USERS, GROUPS, ACCOUNTING, ACCESS CONTROL

- Covers basic concepts
- Files and commands
- Special procedures
- Diagnosing problems
- Recovery

DEFINITIONS

- User
  - log on identity
  - has password, home world in !users
  - member of groups

identity
list of groups

- Group
  - identity for access control purposes
  - granted access to objects
  - limit of 1000

- Access List (ACL) world, file, Ada units (image)
  tree not protected
  - says which groups have what access
  - limit of 7 entries

- Access Types
  - R read to world, file, or Ada unit
  - W write to file or Ada unit
  - C create new objects in a world
  - O owner of world
  - D delete a specific world
  - D and W are synonyms

worlds
resolve names, look at content

- Access Control
  - a feature to make it clear to someone trying to access a protected object that they are doing something wrong
  - not very secure

DEFINITIONS - COMMON CONFUSIONS

- Groups are granted access to objects, not users

- D applies to the world itself, not objects within it

- Each <u>version</u> has an ACL not each <u>object</u>. Different versions can have different ACLs.

if no write access to last version and try to update newest version - will fail at commit since cannot write, i.e. delete, the last to enforce reutention count.

- The default access list associated with a world is what new objects created in that world get as their ACL. Changing the default ACL has no effect on existing objects in the world.

OPERATOR COMMANDS - CREATE USER

- Create_User

  - creates user logon (user object)
  - creates group with same name as user
  - user made member of that group
  - user is made member of group public and network_public
  - world is created in !users for the user; links are set to those in !model.R1000
  - ACL and default ACL are set for the user world from !Machine.User_ACL_Suffix and !Machine.User_Default_ ACL_Suffix, respectively

    default in new universe is (for both)
    NETWORK_PUBLIC => RWCOD

  Additionally,

  - user object is created in !Machine. users
  - group object is created in !Machine.groups

OPERATOR COMMANDS - GROUP OPERATIONS

- Create_Group
- Delete_Group
- Add_to_Group
- Remove_from_Group
- Display_Group
  - shows group members and user information
  - useful diagnosing access problems
- These require Operator_Capability except for Display_Group
- If you remove a user from his home group, confusion will result. This is legal, but probably a mistake.
- You can also remove users from public or network public.
- If a customer has complex group arrangements, you may want to write a skin for create_user that adds new users to other groups.
- Each group has an id. This is a small integer. Ids are not reused when groups are deleted.

  Can be reclaimed

Show_Groups
Show_Identity
in System_Maintenance

Group Identity established at login only - relogin to get any changes into effect

```
with Terminal;

package Operator is

    procedure Disk_Space;

    procedure Create_User (User : String := ">>USER NAME<<";
                           Password : String := "";
                           Volume : Natural := 0;
                           Response : String := "<PROFILE>");
    -- create a user with the given password on volume (0 => Most Available)

    procedure Delete_User (User : String := ">>USER NAME<<";
                           Response : String := "<PROFILE>");
    -- delete user;  Operator capability is required (or priv mode)

    procedure Change_Password (User : String := ">>USER NAME<<";
                               Old_Password : String := "";
                               New_Password : String := "";
                               Response : String := "<PROFILE>");

    procedure Create_Session (User : String := ">>USER NAME<<";
                              Session : String := ">>SESSION NAME<<";
                              Response : String := "<PROFILE>");

    procedure Create_Group (Group : String := ">>GROUP NAME<<";
                            Response : String := "<PROFILE>");
    -- Create the named group.  It must currently not exist.  It has
    -- no initial members.

    procedure Delete_Group (Group : String := ">>GROUP NAME<<";
                            Response : String := "<PROFILE>");
    -- Delete the named group.  This operation cannot be used to delete the
    -- group with the same name as an existent user.  Delete_User will
    -- get rid of the group associated with a user.  Acl entries
    -- that refer to a deleted group become inoperative and will be
    -- reclaimed during the next access list compaction.

    procedure Add_To_Group (User : String := ">>USER NAME<<";
                            Group : String := ">>GROUP NAME<<";
                            Response : String := "<PROFILE>");
    -- Add the specified user to the specified group.
    -- Operator privilege is required to execute this operation.

    procedure Remove_From_Group (User : String := ">>USER NAME<<";
                                 Group : String := ">>GROUP NAME<<";
                                 Response : String := "<PROFILE>");
    -- Remove the specified user to the specified group.
    -- Operator privilege is required to execute this operation.

    procedure Display_Group (Group : String := ">>GROUP NAME<<";
                             Response : String := "<PROFILE>");
    -- Display the names of users in the specified group on Current_Output.

    procedure Enable_Privileges (Enable : Boolean := True);
    function Privileged_Mode return Boolean;
    -- If the caller is a member of the predefined group "privileged",
    -- calling this procedure actually enables or disables the
    -- extra capabilites that such a job can have.  General usage is
    -- to not enable privileged mode unless it is really needed so
```

```
    -- as to avoid accidently doing something that would normally be
    -- stopped by access control.  All tasks in the job become
    -- privileged when the mode is enabled.  No output is produced
    -- by any of these procedures.  Failure to acquire privileged mode
    -- is indicated only by the absence of the privileges.  Privileged_Mode
    -- returns false in this case.


    procedure Enable_Terminal (Physical_Line : Terminal.Port;
                               Response : String := "<PROFILE>");
    procedure Disable_Terminal (Physical_Line : Terminal.Port;
                                Response : String := "<PROFILE>");
    -- (Dis)allow login on the specified terminal port

    procedure Force_Logoff (Physical_Line : Terminal.Port;
                            Commit_Buffers : Boolean := True;
                            Response : String := "<PROFILE>");
    -- Force a user off of the specified terminal.
    -- Try to commit modified buffers if Commit_Buffers is true.
    -- Each of these operations requires operator capability.

    procedure Set_System_Time (To_Be : String := ">>TIME<<";
                               Response : String := "<PROFILE>");
    -- Requires operator capability.

    procedure Shutdown_Warning (Interval : Duration := 3600.0);
    -- Note that Interval is rounded to the nearest minute.  Less than
    -- 30.0 is rounded to 0.

    function Get_Shutdown_Interval return Duration;

    procedure Archive_On_Shutdown (On : Boolean := True);
    function Get_Archive_On_Shutdown return Boolean;
    -- Archive_On_Shutdown causes the next shutdown to store internal
    -- state in "archive" form, allowing upgrades and conversion of
    -- internal data structures.  It typically takes several hours to
    -- complete a shutdown or restart with archive conversions.

    procedure Show_Shutdown_Settings;
    procedure Cancel_Shutdown;

    procedure Shutdown (Reason : String :=
                            "COPS";           -- Customer operations
                        Explanation : String := "Cause not entered");

    -- Shutdown the machine.  Enter the cause and explanation in the system
    -- log, wait for the Shutdown interval to expire, then log users
    -- off and shutdown the machine.
    -- Enter Reason = "?" to get list of reasons.  The shutdown will not
    -- happen unless Reason is a legal value.


    procedure Explain_Crash;
    -- Reads a shutdown cause and explanation from current input and enters
    -- these in the machine's error log.  Corresponds to the information
    -- entered by shutdown.

    procedure Limit_Login (Sessions : Positive := Positive'Last);
    procedure Show_Login_Limit;
    function Get_Login_Limit return Positive;
    -- Control over the number of simultaneously active user sessions

    procedure Internal_System_Diagnosis;
    -- Requires Operator capability

    pragma Subsystem (Os_Commands);
    pragma Module_Name (4, 3926);

end Operator;
```

OPERATOR COMMANDS - GROUP OPERATIONS (CONT'D)

- Only 1000 ids are available. When they are used up, a special procedure (ACL compaction) must be run to recover free ids.

- If a group is deleted and then re-created, it is a "different" group.

```
ACL:        Phil=>RW, Friends =>R
Operator.Delete_Group ("Friends")

ACL:        Phil=>RW, <unknown 377> =>R
Operator.Create_Group ("Friends")          –    no effect on ACL
                                           –    doesn't come back
```

- Reference to deleted group in an ACL apppears as

  <Unknown id#>

- Show_Groups
  - in System_Maintenance subsystem
  - displays current free and used group ids
  - use to see if you are running out of group ids or for diagnosing problems
  - create a new group and look at its id to see where the frontier of allocation is. If the display says an id is "free", this does not mean that it is available for allocation.

ACCESS CONTROL - FILES AND DIRECTORIES

- !Machine.Users
  - contains user "objects"
  - must be readable
  - must have create access to create users

- !Machine.Groups
  - contains group "objects"
  - must be readable
  - must have create access to create users or groups

- !Machine.User_ACL_Suffix
  - contents of this file appended to ACL of new user worlds

- !Machine.User_Default_ACL_Suffix
  - contents of this file appended to default ACL of new user worlds

- !.Machine.Operator_Capability
  - W access to this file allows execution of restricted commands or be member of OPERATOR group

- !Users
  - contains user home worlds
  - must have C access to create users

- !Machine.Network_Public_Session
  - session used for identity Network_Public. Recreated on boot if it is deleted. DØ bug: if this is ever deleted, you're screwed.

ACCESS CONTROL - FILES AND DIRECTORIES (CONT'D)

- !Machine.Public_Session
  - same for identity public
- !Machine.Temporary
  - temp files exist in this world. If ACLs are not proper for it, lots of things will fail to operate. World ACL must allow anyone to create access. Default ACL must allow anyone RW access.

ACCESS CONTROL - BASIC RULES

- Open for Read requires Read access
- Open for update (In_Out) requires write access
- No access required for access to Diana trees
- Creating a new object in a world require create access
- Deleting an object requires write access
- Changing an access list, changing links, frozenness, and switch file associations require owner access.
- Only files, Ada units, and worlds have access control. Directories, Users, Groups, Sessions, Devices, and Pipes do not.
- Viewing or resolving names in a world requires Read access to the world
- Deleting a world requires delete access to that world. This applies only to the world.

ACCESS CONTROL - BASIC RULES (CONT'D)

- A number of other specific rules apply:
  - adding a link to a world requires owner access to the world and Read access to the spec.
  - to promote a unit, you must have write access to the unit and read access to any units it withs.
  - to demote a directly named unit you must have write access to that unit. You need not have any access to the demotion closure of the unit.
  - if you are setting the ACL of a world but do not have owner access to that world, you will still be allowed to if you have owner access to the immediately containing world.
  - executing a command requires Read access to all units directly named in the command.

- There are also a number of specific operator capability checks.

- You have operator capability if:
  - you are a member of group operator
  - you have write access to file !Machine.Operator_ Capability
  - you are running with priviliges enabled

- Affected operations:
  - most commands in operator
  - Job.Kill on sessions not belonging to you
  - terminal set-up commands
  - scheduler set-up commands
  - and many more

ACCESS CONTROL - BASIC RULES (CONT'D)

- Consequences of these rules:
  - Environment operations may fail in strange ways if they try to access objects and are denied access. Delta will be retired before all of these are discovered.
  - This applies ot Editor operations as well.
  - To resolve a name, you will likely need read access to all worlds starting with "!" and on downward.
  - * Wildcard resolution: parts of a naming expression referencing worlds lacking Read access act as though the objects you are not allowed to see are not there. (no access errors)
  - Commands may fail to semanticize because they reference units not visible due to world Read access restrictions. This applies to all access including use clauses and search lists.

```
package Access_List is

    subtype Name is String;  -- an object name

    Read : constant Character := 'R';  -- objects and worlds
    Write : constant Character := 'W';  -- objects only
    Delete : constant Character := 'D';  -- worlds only; same bit as W
    Create : constant Character := 'C';  -- worlds only
    Owner : constant Character := 'O';  -- worlds only

    subtype Acl is String;
    -- String representations of access lists have the following syntax:
    --  Acl       ::= Acl_Entry {',' Acl_Entry}*
    --  Acl_Entry ::= Group '=>' Access
    --  Group     ::= Identifier
    --  Access    ::= Acc_Type+
    --  Acc_Type  ::= 'R' | 'W' | 'D' | 'C' | 'O' |
    --                'r' | 'w' | 'd' | 'c' | 'o'
    --  Examples:  "Phil => R , TRW => rw",  "Public=>RCOD"


    procedure Display (For_Object : Name := "<CURSOR>");

    -- Display the access list of the specified object(s).
    -- Output and error messages are send to current output.

    procedure Set (To_List : Acl := "Network_Public => RWCOD";
                   For_Object : Name := "<SELECTION>";
                   Response : String := "<PROFILE>");

    -- Set the access list for the specified object(s).
    -- Setting the access list requires "Owner" access to the containing world.
    -- Sends messages to a log that is under control of the Response parameter.

    procedure Default_Display (For_World : Name := "<CURSOR>");

    -- Display the default acl of the specified world(s) in an output window.
    -- Error messages are sent to the window in case of any error.
    -- Wildcards in the name are allowed.
    -- Non-world objects are filtered out of the display.
    -- A null display is produced if no worlds are referenced.

    procedure Set_Default (To_List : Acl := "Network_Public => RW";
                           For_World : Name := "<SELECTION>";
                           Response : String := "<PROFILE>");

    -- Set the default ACL for the specified world(s).
    -- Owner access to each world is required.
    -- Sends messages to a log that is under control of the Response parameter.
    -- A log is written indicating success or errors.
    -- Wildcards are allowed in the name.
    -- Any non-world objects referenced are ignored.
    -- A summary of the number of objects affected is included in the log.

    procedure Add (To_List : Acl := "Network_Public => RWCOD";
                   For_Object : Name := "<SELECTION>";
                   Response : String := "<PROFILE>");

    -- Add the access list to the existing value for the specified object(s).
    -- Changing the access list requires "Owner" access to the containing world.
```

```
    -- Sends messages to a log that is under control of the
    -- Response parameter.

    procedure Add_Default (To_List : Acl := "Network_Public => RW";
                           For_World : Name := "<SELECTION>";
                           Response : String := "<PROFILE>");

    -- Add the default ACL to the existing value for the specified world(s).
    -- Owner access to each world is required.
    -- Sends messages to a log that is under control of the Response parameter.
    -- A log is written indicating success or errors.
    -- Wildcards are allowed in the name.
    -- Any non-world objects referenced are ignored.
    -- A summary of the number of objects affected is included in the log.

    pragma Subsystem (Os_Commands);
    pragma Module_Name (4, 3507);

end Access_List;
```

IDENTITY - DEFINITIONS

- Each job has an identity. There are 2 parts to the identity:
  - the base user identity; this is a user name
  - the group identity; this is a set of group names

- The group identity is used to check access when the job attempts an operation requiring access checks.

- A session consists of a core editor job, some object editor jobs, and some user jobs.

- When a job is started, it <u>inherits</u> both base identity and group identity.

- The group identity for a session is established at <u>log in time</u>.
  - adding/removing a user from groups will not affect existing sessions.

- The base identity is associated with a <u>session.</u> This has implications.

```
op.enable_privileges.
def("———");   -- will fail because it tells CE to bring up image,
                 but CE job has not enabled privileges
```

IDENTITY - DEFINITIONS (CONT'D.)

- Special groups

Public	-	all users on a machine
Network_Public	-	all users on a machine and servers processing network requests
Privileged	-	members can override access control (op.enable_privileges)
Mailer	-	the mail system
Spooler	-	the print spooler
System	-	the "system". That is, other system jobs

- Users public and network public exists, but you cannot log on to them

- The public/network public distinction is a convention that network servers must follow

```
with Machine;
with Simple_Status;

package Program is

    subtype Job_Id is Machine.Job_Id;
    subtype Condition is Simple_Status.Condition;

    procedure Run (S : String := "<SELECTION>";
                   Context : String := "$";
                   Response : String := "<PROFILE>");
    -- sets root of job_garbage_unit, dangerous to run concurrently in one job

    procedure Run_Job (S : String := "<SELECTION>";
                       Debug : Boolean := False;
                       Context : String := "$";
                       After : Duration := 0.0;
                       Options : String := "";
                       Response : String := "<PROFILE>");

    procedure Create_Job (S : String := "<SELECTION>";
                          Job : out Job_Id;
                          Status : in out Condition;
                          Debug : Boolean := False;
                          Context : String := "$";
                          After : Duration := 0.0;
                          Options : String := "";
                          Response : String := "<PROFILE>");

    -- Run_Job and Create_Job are identical except that Create_Job
    -- returns the job number of the job just started and a status indicating
    -- success or failure.
    --
    -- Debug => True starts the debugger on the newly started job
    --
    -- The following options are defined:
    --
    --     Output        Specifies the name of the new job's output file.
    --     Input         New job's standard input file.
    --     Error         New job's error file.
    --                   File names given are resolved in the directory
    --                   context of the caller, NOT the Context parameter.
    --
    --     User          Causes the new job to run with the identity
    --                   of this user.  Password must be valid unless
    --                   running job is privileged.  If not specified
    --                   new job runs with same identity as parent.
    --
    --     Password      Password used in conjunction with User.
    --
    --     Session       Session used in conjunction with User.


    function Started_Successfully (Status : Condition) return Boolean;
    -- True => Job has been started successfully

    procedure Wait_For (Job : Job_Id);
    -- Wait until the job specified has terminated.

    procedure Change_Identity (To_User : String := "";
```

```
                               Password : String := "";
                               Options : String := "";
                               Status : in out Condition);

    -- Change the identity of the calling job to the specified
    -- user.  Password must be supplied and correct unless the
    -- caller is privileged.  Options specifies additional
    -- characteristics to be changed.  If To_User is null,
    -- the options are processed.

    -- Note that only the access control identity is changed.
    -- The actual username and session of the job are NOT changed.
    -- This operation should never be used to change identity and
    -- execute untrusted code.  The identity can always be changed
    -- back to the original job identity.

    -- Options presently defined are:
    --      Privileged          -- enable privileged mode.  The specified user
    --                          -- must be a member of group PRIVILEGED
    --      Privileged => False -- disable privileged.  No effect if caller
    --                          -- was not already privileged.
    --      Restore_Identity    -- Change the identity back to the original
    --                          -- identity of the job.  Password is not
    --                          -- required to do this.

    function Current (Subsystem : String := ">>SUBSYSTEM NAME<<";
                      Unit : String := ">>PROCEDURE NAME<<";
                      Parameters : String := "";
                      Activity : String := "<ACTIVITY>") return String;
    -- Constructs a procedure call suitable for Run or Run_Job that references
    -- the appropriate view, has the appropriate quotes, etc.  Unit name is
    -- the Ada name to be called; it will be found anywhere in the
    -- view.  If the procedure being called has parameter they may be
    -- provided.  If the current view of !Subsystem is Rev8_4_0 and package
    -- View is in the Commands directory, then:
    --
    -- Current ("!Subsystem", "View.Initial", "!New_Tool") returns:
    --
    -- "!Subsystem.Rev8_4_0.Units.Commands".View.Initial ("!New_Tool");

    pragma Subsystem (Commands);
    pragma Module_Name (4, 3930);

end Program;
```

IDENTITY - CHANGING

- **Program.Change_Identity**
    - Sets group identity to group membership of a specified user
    - Does not change base identity
- **Program.Run_Job/Create_Job**
    - Options parameter allows setting user (with password)
    - This sets the base identity of the new job (as well as the group identity)
    - If user not specified, it is inherited
    - If the base identity is changed, the job is associated with a particular session. Editor operations, termination messages, scheduling decisions, etc, are based on the session
    - You can specify the session in the options parameter. "S_1" is the default *Can/will affect activity on that session e.g. if someone logged on under same session.*
- **Program.Change_Identity w/options =>"restore"**
    - changes group identity back to that of the base identity group membership
    - Following a Run_Job that changed the base identity, restore changes to that new base identity, not that of the initiating job
- **Program.Change_Identity can also be used to enable and disable privileged mode**

IDENTITY-MACHINE.INITIALIZE

- When Machine.Initialize runs after the R1000 boot, its base identity is "*system" (that is, none) and its group identity is

  Public
  Network_Public
  Privileged
  System

- If initialize executes commands that require operator capability, one of these groups must have write access to !Machine.Operator_Capability

- If initialize starts served with Run_Job, they will inherit this identity unless the user parameter is set

IDENTITY-DISPLAYING A JOB'S IDENTITY

- What.Users or

```
IO.Echo_Line (System_Utilities.User_Name
                  (System_Utilities.Get_Session(job#)))
```

  - will display a job's base identity

- The Show_Identity command (from System_Maintenance) will provide information about a job's current identity. It shows both the base and group identities.

```
Show_Identity (Job#)

Show_Identity                -- defaults to current job
```

PRIVILEGED MODE

- Access checks will always pass if privileges are enabled

- Privileges applies to a job only, not a session. It applies to all tasks in the job

- The editor (core and object) run as jobs. There is no way to enable privileges for these jobs. This implies that porotected objects cannot be brought into editor windows by enabling privileges

- Privileges must be explicitly enabled. Just being a member of group privileged is not sufficient

```
Operator.Enable_Privileges;

if not Operator.Privileged_Mode then
      IO.Put_Line ("Privileges not enabled");
      raise Failure;
end if;
```

RUNNING ACL COMPACTION

- To recover unused group ids, the ACL compaction procedure must be run

- This procedure removes entries for deleted groups from all ACLs on the machine

- It takes about 5 to 20 minutes during which the system is unusable

- Procedure

```
Daemon.Set.Access_List_Compaction;
Daemon.Run ("directory");
Daemon.Run ("file");
Daemon.Run ("Ada");
```

- You can run Set_Access_List_Compaction and let the normal overnight daemon run do the work

- And compaction is automatically disabled after it is run

UNIVERSE ACLs

- Users can set whatever they want for their own stuff and other stuff not in the pre-defined universe

- Setting restrictive ACLs on other objects can cause system problems, including making it impossible to log on or execute commands

- The set-universe ACLs procedure allows you to set up various levels of protection for a system. It sets ACLs to known workable valves

- See handout

- Enable privileges, then call Set_Universe_Acls (see next page)

- Levels

0	None
1	Open
2	Safe
3	Secure

- At this time, operation is not very well understood. Don't set things to safe or secure until further advised

```
procedure Set_Universe_Acls (Level : Natural := 0; -- none
                             Implementation_Okay : Boolean := True;
                             Network_Read_Okay : Boolean := True;
                             Network_Write_Okay : Boolean := True;
                             Trace_Only : Boolean := False);


-- Level = 0 => none   : anyone can do anything.
--       = 1 => Open   : anyone can do anything, but they may have to change
--                       acls to do it.
--       = 2 => Safe   : System and users are protected.  The operator must
--                       change acls to create new areas and allow others to
--                       things that users can do under level=1.
--       = 3 => Secure : Like safe, but more limited network access and less
--                       read access.

-- Set acls for the standard universe to be as described above.
-- Level 3 is about the most restrictive the system can be and still
-- run.  Level 3 will prevent most users other than Operator from
-- successfully executing operator commands even if they have operator
-- capability via write access to !Machine.Operator_Capability.

-- Implementation_Okay => access is given to !Implementation and
-- !Compiler_Interface.  Actually, !Compiler_Interface needs to be
-- readable anyway because it contains the switch file for the
-- standard universe.

-- Network_Read_Okay => Network_Public is granted read to most things, except
-- when Secure (level=3) is specified.

-- Network_Write_Okay is analogous to Network_Read but for Write access.

-- Be sure to update !machine.[user_acl_suffix,user_default_acl_siffix]
-- so that new users will get the acls you wish.
-- Don't forget about !machine.operator_capability, either.
```

UNIVERSE ACLS - DIAGNOSING AND REPAIRING

- Customers may change ACLs and break things

- There is a tool to check that ACLs are properly set. This can be run to see if customer-changed ACLs are the cause of the problem

  Check_Universe_Acls (in System-Maintenance)

- This tool is presently incomplete, but may be helpful

UNIVERSE ACLS - DIAGNOSING AND REPAIRING

- If things get badly messed up, you can run Set_Universe_Acls to reset ACLs. If customers have manually changed ACLs, this may overwrite those changes

- If you can't execute commands or log on at all, drastic measures may be required. The access control system can be diasabled from the IOA console:

```
EEDB: e ed

ED: x ac_off

ED:  – access control ~~if~~ is off

<log on and fix ACLs>

ED: x ac_on     – access control is on

ED: quit
```

- See handout

Running ed tests

```
EEDB: e ed
   _TESTS.9.0.0D                2/08/87 05:53:25
ED: x ac_off
AC_OFF started
AC_OFF finished
ED:     < go fix things >

ED: x ac_on
AC_ON started
AC_ON finished
ED: quit

EEDB:
```

If Ed tests configuration won't elaborate, it can be rebuilt:

```
EEDB: e ed
A subsystem with name NETWORK is already elaborated


EEDB: delete ed
EEDB: running
  D_9_20_1
EEDB: bu ed d_9_20_1                 -- D_9_20_1 is the current
                                     -- running version
Parent subsystem: eoe
Subsystem.Version: ed_tests.9.0.0d
Subsystem.Version:
EEDB: e ed                           -- should work now.
```

ACCESS_LIST_TOOLS PACKAGE

- Programmatic interface for access control
- See handout

• Set/get	- set and retrieve access lists of objects - can be used with directory package
• Check	- test if specified access is allowed - can check based on job identity or user name - also can be used with directory versions
• Check_validity	- checks on ACL string for legality
• Normalize	- remove any references to deleted groups
• Amend	- contruct an amended ACL granting a user access

```
with Simple_Status;
with Bounded_String;
with Directory;

with Machine;

package Access_List_Tools is

    subtype Name is String;  -- an object name

    subtype Access_Class is String;  -- of only the following characters:
    Read : constant Character := 'R';  -- objects and worlds
    Write : constant Character := 'W';  -- objects only
    Delete : constant Character := 'D';  -- worlds only; same bit as W
    Create : constant Character := 'C';  -- worlds only
    Owner : constant Character := 'O';  -- worlds only


    -- An object string name is as defined by the directory
    -- package.  No wilcards are accepted;  each operation in this
    -- package operates on one object.

    subtype Acl is String;
    Max_Acl_Length : constant := 512;  -- max length for access list string
    -- The max size will not be exceeded when an Acl is returned.



    -- String representations of access lists have the following syntax:
    --  Acl       ::= Acl_Entry [',' Acl_Entry]*
    --  Acl_Entry ::= Group '=>' Access
    --  Group     ::= Identifier
    --  Access    ::= Acc_Type+
    --  Acc_Type  ::= 'R' | 'W' | 'D' | 'C' | 'O' |
    --                'r' | 'w' | 'd' | 'c' | 'o'
    --  Examples:  "Phil => R , TRN => rw",  "Public=>RCOD"

    Access_Tools_Error : exception;  -- Raised by functions

    function Get (For_Object : Name) return Acl;
    function Get (For_Object : Directory.Version) return Acl;
    procedure Get (For_Object : Name;
                   List : out Bounded_String.Variable_String;
                   Status : in out Simple_Status.Condition);
    procedure Get (For_Object : Directory.Version;
                   List : out Bounded_String.Variable_String;
                   Status : in out Simple_Status.Condition);

    procedure Set (For_Object : Name;
                   To_List : Acl;
                   Status : in out Simple_Status.Condition);
    procedure Set (For_Object : Directory.Version;
                   To_List : Acl;
                   Status : in out Simple_Status.Condition);


    -- Get or Set the access list for the specified object.
    -- Setting the access list requires "Owner" access.
    -- function Get raises Access_Tools_Error if an error occurs.
    -- The procedure version should be called in that case to get the
```

```
    -- actual error information.
    -- ACL for world must be contain only R, C, O, or D access.  Others
    -- must be only R or W access.

    function Check (User_Name : String := "";
                    Object_Id : Directory.Version;
                    Desired : Access_Class) return Boolean;
    function Check (User_Name : String := "";
                    Object_Name : String;
                    Desired : Access_Class) return Boolean;
    function Check (User_Id : Directory.Version;
                    Object_Id : Directory.Version;
                    Desired : Access_Class) return Boolean;
    function Check (Job : Machine.Job_Id;
                    Object_Id : Directory.Version;
                    Desired : Access_Class) return Boolean;

    -- Check if the specified user has the indicated access to the
    -- specified object.  Only meaningful for Ada objects, Files, and Worlds.
    -- The null string for the User_Name parameter means the identity of
    -- the calling job.  If a user name is specified, the access control
    -- identity of that user (its member groups) is used for the test.
    -- If an error is detected during the test, the value false is returned.
    -- The most common errors are illegal values for Desired and references
    -- to objects that do not exist.  If an object that does not have an
    -- access list is referenced, the value true is returned.

    function Get_Default (For_World : Name) return Acl;
    procedure Get_Default (For_World : Name;
                           List : out Bounded_String.Variable_String;
                           Status : in out Simple_Status.Condition);
    procedure Set_Default (For_World : Name;
                           To_List : Acl;
                           Status : in out Simple_Status.Condition);
    -- Get or set the default ACL for new objects created in the specified
    -- world.  The function raises the exception Access_Tools_Error if
    -- an error is detected.  The procedure version returns a status
    -- that indicates the cause of the error.


    procedure Check_Validity (For_List : Acl;
                              Status : in out Simple_Status.Condition);
    -- Check the validity of the specified access list.  Return status
    -- indicating that it is okay, or the error, if any.

    function Has_Operator_Capability return Boolean;
    -- Return true if the calling job has operator capability.  This is
    -- true if the job has an identity that includes the group
    -- "operator", is on the access list for "!machine.operator_capability",
    -- or is priviledged.

    function Normalize (Initial_Acl : Acl) return Acl;
    -- Scan the acl and eliminate any entries for groups that do
    -- not currently exist.  Return the revised acl.  If the
    -- acl is otherwise illegal, raise Access_Tools_Error.

    function Amend (Initial_Acl : Acl; New_Group : Name; Desired : Access_Class)
               return Acl;
    -- Amend Initial_Acl so that New_Group is granted Desired access.  If
    -- necessary, the right-most acl entry is removed to do this.
    -- Raise Access_Tools_Error if any parameter is illegal.


    pragma Subsystem (Os_Commands);
    pragma Module_Name (4, 3508);

end Access_List_Tools;
```

SIMPLE STATUS

- A status reporting abstraction. Used by some access control operations and other parts of the environment.

- S : Simple_Status.Condition;

```
Access_List_Tools.Check_Validity ("Phil =>RK", S);
if Simple_Status.Error(s) then
  IO.Put_Line ( Simple_Status.Display_Message (S));
end if;
```

- Status condition contains

Severity	Normal, Warning, Problem, Fatal
Condition Name	Up to 63 characters
Message	Up to 400 or so characters

- Usage
  - condition name is a fixed string that identifies the error (program interface)
  - message provides additional information

declare one, pass to lower levels as in out parameter

```
package Simple_Status is

    -- Error status reporting package

    -- A simple_status.condition can be used to return error information from
    -- procedure calls.  They are relatively large and should always
    -- be passed in out (by convention to avoid copies).

    -- A Condition consists of a Condition_Name and a Message.
    -- The Condition_Name indicates the type of error (if any) and how
    -- how serious the error is (or if completion was
    -- successful).  The Message provides additional information about
    -- the error.

    -- In simple applications, A Condition_Name alone can be used to
    -- indicate status.

    -- By convention, condition names in an application should be
    -- standardized so that error conditions can be tested
    -- programmaticly.



    type Condition_Name is private;  -- A short name for the error type
    type Condition_Class is
       (Normal,   -- operation completed normally
        Warning,  -- operation completed, but something unexpected happened
        Problem,  -- operation did not complete, but no harm done
        Fatal);   -- operation did not complete.  Proceeding is dangerous.
    type Condition is private;  -- Contains the above plus a message
    -- Conditions are self-initializing to
    -- severity Normal and null names

    procedure Initialize (Status : in out Condition);
    -- The empty condition has null name and severity normal
    -- a declared condition is initialized;  This procedure will set the
    -- Condition to be Normal (ie, successful).

    function Name (Error_Type : Condition_Name) return String;
    function Name (Status : Condition) return String;
    -- get the human-readable name of this Condition_Name (Condition)

    function Severity (Error_Type : Condition_Name) return Condition_Class;
    function Severity (Status : Condition) return Condition_Class;

    function Error_Type (Status : Condition) return Condition_Name;
    -- provide the Condition_Name on which a Condition is built

    function Error (Error_Type : Condition_Name;
                    Level : Condition_Class := Warning) return Boolean;
    function Error (Status : Condition; Level : Condition_Class := Warning)
                   return Boolean;
    -- True <=> Severity (Error_Type/Status) >= Level;
    -- usage:
    --    Do_Something (Status);
    --    if Simple_Status.Error (Status) then
    --          ... Put (Display_Message (Status);

    function Display_Message (Status : Condition) return String;
    -- given a condition that indicates and error, this function returns
```

```
    -- a string suitable for display to users.  It includes the
    -- string form of the condition name and any additional problem-
    -- specific information recorded in the condition.


    function Message (Status : Condition) return String;
    -- return just the message part of the Condition.

    procedure Create_Condition (Status : in out Condition;
                                Error_Type : String;
                                Message : String := "";
                                Severity : Condition_Class := Problem);
    procedure Create_Condition (Status : in out Condition;
                                Error_Type : Condition_Name;
                                Message : String := "");
    -- Create a new error condition.  The Error_Type is intended to
    -- specify the class of error (limited to 63 characters), generally
    -- in a few words (eg, "Illegal name").  Message is intended to
    -- supplement the error_type with more specific information
    -- (eg, ": '#' is an illegal character").  Function Display_Message
    -- would then return "Illegal name: '#' is an illegal character".


    function Create_Condition_Name
             (Error_Type : String; Severity : Condition_Class := Problem)
             return Condition_Name;


    function Equal (Status : Condition; Error_Type : String) return Boolean;
    function Equal (Status : Condition; Error_Type : Condition_Name)
                   return Boolean;
    function Equal (Status : Condition_Name; Error_Type : String)
                   return Boolean;
    function Equal (Status : Condition_Name; Error_Type : Condition_Name)
                   return Boolean;

    -- return true if the error_type string of Status is equal to the
    -- right error_type string (the second parameter).  The severity
    -- does not participate in the comparison.  The strings must
    -- match exactly (except for index range).  Sample usage:
    --   Directory.Open(File, Status);
    --   if SS.Equal (Status, "Nonexistent file") then ...
    --   elsif SS.Equal (Status, "Internal error") then ...
    --   etc.
    -- The strings in the example should be constants, of course.


    pragma Subsystem (Miscellaneous);
    pragma Module_Name (4, 810);

end Simple_Status;
```

ACCOUNTING

- Same as gamma, except:
  - controlled by existence of file

    !machine.accounting.enabled

    not existence of directory !machine.accounting as in gamma