.app.ChannelListener;
import com.sun.sgs.app.ClientSession;
import java.nio.ByteBuffer;

/**
 * Simple example {@link ChannelListener} for the Project Darkstar Server.
 * <p>
 * Logs when a channel receives data.
 */
class HelloChannelsChannelListener
    implements Serializable, ChannelListener
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

```

```

/** The {@link Logger} for this class. */
private static final Logger logger =
    Logger.getLogger(HelloChannelsChannelListener.class.getName());

/**
 * {@inheritDoc}
 * <p>
 * Logs when data arrives on a channel. A typical listener would
 * examine the message to decide whether it should be discarded,
 * modified, or sent unchanged.
 */
public void receivedMessage(Channel channel,
                           ClientSession session,
                           ByteBuffer message)
{
    if (logger.isLoggable(Level.INFO)) {
        logger.log(Level.INFO,
            "Channel message from {0} on channel {1}",
            new Object[] { session.getName(), channel.getName() }
        );
    }
    channel.send(session, message);
}
}

```

---

## Running HelloChannels

---

To try **HelloChannels** you need a client that will connect and allow you to talk on selected channels. One is provided in Lesson 2 of the *Project Darkstar Client Tutorial* (**com.sun.sgs.tutorial.client.lesson2.HelloChannelClient** in **tutorial-client.jar**).

# Conclusion

---

At this point you know all the basics of writing PDS applications. The applications you write using the PDS API on a single-node system will operate unmodified in exactly the same way on a large-scale multiple-node PDS production back end. In that environment they will scale out horizontally, handle failover, and be fault-tolerant.

There are, however, some best practices to follow to ensure optimal scalability for your application. Failing to follow these can seriously limit how many users your application will be able to support at once.

---

## Best Practices

---

- **Do not design with bottleneck Managed Objects.**

You can think of each Managed Object as an independent worker who can only do one task at a time. When one task is modifying the state of a Managed Object, any other tasks that want to read or change its state must wait. A Managed Object may have many readers at once if no one is writing to it, but any writing turns it into a potential bottleneck. In general, a pattern of one writer and many readers is the best configuration, although it is not always possible.

In the worst case, multiple lockers of the same objects will cause potential deadlock situations that are computationally more expensive to resolve and can result in processing delays (additional latency).

- **Avoid contending for object access.**

Although the PDS will detect and resolve contentions for write access to objects between parallel events such that your code never actually stops processing, this deadlock avoidance can significantly load the CPU and slow response time to the users. In the worst case, contention can actually prevent parallel processing, seriously impacting the scalability of your code.

A classic example of a deadlock is two combatants who have to modify both their own data and that of their enemies when resolving an attack. If all the data is on a combatant Managed Object, then both combat tasks need to lock both objects, typically in the opposite order, which is a formula for deadlock. One solution is to divide the combatants' combat statistics into two groups, the ones they modify themselves and the ones modified by an attacker, and store them on separate Managed Objects. The combatant object would then retain Managed References to these objects, and only lock them as needed.

Another practical solution is partial serialization of the problem. This is especially appropriate if you want the combatants attacks to be processed in a fixed order. Here you have a single “combat clock” task (generally a repeating task) that processes in sequence all the attacks from all the combatants in a given battle.

Not all contention situations are this easy to spot and design around. The PDS gives you feedback at runtime as to what objects are deadlocking. Use this to tune your application.

- **Give all serializable objects a fixed serialVersionUID.**

The PDS uses Serialization internally. If you don't give your **Serializable** classes a **serialVersionUID**, then any change to their public interface could invalidate the stored copies, leading to a need to delete the entire Object Store. Giving them a fixed **serialVersionUID** will allow you to make “compatible changes” without invalidating your existing store. (For what constitutes a compatible change, please see the JDK™ documents on Serialization.)

- **Avoid inner classes in Serializable objects.**

Serialization of objects whose classes contain non-static inner classes gets complicated. It is best to avoid inner classes, including anonymous inner classes. If you must use them, the safest thing is to declare them as static inner classes.<sup>12</sup>

---

<sup>12</sup> In terms of code generated, a static inner class is no different from an outer class, except that its classname exists in the

- **Do not create non-final static fields on Managed Objects.**

The most important reason for this is that static fields exist only within the scope of a single VM, and a PDS back end floats Managed Objects between many different VMs.

- **Do not use the synchronized keyword in Managed Objects.**

First, this is unnecessary in a PDS application. Synchronization is used to prevent contention over data between multiple parallel threads of control. The PDS programming model handles this transparently for you. Second, it won't work, since synchronization is relative to a single VM and a full PDS back-end operates over many VM instances simultaneously. Finally, it causes interactions between tasks that can defeat the system's deadlock-proofing feature and actually cause your code to lock up.

Any task that locks up for too long will be forced by the PDS to yield its control of Managed Objects back to the system, but this is a last-ditch safety feature and will result in significant delays in code execution.

- **Do not make blocking I/O calls or stay in a loop for a long period.**

The system contains the assumption that tasks are short-lived. If a task lives too long, it will be forcibly terminated by the back end. The right way to do blocking I/O and the like is to create an extension manager, do the blocking calls in it, and submit a task to the Task Manager to handle the results when done. (Writing and installing custom managers will be covered in the *Project Darkstar Server Extension Manual*.)

- **Do not catch java.lang.Exception.**

Instead, catch the explicit exceptions you are expecting. The PDS also uses exceptions to communicate exceptional states to the execution environment. Although the system does its best to do the right thing even if you hide these exceptions from it, it will operate more efficiently if you don't.

- **Do not carry a non-transient Java reference on a Managed Object to another Managed Object.**

Instead, use a Managed Reference. Any object that is referred to by a Java reference chain that starts at the Managed Object is assumed to be part of the private state of that particular Managed Object. This means that, while you may set two Java references on two different Managed Objects to the same Java object during a task, they will each end up with their own copy of that object at the termination of the task.

- **Never try to save a Manager Instance on a non-transient field of a Managed Object.**

This is because Manager References are only valid for the life of the task that fetched them. Manager instances are *not* serializable objects. Any attempt to save a reference to one in your Managed Object will cause the Data Manager to throw a non-retriable exception and the entire task to be abandoned.

- **Carefully manage the life cycle of your Managed Objects.**

Remember that the Object Store does no garbage collection for you. Managed Objects are “real objects” in the simulation sense. They don't exist until explicitly created, exist in one and only one state at any given time, and persist until explicitly destroyed. These are very good properties from a simulation programming stance, but if you go wild creating Managed Objects and don't destroy them when they are no longer useful, you can load down the Object Store with garbage and potentially impact performance.

Be aware that any PDS API call that accepts a Managed Object may create a Managed Reference to that object. If this is the first time the Data Manager has been made aware of the Managed Object, this will result in the Managed Object being added to the Object Store. It is still the developer's responsibility to remove the Managed Object from the Object Store when it is no longer in use. For this reason, it is best to avoid passing Managed Objects that your application is not explicitly managing into the PDS APIs.

---

name-space of its outer class.

---

## Things Not Covered in This Tutorial

---

This tutorial is intended to introduce you to the fundamentals of coding a PDS application. Although we present all basic uses of the standard managers, these managers have additional functions and capabilities not covered here. Please see the Javadoc for those other functions.

As discussed, the list of managers in a given PDS back end is extensible. This tutorial does not cover writing or using plug-in managers. For that, see the forthcoming document on PDS extensions.

Although we use PDS client programs in Lessons 5 and 6, this tutorial does not explain how those are written. For those explanations, please see the *Project Darkstar Client Tutorial*.

Finally, for the sake of clarity, this tutorial shows very simple examples. For more complex patterns of PDS usage, please see the community examples at [www.projectdarkstar.com](http://www.projectdarkstar.com).



# Appendix A: SwordWorld Example Code

---

## Sword World

```
/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 *
 * This file is part of Project Darkstar Server.
 *
 * Project Darkstar Server is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation and
 * distributed hereunder to you.
 *
 * Project Darkstar Server is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package com.sun.sgs.tutorial.server.swordworld;

import java.io.Serializable;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.sun.sgs.app.AppContext;
import com.sun.sgs.app.AppListener;
import com.sun.sgs.app.ClientSession;
import com.sun.sgs.app.ClientSessionListener;
import com.sun.sgs.app.DataManager;
import com.sun.sgs.app.ManagedReference;

/**
 * A tiny sample MUD application for the Project Darkstar Server.
 * <p>
 * There is a Room. In the Room there is a Sword...
 */
public class SwordWorld
    implements Serializable, AppListener
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /** The {@link Logger} for this class. */
    private static final Logger logger =
        Logger.getLogger(SwordWorld.class.getName());

    /** A reference to the one-and-only {@link SwordWorldRoom room}. */
    private ManagedReference<SwordWorldRoom> roomRef = null;

    /**
     * {@inheritDoc}
     * <p>

```

```

    * Creates the world within the MUD.
    */
    public void initialize(Properties props) {
        logger.info("Initializing SwordWorld");

        // Create the Room
        SwordWorldRoom room =
            new SwordWorldRoom("Plain Room", "a nondescript room");

        // Create the Sword
        SwordWorldObject sword =
            new SwordWorldObject("Shiny Sword", "a shiny sword.");

        // Put the Sword to the Room
        room.addItem(sword);

        // Keep a reference to the Room
        setRoom(room);

        logger.info("SwordWorld Initialized");
    }

    /**
     * Gets the SwordWorld's One True Room.
     * <p>
     * @return the room for this {@code SwordWorld}
     */
    public SwordWorldRoom getRoom() {
        if (roomRef == null)
            return null;

        return roomRef.get();
    }

    /**
     * Sets the SwordWorld's One True Room to the given room.
     * <p>
     * @param room the room to set
     */
    public void setRoom(SwordWorldRoom room) {
        DataManager dataManager = AppContext.getDataManager();
        dataManager.markForUpdate(this);

        if (room == null) {
            roomRef = null;
            return;
        }

        roomRef = dataManager.createReference(room);
    }

    /**
     * {@inheritDoc}
     * <p>
     * Obtains the {@linkplain SwordWorldPlayer player} for this
     * {@linkplain ClientSession session}'s user, and puts the
     * player into the One True Room for this {@code SwordWorld}.
     */
    public ClientSessionListener loggedIn(ClientSession session) {
        logger.log(Level.INFO,
            "SwordWorld Client login: {0}", session.getName());

        // Delegate to a factory method on SwordWorldPlayer,

```



```
        // since player management really belongs in that class.  
        SwordWorldPlayer player = SwordWorldPlayer.loggedIn(session);  
  
        // Put player in room  
        player.enter(getRoom());  
  
        // return player object as listener to this client session  
        return player;  
    }  
}
```

## SwordWorldObject

```
/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 *
 * This file is part of Project Darkstar Server.
 *
 * Project Darkstar Server is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation and
 * distributed hereunder to you.
 *
 * Project Darkstar Server is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package com.sun.sgs.tutorial.server.swordworld;

import java.io.Serializable;

import com.sun.sgs.app.AppContext;
import com.sun.sgs.app.ManagedObject;

/**
 * A {@code ManagedObject} that has a name and a description.
 */
public class SwordWorldObject
    implements Serializable, ManagedObject
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /** The name of this object. */
    private String name;

    /** The description of this object. */
    private String description;

    /**
     * Creates a new {@code SwordWorldObject} with the given {@code name}
     * and {@code description}.
     *
     * @param name the name of this object
     * @param description the description of this object
     */
    public SwordWorldObject(String name, String description) {
        this.name = name;
        this.description = description;
    }

    /**
     * Sets the name of this object.
     *
     * @param name the name of this object
     */
    public void setName(String name) {
        AppContext.getDataManager().markForUpdate(this);
    }
}
```

```

        this.name = name;
    }

    /**
     * Returns the name of this object.
     *
     * @return the name of this object
     */
    public String getName() {
        return name;
    }

    /**
     * Sets the description of this object.
     *
     * @param description the description of this object
     */
    public void setDescription(String description) {
        ApplicationContext.getDataManager().markForUpdate(this);
        this.description = description;
    }

    /**
     * Returns the description of this object.
     *
     * @return the description of this object
     */
    public String getDescription() {
        return description;
    }

    /** {@inheritDoc} */
    @Override
    public String toString() {
        return getName();
    }
}

```

## SwordWorldRoom

```
/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 *
 * This file is part of Project Darkstar Server.
 *
 * Project Darkstar Server is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation and
 * distributed hereunder to you.
 *
 * Project Darkstar Server is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package com.sun.sgs.tutorial.server.swordworld;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Set;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.sun.sgs.app.AppContext;
import com.sun.sgs.app.DataManager;
import com.sun.sgs.app.ManagedReference;

/**
 * Represents a room in the {@link SwordWorld} example MUD.
 */
public class SwordWorldRoom extends SwordWorldObject
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /** The {@link Logger} for this class. */
    private static final Logger logger =
        Logger.getLogger(SwordWorldRoom.class.getName());

    /** The set of items in this room. */
    private final Set<ManagedReference<SwordWorldObject>> items =
        new HashSet<ManagedReference<SwordWorldObject>>();

    /** The set of players in this room. */
    private final Set<ManagedReference<SwordWorldPlayer>> players =
        new HashSet<ManagedReference<SwordWorldPlayer>>();

    /**
     * Creates a new room with the given name and description, initially
     * empty of items and players.
     *
     * @param name the name of this room
     * @param description a description of this room
     */
}
```

```

    */
    public SwordWorldRoom(String name, String description) {
        super(name, description);
    }

    /**
     * Adds an item to this room.
     *
     * @param item the item to add to this room.
     * @return {@code true} if the item was added to the room
     */
    public boolean addItem(SwordWorldObject item) {
        logger.log(Level.INFO, "{0} placed in {1}",
            new Object[] { item, this });

        // NOTE: we can't directly save the item in the list, or
        // we'll end up with a local copy of the item. Instead, we
        // must save a ManagedReference to the item.

        DataManager dataManager = AppContext.getDataManager();
        dataManager.markForUpdate(this);

        return items.add(dataManager.createReference(item));
    }

    /**
     * Adds a player to this room.
     *
     * @param player the player to add
     * @return {@code true} if the player was added to the room
     */
    public boolean addPlayer(SwordWorldPlayer player) {
        logger.log(Level.INFO, "{0} enters {1}",
            new Object[] { player, this });

        DataManager dataManager = AppContext.getDataManager();
        dataManager.markForUpdate(this);

        return players.add(dataManager.createReference(player));
    }

    /**
     * Removes a player from this room.
     *
     * @param player the player to remove
     * @return {@code true} if the player was in the room
     */
    public boolean removePlayer(SwordWorldPlayer player) {
        logger.log(Level.INFO, "{0} leaves {1}",
            new Object[] { player, this });

        DataManager dataManager = AppContext.getDataManager();
        dataManager.markForUpdate(this);

        return players.remove(dataManager.createReference(player));
    }

    /**
     * Returns a description of what the given player sees in this room.
     *
     * @param looker the player looking in this room
     * @return a description of what the given player sees in this room
     */

```

```

public String look(SwordWorldPlayer looker) {
    logger.log(Level.INFO, "{0} looks at {1}",
        new Object[] { looker, this });

    StringBuilder output = new StringBuilder();
    output.append("You are in ").append(getDescription()).append(".\n");

    List<SwordWorldPlayer> otherPlayers =
        getPlayersExcluding(looker);

    if (! otherPlayers.isEmpty()) {
        output.append("Also in here are ");
        appendPrettyList(output, otherPlayers);
        output.append(".\n");
    }

    if (! items.isEmpty()) {
        output.append("On the floor you see:\n");
        for (ManagedReference<SwordWorldObject> itemRef : items) {
            SwordWorldObject item = itemRef.get();
            output.append(item.getDescription()).append('\n');
        }
    }

    return output.toString();
}

/**
 * Appends the names of the {@code SwordWorldObject}s in the list
 * to the builder, separated by commas, with an "and" before the final
 * item.
 *
 * @param builder the {@code StringBuilder} to append to
 * @param list the list of items to format
 */
private void appendPrettyList(StringBuilder builder,
    List<? extends SwordWorldObject> list)
{
    if (list.isEmpty())
        return;

    int lastIndex = list.size() - 1;
    SwordWorldObject last = list.get(lastIndex);

    Iterator<? extends SwordWorldObject> it =
        list.subList(0, lastIndex).iterator();
    if (it.hasNext()) {
        SwordWorldObject other = it.next();
        builder.append(other.getName());
        while (it.hasNext()) {
            other = it.next();
            builder.append(" ,");
            builder.append(other.getName());
        }
        builder.append(" and ");
    }
    builder.append(last.getName());
}

/**
 * Returns a list of players in this room excluding the given
 * player.
 */

```

```

    * @param player the player to exclude
    * @return the list of players
    */
    private List<SwordWorldPlayer>
        getPlayersExcluding(SwordWorldPlayer player)
    {
        if (players.isEmpty())
            return Collections.emptyList();

        ArrayList<SwordWorldPlayer> otherPlayers =
            new ArrayList<SwordWorldPlayer>(players.size());

        for (ManagedReference<SwordWorldPlayer> playerRef : players) {
            SwordWorldPlayer other = playerRef.get();
            if (! player.equals(other))
                otherPlayers.add(other);
        }

        return Collections.unmodifiableList(otherPlayers);
    }
}

```

## SwordWorldPlayer

```
/*
 * Copyright 2007-2008 Sun Microsystems, Inc.
 *
 * This file is part of Project Darkstar Server.
 *
 * Project Darkstar Server is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation and
 * distributed hereunder to you.
 *
 * Project Darkstar Server is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <http://www.gnu.org/licenses/>.
 */

package com.sun.sgs.tutorial.server.swordworld;

import java.io.UnsupportedEncodingException;
import java.nio.ByteBuffer;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.sun.sgs.app.AppContext;
import com.sun.sgs.app.ClientSession;
import com.sun.sgs.app.ClientSessionListener;
import com.sun.sgs.app.DataManager;
import com.sun.sgs.app.ManagedReference;
import com.sun.sgs.app.NameNotBoundException;

/**
 * Represents a player in the {@link SwordWorld} example MUD.
 */
public class SwordWorldPlayer
    extends SwordWorldObject
    implements ClientSessionListener
{
    /** The version of the serialized form of this class. */
    private static final long serialVersionUID = 1L;

    /** The {@link Logger} for this class. */
    private static final Logger logger =
        Logger.getLogger(SwordWorldPlayer.class.getName());

    /** The message encoding. */
    public static final String MESSAGE_CHARSET = "UTF-8";

    /** The prefix for player bindings in the {@code DataManager}. */
    protected static final String PLAYER_BIND_PREFIX = "Player.";

    /** The {@code ClientSession} for this player, or null if logged out. */
    private ManagedReference<ClientSession> currentSessionRef = null;

    /** The {@link SwordWorldRoom} this player is in, or null if none. */
    private ManagedReference<SwordWorldRoom> currentRoomRef = null;

    /**
```



```

* Find or create the player object for the given session, and mark
* the player as logged in on that session.
*
* @param session which session to find or create a player for
* @return a player for the given session
*/
public static SwordWorldPlayer loggedIn(ClientSession session) {
    String playerBinding = PLAYER_BIND_PREFIX + session.getName();

    // try to find player object, if non existent then create
    DataManager dataMgr = AppContext.getDataManager();
    SwordWorldPlayer player;

    try {
        player = (SwordWorldPlayer) dataMgr.getBinding(playerBinding);
    } catch (NameNotFoundException ex) {
        // this is a new player
        player = new SwordWorldPlayer(playerBinding);
        logger.log(Level.INFO, "New player created: {0}", player);
        dataMgr.setBinding(playerBinding, player);
    }
    player.setSession(session);
    return player;
}

/**
 * Creates a new {@code SwordWorldPlayer} with the given name.
 *
 * @param name the name of this player
 */
protected SwordWorldPlayer(String name) {
    super(name, "Seeker of the Sword");
}

/**
 * Returns the session for this listener.
 *
 * @return the session for this listener
 */
protected ClientSession getSession() {
    if (currentSessionRef == null)
        return null;

    return currentSessionRef.get();
}

/**
 * Mark this player as logged in on the given session.
 *
 * @param session the session this player is logged in on
 */
protected void setSession(ClientSession session) {
    DataManager dataMgr = AppContext.getDataManager();
    dataMgr.markForUpdate(this);

    currentSessionRef = dataMgr.createReference(session);

    logger.log(Level.INFO,
        "Set session for {0} to {1}",
        new Object[] { this, session });
}

/**

```

```

    * Handles a player entering a room.
    *
    * @param room the room for this player to enter
    */
    public void enter(SwordWorldRoom room) {
        logger.log(Level.INFO, "{0} enters {1}",
            new Object[] { this, room }
        );
        room.addPlayer(this);
        setRoom(room);
    }

    /** {@inheritDoc} */
    public void receivedMessage(ByteBuffer message) {
        String command = decodeString(message);

        logger.log(Level.INFO,
            "{0} received command: {1}",
            new Object[] { this, command }
        );

        if (command.equalsIgnoreCase("look")) {
            String reply = getRoom().look(this);
            getSession().send(encodeString(reply));
        } else {
            logger.log(Level.WARNING,
                "{0} unknown command: {1}",
                new Object[] { this, command }
            );
            // We could disconnect the rogue player at this point.
            //currentSession.disconnect();
        }
    }

    /** {@inheritDoc} */
    public void disconnected(boolean graceful) {
        setSession(null);
        logger.log(Level.INFO, "Disconnected: {0}", this);
        getRoom().removePlayer(this);
        setRoom(null);
    }

    /**
     * Returns the room this player is currently in, or {@code null} if
     * this player is not in a room.
     * <p>
     * @return the room this player is currently in, or {@code null}
     */
    protected SwordWorldRoom getRoom() {
        if (currentRoomRef == null)
            return null;

        return currentRoomRef.get();
    }

    /**
     * Sets the room this player is currently in. If the room given
     * is null, marks the player as not in any room.
     * <p>
     * @param room the room this player should be in, or {@code null}
     */
    protected void setRoom(SwordWorldRoom room) {
        DataManager dataManager = ApplicationContext.getDataManager();

```

```

        dataManager.markForUpdate(this);

        if (room == null) {
            currentRoomRef = null;
            return;
        }

        currentRoomRef = dataManager.createReference(room);
    }

    /** {@inheritDoc} */
    @Override
    public String toString() {
        StringBuilder buf = new StringBuilder(getName());
        buf.append('@');
        if (getSession() == null) {
            buf.append("null");
        } else {
            buf.append(currentSessionRef.getId());
        }
        return buf.toString();
    }

    /**
     * Encodes a {@code String} into a {@link ByteBuffer}.
     *
     * @param s the string to encode
     * @return the {@code ByteBuffer} which encodes the given string
     */
    protected static ByteBuffer encodeString(String s) {
        try {
            return ByteBuffer.wrap(s.getBytes(MESSAGE_CHARSET));
        } catch (UnsupportedEncodingException e) {
            throw new Error("Required character set " + MESSAGE_CHARSET +
                " not found", e);
        }
    }

    /**
     * Decodes a message into a {@code String}.
     *
     * @param message the message to decode
     * @return the decoded string
     */
    protected static String decodeString(ByteBuffer message) {
        try {
            byte[] bytes = new byte[message.remaining()];
            message.get(bytes);
            return new String(bytes, MESSAGE_CHARSET);
        } catch (UnsupportedEncodingException e) {
            throw new Error("Required character set " + MESSAGE_CHARSET +
                " not found", e);
        }
    }
}

```