

---

# *Lesson 1*

## *Introduction to Programmers*



### *Objectives*

**When you complete this lesson you will know the Eclipse:**

- Vision
- Products
- Company
- Programming Environment

# ***Introduction***

## **Eclipse**

Eclipse represents a revolutionary advance in distribution management software. The Eclipse system was designed to put wholesalers in total control of their distribution operations.

## **Our Company**

Eclipse Inc. was incorporated in 1991. The corporate headquarters are located in Shelton, Connecticut, with a customer support center in West Yarmouth, Massachusetts. Boulder, Colorado, is home for the Research and Development, Quality Assurance, Documentation, and Marketing departments.

Six veterans of the hard good distribution industry developed the Eclipse system. Their software savvy and knowledge of the warehousing business led them to create an innovative system—the first real-time, PC-based system for wholesale distribution management. The Eclipse system was designed to put wholesalers in total control of their distribution operations.

## **Our Product**

The Eclipse Distribution Management System completely automates every facet of wholesale distribution: sales, inventory management and forecasting, purchasing, accounting, and much more. Wholesalers become more productive, and can plan ahead. Real-time processing means Eclipse users always have accurate, up-to-the-minute inventory, sales, and order information. Instant system-wide updating empowers sales staff and warehouse workers, while at the same time automating business management tasks.

Today Eclipse, Inc. is a leading provider of enterprise software for end-to-end supply chain management. Since 1992, Eclipse has enabled hundreds of distributors across the United States, Canada, and Mexico to improve their business efficiency by streamlining several traditional distribution processes. Eclipse has established itself as a leader in several market segments including Plumbing, HVAC, PVF, Electrical, Building Materials, Industrial, Paper and other durable goods supply industries.

From Palm-based, wireless solutions to Internet-centric applications for collaborative eBusiness, Eclipse can help its customers achieve a true, competitive advantage. Integrated business applications including Sales Management, Inventory Management, Finance & Accounting, Business Intelligence and Warehouse Automation are among the essential elements that leverage the business rules set forth by the Eclipse Application Framework. Collectively, these components work together to help optimize the flow of information throughout the supply chain.

In *FY 1999*, Eclipse experienced 80% year-over-year revenue growth, spurred on by a record number of new installations. Today, with over 185 employees, Eclipse is one of the fastest growing private companies in the computer software industry.

# Our Vision

Eclipse has developed a business vision that focuses on empowering customers through the use of great software. As part of this vision, Eclipse intends to:

- Develop innovative new solutions through the integration of current and emerging technologies;
- Hire, train and retain outstanding professional employees with a broad understanding of best-practice techniques;
- Enhance customer support and product education through the utilization of emerging technologies;
- Grow market penetration by extending sales and marketing efforts to targeted verticals within the durable goods.

Integral to the Eclipse vision is a customer-centric business philosophy. By strategically placing the “needs and wants” of customers as the driving force behind innovation, Eclipse intends to extend its leadership position as the dominant supplier of distribution management systems.

## Programming Environment

Eclipse is written on top of the IBM database called UniVerse. The native programming language for UniVerse is UniVerse Basic, which is a form of PICK Basic. PICK has been around since 1968 and is a very fast, comprehensive database language. However, to take advantage of new technologies and to stay on the leading edge of Distribution Management Systems, the founders of Eclipse wrote a pre-compiler around the UniVerse Basic language. This pre-compiler lets us, as Eclipse developers, develop in a language we fondly call “Eclipse Basic.” Along with making developers more efficient, “Eclipse Basic” also allows windowing, buffered input, and other capabilities. To take advantage of these additional functionalities, the client must be running the Eclipse terminal emulator Eterm.

# About this Manual

This manual is organized in a logical progression from the first questions you'll need to answer in order to program in Eclipse to advanced programming techniques and hints. Each lesson covers a different topic and provides multiple exercises and examples that help teach that topic.

This manual will help experienced PICK developers transition quickly into the Eclipse environment and will teach new developers with no PICK experience all they need to know to program the Eclipse DMS. You will also look at many of the Eclipse programming standards, which all programmers are responsible for knowing and following.

This manual is intended to be read and absorbed over the course of many weeks.

## Lesson Structure

In each lesson you are presented with concepts that relate to the Eclipse language and the creation of new applications. Each chapter builds on the one before, so it is imperative that you understand the chapter you're working on before moving to the next chapter.

### Exercises

Each lesson offers exercises that will help you master the concepts being taught. You may not need to do every exercise in a chapter in order to master that topic and are free to move at your own pace through these lessons.

### Frequently Asked Questions

Each lesson has a section of frequently asked questions and answers to those questions.

### Assignments

Each lesson will have one or more assignments for you to perform and turn into your training manager. You may be required to do all or only one of each assignment. The individual training manager you're working with will determine this.

# Conventions

The following conventions are used for presenting information in this manual.



**Tips** *A tip offers advice on the current topic of discussion and easier ways to accomplish specific tasks.*



**Cautions** *A caution advises you of potential problems related to the current topic of discussion and gives advice on how to avoid them.*



**Notes** *A note presents interesting, sometimes technical pieces of information related to the current topic of discussion.*

## Code Listings

**TYPE** - All full code listings will be headed with this tag and then the name of the code. Note that all programs we write in this manual will start with your initials (such as, **JTS**) and then the rest of the code ID.

Program or subroutine listings will be printed in a different font, as shown in the following example:

```
01      SUBROUTINE (PASSER)
02  ** Version# 90 - 02/23/2001 - 05:33pm - CHRISM - develop
03
04  *** Subroutine: CUSTOMER.MAINT
05  *-----*
06  *** This program allows user to update static information concerning
07  *** customers (such as "Name" or "Address"). The customer records live in
08  *** the ENTITY file which is open to the file handle CUSFILE in
09  *** OPEN.STANDARD.FILES.
10  *-----*
11  *** PASSER<1> - Internal customer ID - If passed in they will only
12  ***      - be allowed to se/maintain this one record before leaving
13  ***      - the routine. If passed in as '' we will prompt for the
14  ***      - customer ID.                                (IN)
15  *** PASSER<2> - view only override (set to YES will force view mode) (IN)
16  *** PASSER<3> - branch account override - will for CUS(7)<1,4> = YES (IN)
17  *-----*
18  *** NOTE: everything in passer is optional. If sent in as '' we will
19  *** default or prompt the information from the user rather than overriding
20  *** it.
21  *-----*
22  *** COMMON VARIABLES USED/SET/CHANGED: CUS, OID.DATA$, ACT.ID$,
23  *** USE.ZIP4$
24  *-----*
25  *** Fill in the default variables from PASSER (Override Customer ID, View
26  *** Only Override, and Branch Account Override)
27          INIT.CN  = PASSER<1>
28          VIEW.ONLY = PASSER<2>
29          BRCH.ACCT = PASSER<3>
30  *** The user is NOT in an activity log at this point so clear out
31  *** attribute 7 of the common array OID.DATA$
32      OLD.DATA$<7> = ''
```

## *Lesson 2*

# *Eclipse's Operating System, UniVerse Database, and Distribution Management System*



### *Objectives*

**When you complete this lesson you will understand:**

- The Eclipse Distribution Management System
- The UniVerse Database
- The Operating System

# Overview

In this chapter we are going to introduce the three major components of the Eclipse system. It is important that as a developer at Eclipse you fully understand the roll of each of the components.

The three layers are:

- The Eclipse Distribution Management System
- The UniVerse Database
- The Operating System (AIX, Windows NT, HPUX etc.)

By the end of this chapter it should be clear what role each of the above layers plays in the Eclipse developer's life and the differences each layer has from the next.

## Operating System

An operating system is an integrated collection of items that service the sequencing and processing of programs by a computer. This system can provide many services such as resource allocation, scheduling, input/output control, and data management.

The above paragraph briefly describes what an operating system is. So what does this mean to you? It's very simple; everything you program within Eclipse (or in any programming language for that matter) is eventually just turned into a bunch of commands that the operating system performs. Now that doesn't mean that our code is compiled to machine code, because it isn't. As a programmer at Eclipse you will rarely need to write an OS command, but it is important that you understand what the OS is doing for you. The operating system is responsible for making sure all of our programs run, file updates happen, reads are taken care of, etc. The operating system for Eclipse can be Windows NT, Windows 2000, AIX, HPUX, or Solaris. The nice thing about having UniVerse in the Database layer is that we have very few programs that must be changed when porting onto a different platform. UniVerse takes care of all this for us.



## Possible Operating Systems

- **AIX** is an IBM system, based upon a version of UNIX. The most advantageous aspect of AIX is that it is an open operating system. This means it adheres to a publicly known and sometimes standard set of interfaces, so anyone using it can also use any other system that adheres to the standard. Users can learn a single set of skills and find that they are portable across the entire industry. AIX is also the operating system for one of the most adopted and performant server/hardware lines in the industry (the RS6000).
- **Solaris** is a product created by Sun Microsystems (Famous for the Java programming language). It is another version of the UNIX standards and is similar to AIX in many ways.
- **HPUX** is yet another UNIX-based operating system; Hewlett Packard created this version for its server line.
- **Windows NT 4.0 (Server)** and **Windows 2000 (Server)** are the only two non-UNIX operating systems Eclipse operates under. For these operating systems, we do have to make some changes to our code. These changes are necessary because Microsoft created these Operating systems to a proprietary standard not compliant with UNIX or any other standards. These operating systems have the following limitations:
  - Many things are nearly impossible to do from the command prompt of an NT or 2000 server.
  - Windows servers are also not nearly as efficient in memory and disk management as the UNIX platforms mentioned above.
  - NT and 2000 server hardware is limited. The UNIX entry and mid-level servers consistently out perform even the best NT machine ever made.

Because of these many limitations, Windows NT and 2000 are limited to our smaller clients who need to keep the hardware cost of Eclipse to a minimum.

# **The UniVerse Database**

## **What is a Database?**

A database is a collection of related files. A file is a collection of related records. A record is a collection of fields (attributes). A unique identifier (ID) or Key identifies each record within the file. In multi-dimensional databases such as Universe, attributes can also consist of multi-values and values can consist of sub-values.

## **Files**

Files are collections of logically related items or records. For example, a file cabinet contains folders, which in turn contain similar types of information. We all remember in grade school that everything we did (good and bad) was recorded in our record. Well, I'm sure they had a big filing cabinet filled with all of our records. At Eclipse, we have files about our customers, the products our customer sell, and our vendors just to name a few. Each of these files contains records made up of similar types of information. For example, at Eclipse we have about 400 customers so as you might guess, there are about 400 records in our customer file, each containing similar types of information. Examples of that information would be the customer name, address, and phone number.

## **Records**

A record is a collection of logically related attributes or fields. So all of these 400 records in our customer file will contain the name of the customer, address, contacts, billing information, and other information for that ONE customer. Each record in the customer file is formatted in the exact same way. A record would be like a file within a section of the file cabinet. Going back to the grade school example, each "file" (one per student) would contain our grades, names, mom's name, and so on.

## **Attributes**

An attribute is simply a dividing mark between each data element. When you access the data, you can quickly get to the field of data (attribute) you need. So the first attribute or field in the customer file might be name, the second; address, the third; contacts, etc.

## Record ID / Key

Each record in a file must have its own unique identifier. This is called the key to the record. This key may be any combination of alphabetic, numeric, and most punctuation characters. At Eclipse we mainly use '.' and '~' as punctuation characters. No spaces are allowed in the key. Because of the uniqueness of the key, a programmer can pull data very easily from a database to be used in any program they write. The ID for our grade school records may have been our name or our social security number.

### *Exercise 2.1: Creating Database Files*

Create your own simple example of multiple records within one file. Using a simple Microsoft Excel spreadsheet, create a list of all of your relatives, recording for each person their address, phone number, spouse's name, and the day they got married.

1. Open MS Excel.
2. Start a "new" table/spreadsheet called something like "Relative Information." This will be the "file name."
3. Once you've created this new table, enter the following column headings:
  - Relative Name
  - Street Address
  - City
  - State
  - Zip Code (Postal Code)
  - Phone Number
  - Spouse's Name
  - Marriage Date
  - Relationship to You



**Tip:** *If you need help creating this spreadsheet in Excel please see your training manager on how to get started.*

4. Now that you've got the column headings, enter some data. Under the "Relative Name" fill in two or more of your relatives. Go ahead and fill out all of the columns for each relative. Remember this doesn't have to be exact; it's just for practice. If you don't know a piece of information leave it blank or make it up!
5. Once you have filled in all of the data, save the spreadsheet into your C: drive and you're done.

## ***Review***

Let's analyze what you just did.

1. You created a **file** called "relative information."
2. The file contained information about each one of your relatives (**records**).
3. Next you created your columns. These columns represent **attributes**. Notice that every record (each relative is one **record**) has the same attribute information in the same column. The data within the record itself is not the same but every relative's name is in the first column (**attribute**) of the record and every relative's marriage date is in the 8th column (attribute) of the record and so on.
4. What was the key for each record? Excel automatically generated a unique key as you added records. What was it? It was the "Line #" on the left hand side of your screen. That incremental number for each relative you entered gave those records a unique identifier. So, even if you type the same name on multiple lines in the Relative Name column, the program (Excel in this case) sees each line as totally different information.

UniVerse is just the name given to the database Eclipse uses to store data in. Along with being a database, UniVerse also provides a native programming language called UniVerse BASIC (just another "flavor" of PICK BASIC). This is the language we use to program Eclipse. There are many advantages to using a native programming language when accessing the database. A few of the most important advantages are:

- Getting data from and to disk is usually a very easy task;
- There are a lot of native query statements that are much more efficient within the language;
- The programmer can fully utilize all of the databases strong points and stay away from its weaknesses;
- It is very easy to create new fields, files, and records in the native language.



**Note:** *Going back to the introduction, we do have a proprietary pre-compiler that allows us to write Eclipse Basic, but that code is just turned into native UniVerse BASIC and compiled using the UniVerse compiler.*

*PICK BASIC was created back in 1968 as all three BASIC layers: the programming language, database, AND operating system. This evolved (although the original PICK Systems company is still around and dying) into our model with a database provider (Informix, UniVerse and UniData) integrating the PICK programming language as its native language and supporting the traditional file structures of PIC, but they are not operating systems. These databases run on top of other operating systems that handle all of the commands to the CPU, disk, etc. This gives an advantage in that all you have to do to port over to other hardware platforms is have a different compiler to make your code work on that OS.*

We will go into much more detail on the UniVerse database in **Lesson 4**. For now it is just important that you understand why we have UniVerse along with a separate operating system running under Eclipse, and what the role of each system is.

# Eclipse Distribution Management System

Eclipse is a programming application. We write programs using the UniVerse BASIC language that talks to the UniVerse Database, which in turn has code (written in C) that talks to the operating system to make all of the commands happen. So if you think about it, Eclipse has three layers of software:

1. Operating System
2. Database
3. Application

The **Eclipse Application** is all that truly matters to our end users. The application is responsible for maintaining all of the files and data, and doing all of the calculations the customers need to run a successful distribution company. We also have the entire user interface in this application that draws the screens, asks for information, and interacts with our end user. This Application is what you'll be programming, but you must know how to interact with the Database and OS layers to be a truly successful programmer in any application.

# Summary

In this chapter you have learned about the three different software layers that concern us as programmers at Eclipse:

1. Operating System
2. Database
3. Application

You also learned a little bit about how each layer interacts with the layers below and above it and why Eclipse chose to use a native programming language when talking to the UniVerse Database.

Finally, we discussed some of the advantages of each operating system Eclipse uses and did a short exercise (using Microsoft Excel) to understand the relationship of data within a file in a database.

# Frequently Asked Questions

## **Question:**

Why doesn't Eclipse just use the old PICK systems approach and use one operating system and database combined?

## **Answer:**

There are many drawbacks to this old model. The main one being that very few hardware platforms support the PICK operating system. The nice part about having a database like UniVerse is that most of the customization you would do to port onto different platform models is done by them. All databases and products that run on an OS (or hardware platform) must get "certified" by that manufacturer, which can take a lot of time and money. For Eclipse we just install the correct version of UniVerse on that OS and we're ready to go.

## **Question:**

With all of the more popular databases (like Oracle) in the software industry, why did Eclipse choose a seemingly smaller and unknown database like UniVerse as its database provider?

## **Answer:**

Oracle is a very high-end database platform with a very high price point. Oracle (and almost every other relational database including Access) is also limited to the 2-dimensional approach within a file and its record (like our Excel example). The file contains records and the records contain data across the columns. The UniVerse database is very powerful and can easily handle up to 5 and 6 dimensions within each record. Finally, the PICK database (and UniVerse in particular) is very easy to maintain and optimize. We can easily add fields to files (tables) without changing the whole file structure and our clients do not need a full-time DBA to make sure the database is "tuned" and running properly.



## Assignment: Lesson 2

1. On paper, design a file layout that you would recommend as a “contact” database. It should be designed to store:
  - First name
  - Middle initial
  - Last name
  - Home phone number
  - Work phone number
  - Email address
  - Address 1 and 2
  - City
  - State
  - Zip code.
2. Create this file in Excel.
3. Once you have completed it, email it to your training supervisor.

---

# *Lesson 3*

## *Call Tracking for Programmers*



### *Objectives*

**When you complete this lesson on the Tracking System, you will be able to:**

- Create Trackers
- Assign Trackers
- View Trackers
- Use the Tracker Stopwatch
- Append Trackers

# Overview

In this chapter you will learn how to use the Eclipse Call Tracking System. It was designed for Eclipse employees to track the time spent researching, problem-solving, programming, testing, and documenting existing and new application programs.

Programmers, in particular, use the Call Tracking System to track which projects we have open, how long we expect each project to take, and when we expect to complete each project. The release control department, which decides what programs are promoted into each release, also uses this system. We also use this system for tracking the actual hours spent working on each project.

Consequently, it is very important that all programmers know how to correctly use the Call Tracking System.



**Note:** For a complete explanation of the Call Tracking System, refer to the *Call Tracking System* document in the Eclipse Elibrary.

## Creating a New Tracker

While it is very rare that a programmer would ever create a new tracker, it is important that you know the procedures for correctly creating a tracker. The following example walks you through the steps of creating a new tracker.

### *Exercise 3.1: Assigning a Tracker to a Customer Account*

1. Assign a tracker in Eclipse to a customer account. This is how we know for whom a programming project is being done.

For Example: When Eclipse receives a call from a customer about a problem with their system or requesting a custom modification, a tracker is created and assigned to that customer's account.



**Caution:** It is very important to assign trackers to the correct account or you could end up putting code onto a customer system that had no need for that code.

2. Assign Internal Eclipse trackers to the correct customer "account" or "branch" of Eclipse from which the tracker is initiated.
  - For example, if you as a programmer were actually initiating an internal tracker you would assign it to the "Eclipse Boulder" customer account, because you're located in the Boulder office. However, if someone in the corporate office were initiating an internal tracker, they would assign it to the "Eclipse Shelton" customer account, because they're located in the Shelton office.



**Caution:** We should NEVER assign a tracker to a user or vendor account within Eclipse. This functionality exists only for our customers' use.



**Note:** An internal tracker is a project or job that needs to be completed for Eclipse as a company and is in no way affecting or needed on a client's site.

### Exercise 3.2: Creating a New Tracker

1. To display the Call Tracking Entry screen shown below select **Customer Activity Log Entry** on the F4-A/R menu or key **F4-Y**.

Call Tracking Entry											
ID: _											
Customer :				Contact :				Next Action :			
Category :				Priority :				Final Action :			
Work Area :				Ext Status :				Expected Dt :			
Sub Area :				Int Status :				Delivery Dt :			
Source :				Sec Level :							
View	Edit	Log	FInd	Forward	Append	Print	CopY	ConTact	Hours	PrOgram	
SoRt	Show App	Close	Notes	Keys	InQ	Bid	SuM Hrs	QUote	SpaWn		

2. The first thing that Eclipse prompts you for on this screen is an "ID." To start an internal tracker assigned to the Boulder office, type **eclipse boulder** in the **ID** field and press **Enter**.
3. Because there is an exact match for the ID, the system automatically displays the Call Tracking Select screen shown below and fills in a type of **New**. This lets you tell the system that you want to create a new tracker. Press **Enter** on the Call Tracking Select screen to start the new tracker.

Call Tracking Entry																	
ID: ECLIPSE BOULDER																	
TZ: MST																	
Customer :Eclipse Boulder				Contact :				Next Action :									
Category :				Priority :				Final Action :									
Work Area :				Ext Status :				Expected Dt :									
Sub Area :				Int Status :				Delivery Dt :									
Source :				Sec Level :													
<table border="1" style="margin: auto;"> <thead> <tr> <th colspan="2">Call Tracking Select</th> </tr> </thead> <tbody> <tr> <td>Type :</td> <td>New</td> </tr> <tr> <td>Word String :</td> <td></td> </tr> </tbody> </table>												Call Tracking Select		Type :	New	Word String :	
Call Tracking Select																	
Type :	New																
Word String :																	
View	Edit	Log	FInd	Forward	Append	Print	CopY	ConTact	Hours	PrOgram							
SoRt	Show App	Close	Notes	Keys	InQ	Bid	SuM Hrs	QUote	SpaWn								

- A tracker ID is assigned and the Call Tracking Entry screen is filled in with all of your and the customer default information, similar to the example shown below.

Call Tracking Entry			
ID: BCR726	Cust ID: 1948	TZ: MST	Entered: 06/28/2001-10:34am User
Customer :Eclipse Boulder	Contact :-	Next Action : User	
Category :APPLICATION	Priority :-	Final Action: User	
Work Area:	Ext Status:Newitem	Expected Dt :	
Sub Area :	Int Status:Newitem	Delivery Dt :	
Source :INTERNAL	Sec Level : 99		
All Comments			
View	Edit	Log	Find
Forward	Append	Print	Copy
Contact	Hours	Program	
Sort	Show App	Close	Notes
Keys	Inq	Mid	Su. Hrs
Quote	Spa. n		

- The “1948” in the **Cust ID** field is the Eclipse internal ID for the Eclipse Boulder customer record. You will learn more about internal IDs in the *Introduction to the UniVerse Database* chapter of this manual.
- The **Category**, **Source**, **Priority**, **Ext Status**, **Int Status**, and **Sec Level** fields are automatically filled in if defaults have been set up for your User ID in User Maintenance.



**Note:** If you are entering an internal tracker, you can leave the **Contact** field blank, because your User ID is already displayed in the **Entered** field. However, if you are creating a non-internal tracker assigned to a customer account, you **must** fill in the **Contact** field with the name of the customer’s on-site person with whom we can discuss this tracker.

4. Press **Enter** until the cursor reaches the **All Comments** field. This is where you will type the “initial comment” of your new tracker. The initial comment is what drives the life of the project from this time forward. If the initial comment doesn’t contain all of the information we need to accomplish the project, people will waste time trying to find out what the true issue is, or even worse, waste time programming something totally incorrectly for the customer.

- The initial comment of a tracker should always answer the following questions:
  - **Who** is the person asking for or forcing this project through?
  - **Why** are you writing this tracker? What is the “problem” we are trying to solve by changing an Eclipse program?
  - **What** is the proposed solution to the problem stated by the contact of this tracker? Is this a bug? Semi-Custom mod? True-Custom mod?

- **When** should this project be complete and why are we under this time constraint? (Is it causing the customer great pain or is it easily worked around at this time?)
  - **Where** should these changes be made? Only in develop, on site and in develop (semi-custom), or only on site (true custom)?
  - This should be a comment that you would feel comfortable with the client reading, because they will be able to see everything you just entered in this “initial comment” on the customer support site of [www.eclipseinc.com](http://www.eclipseinc.com) and [www.eclipsesupport.com](http://www.eclipsesupport.com).
5. After you have completed the initial comment, press **Esc**. The Forwarding screen automatically appears. Eclipse uses this screen for designating who should be seeing or working on this tracker.



***Note:** You can also display this screen by pressing the **Forward (Alt-F)** hot key on the Call Tracking Entry screen. In this case, however, the Forwarding screen appeared automatically, because the Forward To list contained no users and Eclipse will not let you exit a new tracker that is not closed unless there is at least one name on the Forward To list.*

### ***Exercise 3.3: Using Call Tracking System Forwarding***

1. Enter your name and your training manager’s name in the **Forward To** column.
2. After you enter a name, Eclipse automatically enters **Newitem** in the **Status** column. Press **F10** in this field to view all the valid statuses and select another one.



***Note:** The “Newitem” status is determined by the control record.*

- When assigning a tracker to you, your manager will use one of the following statuses:
  - **Hot** has an expected completion date within 24 hours of assignment to you (the programmer). Hot is a “drop everything” status and the tracker must be addressed immediately. It implies that the problem stated within that tracker is preventing that client from doing their day-to-day business.

- **Urgent** has an expected completion date within one week of being assigned to you. This status implies that although the client can still perform most business tasks, something is happening that is making life very difficult. This is not a “drop everything” status, but the tracker should be addressed as soon as possible. One week is the very maximum for an Urgent tracker.
- **High** is one of the two most common statuses you will see at Eclipse. This status is usually used for a bug at a site. High usually (but not always) has an expected completion date within 2 to 3 weeks of it being assigned to you. The date will be determined by your current workload and the size of the project.
- **Medium** is the other most common status for programmers. This status is used for things like custom modifications to a site or enhancements to the Eclipse package. If it is assigned to you on a bug fix tracker, it will have an expected completion date within 3 to 4 weeks of being assigned. For custom work this same time frame is also valid but may be extended up to 6 weeks. For enhancements it can be anywhere from 3 to 8 weeks, depending on the size of the project.
- **Low** is a status you will rarely see. This status is used for items that really are not affecting the Eclipse system in any way but at some point need to be addressed. The normal expected completion date for a “bug” with a “low” status is 2 months, but you may not even get an expected date. It more than likely is a tracker that your team supervisor wants you to take a look at if you run out of all other projects.



***Note:** With respect to programming, status and priority mean the same thing for a tracker. Technically this is your “followup status,” but programming supervisors use this field to let you know the priority they have assigned to the project. Throughout this manual, these terms are used interchangeably and mean the same thing.*

3. For your training manager’s entry, you’ll change the follow up status from **Newitem** to **F.Y.I.** This status lets the designated recipient know that the tracker is being forwarded just to keep them informed and action on their part is not expected.
4. Notice that the **Final Action By** and **Action Req’d By** fields on the Forwarding screen are both defaulted to you. This same information also appears on the main Call Tracking Entry screen in the **Final Action By** and **Next Action By** fields.

- Because there can be multiple people on a tracker's Forward To list, it is very important that it be very clear who is responsible for the next action. If your ID is in the **Next Action By** field, everyone is waiting on you to do something with this project before the next step is taken. Until you make an append (we'll talk about how to do this next) and change the "next action by," you're totally responsible for completing this project's "next action."
  - The person entered in the **Final Action By** field is responsible for closing the tracker when everything is complete. This is usually the person who created the tracker but not always.
  - Normally, a tracker will come to you from your programming team supervisor.
    - In the tracker header, your User ID will be in the **Next Action By** field.
    - There will be a date in the **Expected Completion Date** field. This date is the latest that you can finish your tracker. It is your "due date."
    - In the **All Comments** section, there will be an append saying "Assigned to: [Your User ID]."
    - The assigned **Status** will be one of the five described in step 6.
5. Press **Esc** to save the information entered on the Forwarding screen and return to the Call Tracking Entry screen.



***Note:** In Eclipse pressing **Esc** saves all changes made on a screen, while pressing **F12** aborts without saving. Also note that if you change a field by accident and have not terminated that input yet, you can press **Ctrl-R** to refresh the value that was initially in that field.*

6. Press **Esc** again.



***Note:** Step 6 is not necessary if you got into the Forwarding screen using **F2-S-C** rather than **Alt-F**.*

7. Eclipse automatically displays the Tracking Hours screen, where you need to assign a task code to the time you spent in this tracker. Eclipse automatically displays the amount of time the tracker has been open, but you can change that number, if necessary. It is very important that you enter the correct number of hours and the correct task code, which explains why you had the tracker open. The task codes you will be using are:
- **TRAINING** - Use this code for any time you spend in training classes, working through this manual, in training meetings, etc.



- **DEVELOP** - Use this code any time you're programming a new module within Eclipse that is NOT billable to the customer.
- **CUSTOM** - Use this code for the time you spend on a tracker (programming, researching, documenting, testing) that is billable to the customer, whether it's a true custom or a semi-custom mod.
- **INTEGRATE** - Use this code for the time you spend integrating code done on a separate tracker to another place. INTEGRATE should not be used on the original tracker that coding changes were made against to get that code onto site.
- **RESEARCH** - Use this task code only for the time you spend on a tracker that doesn't require any coding changes and is not billable to the customer.
- **BUGFIX** - Use this code for the time you spend on a tracker changing code that was a bug (not enhancement, custom, or new feature).



***Tip:** The number of hours defaulted in the Tracking Hours screen can happen two ways. The first is just like above for the number of hours you were actually in that tracker. The second is the "stopwatch" which we will learn about in a few minutes.*

8. Log your time in this tracker to the TRAINING task code.
9. Press **Esc** to save the changes, exit the Tracking Hours screen, and send the tracker on its way.
10. Each person listed on the Forwarding screen will receive a message in the Eclipse Message System indicating that this tracker is now in their User Job Queue. You will receive a message every time someone adds your name to the Forward To list of a tracker or someone appends to or changes a tracker whose Forward To list your name is on. Any time a tracker in your job queue changes, you receive a message.



***Caution:** While this method is quick and good for reading the comments recently added to trackers, you should never actually work on a project (tracker) from the message system. The primary purpose of the message system is to notify you that something changed. You should always "work" from your User Job Queue, where you can sort and work on the trackers in order according to their assigned expected dates and priorities.*

### ***Exercise 3.4: Creating an Internal Tracker***

In this exercise you will create an internal tracker that you will use throughout your initial programmer training for tracking your hours.

1. Display the Call Tracking Entry screen.
2. Start a new internal tracker assigned to the Eclipse Boulder office.
3. Enter the following text as the “initial comment” for your tracker:  
  
[Your Name] - This is the tracker I will use to log my hours and to which I will attach my programs during my initial Eclipse training. When I have finished this training I will close this tracker and delete all of my temporary programs.
4. Press **Esc**.
5. Enter your name and your training manager’s name in the **Forward To** column of the Forwarding screen.
6. Change the followup status for your training manager to **F.Y.I.**
7. Press **Esc** twice.
8. Enter the task code **TRAINING** and check that the number of hours displayed on the Tracking Hours screen is accurate.
9. Press **Esc** to save your changes and send the tracker on its way.

***Congratulations!*** You just created a perfect (or close to perfect) tracker within the Eclipse system.

# Viewing Trackers

Your User Job Queue contains a copy of each tracker for which your name is on the **Forward To** list. There are two locations from which you can view your trackers:

- The Message System
- Your User Job Queue

## *Exercise 3.5: Viewing Trackers from the Message System*

When your name is on the **Forward To** list of a tracker, every time that tracker is updated you receive an instant Eclipse message. The message scrolls across the bottom of your Eclipse screen and is added to your received messages in the Message System. A message of this sort looks like the following example:

- “JQ Rec’d: EL#ABC123 - Status:Newitem”

The message shown above indicates that your job queue received a copy of tracker number ABC123 (which permanently resides in the Entity Log) with an assigned status of **Newitem**.

To view an updated tracker from the Message System:

1. Select **Message System** on the F2-System menu (**F2-M**).
2. Press the **Received** hot key (**Alt-R**) to view all your received messages.
3. Position your cursor on the tracker message and press the **View** hot key (**Alt-V**).
  - Eclipse displays the designated tracker in the Call Tracking Entry screen.
4. After reading the message appended to the tracker, press **Esc** to exit the tracker and return to the message system.
5. With the cursor still positioned on the tracker message, press the **Delete** hot key to delete the message.
6. When you are finished checking your received messages, press **Esc** to return to the message screen and press **Esc** again to exit the message system.



**Caution:** While this method is quick and good for reading the comments recently added to trackers, you should never actually work on a project (tracker) from the message system. The primary purpose of the message system is to notify you that something changed. You should always “work” from your User Job Queue, where you can sort and work on the trackers in order according to their assigned expected dates and priorities.

### ***Exercise 3.6: Viewing Trackers from your User Job Queue***

Whenever your name is on the **Forward To** list of a tracker, a copy of that tracker resides in your User Job Queue. The tracker remains in your queue until you remove your name from the **Forward To** list.

In other words, your User Job Queue contains all the trackers that for which your name is on the follow up list. This is where you should go to see what you should be working on and when each project is supposed to be completed.

To view your User Job Queue:

1. Select **User Job Queue Viewing** on the F2-System menu (**F2-Q**), or
2. Press **Shift-F3** from anywhere within Eclipse.
  - This displays the User Job Queue Viewing screen, with your User ID in the **User ID** field, today's date in the **End Dt** field, and "Followup" in the **SelType** (Selection Type) field.
3. Press **Enter** three times. (Press **Enter** two times if you are coming from **Shift F3**.)
  - This moves the cursor through the header fields without changing the default data and then displays your trackers.
4. Press the **Sort By** hot key (**Alt-T**) and choose the "Expected Date" sort option. This sorts your trackers so the ones at the top are the ones you must complete first.
  - The trackers are sorted as follows:
    - Trackers that you have not read since they were last updated appear at the top of the list and have a caret ( ^ ) preceding the tracker description. These are followed by remaining trackers.
    - Once you go into and out of a tracker displaying the caret ( ^ ), it will go away (you've now seen the latest changes). The next time you sort the queue, the tracker will be listed with the second group.
    - Each of these two groups is sorted by expected date, with the oldest date listed first and most recent date listed last.
    - Trackers with the same expected date are sorted by the status assigned to your name on the Forwarding screen.

5. Press the **Change View** hot key (**Alt-G**) and select the “Expected Date” option.
  - The screen shown below now shows each tracker’s expected completion date, status and the first line of the tracker comment.

User Job Queue Viewing		
User ID	User	- User
Exp Date	Status	Trackers
	Newitem	Programming: Complete Training Manual Exercises

1 of 2

View	Edit	Select	New Item	Edit Item	Close Item	Forward	Print Q
Change View	Sort By	Expand	Append	Display Opt	Expand Line	Review	

6. To open the tracker, position the cursor on the tracker and press the **Edit Item** (**Alt-I**) hot key.
  - The complete tracker is displayed in the Call Tracking Entry screen.



**Tip:** In addition to using the Page Up, Page Down, and Arrow keys to navigate the job queue, you can use your mouse to click on the line you want to select. You can also use the mouse to click on hot keys rather than using the Alt-Key combination.

You now know how to enter a new tracker, see what trackers have changed, and look at a list of what projects (trackers) you are working on in expected date order. The next time that you come into a user job queue, the program will remember how you sorted and what view you displayed. Programmers should always sort by **Expiration Date** and view by **Expiration Date** because it provides a perfect “list” of the order you should work on each queue.

# Tracker Stopwatch

While a tracker is open, Eclipse tracks the time. When you close the tracker, Eclipse prompts you to assign a task code and record your time worked. It is not necessary, however, to keep a tracker open while you are working on it.

The Tracker Stopwatch feature lets you track the time you are working on one or more trackers without keeping the trackers open. (The Stopwatch is displayed on your screen.)

## *Exercise 3.8: Using the Tracker Stopwatch*

1. To display the Tracker Stopwatch screen press **Shift-F10** from anywhere in Eclipse.
  - If you press **Shift-F10** from a Call Tracking Entry screen that has a tracker displayed, Eclipse opens the Tracker Stopwatch screen and automatically adds that tracker to the list of active trackers, along with the amount of time you have already spent in the tracker. If the tracker was already on the stopwatch, then your cursor will be placed on that line.



**Caution:** *It is important that you add your tracker to the stopwatch before editing another tracker or pushing any levels, or all the time you were in the tracker up to that point will be lost.*

- If you press **Shift-F10** from any other screen, Eclipse opens the Tracker Stopwatch screen and displays the current list of trackers on the stopwatch. To add a new tracker to the list, position the cursor on a blank line and enter the tracker number in the **Trackr#** column. The cursor will either be on the first non-paused tracker or the first line.
  - Once you have entered a tracker on the Tracker Stopwatch screen, you can go anywhere you want without losing the amount of time you've been working on the tracker.
2. Press **Esc** to exit the Tracker Stopwatch screen and return to the tracker from which you pressed **Shift-F10**.
    - The stopwatch continues tracking time for active trackers listed on the screen. You can always go back into the stopwatch to stop or pause the active trackers if you take a break or work on a different tracker for awhile.
  3. To re-display the screen at any time, press **Shift-F10**.
  4. To pause the stopwatch for a particular tracker, position the cursor on the tracker and press the **Pause** hot key.
  5. To restart the stopwatch for a paused tracker, position the cursor on the paused tracker and press the **Start** hot key.

6. To stop the stopwatch for a tracker and record the time worked, you can use the **Stop** hot key, but there is another preferred method described in a following topic.
7. Press **Esc** again to exit the tracker and return to the User Job Queue Viewing screen.
  - You will now be in the User Job Queue with the “^” gone from the tracker you just edited.
8. Press **Esc** again to exit the User Job Queue Viewing screen and return to the main menu.
9. Press **Shift-F10** to display the Tracker Stopwatch screen again.
  - The screen displays the current list of trackers on the stopwatch.

### ***Exercise 3.9: Adding your Tracker to the Tracker Stopwatch***

In this exercise you will add your tracker to the Tracker Stopwatch.

1. With the tracker from Exercise 3.7 displayed, add the tracker to the Tracker Stopwatch screen.
2. Exit the Tracker Stopwatch screen.
3. Exit the tracker.
4. Exit the User Job Queue Viewing screen.
5. From the main menu, display the Tracker Stopwatch screen again.
6. Note that your tracker is still listed.
7. Exit the Tracker Stopwatch screen again, but keep this tracker active. From this point on you should keep this tracker on the stopwatch any time you work on this manual.

### ***Exercise 3.10: Editing a Program and Tagging the Change to a Tracker***

Before doing any programming related to a tracker, you should add the tracker to the Tracker Stopwatch screen. Then you can start making the related programming changes using the Program Editor.

1. To display the Program Editor screen shown in Figure 3.5 select **System Programming** on the F2-System menu, and then select **Program Editor** on that menu.
2. Enter **UBP** in the **File Name** field.
  - “UBP” means “user-defined basic program.” This and other valid entries for the **File Name** field are explained in the *Program Editor* chapter.
3. Enter **INI.PROG.MANUAL1** (where INI is your first, middle, and last initial) in the **Edit Program** field.
  - Eclipse displays a Program Names prompt with the option “New.”

4. Press **Enter** to confirm that this is a “New” program.
5. Enter **E** (Edit) in the action field. This field is not named, but shows a string of action codes.
  - Entering “E” in this field displays the Eclipse Program Edit screen. When creating a new program, as you’ll be doing in your next exercise, a seemingly blank screen with a number 1 in the upper left corner appears.
6. Type “\*THIS IS A TEST PROGRAM” on line 1.



**Note:** You will learn more about writing and editing programs in the *Program Editor* chapter.

7. Press **Esc**.
  - This saves the program and displays the Program Change Log Entry screen.
  - The Program Change Log Entry screen (shown below) automatically appears when you save and exit a program. Use this screen to document the program change and tag the change to the tracker for which it was made.

The screenshot shows the 'Program Change Log Entry' dialog box. The title bar reads 'Program Change Log Entry=BCN399'. The fields are as follows:

Program :	INI.HELLO.WORLD	Version# :	0.01
FileName :	UBP	Entered By :	User
			06/28/2001 - 12:23pm
Actv Log # :			
AL Comment :			
Rel Level :			
Change Type:			

Below the fields is a large text area for 'Comment'. At the bottom of the dialog are buttons: 'View Log', 'View', 'User Queue', 'Log View', 'Copy', and 'Active Log Index'. To the left of the dialog is a sidebar with 'File Na' and 'UBP' fields, and buttons 'Log', 'Open L', 'Patch Create', and 'Send/Receive'. To the right are buttons 'cs', 'Compare', and 'Phantom Run'.

8. Describe the changes you made in the **Comment** field.
  - The comment you enter here is very important. This helps other programmers, QA, and Documentation track the changes made in each “version” of an Eclipse program. So, concisely and precisely describe the changes you made.
  - In this test case, enter “Program for training exercise.”
9. Press **Esc**.



- Because you did not fill in the **Actv Log #** field yet, the system automatically places the cursor in that field and displays the prompt “Valid Activity Log # Required.”



**Note:** Because trackers are stored in Activity Logs, “Activity Log #” is synonymous with “Tracker #.”

10. Enter the tracker number for which you made the programming changes in the **Actv Log #** field. If you can’t remember the tracker number:

- Press **F10** to display the Tracking Log Viewing screen. This screen lists the last 100 trackers you have edited. Position the cursor on the correct tracker and press **Enter**. Eclipse copies this tracker number into the **Actv Log #** field and fills in the **AL Comment** field with the first line of the tracker comment.



**Tip:** This **F10** “quick access list” of the last 100 trackers you’ve edited is available from every tracker # prompt in Eclipse.

- You can also press **Shift-F10** to display the Tracker Stopwatch screen, where the tracker you’re working on should be listed. Note the number, press **Esc**, and then enter it in the **Actv Log #** field. When you press **Enter**, Eclipse fills in the **AL Comment** field with the first line of the tracker comment.



**Note:** The tagging of the correct tracker to the program change log is the most important thing you will learn in this chapter. If you have any questions at all open up another test program with a new name and keep doing this until this is perfectly clear.

11. Press **Esc**. The Program Change Log Entry screen closes and Eclipse returns you to the Program Editor screen shown below.

File Name		Edit Program		Run Program	
UBP		INI.PROG.MANUAL1		INI.PROG.MANUAL1	
E,ED,C,CF,R,S,DS,T,C*,P,U...: C					
Version#	Last Updated By	Open to			
0.01[1]	User 06/27/01 14:43	User	1 Open Pgms		
Log	Open List	Close	Copy	Copy	Open
Run	Undo Chg	Docs	Compare		
Patch Create	Bld Pgm Idx	Find Obj	Show Calls	Phantom Run	
Send/Receive Data	Comm				

- The Program Editor screen now shows that this program is open to you and the action code is a “C.” This is part of Eclipse’s version control and will be discussed in detail in the *Program Editor* chapter.

- Eclipse only allows one programmer at a time to make a change to a program. This way we guarantee that the program has a single line of changes and that not more than one change happened within each version. However, if you keep a program open for a long time other people may be waiting to change that same program. Thus, it is imperative that you close a program as soon as possible.

12. Press the **Close** hot key (**Alt-C**).

- Eclipse closes the program and once again displays the Program Change Log Entry screen for this program.

13. Make sure that you filled in the **Actv Log #** field and entered a valid “reason for change” in the **Comment** field and then press **Esc**.

The program is now closed and available for other programmers to edit.

# Appending to a Tracker

As work on a project progresses, various people will append related information and milestones to the tracker. For example:

- A support person or QA tester appends a question or describe a problem for a programmer.
- A developer describes a new project or custom modification to be programmed.
- A customer adds additional information regarding their requirements for the project.
- A manager appends a message assigning the project to a programmer.
- A programmer or manager appends the decisions made at a design review.
- A programmer appends a question (or answer) directed to the customer, the Support office, a manager, a tester, a writer, or another programmer.
- Any of the people listed above appends an answer (or question) answer directed to the programmer.
- A programmer can append an explanation of the changes made to a program.



***Note:** Whenever you are the “Next Action” person on a tracker and think that you will not meet the expected completion date, it is your responsibility to make an append explaining why and suggesting a new expected completion date. When a new date is agreed upon, you can inform the customer in an “External” append.*

## ***Exercise 3.12: Using Tracker Appends***

1. To track progress, or to add comments, questions, or other information to a tracker, display the tracker in the Call Tracking Entry screen and use the **Append** hot key.
  - So far you’ve learned how to access the Call Tracking Entry screen from your Received Messages screen and from your User Job Queue Viewing screen. In this example, you’ll use another way.
2. Select **Customer Activity Log Entry** on the F4-A/R menu. (**F4-Y**)
  - The Call Tracking Entry screen appears.
3. Enter the tracker number in the ID field.



***Tip:** If you don’t remember the tracker ID, press **F10** and select from your quick access list of the last 100 trackers you’ve viewed.*

- The tracker appears with all the information and comments entered to this point.
4. Press the **Append** hot key.
    - This displays the Append Message screen. By default, the message entered on this screen will be an “Internal Append.”
  5. To change the type of append, use the **Append Type** hot key or key **Alt-T** and type in the message. The valid types of appends are as follows (although, for now let’s leave the type “internal”):
    - **Internal** - Only Eclipse employees (but every Eclipse employee) can see this append. The customer to which this tracker is assigned cannot see these appends.
      - Your appends should almost always be “Internal” (which is why it’s the default).
    - **External** - All Eclipse employees and the customer to which this tracker is assigned can see this append. The customer accesses the Call Tracking System through the customer support site of [www.eclipseinc.com](http://www.eclipseinc.com).
      - Use an “External” append to request information from the customer or to provide specific information to the customer.
      - Do not use an “External” append to discuss programming issues with your manager or other programmers.



**Caution:** *The “Original” comment and all External appends of the tracker can be seen by all contacts for this customer on the customer support site. Spell correctly, use good grammar, and be politically correct when making these appends. If you have any questions about making an append safe for all clients, please see your manager*

- **Secure** - Only the people you add to a “secure list” can see this append.
    - Enter the message to be appended and press **Esc**.
    - This option will automatically ask for the secure list.
    - You can display a message that is only appropriate for certain users.
6. Press **Esc** to save the updated tracker.
    - Eclipse sends a message to everyone on the Forward To list, notifying them that the tracker has been updated.



**Tip:** *Although a newly appended message appears in the **All Comments** field of the tracker before you exit the tracker, pressing **F12** instead of **Esc** will abort the addition of this append.*

- Eclipse also displays the Tracking Hours screen.

7. Enter a task code for the time worked and press **Esc**.
  - Eclipse records the time worked and, because the tracker you are exiting is also listed on the Tracker Stopwatch screen, then displays the prompt.
    - **Stop** - stops the stopwatch and removes the tracker from the Tracking Stopwatch screen.
    - **Reset** - resets the stopwatch to zero and leaves the tracker listed on the Tracking Stopwatch screen.



***Note:** If you press **Esc** to exit the Tracking Hours screen without entering a task code, the time is **not** recorded and Eclipse displays the Tracker Stopwatch screen. At this screen you can leave the stopwatch running for this tracker, pause the stopwatch, or stop the stopwatch.*

8. Press **Enter** to accept the “Reset” option.
  - Eclipse removes the tracker from the Tracker Stopwatch screen.

You now know how to use tracker appends to ask questions about a project and update other people on your status for a project.

### ***Exercise 3.13: Making an External or Secure Append to Your Tracker***

In this exercise you will make some appends to the tracker currently listed on your Tracker Stopwatch screen.

1. Display the Tracker Stopwatch screen.
2. View the tracker listed on the Tracker Stopwatch screen.
3. Display the Append Message screen.
4. Change the “append type” to “External Append.”
5. Enter the message: “This is a test external append.”
6. Save the append.
7. Display the Append Message screen again.
8. Confirm that your append type is “Secure Append.” A confirmation box will appear. Enter your and your tracking supervisor’s name to secure the list. Press the **Esc** key.
9. Enter the message, “This is a test secure Append.”
10. Save the Append.
11. Save your updated tracker.
12. Enter a task code and record your time worked.
13. Reset the tracker on the stopwatch.

### ***Exercise 3.14: Editing an Existing Append***

1. Display the tracker you have been working on.
2. Place the cursor on any of the 3 appends you created.
3. Press the hotkey **F2** for Edit Append.
4. The Append screen is displayed.
5. Add “This is line 2” to the Append.
6. Press **Esc** twice.

### ***Exercise 3.15: Closing a Project From Your Queue***

When you have completed a project, you need to fill in the tracker’s Problem Statement and Solution.

1. From the Call Tracking Entry screen, press the **Program** hot key.
  - This displays a screen that has two sections: one for the Problem Statement and one for the Solution. Regardless of how many appends a tracker has, this is where anyone can go to read a clear description of the problem addressed by the tracker and its final solution.
  - Eclipse automatically fills in the Problem Statement section with a copy of the original tracker comment, but this field is editable. Quite often, once a problem has been solved, the original comment does not adequately describe the problem. When appropriate, you should revise the text in the Problem Statement section to clearly describe the problem. This helps us gain a solid knowledge base so we do not waste time solving something twice.
2. When appropriate, revise the text in the Problem Statement section to clearly describe the problem.
3. In the Solution section, enter a detailed description that clearly describes the programming you did and answers all of the questions listed in step 4.
4. Following the solution, insert a blank line and then add a paragraph for the documentation and QA departments that tells them what to test and what to document. A well-written solution should answer the following questions:
  - Where was the change made (develop, on site or both)?
  - Who was the change made for (customer, future release)?
  - What kind of change was this (bug, custom, enhancement)?
  - What were the changes and enhancements (detailed description, human readable)?
  - What do QA and Documentation need to know about this project?
5. Press the **PG Complete (Alt-P)** hot key on the Problem Statement/Solution screen.
  - This displays the Call Tracking Detail Information screen.

6. Correctly fill out all information on this screen and press **Esc** to save your changes.
7. Press the **Append** hotkey to display the Append Message screen, with the text of the “solution” already filled in. At the beginning of this append, insert a description of where these changes were made and if you contacted the client and how.
8. Press **Esc**.
  - Eclipse automatically removes your name from the Forward To (which, in turn, removes the tracker from your queue) and assigns the “next action by” to your team supervisor.

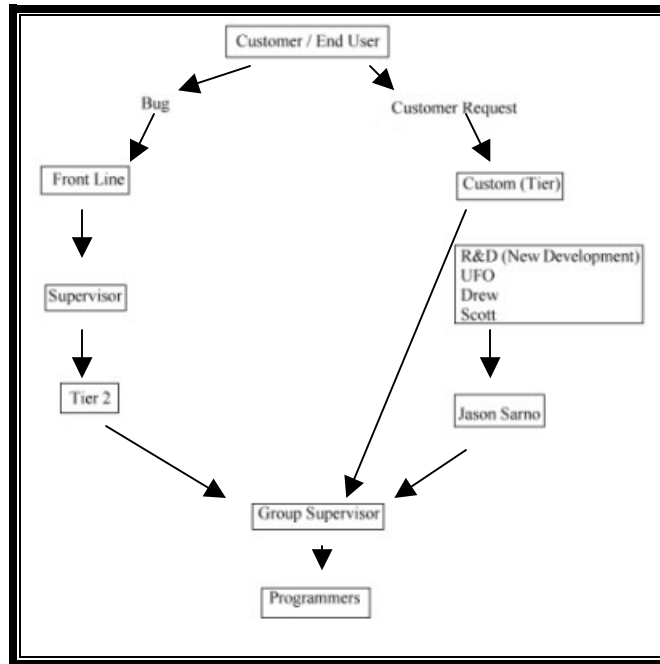


**Caution:** *It is very important that you follow all of the above completion steps every time you finish a tracker (project). This is where QA, DOC, and Support get all of the information from you as a programmer on how the system was changed and who was notified. Any lapses here cause major distress later in the release cycle.*



**Tip:** *If you inadvertently closed your test tracker from your queue, you can get it back by doing the following. Open the tracker from F4-Y (use **F10** for the last 100 trackers if you don't remember the number), display the Forwarding screen and add your name to the Forward To list. Then move the cursor to the Next Action By field and fill in your User ID. Press **Esc** until you have exited the tracker. It will now be listed again in your user job queue.*

# Summary



## Programmer Work Flow

In this chapter you learned about the life cycle of a project (or tracker) at Eclipse. This life cycle is basically as follows:

- Somebody in Eclipse creates a valid tracker for enhancement, custom modification, or bug fix. (See above graphic.)
- The creator/supervisor of trainer of the tracker sends the tracker to a programming team supervisor for approval.
- The programming team supervisor assigns the tracker (project) to a programmer (or multiple programmers on large projects).
- The programmer finishes the project and closes the tracker using the **Program** hot key from the body of the tracker.
- The tracker returns to the programming team supervisor.
- The tracker goes to QA for testing and approval of the changes.
- When required, the tracker also goes to Technical Publications for documentation.
- When testing is complete, the changes made in relation to the tracker are promoted into the correct release of Eclipse to get out to customer sites on upgrades.

This is the life cycle of all programming trackers within Eclipse. Some may go back and forth on the above steps but eventually (if programming was actually required and performed for that tracker) they all hit every step.



- You have also learned the very basics of the Eclipse program editor and how to correctly attach programming changes to the tracker that initiated them. You learned why the tagging of program changes to the correct tracker is essential at Eclipse for many reasons, especially to correctly handle release control.
- You performed a simple assignment to clarify all of the above information.

Although there are many hot keys in the Call Tracking System that were not explained, for almost everything you do, you will only need to know about what was presented in this chapter.

For complete information, refer to the *Call Tracking System* document.

## Frequently Asked Questions

### Question:

If I have two trackers in my queue, one being urgent priority with an expiration date of next week and one being medium priority with an expiration date of today, which one should I work on?

### Answer:

Ninety Percent of the time, the medium priority should be the tracker you should finish today in order to move onto the urgent tracker. However, if the medium tracker will take more than “today” to finish, you should talk the priority over with your supervisor. We almost always work based on the expiration date list.

### Question:

Why did this chapter cover what seems to be only about 20 percent of the Eclipse Call Tracking System?

### Answer:

The Call Tracking System was designed for all departments in Eclipse, so there is functionality built in that programmers will never need to use. Rather than clutter your head with too much information about this system, we felt it best to present just what need to know to do your job well in this complex system. As you become more familiar with the system, refer to the *Eclipse Call Tracking System* document in the Eclipse Elibrary for complete detail.

### Question:

The two programs we created don’t do anything. Why?

### Answer:

The intent of this chapter is just to introduce you to the Eclipse project flow and make sure you understand each piece of a project. It is much more important that you understand how to use the Call Tracking System and tag trackers to programs than how to write any code. The chapters to come will focus on developing your programming skills.

**Question:**

Why is the tracking of hours so important to Eclipse?

**Answer:**

“Time is money” in the software industry. Tracking hours enables the management team to see how our programming time is being spent. Is it being spent fixing bugs, assisting QA or Documentation, or developing new enhancements?

The most important way Eclipse makes money is by selling the Eclipse DMS. And to maintain our position as the leader in the DMS marketplace, we must continually improve our current package. This is part of your job as a programmer. You will also fix bugs, help QA and Documentation, and develop custom software for clients. It is very important that the management team make sure that we as developers are spending as much time as possible developing software that can be resold and as little time as possible on unrelated tasks. Logging hours on our projects is how the management team obtains this critical information.



**Note:** For a comprehensive explanation of the Eclipse Message System and your User Job Queue, refer to the **Message System, Job Queues and Logs** document in the Eclipse Elibrary.

## Assignment: Lesson 3

1. Go to your training manager to request the Chapter 2 assignment. Your training manager will create a test tracker and assign it to you.
2. Watch closely as your training manager creates this tracker. Do not be concerned when you see the “assignment” screen; this is only used at the team supervisor level to assign projects to individual programmers. If you have any questions at all about the correct methods for creating a tracker, this is a perfect time to ask them. If you notice an error in what the training manager is doing while creating the tracker, point it out. It may be a test!
3. When you get back to your own computer, you will see the instant Eclipse message that this new tracker has been assigned to you.
4. Open the User Job Queue Viewing screen and display the list of your assigned trackers.
5. Make sure the list is still displaying and sorting by Expected Date.
6. Open the tracker your training manager sent to you.
7. Add this tracker to the stopwatch.
8. Exit the Stopwatch screen.
9. Exit the tracker.
10. Go into the Eclipse program editor and create a one-line program named “INI.PROGRAM.MANUAL2” and save it.



*Note: Here again INI represents your first, middle, and last name initials and then .PROGRAM.MANUAL2 - From here on out in this manual this should be assumed any time you see a program name starting with INI.*

11. Tag the tracker from this assignment to this program and enter a change log comment.
12. Close the program.
13. make an internal, external, and secure append using our closing procedures.
14. Send the tracker back to your training manager.
15. Congratulations, you have just completed the full cycle of a tracker for a programmer at Eclipse.
16. Just to be safe, check with your training manager and make sure you completed all the steps of this assignment correctly.

# *Lesson 4*

## *Introduction to the Eclipse Program Editor*



### *Objectives*

**When you complete this lesson you will be able to:**

- Use the Program Editor
- Create a subroutine
- Distinguish between Basic Programs, Field Basic Programs, and User Specific Basic Programs
- Know other Eclipse Programming files

# Introduction to the Eclipse Program Editor

In this chapter we will discuss the next tool you will use to write code at Eclipse: the Program Editor. You will make all programming changes in this editor. The Program Editor can also be used to handle version control, send routines out to site, and check for changes made to each program in each version. Become very familiar with this tool, as you'll need it for the rest of your career at Eclipse.



**Caution:** Because the program editor is so tightly linked with version and release control it is imperative that you make all programming changes through this utility.

## Most Important Fields

The three most used fields in the program editor are the following (See Figure below):

- File Name
- Edit Program
- Option to Perform (the unlabeled field box in the center of the screen)

File Name		Edit Program		Run Program	
E,ED,C,CF,R,S,DS,T,C*,P,U...:					
Version#	Last Updated By	Open to	1 Open Pgms		
Log	Open List	Close	Copy	Copy	Open
Run	Undo Chg	Docs	Compare		
Patch Create	Old Pgm Idx	Find Obj	Show Calls		Phantom Run
Send/Receive Data		Comm			

## File Name

The File Name field directs the Program Editor to the program file you want to edit. The file you will use throughout this manual is UBP.



***Note:** UBP is the file where all totally custom Eclipse programming is done. Because you're in training and these programs are never to get out to any sites (from develop) we want them all in the UBP file. We will discuss the other Eclipse program files later in this chapter.*

## Edit Program

This field directs Eclipse to the record (program) you want to edit within the file that you designated in the File Name field. In the figure below, I have specified that I want to work on the JTS.TRAINING1 program that lives inside the UBP file. Remember the example spreadsheet that you created in Lesson 2? In PICK Basic the programs are just records within a file. In this case the record is the program name and the file it lives in is UBP.

## Option to Perform

In this field, select the option you want to run the edit for the selected program. The options are all listed and described below.

- **C** - Compile this program (edit program) using the Eclipse pre-compiler.
- **C\*** - Compile this program without using the Eclipse pre-compiler. This option will put your code straight into the BASIC compiler and should not be used very often. You will never use this option in this manual, however we will discuss areas of our code that differ from true PICK code.
- **CD** – Compile this program with debug window inputs.
- **CF** – Compile a list of programs (your active list). This option can be done after you do a “Find” from the program editor to compile all of the programs you found in that list.
- **CT** – Compile test(s) on the edit program. This option will perform a test compilation before activating the new code. It is very important when you compile high profile routines in develop or routines on a site. When a compile blows up, that code cannot be accessed until you complete a successful compile on that routine. As you can imagine, clients do not like when things will not work because we blew up a compile!
- **D** – Delete the edit program from the file you specified in File Name. This option will save a copy of the routine in the DP file.
- **DS** – Send a grid layout of the screen to the printer.
- **DB** – Database allows you to enter a UniVerse BASIC command and get help from the on-line UniVerse help documents.

- **DU** – **DU** let's you put in a Unix command and get help from the on-line AIX/UNIX manual on that command.
- **E** – Edit the “edit program” with the Eclipse program editor.
- **ED** – Edit the “edit program” with the UniVerse line editor (not recommended for use at Eclipse).
- **EF** – Use the Eclipse program editor to edit all of the programs in your active find list.
- **P** – Print the edit program to the active printer.
- **R** – Run the edit program from this editor.
- **S** – Invoke the Eclipse screen editor to create or edit a screen with the same name as the edit program.
- **T** – Drop down to Eclipse TCL (True Command Language).
- **V** – View the edit program in view only mode (no changes can be made).
- **EV** – Edit the “edit program” using the new GUI program editor.



**Note:** *This new editor will be covered in detail in **Lesson 15: New GUI Program Editor.***

We will go over many of the above options in greater detail throughout this lesson. Others, which you will not use very often, will be discussed in later lessons.

### ***Exercise 4.1: Creating A Subroutine***

It's time to create our first piece of true code at Eclipse. This code will be the simple “Hello World” that we all learn when using a new programming language. It will simply open a blank window, print out “Hello World” to the user and then go away.

1. Go into the Eclipse program editor (**F2-S-P**).
2. Put in a file of UBP.
3. Name this program INI.HELLO.WORLD.
4. Using the **E** option to edit this routine with the Eclipse editor, pull up your blank screen. See the figure below.

File Name		Edit Program		Run Program	
UBP		INI.HELLO.WORLD		INI.HELLO.WORLD	
E,ED,C,CF,R,S,DS,T,C*,P,U...: E					
Version#		Last Updated By		Open to	
				1 Open Pgms	
Log	Open List	Close	Copy	Copy Open	Run
Undo Chg	Docs	Compare			
Patch Create	Old Pgm Idx	Find Obj	Show Calls	Phantom Run	
Send/Receive Data		Comm			

5. Once you press <ENTER> to accept E as your option you will get a blank screen. You are now editing your HELLO WORLD program.
6. On the very top line of this program we will put the following:
  - “SUBROUTINE (PASSER)”



**Note:** This tells PICK that this is actually a subroutine and not a true program. Everything that runs in Eclipse except for phantom programs is a subroutine. This is how we handle the pushing of menu levels. A subroutine always returns control of the process (thread) to the person who called it. In this manner we can go anywhere throughout Eclipse as long as it's a subroutine with only one argument. Don't worry about any of the other details at this point, as we will cover this topic in detail in Lesson 8.

7. Next we need to put in our standard comments. At the top of every subroutine we have the following mandatory sections (all delimited by our dashed line):
  - Subroutine name,
  - Description of what this subroutine is suppose to do,
  - List of arguments, what the arguments are for, and if they are passed into or out of this routine,
  - Common variables used in this routine.



**Note:** Subroutine name is very easy; it's just a comment line saying what you named this subroutine. This field is here so that if you start multi-tasking and forget what routine you were in you can just go to the top rather than go out and back into the routine.

8. Put in the following code (after one blank line):
  - “\*\*\* Subroutine : INI.HELLO.WORLD”





**Tip:** There are two ways to get onto the far left side of the program editor for these comments. The first is to use your **<HOME>** key and then type **\*\*\*** and the comment you want after the stars. The second method is to use our shortcut **SHIFT-F7** which inserts a blank line where your cursor is, puts in three stars, and puts your cursor next to the stars so you can start typing the comment. The author recommends using **SHIFT-F7**, as you will be entering many comments.



**Caution:** It is imperative that the subroutine name be exactly one blank line below the **SUBROUTINE (PASSER)** line for Eclipse version control. Having this name at the direct line below the top line will cause major problems.

9. To go onto the description of this routine we need to end the name section. We use the dashed delimiting line to do just that. Make sure your cursor is on a blank line and press **SHIFT-F8**. This will draw a delimiting line where the cursor was on the screen.



**Caution:** Be very careful when you press **SHIFT-F8**, as it does not insert a blank line before drawing the dashed line. If your cursor is on a line with information or code on it the dashed line will delete that code and put itself in.

10. Under your new dashed line press **SHIFT-F7** and type in the description of what this routine will do. End that section with another dashed line.
11. Now start your arguments section with another **SHIFT-F7**. We only have one argument in this subroutine, which is **PASSER**. It should look like the following.
  - **\*\*\* PASSER** – This argument is not used in this routine.
12. Finish the argument section with a dashed line and start the common variables section by entering:
  - **\*\*\* COMMON VARIABLES**
  - **\*\*\* None are used in this routine**
13. End this section with a final dashed line and you've completed the mandatory header for all Eclipse programs. If you have any questions on what code you should have now please see code listing 4.1 later in this section.

## ***Exercise 4.2: Writing the Program***

Now let's write the actual program.

1. The first command you need for this program is **WINDOW**. This is an Eclipse subroutine that you will call to tell Eterm you want a blank screen opened up. For now we will not pass any arguments, which tells it to be a full screen window.

2. The second command you need for this routine is PRINT. This is a PICK command that sends the string after itself to the active standard out (in our case the window we just opened). To display “Hello World” on the blank screen simply insert the following line under your WINDOW command:
  - “PRINT ‘Hello World’”
3. To finish off this subroutine close the window using WINDOW.CLOSE. Select RETURN to give control to the calling program. You should have code that looks very similar to listing 4.1 below.



**Note:** Every WINDOW that’s opened in a subroutine must also be closed before leaving that subroutine.

#### Listing 4.1 – Full listing of INI.HELLO.WORLD

```

1:      SUBROUTINE (PASSER)
2:
3: *** Subroutine : JTS.HELLO.WORLD
4: *-----*
5: *** This subroutine will open a blank window and display "Hello World" to
6: *** the user in it.
7: *-----*
8: *** PASSER - This argument is not used in this routine
9: *-----*
10: *** COMMON VARIABLES
11: *** None are used in this routine
12: *-----*
13:
14:      WINDOW
15:
16:      PRINT 'Hello World'
17:
18:      WINDOW.CLOSE
19:
20:      RETURN

```

Again, this routine just opens a blank screen, prints out “Hello World” and closes that screen. The final line RETURN tells PICK that this routine is done running and returns control back to the calling program.



**Caution:** When writing an Eclipse subroutine, it is important that you never use the PICK STOP command. IF you do use it, then it will drop the user down to UniVerse command language and end the Eclipse session they were running.

4. To save your new program press <ESC>, put in a reason for the change, and tag this program to your test tracker we created in Lesson 2.



**Note:** Please make sure that your test tracker is also running in the stopwatch so that you can correctly log all of this time to “TRAINING”.

5. To test this routine you must first compile it. Remember that we are writing Eclipse code, so make sure to use the **C** option to compile this routine. If you typed everything correctly you should have a screen like the one shown below (successful compilation screen). If your compile was not successful please look to listing 4.1 and make sure you do not have any typos in the routine.

```
Connect Dial Disconnect Edit Configure Print Attachments DeBug Help Message Source
UBP INI.HELLO.WORLD Line: 22 of 22

Compilation Complete.
"INI.HELLO.WORLD" cataloged.

PRESS <RETURN> TO CONTINUE :
```

6. Once the compile is complete use the **R** option (**R**, <ENTER>) to run this program. This command will pull up the following screen shown below.

```
Run Subroutine Program: INI.HELLO.WORLD
PASSER.....
```

7. This screen is simply giving you (the programmer) a chance to “pass in” the argument **PASSER** with something other than an empty string to test your routine. In this case we never look to **PASSER**; so just press <ESC> and your program will run.

Now, unless you have very quick eyes or develop is running exceptionally slow, you didn't see anything. Why not? Let's evaluate the routine we just wrote. Where would this routine stop? That's correct: it never will. All we told it to do was open a window, print out the string "Hello World," and then close the window. Nothing was written in the routine that said "Wait until the user sees the message before going away". You as a programmer would have to write this into the code.

8. There are many different ways to make a routine stop; we will use the SLEEP command here. After you print "Hello World" put in SLEEP 5, which will tell the routine to sleep (or pause) for 5 seconds before it goes onto the next line of code.



**Note:** *SLEEP is another UniVerse function native to its BASIC language. If you would like more information on SLEEP press F9 while you're in the editor and just after typing SLEEP, it automatically will put you in the on-line UniVerse help for SLEEP.*



**Caution:** *Make sure you put the SLEEP command after the PRINT but before the WINDOW.CLOSE or you will just see what you saw before. See listing 4.2 for the fully revised HELLO.WORLD routine.*

#### Listing 4.2 – Revised listing of INI.HELLO.WORLD

```

1:  SUBROUTINE (PASSER)
2:
3:  *** Subroutine : JTS.HELLO.WORLD
4:  *-----*
5:  *** This subroutine will open a blank window and display "Hello World" to
6:  *** the user in it.
7:  *-----*
8:  *** PASSER - This argument is not used in this routine
9:  *-----*
10: *** COMMON VARIABLES
11: *** None are used in this routine
12: *-----*
13:
14:  WINDOW
15:
16:  PRINT 'Hello World'
17:  SLEEP 5
18:  WINDOW.CLOSE
19:
20:  RETURN

```

9. To test your revised HELLO.WORLD press <ESC> to save it, compile the latest version, and run it.
10. You should now get the “Hello World” statement on your screen for 5 seconds.



***Note:** Like everywhere else in Eclipse <ESC> is the key that tells the editor you want to save your changes. This takes all the changes you made since pulling up that routine and writes them out to disk. Because our version of BASIC (UniVerse) is not totally interpretive (it is compiled before being run for speed issues), we must compile this code. When you’re writing Eclipse code it is almost always put through our pre-compiler and then through the BASIC compiler. This is what the C command did for you. It also “catalogs” this routine in memory for fast access from all processes. If you were editing a program and wanted to abort the changes you had made (all of the changes in that session) you can use the **F12** key and you will be prompted, “Are you sure”. **F12** will undo ALL changes you’ve made since pulling up that program. We do not yet have a **CTRL-Z** function that just backs out one change at a time. If you forgot to press **F12** instead of <ESC>, we do have incremental “Undoes” that we will discuss later in this lesson.*

### ***Exercise 4.3: Copying, Inserting, and Deleting code from the Eclipse Editor***

1. Let’s pull up our “Hello World” routine one more time.
2. Go onto the line where SLEEP is currently located and press **ALT-INSERT**. You will notice that a blank line appears above the SLEEP command.
3. Notice that we now have a blank line right above the SLEEP 5 command. To delete this line press **ALT-DELETE** with your cursor on that blank line.
4. Now, put your cursor on the SLEEP line again and press **ALT-C** twice.
  - This command will highlight that line of code. This is our “copy” command. The first **ALT-C** is the “start” line, and the second is the last line you want copied (or moved).
5. Once you have a section of code highlighted, move your cursor to the line the code should start on and press **ALT-P** (Copy) or **ALT-M** (Move).
  - For practice let’s highlight just the line of code where your SLEEP command is (line 17 in Listing 4.2) by pressing **ALT-C** twice with your cursor on that line.
6. Next move your cursor to the line below the SLEEP command and press **ALT-M**. You will notice that you just moved the command down one line.

7. Repeat the highlight of this line of code (the one with SLEEP on it) and this time move your cursor down one line and press **ALT-P**. Notice that this time you have a second line of code exactly the same as your first, just one line below it. Let's go ahead and leave our "Hello World" routine this way (See code listing 4.3 for a full listing of this revised code).



**Tip:** *The difference between ALT-M (Move) and ALT-P (copy) after using the ALT-C, ALT-C to select code is that the ALT-M will remove the highlighted code from it's original position in the routine and put it where you said to start (the line your cursor was on when you pressed ALT-M) while ALT-P will leave the code where it was originally as well as add that same code in the starting position your cursor was on when you pressed ALT-P.*

8. Press **<ESC>** to save your changes and then compile the routine.
9. Now, run this routine one last time and you should notice that this time instead of pausing for 5 seconds it pauses for around 10 seconds. This is because you have two SLEEP commands in sequence, both of which say "pause for 5 seconds" because one happens right after the next it just looks the same as one SLEEP command of 10 seconds.



**Caution:** *In true programming, you would never write code like we just did. Using two commands in a row, like this exercise does, when one line of code would do the same is poor practice.*

### Code Listing 4.3 (Revised INI.HELLO.WORLD):

```
1:  SUBROUTINE (PASSER)
2:  ** Version# 0.01[3] - 04/02/2001 - 05:10pm - JASONS - develop
3:
4:  *** Subroutine : JTS.HELLO.WORLD
5:  *-----*
6:  *** This subroutine will open a blank window and display "Hello World" to
7:  *** the user in it.
8:  *-----*
9:  *** PASSER - This argument is not used in this routine
10: *-----*
11: *** COMMON VARIABLES
12: *** None are used in this routine
13: *-----*
14:
15:  WINDOW
16:
17:  PRINT 'Hello World'
18:
19:  SLEEP 5
20:  SLEEP 5
21:  WINDOW.CLOSE
22:
23:  RETURN
```

# File structure of Eclipse Programming Files (that are put through the Eclipse Pre-compiler)

So far everything we've been programming has lived in the UBP file. As explained earlier in this chapter, UBP stands for "User Specific Basic Programs." This file contains all basic code that will never be included in any Eclipse release. We use this file on site for true custom modifications, and we use this file in develop for any modifications we are doing that we do not want to accidentally get into a release.

Let's take a look at the other files you will need to know when programming.

1. The first (and lowest level) file is SP, which stands for System Programs. This file contains basic code that is vital to many things throughout the Eclipse system (such as the WINDOW subroutine). It also contains our display functions, print functions and other important functions.
2. The next level up for basic programs is BP. This file contains over 90% of Eclipse code and contains all subroutines that are not "system level" subroutines. Anything in develop in files BP or SP will be put into an Eclipse release at some point.
3. FBP stands for "Field (or Fixed) Basic Programs." We put any semi-custom (mods the customer paid for but that will eventually be in a release) or on site bug fixes in this file. We *never* use FBP in our development accounts. The reason we put code on sites in FBP is so that EVERY eclipse site on a release (such as 7.0.1.8) will have the exact same SP and BP file. Only UBP and FBP can be different from the base release.
4. Finally we have CBP, which stands for "Client Modified Basic Programs". This file holds any BASIC code that our actual clients have modified themselves. Eclipse personell will never edit a routine in CBP unless it is approved by their supervisor.

Because each file has a different use within Eclipse, a corresponding hierarchy exists to determine which programs will be compiled when the same program lives in more than one of the files. The pre-compiler looks in the following order to compile the routine: CBP, UBP, FBP, BP, and finally SP. Thus if I have the same routine in all files, the one in CBP will actually get compiled. If I have the same routine in UBP and BP, the one in UBP will get compiled. The reasons behind this structure should be clear from the previous paragraph explaining what each file does. If we've fixed a program on site, for example SOE.TOTALS, so that it displays properly on Canadian orders and the pre-compiler still compiled in the BP version, our fix would never show up (we only edit FBP or UBP on site remember). So it must look to FBP first. On the same note, if we added any custom code to a program on site, that code should always take precedence over any other changes.



**Table 4.1: Eclipse Programming Files**

<b>SP</b>	<b>System Programs</b>
<b>BP</b>	<b>Basic Programs (90 % of our code)</b>
<b>FBP</b>	<b>Field (or Fixed) Basic Programs</b>
<b>UBP</b>	<b>User Specific (Custom) Basic Programs</b>
<b>CBP</b>	<b>Client Modified Basic Programs</b>

### ***Exercise 4.4: Understanding the Difference Between BP, FBP, and UBP in Relation to the Pre-Compiler***

Let's change gears for a minute and do an exercise to help you get a better understanding of the hierarchy we just talked about on the programming files. Let's start by creating a new subroutine in develop in BP.



**Note:** *This exercise will require answering “yes” to the prompt “will this ever be included in an Eclipse release.” But don't worry because we will delete this when we are done with this exercise.*

1. Name the routine `INI.FILE.TEST`.
2. Explain to the user that this routine will just print different words based on what file it lives in.
3. Complete the required comments. Again we will not use any arguments, but we must include one generic “PASSER” at the top.
4. Open a blank window.
5. Print out the words “I live in BP” on the screen.
6. Hold the window open 7 seconds.
7. Close the window and return control to the calling routine.
8. Press <ESC> to save this routine in BP.
9. Fill in the required change log comments.
10. Attach this program to your training tracker.

## Final Listing for INI.FILE.TEST (In BP File)

```
1: SUBROUTINE (PASSER)
2:
3: *** Subroutine : JTS.FILE.TEST
4: *-----*
5: *** This subroutine will simply open a blank window, display the message
6: *** "I live in BP", stay open for 7 seconds and then close the window.
7: *** I am writing this routine for exercise 4.4 in the programmers guide.
8: *-----*
9: *** PASSER - This argument is not used in this routine.
10: *-----*
11: *** COMMON VARIABLES:
12: *** None are used or set in this routine.
13: *-----*
14:
15: WINDOW
16:
17: PRINT 'I live in BP'
18:
19: SLEEP 7
20:
21: WINDOW.CLOSE
22: RETURN
```

11. Compile this program and run it. You will notice that you get a blank screen with the words "I live in BP" on it.
12. Now that we have this routine running let's copy it over to FBP. Go into the program editor and make the file BP. The "Edit Program" should still be INI.FILE.TEST.
13. Press the **ALT-P** (Copy) Hotkey, fill in a file name of "FBP", and leave the program name INI.FILE.TEST.
14. You will immediately be asked for a reason for copy, and get the program change log. Fill it out and tag this to the correct training tracker.
15. Make sure that the file is still FBP and that the edit routine is still INI.FILE.TEST.
16. Edit this routine (in FBP) by changing the one line that says "I live in BP" to "I live in FBP".
17. Save this routine.
18. Compile the routine and notice that the compiler automatically compiles the FBP routine and not the one you originally had in BP. To prove that it's not based on the "File", go ahead and change the "File" to your "BP" version and compile again. Notice it still uses the FBP version.

19. For one final step run this routine now and you'll see a blank window with the words "I live in FBP" on it.

This may seem like a very strange exercise but it is very important that you understand this file hierarchy or you could end up making changes in the wrong spot in a customers system.

## Actual PICK code

Another file relevant to your programming at Eclipse is the OC file (Object Code). OC contains all of your code before we put it into the UniVerse BASIC compiler. This file is where the “true” PICK code for Eclipse lives.

### ***Exercise 4.5: Differences Between Eclipse and PICK***

1. To see some differences in our code from PICK code, let’s look in OC at our INI.FILE.TEST.
2. Go into the Eclipse program editor.
3. Change the file to be “OC” and leave the “Edit Routine” as INI.FILE.TEST.
4. Press **V <ENTER>** to pull this routine from OC up in view only mode.



***Caution:*** It is very important to use the “view only” mode in the OC file or all changes you make will be lost on the compile of this subroutine. Remember the pre-compiler only writes out to OC, it never reads what’s in there to make sure you didn’t change anything.

You should see the code from listing 4.4.

#### Listing 4.4: INI.FILE.TEST from OC

```
1:  SUBROUTINE (PASSER)
2:  $INCLUDE CC COMMON
3:  *
4:  *
5:  *
6:  *
7:  *
8:  *
9:  *
10: *
11: *
12: *
13: *
14: *
15: *
16:  CALL WINDOW("","","","")
17: *
18:  PRINT 'I live in FBP'
19: *
20:  SLEEP 7
21: *
22:  CALL WINDOW.CLOSE("")
23:  RETURN
24: * Compiled by JASONS on 06/14/01 14:11 from BP:JTS.FILE.TEST
25: *** Version# 1[1] - 06/14/2001 - 02:11pm - JASONS – develop
```

5. Notice the differences between line #2, all of the comment or blank lines, line 22, and finally lines 24 and 25. Notice that on the WINDOW and WINDOW.CLOSE lines the pre-compiler automatically added the word CALL in front of them. Why? It did this because it found those to be Eclipse subroutines (they live in OC as well). Instead of requiring you to type CALL every time you want to call an external routine, we have the pre-compiler do it for you.
6. Also, notice that it filled in empty parameters within parenthesis (“”) for the subroutine calls. This is the number of arguments this subroutine expected. In this case WINDOW.CLOSE expected one argument and we didn’t supply any.

Well, while our subroutine may be smart enough to handle an empty string for a parameter, it still must be called with the correct number or we would receive a run time error saying “Subroutine WINDOW.CLOSE called with X arguments when Y were expected”. So again, instead of making you the programmer type all of these extra, unneeded “” parameters, the pre-compiler filled them in for you. If you had supplied too many parameters for this subroutine, the file would have truncated the parameter list at the highest number the subroutine could take.



**Caution:** *While this parameter addition/subtraction is very nice in terms of not having to type this extra parameter(s) every time, it also makes it necessary for you to re-compile all code that calls this subroutine whenever you change the number of arguments a routine can take. We will discuss how to do this in detail in Lesson 6 “Standard Eclipse operating environment”.*

# Other Eclipse programming files

There are four other programming files that you will need to know about before the end of this lesson. The first three are where we store code such as our pre-compiler and any macros that we want run in true PICK Format. These files never get put through the Eclipse pre-compiler. They are instead compiled with the normal UniVerse BASIC compiler.

The three files follow:

1. MP – Macro Programs
2. PP – Pre-compiler Programs (anything to do with the pre-compiler lives here)
3. CC – This is where we store all the common (non-program) related information.

We don't need to go into any more detail on these files in this lesson as long as you know they exist. We will cover each one of them in a later lesson.

The last file you need to know about is the SCREEN file. This is where all of our Eclipse screens live. We will be designing and editing screens throughout this manual and they will all live in the SCREEN file.

# Displaying Strings in Different Places Around the Screen

So far in our HELLO.WORLD example we have always displayed the text in the upper left hand corner of the screen using the PRINT command. The reason for this positioning is that PRINT puts the string out wherever the cursor is located. Because we started with a blank window, the cursor started in the top left of the window. If we wanted to move the string around in the screen, we would have to use multiple PRINT commands and spaces to get the cursor where we wanted to print our string. Unfortunately if we then wanted to print something up above this point we could not do it. No way exists to move the cursor back up to a point before its current location.

## ***Exercise 4.6: Moving Text Around the Screen with the PRINT Command***

To play with this function let's edit our HELLO.WORLD routine by changing it to print "HELLO.WORLD" on the 5<sup>th</sup> line, 20 spaces from the left.

1. Edit your INI.HELLO.WORLD and add a single PRINT command on the line above the PRINT "Hello World".
2. Copy this line 3 times to get a total of four PRINT lines before the actual printing of the string.
3. Change the PRINT "Hello World" line to PRINT SPACE(19):"Hello World". Your code should look like Listing 4.5 now.



**Note:** The SPACE function we used in step 4 is another handy PICK function that will put out a string of spaces with the integer number you passed into it – in our case a string of 20 spaces.

4. Save, compile and run this routine. This time you will get the results shown in the figure below with the string "Hello World" down 5 lines and 20 spaces in from the left. It appears in this location because the first 4 prints put out carriage return and line feeds (moving the cursor down one line each time). Then we added the SPACE(19) (for 20 spaces) with a ":" between it and out "hello world" string. This ":" tells the PRINT command not to put out a carriage return and line feed after it prints the string. So the 19 spaces and the string "Hello World" are kept on the same line. You can also end a PRINT command with a ":" which will make the next PRINT command start at the same place the previous one did. So to get the same results we just did we could have inserted a PRINT SPACE(19): above the PRINT "Hello World" and the program output would not change.



### Code Listing 4.5 (Newly Revised INI.HELLO.WORLD):

```
1:  SUBROUTINE (PASSER)
2:  ** Version# 0.01[3] - 04/02/2001 - 05:10pm - JASONS - develop
3:
4:  *** Subroutine : JTS.HELLO.WORLD
5:  *-----*
6:  *** This subroutine will open a blank window and display "Hello World" to
7:  *** the user in it.
8:  *-----*
9:  *** PASSER - This argument is not used in this routine
10: *-----*
11: *** COMMON VARIABLES
12: *** None are used in this routine
13: *-----*
14:
15:  WINDOW
16:
17:  PRINT
18:  PRINT
19:  PRINT
20:  PRINT
21:  PRINT SPACE(19):'Hello World'
22:
23:  SLEEP 5
24:  SLEEP 5
25:  WINDOW.CLOSE
26:
27:  RETURN
```



Now let's look at how we could have accomplished the same thing without using all of the extra print statements.

## Introduction to PRINT @ Commands

PICK comes with another PRINT function that can only be used on screen display. This is the PRINT @ function. You follow the @ sign with two parameters in parentheses. These parameters are the X and Y (Horizontal and Vertical) positions you would like this string displayed on. So if I wanted to print "Hello World" at line 5, on the twentieth position, I don't have to enter four blank PRINT lines and SPACE(19) to get to that position. Instead I could write one line of code that says:

- "PRINT @(20,5):'Hello World'"

Again, the program output would not change (see the figure below).

### *Exercise 4.7: Using Print @ Commands*

1. Let's make one more edit to your INI.HELLO.WORLD routine. Delete all of the PRINT lines.
2. Insert the PRINT @ command and let's use the same coordinates (20,5). Your code should now look like Listing 4.6 below.
3. Save, compile and run this routine and you will see the exact results you saw before.

#### **Code Listing 4.6 (Newly Revised INI.HELLO.WORLD):**

```
1:  SUBROUTINE (PASSER)
2:  ** Version# 0.01[3] - 04/02/2001 - 05:10pm - JASONS - develop
3:
4:  *** Subroutine : JTS.HELLO.WORLD
5:  *-----*
6:  *** This subroutine will open a blank window and display "Hello World" to
7:  *** the user in it.
8:  *-----*
9:  *** PASSER - This argument is not used in this routine
10: *-----*
11: *** COMMON VARIABLES
12: *** None are used in this routine
13: *-----*
14:
15:  WINDOW
16:
17:  PRINT @(20,5):'Hello World'
18:
19:  SLEEP 5
20:  SLEEP 5
21:
22:  WINDOW.CLOSE
23:  RETURN
```

So, as you can see the PRINT @ command gains you a lot of flexibility. We can move the cursor anywhere we want on the screen to display strings, and in effect, we don't have to print everything in order from left to right and top to bottom. We can also easily replace strings that are already on the screen with a new screen based on user input without re-displaying the entire screen. This function makes the user interface flow much better on large data entry screens. It also makes the code much easier to follow and understand. Because of all of these advantages, always use PRINT @ when displaying static information on a screen. Unfortunately there is no counterpart to PRINT @ when working with a printout, we must use the PRINT command. We will discuss printing to reports and the hold file in detail in a further lesson.

### ***Exercise 4.8: Displaying and Re-displaying Different Strings on the Same Screen***

1. Create a new subroutine in UBP called INI.STRING.DISPLAY.
2. Enter the appropriate comments in the header of the subroutine.
3. Open a blank window.
4. Display the string "String Number 1" at position 35,6 on the screen.
5. Pause 4 seconds.
6. Overwrite the initial string with the string "String Number 2" at the same position.
7. Pause 4 seconds.
8. Close the window and return control to the calling routine.
9. Save, compile and run this routine.

Your program should initially display the string "String Number 1" for 4 seconds and then display the string "String Number 2" for 4 more seconds before the window is closed the Program Editor screen is displayed. Your code should look very similar to listing 4.7 below.

### Code Listing 4.7 (INI.STRING.DISPLAY in UBP)

```
1:  SUBROUTINE (PASSER)
2:  ** Version# 0.01[3] - 04/02/2001 - 05:10pm - JASONS - develop
3:
4:  *** Subroutine : JTS.STRING.DISPLAY
5:  *-----*
6:  *** This subroutine will open a blank window and display two strings to
7:  *** the user in it. Each string will be on screen for 4 seconds.
8:  *-----*
9:  *** PASSER - This argument is not used in this routine
10: *-----*
11: *** COMMON VARIABLES
12: *** None are used in this routine
13: *-----*
14:
15:  WINDOW
16:
17:  PRINT @(35,6):"String Number 1"
18:
19:  SLEEP 4
20:
21:  PRINT @(35,6):"String Number 2"
22:
23:  SLEEP 4
24:
25:  WINDOW.CLOSE
26:
27:  RETURN
```

Notice that because we displayed the second string in the exact same position and length, the first string was overwritten and the user only sees the second string. Unfortunately many times when we need to remove one string and overlay it with another string, the second string is shorter in length. What would happen if, in our above example, we changed the second string back to "Hello World"? Let's go ahead and do this.

#### ***Exercise 4.9: Replacing a String with a Second, Shorter String***

1. Edit your INI.STRING.DISPLAY in UBP
2. Change line 21 (String Number 2) to display "Hello World"
3. Save, compile and run this new routine.

Now, after the 4 second delay you will see the output “Hello World”, which is printed at 35,6 and the first part of “String Number 1” is gone. However, the last part is not. This is because the second string was shorter than the first and the PRINT @ only prints the string we told it. If we don’t force the length of the strings to be the same, PRINT @ will only re-display the characters in the second string.

Three possible solutions exist to fix this problem.

- Clear the entire screen before displaying the second string
- Clear the entire line (6) after the second string (from the char after d on)
- Make the string lengths the same so that the second string completely overwrites the first.

While option 3 is the one we most often use, let’s look at all 3 possibilities.

- The 1<sup>st</sup> option is very easy to accomplish. All we have to do is insert a PRINT @(-1) right before the PRINT @ (35,6):”Hello World”. The PRINT @(-1) tells our terminal emulator to clear the entire screen. This will obviously give us the results (figure ccc) that we’re looking for. However, this is like taking a sledgehammer to fix a watch. See listing 4.7.

### Code Listing 4.8 (Revised INI.STRING.DISPLAY in UBP)

```
1: SUBROUTINE (PASSER)
2: ** Version# 0.01[3] - 04/02/2001 - 05:10pm - JASONS - develop
3:
4: *** Subroutine : JTS.STRING.DISPLAY
5: *-----*
6: *** This subroutine will open a blank window and display two strings to
7: *** the user in it. Each string will be on screen for 4 seconds.
8: *-----*
9: *** PASSER - This argument is not used in this routine
10: *-----*
11: *** COMMON VARIABLES
12: *** None are used in this routine
13: *-----*
14:
15: WINDOW
16:
17: PRINT @(35,6):"String Number 1"
18:
19: SLEEP 4
20: PRINT @(-1)
21: PRINT @(35,6):"String Number 2"
22:
23: SLEEP 4
24:
25: WINDOW.CLOSE
26:
27: RETURN
```



**Note:** While `PRINT @(-1)` will clear the screen there is also an eclipse subroutine `CLEAR.SCREEN` which will usually be used to do this. We will look into this subroutine in later lessons.

- Now let's look at option 2. This option is a little better than the first option we looked at, but it still has a drawback. If there was any data other than this string on the right side of the screen, then we would be overwriting that data. However to accomplish the clearing of the rest of the line (beyond "Hello World"), we could do the following:

1. Use a `PRINT @(-4)` on the line after the printing of Hello World.
2. Add a colon (:) to the end of the "Hello World" line to give us our desired results. The reason we need to add ":" at the end of the print is to keep our cursor on that line. Again, the `PRINT @(-4)` clears the "remaining characters after the cursor on the current line". A normal `PRINT` or `PRINT @` that doesn't end with a colon will move the cursor down to the next line.

### Code Listing 4.9 (Revised INI.STRING.DISPLAY in UBP)

```
1:  SUBROUTINE (PASSER)
2:  ** Version# 0.01[3] - 04/02/2001 - 05:10pm - JASONS - develop
3:
4:  *** Subroutine : JTS.STRING.DISPLAY
5:  *-----*
6:  *** This subroutine will open a blank window and display two strings to
7:  *** the user in it. Each string will be on screen for 4 seconds.
8:  *-----*
9:  *** PASSER - This argument is not used in this routine
10: *-----*
11: *** COMMON VARIABLES
12: *** None are used in this routine
13: *-----*
14:
15:  WINDOW
16:
17:  PRINT @(35,6):"String Number 1"
18:
19:  SLEEP 4
20:
21:  PRINT @(35,6):"String Number 2":
22:  PRINT @(-4)
23:  SLEEP 4
24:
25:  WINDOW.CLOSE
26:
27:  RETURN
```

- Finally, let's look at option 3. This is the option we will most often use in our code because it forces the two strings to be the same length. We will accomplish this task by using formatting syntax that is native to PICK, which let's us justify our strings right or left, pad them with spaces or characters, and output numbers in many different formats, as well as many other things. In the current example, we could do the following:

1. Left justify both of our strings the same number of spaces. This will make them both the same length. When the smaller string is printed over the larger, the extra spaces will still clear out the remainder of that larger string.
2. Next we will just add the following to the end of our "Hello World" line with spaces or without a delimiter between the "" and formatting information. Make your Hello World print line look like the following:

- "PRINT @(35,6):'Hello World' 'L#15'"

This will left justify the string “Hello World” and pad is up to the 15 characters. If the string was more than 15 characters it would only take the leftmost 15 characters of that string. Using these formatting commands we can make our strings match whatever specifications we need to make the program work as we want. Again we will still see the results we desire and still only have the two PRINT @ commands. See Listing 4.10 below

#### **Code Listing 4.10 (INI.STRING.DISPLAY in UBP)**

```
1:  SUBROUTINE (PASSER)
2:  ** Version# 0.01[3] - 04/02/2001 - 05:10pm - JASONS - develop
3:
4:  *** Subroutine : JTS.STRING.DISPLAY
5:  *-----*
6:  *** This subroutine will open a blank window and display two strings to
7:  *** the user in it. Each string will be on screen for 4 seconds.
8:  *-----*
9:  *** PASSER - This argument is not used in this routine
10: *-----*
11: *** COMMON VARIABLES
12: *** None are used in this routine
13: *-----*
14:
15:  WINDOW
16:
17:  PRINT @(35,6):"String Number 1"
18:
19:  SLEEP 4
20:
21:  PRINT @(35,6):"Hello World"    "L#15"
22:
23:  SLEEP 4
24:
25:  WINDOW.CLOSE
26:
27:  RETURN
```

In summary we will always use the PRINT @ command to display static information on the screen. This command has many options but the most used option is to display the information in the correct spot (X,Y) on the screen. Finally, we have just introduced the formatting syntax within PICK, we will discuss this in greater detail as we start doing more complex work in this manual.



## ***Exercise 4.10: Using the Undo Option in the Program Editor***

1. Let's go into our last routine `INI.HELLO.WORLD` and delete all of the code.
2. Save the empty program and get back into the program editor. Now, you're saying, what was the point? The point is sometimes we make mistakes, on accident, or break our code to the point where we're better off starting over. The Eclipse program editor has a very nice feature called "Undo" that protects us against ourselves in these cases. So, now that we've completely destroyed your Hello World routine, let's get it back.
3. Go into the program editor and make sure the file is `UBP` and the Edit program is "`INI.HELLO.WORD`"
4. Press the **Undo Chg** hotkey (**ALT-U**)
5. You will receive the prompt asking if you want to undo version, mod, or nothing
  - **Version** will undo every change you have made since opening this program. You will use this option in cases where you:
    - Edit a routine for debug and find the bug in another routine,
    - Where you changed this routine but decided the changes are better off somewhere else or the changes didn't fix the problem,
    - Or you really broke the code and need to start fresh.
  - **Mod** will only undo the changes you made in the last edit session. This option exists in case your latest changes broke the routine.
    - You can use it to get back to the file before you made those changes.
    - You can also use it if you accidentally deleted or changed code and pressed **<ESC>** instead of **F12** to abort.
  - **Nothing** is there in case you accidentally pressed **Undo** and don't need this option. It aborts the Undo function.
6. In this case we only want to undo our very last changes, so choose **M** for "mod"
7. Edit this routine
8. You will see that all of your code is back just the way it was before we deleted everything



**Tip:** *Undo can only be done if the subroutine is still open to you. If you made changes and need to get them out of a closed version of a subroutine, you must open another version and manually back out your code's changes. To find what changes you made you can use the "Compare" option in the Program Editor which we will discuss in the next exercise.*

# Using the Compare hotkey in the program editor

The Compare option in the Eclipse Program Editor let's you compare different versions of the same subroutine. You can also compare one subroutine to another or the same subroutine in BP and FBP (or any other file) to itself. This option is helpful when trying to find out what change was actually made in a version, what changes were made on site, or what custom mods are in site versus in the BP version. You can even compare "mods" of a program so you can see what changes you made in each "edit" session.



***Note:** An edit session is from the time you pull the program up in edit mode to the time you pressed <ESC>. We don't save a "mod" unless you press <ESC> and save the routine to disk.*

## ***Exercise 4.11: Using the Compare Hotkey***

1. Go into the Program Editor for UBP, INI.HELLO.WORLD and let's compare two "mods" of this routine.
2. Press the hotkey **Compare (ALT-M)**. This screen defaults to comparing the last mod you made on an open program.
3. Press the **Begin** hotkey (**ALT-B**).
  - You could also use the hotkey **Strip comments Then Compare (ALT-S)**. The nice feature on this key is that it will not clutter your compare screen with comment changes that you're not worried about.
4. Enter UBP for the source file
  - This option is what you would change on site when comparing a UBP version to a BP version of a program. You would change this file to the file you wanted to compare the program you had chosen in the Program Editor. So in our example if we had a BP version of our routine and had picked the UBP in the PE we would change this value to BP and it would compare our UBP to our version.
5. You will see a screen listing all changes, new lines, or deletions that you made between the two mods. Press <Esc> to leave this screen.
6. Answer **N** (no) to saving a report in your hold file
  - This lets you save this report for later reference in the Eclipse hold file so that you don't have to re-run the compare later. Usually one compare is enough.
7. You should be back in the Eclipse Program Editor.

Again, Compare lets you compare versions, mods, and programs to each other in order to find out what differences exist in the code.

# Summary

In this lesson we introduced you to the Eclipse program editor and the Eclipse programming environment. We talked about what files your programs will live in, and how to create, compile, edit, and run programs. We also talked about some of the nice features of Eclipse BASIC vs. PICK BASIC and of the Eclipse pre-compiler. We introduced the PICK “SLEEP” function. You also learned how to open a blank window and how to display information in that window. Finally, we discussed the OC file, and other eclipse programming files such as SCREEN and MP.

## Assignments: Lesson 4

1. Create a program that will display “Hello My Name is: Your Name” in the center of a blank window for 15 seconds. Name this program INI.ASGNMT.41.
2. Create a program that will display the string “Scrolling Text” starting on the right hand side of the screen and scrolling to the left and off the screen (just like your scroll bar for the Eclipse Instant Messenger). This message should be on the center line of the blank window. Scroll the message two times and then leave the routine. Name this program INI.ASGNMT.42.
3. Create a routine in which you will display the text “Random Position” 10 times on the screen in a random X and Y position. Each time it displays it should pause for two seconds before re-displaying in a new position. Name this program INI.ASGNMT.43.

---

# *Lesson 5*

## *Introduction to the UniVerse Database*



### *Objectives*

**When you complete this lesson you will be able to:**

- Create files in UniVerse
- Create dictionaries in Universe
- Use the Edit command
- Use the COUNT command
- Use the STACK command
- Use the SELECT command
- Use the SORT command

# Overview

As we stated in a previous lesson, the UniVerse database is a multi-dimensional database that was started to support the original PICK database. Since its conception UniVerse has added many more file types to the database, as well as enhancements, such as secondary indexes and ODBC compatibility within the database. In this lesson we will take a high level overview of the UniVerse database. We will discuss what a file, record, and attributes are within the file, as well as how dictionaries play into all of this. We will learn how to create a file, record and dictionaries by using TCL (True Command Language). We will also see what exists in a file, learn how to select data out of a file, and learn how to clear and delete files. Further, you will learn about the types of files we use and how to correctly size files based on the data you will put in it. Within this topic, we will cover the internal workings of files so that you as a programmer will have the knowledge you need to correctly create and maintain files (tables) within the Eclipse application.

This lesson is a necessary step in your development as a programmer as most of what you will program within Eclipse will at some time need to interact with the database and files. It is important for you to know what is happening behind the scenes before you start using the programs to access this information.

## What is a file?

As stated in Lesson 2, a file is a grouping of like items on disk within the database. This means that, like a file cabinet, a file contains records which all contain similar information. Back to our grade school example, our file contains our records for grades, sick days, etc. For another example go back to our Excel spreadsheet and look at our “file” that contained our relatives information. The file contained records (each row was a record in that example) and each record contained the same information about each of the relatives in the same columns. So, in summary, a file is just a collection of similar data in the database stored on the disk drive.

## What is a record?

A record lives within a file, and within the record is where the actual data of a file lives. Each record must have a unique ID. Again looking at our grade school example, our record would probably be something like your social security number. For our relative information, we let Excel assign the unique key for each record, which was the line (or row) number on the far left of our screen.

Within UniVerse files, we as programmers get to pick what kind of record ID (key) to use. It can contain numbers, alpha characters and most punctuation characters. However, each record must be identified by a unique ID (meaning that no other record can share this ID). Again if you were looking at a file cabinet as an example, this ID would be the label printed on the folders inside the drawer. If you had two folders with the same label, you would have to look in both to find the record that you wanted. This identity problem is much worse in a database file because a computer can't ask "which one did you really want". Instead it will overwrite the old record with the new one if you use the same ID.

Within the UniVerse database we can store data at the attribute (column in our Excel example). Every record within the file should contain the same piece of information in the same attribute. So for a contact database the first name of every contact would all live in attribute 1. As programmers, no matter which record we work on, we know we can use the first attribute to find the first name that belongs to that contact. The nice thing about the UniVerse database is that we are not limited to storing one piece of information at the attribute level like a relational database. We can store up to 5 and 6 dimensions very easily. The most popular option is to put a value level in each attribute, which allows multiple values within each attribute. The sub-value level, which is the next level down, allows multiple values within each value position that exists within each attribute and so on.

In UniVerse the attribute, value, and sub-value positions are held by special characters.

- The **Attribute level** is held by the attribute mark (ASCII character 254).
- The **Value positions** within each attribute are separated by the value mark (ASCII character 253)
- The **Values** within each value position are separated by the sub value mark (ASCII character 252).

As you can see, no limit exists to the number of dimensions this structure can support unless you start getting into printable characters.



**Note:** Each record within a file is separated by the Record delimiter (ASCII character 255) in case you're wondering why they started at 254. See table 5.1

**Table 5.1 – Data Delimiters within the UniVerse database**

Description	ASCII Abbreviation	UV Abbreviation	Eclipse Name	ASCII Decimal
Record Delimiter	IM			255
Attribute Mark	FM	@FM, @AM	AM	254
Value Mark	VM	@VM	VM	253
Sub Value Mark	SVM	@SM	SVM	252

To remind yourself how files and records work together, remember that files contain records and records contain data. Each record will contain the same type of data in the same attribute, so every contact in the CONTACT file will have it's last name in attribute 3.

## Dictionaries and the UniVerse Database

Every file within UniVerse contains a second file, which is the Dictionary file. This file defines what data lives in what attribute of our main file. If we look at a CONTACT file there will be another file, more than likely named D\_CONTACT. This file contains records that define what lives in attribute 1, 2, 3 and so on. If a first name was suppose to be in attribute 1 of the CONTACT file, we may have a dictionary ID **FIRST.NM** that pointed to attribute 1 of the CONTACT file. The dictionary allows us to see what data lives where in the file, as well as what format we should be putting the data in when we enter it into the file. A benefit of UniVerse dictionaries as compared to relational table setups is that UniVerse doesn't force our data to match what the dictionary says. Yet this benefit can also have its negative effects if programmers are not careful in putting data into the correct position of the records they are updating.

Any time you create a new field in a file at Eclipse, you must first define the dictionary item, search in OC to make sure it's not already used, and get your supervisor to sign off on the new attribute before entering data. This process ensures that we do not create attributes that we do not need and that when we do create new attributes they don't overlap with another programmers code. We will discuss creation of dictionaries later in this lesson.

The types of dictionaries that define what data is in a record in the file are one of three types:

1. **A Type** – “Attribute” dictionary item

2. **S Type** – “Synonym dictionary item”
  - S Type was really almost exactly the same as “A” but put in place for people who couldn’t understand how one attribute in UniVerse could really contain more than one piece of information.
3. **D Type** – “Field Definition dictionary item”
  - Both A and S Type Dictionaries have been deprecated and all programmers should be using D Type Dictionaries to define fields in files. We should never use types A or S.

All 3 dictionary types above define what data lives in each attribute of a file. As stated we only use D Type at Eclipse since A and S Types are being phased out. Along with telling us what lives in files these dictionary items can also be used to query the database, as we will see later in this chapter.

### ***Exercise 5.1 – Creating a File Within UniVerse***

1. Go into Eclipse TCL, either by going to the program editor and using **T** in the option field or by selecting the menus **F2-System/TCL**.
2. Type in the command **CREATE-FILE** and hit **ENTER**.
3. You will be prompted for “File Name”. Type **INL.MANUAL.TEST** and press **ENTER**.
4. Next you are prompted for “Modulo”. Type **13** and press **ENTER**.
5. You will be prompted for “Separation”. Type **1** and press **ENTER**.
6. The next prompt will be for “File Type”. Type **18** and press **ENTER**.
7. Enter the same data in the next three corresponding prompts for the Dictionary file.



**Caution:** *We are prompted for two pieces of information because every file we create actually has a Data file and a Dictionary file. In this case, because we are creating a very small test file, the dictionary has the same information as the file. In most cases this will not be the case. You will rarely have a dictionary file any bigger than the one we just created and you will never use any type other than 18 for a dictionary file.*

8. Enter **Test file for class** in the file description.
  - You have now created the UniVerse file **INL.MANUAL.TEST**. So far nothing is in this file, it’s just empty waiting for data to be entered into it.
9. To prove this type in **COUNT INL.MANUAL.TEST**. This command (**COUNT**) will return you the number of items (records) in the file you ask it to count. In this case, the screen will display **0 Records** counted.



10. Another way to see what lives in a UniVerse file from TCL is the **LIST** command. Still in Eclipse TCL type in **LIST INI.MANUAL.TEST**. This time you will receive the message **0 records listed** from UniVerse.

To see what occurs when data lives in a file, we will create that data. However, before we put any data in the file we should (we don't have to but we should) create the dictionaries in order to determine where everything will live.

## ***Exercise 5.2 – Adding data manually to a UniVerse file using ED command***

Let's make this file contain information about cars, we'll make the record IDs a sequential counter. We will make **Attribute 1** contain the make, **2** will contain the model, **3** will contain the number of doors, **4** will contain number of passengers, and **5** will contain the year that this data is valid for this make/model. We will manually enter two records to get started on this file and also introduce you to the UniVerse line editor **ED**.

1. Enter **ED INI.MANUAL.TEST 1**.
  - This command will put you into the UniVerse line editor (**ED**) as well as prompt you that this is a new record. This prompt means that you are editing a record in the file that was not already there.



**Caution:** While I am introducing the line editor here and want you to be comfortable using it (it can be a very powerful cleanup and debugging tool), very few incidents at Eclipse occur prompting need of this tool. It should never be used to edit programs. You must always use the Eclipse program editor instead.

2. Enter **I <ENTER>**. This command enters you into **Insert** mode and will place your cursor in **Attribute 1 (0001=)** to enter data.



**Note:** When you enter into **Insert** mode in the UniVerse line editor, you start on the line **AFTER** your cursor. Because this record is empty, our cursor started on line 0 and thus we started inserting data at line (attribute) 1. If we had a record with data in it, we would have jumped to the line before we wanted the new data (by entering the line#<**ENTER**>) and then pressed **I<ENTER>** to begin inserting this data.

3. Let's put in a Lamborghini Diablo to start. **Attribute 1** should be **Lamborghini**. Press <**ENTER**> once you have put this in and the editor will put you on line 2.



**Note:** This is a "Line Editor" which means that once you are on line 2 you cannot get back to line 1 without terminating your **Insert** mode. The only way to terminate the **Insert** mode is to press <**ENTER**> on a blank line. However, this line (attribute) will not be saved (the blank line you pressed <**ENTER**> on). If

*you made a mistake in step 2, do not worry now. We will discuss editing this data later in this exercise.*

4. Attribute 2 is the model. Enter “Diablo” and press <ENTER> to go into Attribute 3.
5. Attribute 3 is the number of doors. This car has 2, so enter **2** and press <ENTER> to go on.
6. This is a 2 seated car so the next attribute (4) is also going to be 2. Press <ENTER> to continue.
7. The last attribute (5) contains the year that this information is valid. For now let’s assume it’s only valid for 1 year and enter 1991. Press <ENTER>.
8. We have completed entering data in this record, and we are now on line 6. To exit **Insert** mode press <ENTER> on this blank line and you will be taken out of **Insert** mode and back to the ----: prompt of the editor.
9. Type **T**<ENTER>. This command will take you to the top of the record.
10. Type **P**<ENTER>, which will print out 20 lines (or as many are left in the record) below where your cursor was on the screen. If you correctly entered this first record you will see a printout like Figure 5.1 below.
11. To save this record in the file. type in **FI** and press <ENTER>.
12. Do steps 1 through 11 again for Record ID 2 and create a Ferrari, F40, 2 doors, 2 seats, and again 1991 for the year.

```
Top.
----: p
0001: Lamborghini
0002: Diablo
0003: 2
0004: 2
0005: 2001
Bottom at line 5.
----: _
```

### ***Exercise 5.3: Previewing Your Data***

You now have two records inside your new file (**INI.MANUAL.TEST**).

1. To see these records type in **LIST INI.MANUAL.TEST** and you will see these two records listed on the screen.
  - You will see each ID, but you will not see the actual data inside the record.
2. To see the actual data you have a few choices. First let's use **LIST.ITEM**. At TCL type in **LIST.ITEM INI.MANUAL.TEST** and you will see both records, but this time you will also see all data within each record. (See the figure below.)

```
LIST.ITEM INI.MAN.TEST 02:34:26pm 10 Jul 2001 PAGE 1

      I
001 FERRARI
002 F40
003 2
004 2
005 2001

      1
001 Lamborghini
002 Diablo
003 2
004 2
005 2001
;
```

3. Select the **CT** command (Copy to Terminal) with a \* after the file name for a second way to list everything in the file. This \* tells the CT command that you want everything in the file listed out. This command will look like the following:
  - **CT INI.MANUAL.TEST \***

- You should see the results in the figure below.

```

:CT      INI.MAN.TEST 02:34:26pm 10 Jul 2001 PAGE 1

      I
001 FERRARI
002 F40
003 2
004 2
005 2001

      1
001 Lamborghini
002 Diablo
003 2
004 2
005 2001
;
```

4. If you wanted to see the information in only one of the records you can use either the command **CT** or **LIST.ITEM**. Type **command FILE RECORD** where command is either **CT** or **LIST.ITEM**.
5. FILE is the file you want to look at and RECORD is the record ID you want displayed. Go ahead and do one of each command **CT** and **LIST.ITEM** on Record 1.
  - You will see very similar results with both commands. The only major difference between the **CT** command and the **LIST.ITEM** command is that the **LIST.ITEM** command will not display empty attributes inside the record and the **CT** command will. Besides that difference, they are almost interchangeable and you can use whichever command you prefer to view data inside records.

## ***Exercise 5.4: Creating Dictionaries Using ED on Our New File***

Now we should create our dictionary items for this file, so we will remember how we stored our data and so that programmers to come will be able to quickly find what data lives where as well as add new attributes to this file.

1. Type in **ED DICT INI.MANUAL.TEST AUTO.MAKE**.
  - This command will put you into the record AUTO.MAKE in the dictionary file for INI.MANUAL.TEST. Because we created this file through normal means, the actual dictionaries will live in D\_INI.MANUAL.TEST.
  - If you ever want to see what file the dictionaries live in for a specific file, type in **CT VOC INI.MANUAL.TEST**. The VOC is where UniVerse defines all files, master dictionaries, TCL Commands, UniVerse subroutines, and PICK subroutines. I will not go into detail on the VOC in this manual as all UniVerse documentation contains this information. The dictionary file name will be the third attribute of this record if Attribute 1 is **F** for File. To see what I'm referring to go ahead and push a level (F2-T) and type in **CT VOC INI.MANUAL.TEST** and you'll see attribute 1 is **F**, Attribute 2 is your file name, and Attribute 3 is **D\_yourfile** name. Now <ESC> back to the Line Editor in step 1.
2. Enter **Insert** mode and put a **D** in Attribute 1.
3. Put in a 1 in Attribute 2.
4. Leave Attribute 3 empty. By leaving it empty, though, you will have to put in a **space** <ENTER> or you'll leave **Insert** mode.
5. Fill in **Automobile Maker** in Attribute 4.
6. Enter **25L** for Attribute 5
7. Enter **S** for Attribute 6. Leave **Insert** mode and save this record.
  - Attribute 1 is the dictionary type (**D**), 2 is the attribute the data lives in the file, 3 is the UniVerse conversion code, 4 is the dictionary description, 5 is the length (25) and justification (L) of the attribute (this is for display only), and 6 tells us if the attribute is Single (S) or Multi-Valued (M).
8. To quickly create the next 4 dictionaries let's use a shortcut in the editor. Type **ED DICT INI.MANUAL.TEST MODEL DOORS PASSENGERS YEAR** and press <ENTER>
9. This command will put you into the line editor for the first record (**MODEL**). Notice that we will edit each record in sequence, so once you **FI** the **MODEL** dictionary the editor will take you into the next (in this case **DOORS**). Fill in all of the correct information and save all of these dictionaries. Make sure that the doors and passengers' dictionary items are right justified, as they are numbers.

10. To see all of the dictionaries you created type in **LIST DICT**  
**INI.MANUAL.TEST** and you will see a sorted list (by attribute) of every dictionary on this file. That output should look very similar to the figure below.
11. To end the edit session completely you will just press **<ENTER>**. Through the last prompt of **Record Name =**, the editor asks if there were any other records you wanted to edit after completing the active list.

DICT INI.MANUAL.TEST 05:02:16pm 10 Jul 2001 Page 1					
Field..... Name.....	Type & Field. Number	Field..... Definition...	Conversion.. Code.....	Column..... Heading.....	Output Depth & Format Assoc..
@ID	D 0			INI.MANUAL.TEST	10L S
AUTO.MAKE	D 1			Automobile Make	25L S
DOORS	2 2			r	
MODEL	D F40				S
4 records listed.					
;■					

Why are there 6 dictionaries in this list rather than just the 5 we created? Six dictionaries exist because UniVerse uses the special dictionary **@ID** as the “**KEY**” dictionary on every file. Every file must contain at least one dictionary in the **@ID**, which will point to Attribute 0. This dictionary is actually pointing to the record ID for each record inside of this file. As you know the final 5 dictionaries point to Attributes 1 through 5 of this file.

Now we have a complete file with a good listing of dictionaries telling us what lives in each attribute of the file. It's time to learn how to select certain pieces of data out of the file. Before we do so let's do one more quick exercise to get more data in our test file.

## ***Exercise 5.5: Creating More Data in INI.MANUAL.TEST File***

1. Create 6 more automobile records in this file.
2. The record IDs will start at 3 and go up to 8 for these records.
3. The automobiles' attributes will be the following:
  - Lamborghini, Countach, 2 doors, 2 seats, 1978
  - Ferrari, Testarosa, 2 doors, 2 seats, 1983
  - Porsche, 911, 2 doors, 2 seats, 1989
  - Lotus, Esprit, 2 doors, 2 seats, 1985
  - Porsche, Boxster, 2 doors, 2 seats, 2001
  - Dodge, Viper, 2 doors, 2 seats, 1997
4. You should now have a total of 8 records in this file, with record IDs 1 through 8. Perform a **LIST.ITEM** on the file to make sure you are happy with the data.
5. **ED** one final record with record **ID 10**.
6. Put in **Ford** on line 1.
7. Type **Mustang** on line 2.
8. Type **2** on line 3.
9. Type **8** on line 4. (Yes. Make this 8 on purpose!)
10. Type **2001** on line 5.
11. Exit **Insert** mode. Notice we made an error in Attribute 4. The Mustang only has 4 seats.
12. Type in **4 <ENTER>** to jump to line 4 and type in **R/8/4<ENTER>**. This command tells the editor you want to replace the character 8 with the character 4 on this line.
13. The other way we could have completed this command is to enter **R 4<ENTER>**. This command tells the editor to replace everything on this line with the string following the space.



**Note:** If you ever make a mistake in the Line Editor (and are not in **Insert** mode) you can type **OOPS<ENTER>** to back out of that latest change you made. If you want to abort the entire edit session use **Q<ENTER>** to quit without saving the changes to that record.

14. Go to the top of this record and make sure the data is now correct for the Mustang. Save the record.



**Tip:** You can also insert a line in the Line Editor with data and not have to go all of the way into **Insert** mode. You can perform this action by putting your cursor on the line you want to insert data into and type **I DATA<ENTER>**, which will enter **DATA** on the line you had your cursor on without putting you into **Insert** mode. You can delete lines in the editor by using the **D** option with your cursor on the line you want deleted. Finally, you can delete a record by using the **FD** option instead of **FI** to exit the record.

## ***Exercise 5.6 – Counting Records in a File Without Using the LIST Command***

One thing that can be useful when working with a file is knowing how many records you're dealing with. We can accomplish this task by using the **COUNT** command. This command goes out to the file and counts how many records are in it. Let's find out how many records we have in our **INI.MANUAL.TEST** file.

1. Go into Eclipse TCL (**F2-T**).
2. Type in **COUNT INI.MANUAL.TEST** and press **<ENTER>**.
3. You will receive a message back displaying **"9 records counted."**

That's all it takes to count records! Unfortunately on a large file this action could take a few minutes to count each record. In Eclipse TCL we have given you a hotkey (**ALT-X**) that let's you execute the TCL command in the phantom (background) so that while this command is being executed you can go do other things. This action is also very handy in the next section for selecting information out of the file.



**Caution:** While everything in Eclipse TCL gets converted to uppercase and thus seems case insensitive TCL is truly case sensitive. When executing a command in the phantom be sure the case is correct or the command will not be performed as you expect.

## ***Exercise 5.7: Execution of COUNT in the Background***

1. Go into Eclipse TCL.
2. Type in **COUNT INI.MANUAL.TEST**.
3. Press **ALT-X**.
4. Answer **"Y"** to the **"Execute Command in Phantom"** prompt.
5. You will see **"Phantom process started with process ID xxx"** on your screen.
6. As soon as this command is complete, you will get an Eclipse message **"From Phantom: 9 records counted."**



In the above exercise, we did not save much time running this task in phantom because counting 9 records takes little time anyway. However, if I wanted to see how many order records were out in a system, this command comes in very handy.

Another nice feature of Eclipse TCL is the command “**stack**”. This command keeps track of the last 100 commands you executed in Eclipse TCL. You can easily access those commands using your arrow keys (**Up Arrow** goes up the stack and **Down Arrow** goes down the stack) or by using the **ALT-P** (Previous Commands) hotkey. Let’s explore this a little further.

### ***Exercise 5.8: Command Stack in Eclipse TCL***

1. Go into Eclipse TCL.
2. Type in **COUNT INI.MANUAL.TEST** and press **<ENTER>**.
3. Type in **LIST INI.MANUAL.TEST** and press **<ENTER>**.
4. Now press the **Up Arrow** key one time. Notice that it will fill in the TCL prompt with your last command “**LIST INI.MANUAL.TEST**”. You can either press **<ENTER>** to submit this command again, change the command around if you had a typo, or **Up/Down Arrow** to move to different commands.
5. If you wish to change a command in your stack before resubmitting, be sure you do not start typing in the first character or you will end up overwriting the entire command with your new characters. If you need to edit the first character, press the **Right Arrow** and **Left Arrow** to navigate and then type in your changes.
6. You can also use the **<END>** key to quickly jump to the end of the command and make changes there.
7. Play with your command stack for a few more minutes (In Eclipse TCL) and let your training manager know if you have any questions.

We now know how to create a file, create dictionaries, see all the records in a file, see all of the data in the records in the file, and see all of the dictionaries in a file. The next step is to be able to select data out of our file. There are 3 commands we will use to select data out of our file. The first two are **SELECT** and **SSELECT**. The only difference in the two commands is that **SSELECT** will sort the IDs before returning the select list. The third command we will look at is **SORT**, which behaves a little differently than the other two.

### ***Exercise 5.9: Selecting Specific Records out of a File in TCL***

Let’s start by saying we only want to see those records in our file that were made by Lamborghini. How would we do this? It’s actually very simple in the UniVerse Database using the **SELECT** command.

1. Get into Eclipse TCL.
2. Type in **UL** and press **<ENTER>**.



**Note:** This command let's us use lower and upper case in Eclipse TCL. If you do not run this command, the **SELECT** will fail because we did not enter "Lamborghini" in all uppercase in the database. As the above **Caution** stated, all of these TCL commands are case sensitive. We convert everything to uppercase in Eclipse TCL to make your life easier most of the time.

3. Type in **SELECT INI.MANUAL.TEST WITH AUTO.MAKE = "Lamborghini"**.
4. If you didn't make any typing errors, you will get the message "**2 record(s) selected to SELECT list #0.**"

This message is telling you that your "active" list of record IDs now contains two IDs. A benefit with a normal UniVerse **SELECT** (like we just did) is that it does not store any record information with the IDs. Instead it stores only the IDs themselves. If I have related files with the same IDs, I can select off one and then go back and get the data out of another. We will look more into this function at a later time. The important thing to know about your active list is that while you have it active, every command you execute will only execute off that list of IDs. Let's try an example.

### ***Exercise 5.10: Selecting Records from an Active List***

1. Type in **SELECT INI.MANUAL.TEST WITH AUTO.MAKE = "Porsche"** and press <ENTER>.
2. You will get returned the message "**0 record(s) selected to SELECT list #0.**"
3. This message is displayed to demonstrate that the only records the second **SELECT** looked at were the two we had in our active select list.
4. Repeat Step 4 in the previous exercise to get only the Lamborghini's record.
5. Now press **ALT-C** hotkey to clear out your selected list.
6. Repeat the Porsche Selection and you will now receive the message "**2 record(s) selected to SELECT list #0.**" This message appears to tell us that, before our second selection, we cleared out our active list of record IDs. You can also clear this active list by executing a **LIST** command on the select list.
7. With this list still active run a **LIST INI.MANUAL.TEST** and you will see records 5 and 7 listed on your screen. These were the two records we created for Porsche.
8. Execute Step 4 from the previous exercise again, and then run a **LIST.ITEM INI.MANUAL.TEST**. You will see results similar to those in Figure 5.1 with record IDs 1 and 3 displayed and their data below.

So in review, the **SELECT** statement find records within a file that match the criteria defined after the **WITH** clause. If you need more than one piece of data you can use the **AND/OR** clause to specify conditional statements that must be met to select out the records.

### ***Exercise 5.11: Using the AND, OR and Wild Card Clauses in the SELECT Statement***

1. Go into Eclipse TCL.
2. Type in **SELECT INI.MANUAL.TEST WITH AUTO.MAKE = “Lamborghini” OR WITH AUTO.MAKE = “Porsche”** and press <ENTER>.
3. You will receive the message **“4 record(s) selected to SELECT list #0.”**
4. Do a **LIST.ITEM** on this list and see what records were selected from the file. You will see that all records that had either Lamborghini or Porsche in attribute 1 are now in this active list of record IDs.
5. Type in **SELECT INI.MANUAL.TEST WITH AUTO.MAKE = “Porsche” AND WITH MODEL = “Boxster”** and you will receive the message **“1 record(s) selected to SELECT list #0.”**
6. Use **LIST.ITEM** or **CT** to see what record was selected and you will see it was record **ID 7**. The record for the Porsche Boxster came up on the screen because we told the select we only wanted records that were made by Porsche and had the model Boxster.
7. We can also use what is called the implicit **OR** within the **SELECT** statement. To do this let’s make the same selection as in Step 2 but without the second **OR WITH** clause. Just type in **SELECT INI.MANUAL.TEST WITH AUTO.MAKE “Lamborghini” “Porsche”** and press <ENTER>
8. You will receive the message **“4 record(s) selected to SELECT list #0”** and you will see that they are the exact same record you selected in Step 4.



**Note:** *If you don’t put the AND/OR in between dictionaries, then you have an implied OR in the dictionary before the values.*

9. Wild card characters also exist in the **SELECT** values. They allow you to select all records that have an auto maker starting with an **“L”** or all records that have an **“L/I”** anywhere in the record.
10. To do the first selection (all records with attribute 1 starting with an **“L”**) type in the following: **SELECT INI.MANUAL.TEST WITH AUTO.MAKE = “L”**. You will get 3 records: the two Lamborghini records and the one Lotus record.
11. Let’s select anything ending in an **“e”**
12. Type in **SELECT INI.MANUAL.TEST WITH AUTO.MAKE = “[e”** and press **ENTER**.

13. You will get 3 records in this list: two for the Porsche records and one for the Dodge (they all end in e). So the wild card characters are the “[“ and the “]”, where the “[“ tells us we do not care what the string starts with and the “]” tells the **SELECT** we do not care what the string ends with. You can also use these characters together in the **SELECT** command.
14. Let's select anything with the letter “s” in the auto make. Type in the following:
  - **SELECT INI.MANUAL.TEST WITH AUTO.MAKE = “[s]”** and you will get three records: two for the Porsche and the one for the Lotus. You don't have to use only one character with these wild cards. The strings can be as long as you need them to be.
15. If we wanted to save this list of record IDs for later reference we can now use the **SAVE-LIST** command. This will save this list of record IDs out to the **&SDAVEDLISTS&** file where we can edit the list of ID (**EDIT-LIST**) or retrieve the list back into our active list using **GET-LIST** with the list name. Save this list off to **INI1 (SAVE-LIST INI1)**.
16. Edit this list using **EDIT-LIST INI1** and press **p <ENTER>**. You will see the three record IDs we selected.
17. Quit the edit session.
18. Retrieve this list back into your active list by using **GET-LIST INI1**.
19. Run **LIST.ITEM INI.MANUAL.TEST** to see that you are again looking at these three records. These last few steps are again important time savers when working on large files. You could have performed the initial **SELECT** in phantom (**ALT-X**), gone to work on other things, saved the list once it was selected, done some more research, and then never have to re-do this large select.



**Tip:** Another nice thing about Eclipse TCL is that it also does automatic **SAVE-LISTS** every time you select data out. You can see these automatically saved lists by pressing the **ALT-L** hotkey in TCL. To re-activate a list just highlight it and press **<ENTER>**. This list will now be your active list. This command is helpful if you forgot to save a list or made a mistake in the secondary selection. You can use it to get back to the initial lists.

So we can easily select out just about any records from a file with the **SELECT** statement and dictionaries defined on that file. As we also saw UniVerse creates what we call an *active list* of the record IDs that we selected out of this file. We can then further select off that active list or use that active list to find which records were selected from our statement, and we can use *wild card* and conditional statements to further enhance our selection capabilities.

The final capability we will discuss in this lesson is the sorting of records after selecting them. This capability allows us to display records in a logical manner rather than in the order they were hashed into the file. We can still use the **SELECT** statement, but we can also use the **SSELECT** statement. The only difference in the two statements is that **SSELECT** will sort the records by record ID before we have set up any secondary criteria. This command can be an advantage if we need the records sorted by record **ID**. However, if we want the records sorted in any other way, this command is just an additional overhead on the system. For this reason we will continue using **SELECT**. Know that everything we do in **SELECT** would work exactly the same using **SSELECT**.

### ***Exercise 5.12: Sorting Records within a File/Select List***

1. Go into Eclipse TCL.
2. Type in **SELECT INI.MANUAL.TEST WITH AUTO.MAKE = "LJ" BY AUTO.MAKE**.
3. You will get three records. This time run **LIST.ITEM** on these records and see that the two Lamborghini records are sorted to the top and the Lotus is listed last. Our **BY** clause tells the **SELECT** statement that we want the final list in ascending **AUTO.MAKE** order.
4. Type in **SELECT INI.MANUAL.TEST BY AUTO.MAKE**.
5. Type in **LIST INI.MANUAL.TEST AUTO.MAKE MODEL**.
6. You will see the results in the first figure below.



**Tip:** With the **LIST** command, you can show more than just the record IDs by entering the dictionary IDs you want displayed after the file name that you entered. If you didn't want to see the record IDs, you could type **LIST INI.MANUAL.TEST AUTO.MAKE MODEL** (I. This command tells UniVerse not to display the internal record IDs on the screen – See the second figure below.

7. You can also sort by more than one field by putting in more than one **BY** clause. Type in **SELECT INI.MANUAL.TEST BY AUTO.MAKE BY MODEL** and you will receive a sorted list with the auto maker as the primary sort by. Next you will see a list sorted by the model as the secondary sort by.
8. We can also sort in descending order by using the **BY-DSND** clause. Type in **SELECT INI.MANUAL.TEST BY-DSND AUTO.MAKE** and you will receive a sorted list with the highest auto maker value at the top of the list.

```

LIST INI.MANUAL.TEST AUTO.MAKE MODEL 05:16:24pm 10 Jul 2001 PAGE 1
INI.MANUAL.TEST Automobile Maker.....

Dodge          Dodge
Ferrari         Ferrari
INI.MANUAL.TEST Ford
I               Lamborghini
Lamborghini     Lamborghini
Lotus           Lotus
Porsche         Porsche

7 records listed.
;

```

```

LIST INI.MANUAL.TEST AUTO.MAKE MODEL 10-SUPP 05:18:37pm 10 Jul 2001 PAGE 1
Automobile Maker.....

Porsche
Lotus
Dodge
Ford
Lamborghini
Lamborghini
Ferrari

7 records listed.
;

```

The final command I would like to look at in this lesson is the **SORT** command. The **SORT** command let's you **SELECT**, **SORT**, and **LIST** records within the file all in one command. The drawback to the **SORT** command is that you cannot use it within a program or run it in phantom; it relies on putting the final listing on the screen.

### ***Exercise 5.13 – Using the SORT Command in TCL***

1. Go into Eclipse TCL.
2. Type in **SORT INI.MANUAL.TEST BY AUTO.MAKE**.
3. You will see a list of record IDs sorted in order of the **AUTO.MAKE** file (attribute 1).
4. Type in **SORT INI.MANUAL.TEST BY AUTO.MAKE AUTO.MAKE**.
5. This time you will see the same sorted IDs, but you will also see the **AUTO.MAKE** field displayed next to the IDs on the screen
6. Type in **SORT INI.MANUAL.TEST BY AUTO.MAKE BY MODEL AUTO.MAKE MODEL**.
7. You will see a list sorted by maker, model, or IDs.
8. Type in **SORT INI.MANUAL.TEST WITH AUTO.MAKE = "L]" BY AUTO.MAKE BY MODEL AUTO.MAKE MODEL** and you will see only the records that have attribute 1 starting with "L" sorted by maker and by model on the screen.
9. If you want to hide the record IDs you can still use the **"(I switch on the end of the"** command. You can still sort in descending order by using the **BY-DSND** clause as in the **SELECT** statement.

## Summary

In this lesson you learned how to create UniVerse files, and you learned about the different types of files that we use at Eclipse. You also learned how to properly size the files, how to create and edit records within a file, how to manually create dictionary items for a file, how to read a VOC entry for a file, and how to select and sort information from a file. This lesson is an important step in learning the PICK programming language, as most of what you do at Eclipse will require interacting with the database in some way. So far we only created a two-dimensional file. We will expand on this topic in the lessons to come in order to find the advantages of our database over all of the relational databases of the world.

## Assignments: Lesson 5

1. Explain the steps necessary in creating a file in detail. Explain what to look for when deciding the initial type and size of a new file in a word document. Email this assignment to your Training Manager.
  2. Select all of the records in your INI.MANUAL.TEST that have a model containing a “t” anywhere in it. Save the list to a list named INI2 and show your training manager these steps.
  3. Using two different methods, get a sorted list of autos from our new database by maker, by model, by cars that have 2 seats. Display the maker, model, and year of the automobile on the screen.
-

## *Lesson 6*

# *Retrieving and Saving Data from a Program in Universe*



### *Objectives*

**In this lesson, you will learn to:**

- Use both dynamic and dimensioned arrays
- Retrieve information from arrays
- Write information into arrays



# Overview

In the previous lesson, you learned the definitions of file, record, and attribute within the UniVerse database. We discussed selecting records out of a file by using dictionaries that define structures of records in files. We also looked at manually creating, changing, and saving information within each of these records.

Such manual work is a tedious process, but, as programmers, our task is to enable users to create, edit, and save information within a record. The following lesson discusses how to complete this task.

In this lesson, you will learn numerous ways to retrieve and save data within the UniVerse BASIC language. You will cover the following two different types of arrays and what they do for us:

- Dynamic
- Dimensioned

## Retrieving Data from a Record into a Dynamic Array

In the following exercise, you are introduced to the SHOW function, READ statement, dynamic arrays, and the OPEN statement. You will retrieve the information needed for this exercise from our INI.MANUAL.TEST file and display the record on the screen.

The OPEN statement is used to give our program access to the file from which we want to read data. The OPEN statement is not complicated to write in a program, but it is expensive for the operating system to run.

The OPEN statement stores the file header information in a variable called the file handle. The file handle variable enables our program to read and write from the file.

The READ statement is used to read data into the file from a record. The SHOW statement is used to display the contents of a record's variable.

## ***Exercise 6.1: Retrieving Data from a Record into a Dynamic Array***

1. In develop, create a new program in UBP. Name it INI.DYN.READ.
2. Generate comments on the top of the routine that indicate the program reads a record into a dynamic array and displays the information on the screen.
3. Insert the **OPEN “INI.MANUAL.TEST” TO TEMPFILE ELSE RETURN** directly after the comments in order to use the INI.MANUAL.TEST file in this routine:
  - If UniVerse is unable to open INI.MANUAL.TEST, it executes the ELSE clause.
  - UniVerse is unable to open INI.MANUAL.TEST if the file no longer exists or if the file has permissions that keep this process from using it at the OS level. If UniVerse successfully opens the file, a THEN clause in the OPEN statement is executed. In this exercise, you will assume the file can be opened.
4. Enter **READ AUTO.REC FROM TEMPFILE, 1 ELSE AUTO.REC = “”**.
  - This command instructs the routine to read in Record ID 1 from the file.
  - The READ command is broken into the following parts:

Part	Definition
<b>Command</b>	The command itself.  For example: <b>READ</b>
<b>Dynamic Array</b>	The dynamic array you want applied to the record for this program.  For example: <b>AUTO.REC</b> .
<b>FROM Statement</b>	The file handle, a comma, and the Record ID.  For example: <b>FROM TEMPFILE, 1</b>
<b>Exception Clauses</b>	<ul style="list-style-type: none"><li>• <b>THEN</b> is executed if the record is found in the file.</li><li>• <b>ELSE</b> is executed if the record is not found in the file. If you do not include an <b>ELSE</b> statement and the record is not found in the file, an error message stating the variable is <b>Undefined</b> occurs at run-time.</li></ul>

5. Enter **SHOW AUTO.REC** to display the contents of the variable on the screen.
- The **SHOW** function takes up to 10 arguments and displays them in a separate window. The window displays an instruction for the user to press **RETURN** in order to continue.
- Use **SHOW** for debugging a subroutine. You can also use it to display variable values at key points within a routine.



**Caution:** A program containing a *SHOW* statement cannot be closed with the Eclipse Program Editor.



**Tip:** If you use *SHOW* in a subroutine that can affect other users, pass an asterisk (\*) and your User ID in the first argument of the *SHOW* call. This action displays the *SHOW* only for you.

For example: **SHOW “\*User.ID”, AUTO.REC**

6. Enter **RETURN** on the next line to return control to our calling subroutine.
7. Save, compile, and run the routine from the program editor. You should see the same results and listing as below:

```
Run Subroutine Program: JTS.DYN.READ
PASSER.....
Lamborghini^Diablo^2^2^1991

Press <RETURN> to continue : _
```

## Listing 6.1: INI.DYN.READ

```
1. SUBROUTINE (PASSER)
2. ** Version# 0.01[1] - 07/31/2001 - 02:28pm - JASONS - develop
3.
4. *** Subroutine : JTS.DYN.READ
5. *-----*
6. *** This subroutine will read in record ID "1" from the JTS.MANUAL.TEST
7. *** file and display the contents on screen to the user.
8. *-----*
9. *** PASSER is not used in this subroutine.
10. *-----*
11. *** No Common variables are used in this routine.
12. *-----*
13.
14. *** Open the JTS.MANUAL.TEST file for use in this subroutine.
15. OPEN 'JTS.MANUAL.TEST' TO TEMPFILE ELSE RETURN
16.
17. *** Put the contents of record ID "1" in the JTS.MANUAL.TEST file
18. *** into the variable AUTO.REC making it a dynamic array. If the
19. *** file doesn't contain record ID "1" make the variable equal
20. *** an empty string.
21. READ AUTO.REC FROM TEMPFILE,1 ELSE AUTO.REC = ""
22.
23. *** Display the contents of AUTO.REC variable to the user using
24. *** the Eclipse function SHOW. This function will let the user
25. *** hit <RETURN> before closing the window and returning control
26. *** to here.
27. SHOW AUTO.REC
28.
29. RETURN
30.!!JASONS~07/31/01~14:28
```

### *Why we use Dynamic Arrays*

From this example, you notice that you access a file stored on a disk by:

1. Using the OPEN statement to open the file to a file handle variable.
2. Using the READ statement to put the file's record into a dynamic array.

If you are familiar with programming, you might wonder how a variable can be an array without indicating how big the array is. In PICK you do not need to indicate how big a dynamic array is. A dynamic array adjusts its size to the data put into it. By using dynamic arrays you do not need to worry if the array is large enough to handle the data.

Dynamic arrays are long strings. For example, when you printed the entire string on our screen by entering **PRINT DYN.ARRAY**, it compiled and ran without any errors. Although ^ characters displayed where the attribute marks (**Char(254)**) appeared, everything else looked like a string. You could have changed ^ to ~ and then written your own subroutines to parse the string. UniVerse already parsed the string for you, though, so keep these delimiters.

Dynamic arrays' storage in memory is a drawback. Because dynamic arrays are long strings, accessing the elements within them becomes slower as the strings grow.



***Note:** Do not use your own delimiter instead of AM, VM, SVM in the dynamic array. UniVerse has special caching logic so the retrieval of strings from a dynamic array attribute is faster than parsing our own delimiters.*

## Accessing One Element of a Dynamic Array

To access one element of a dynamic array, place the <> characters around the element you want to access. For example, for a dynamic array variable named **DREC**, you access just the first attribute by using the following syntax:

**DREC<1>**

If you want to place this dynamic array into another variable called **FIRST.ATTB**, use the following syntax:

**FIRST.ATTB = DREC<1>**

This syntax takes the first attribute of the dynamic array **DREC** and places it into the new variable **FIRST.ATTB**. If **DREC<1>** has multiple values within the attribute (separated by value marks, sub value marks, and so on), the syntax places them into **FIRST.ATTB**, as well. To access the first attribute and its first value, use the following syntax:

**FIRST.ELEM = DREC <1,1>**

To access the second value position for the first attribute, use the following syntax:

**FIRST.ATTB.SECOND.VAL = DREC <1,2>**

To access the second sub value from the third value in the fifth attribute, use the following syntax:

**NEW.VAR = DREC <5,3,2>**



***Note:** If further dimensions within the dynamic array exist, you must use the **RAISE** command in conjunction with multiple assignment statements to access those further dimensions. **PICK** only supports three dimensions in the dynamic array syntax. You will learn this in a later lesson.*

## ***Exercise 6.2: Accessing Elements in a Dynamic Array***

In this exercise, you will access specific elements of the dynamic array. Take the model of the car that you pulled from the INI.MANUAL.TEST file, record ID 1.

1. Create a new subroutine and name it **INI.DYN.ACCESS**.
2. Open the INI.MANUAL.TEST file.
3. Read Record ID 1 from the file into a dynamic array.
4. Place the second attribute from your dynamic array into a variable called MODEL.
5. Open a blank window and display the model of this car on the screen for 5 seconds.
6. Close the window.
7. Return control to the calling routine. You should see the output **Diablo** on the screen.

You should see the same listing as below.

### **Listing 6.2: INI.DYN.ACCESS**

```
1.      SUBROUTINE (PASSER)
2.
3.      *** Subroutine : JTS.DYN.READ
4.      *-----*
5.      *** This subroutine will read in record ID 1 from the JTS.MANUAL.TEST file
6.      *** into a dynamic array and then access attribute #2 (Model) and display
7.      *** it on a blank screen for 5 seconds.
8.      *-----*
9.      *** PASSER is not used in this routine.
10.     *-----*
11.     *** COMMON VARIABLES:
12.     *** None are used in this routine.
13.     *-----*
14.     OPEN 'JTS.MANUAL.TEST' TO JTSFILE ELSE RETURN
15.
16.     *** Read in record ID 1 from the JTS.MANUAL.TEST file into the dynamic
17.     *** array AUTO.REC.
18.     ID =1
19.     READ AUTO.REC FROM JTSFILE,ID ELSE AUTO.REC = "
20.
21.     *** Pull the model of the car from attribute #2 of this dynamic array.
22.     MODEL = AUTO.REC <2>
23.
24.     ***Open a blank window and display what we got out of attribute 2.
25.     WINDOW
26.
27.     PRINT @ (10,2):MODEL
28.     SLEEP 5
29.
30.     WINDOW.CLOSE
31.
32.     RETURN
```

## Introduction to READV for Accessing Single Attributes

You have learned how to open a file for access, read data from an entire record into a dynamic array, and access elements from that dynamic array. You have also learned that PICK treats all variables as strings until you change them, such as a mathematical operation, a file open, or a dimension for dimensioned arrays.

In this section, you will learn how to speed up the read time (as READ/Retrieves of large records are expensive) for the program.

If you need to access only one or two attributes from a record within the file, UniVerse provides the READV command. This command instructs the system to go out to the file for one record and access one attribute of that record. It then places that attribute into the dynamic array. In Exercise 6.2, you could have used READV to accomplish the code instead of reading the entire record. Using READV would have been more efficient.

Break the READV statement into the same parts as you did with the READ statement in Exercise 6.2, except enter the attribute number after the record ID. Make file handle **TEMPFILE** read in only the second attribute from record ID **JASON** into a variable called **SECOND.ATTB**. Use the following syntax:

```
READV SECOND.ATTB FROM TEMPFILE,"JASON",2 ELSE  
SECOND.ATTB = ""
```

The READV statement has the same clauses as the READ statement for THEN and ELSE.

- The THEN statement is executed when the record is found in the file.
- The ELSE statement is executed when that record is not found in the file.

### ***Exercise 6.3: Changing `INI.DYN.ACCESS` to `READV` Instead of `READ`***

1. In the Program Editor, pull up `INI.DYN.ACCESS`.
2. Change the `READ` into `AUTO.REC` command to a `READV` in your model variable command.
3. Display the `MODEL` on the screen for five seconds.
4. Run this updated routine. You should see the same results as in Exercise 6.2.

You can also use the `READV` command in checking for a record in a file. UniVerse allows you to use attribute 0 as the attribute piece of `READV`. This command tells Eclipse to look only in the hash table rather than accessing data from disk, which makes this command quick to execute.

### ***Exercise 6.4: Creating `INI.READV.TEST`***

1. In UBP, create a new routine named **`INI.READV.TEST`**.  

This routine will access the **year** attribute of record ID 1 and record ID 4, and display them on the screen.
2. Create the correct comments in the header of this routine.
3. Use **`READV`** to access the year of the cars in Records 1 and 4.
4. Display both of the years on the screen simultaneously for five seconds.
5. Return control to the calling routine.
6. Compile and run this routine. You should see the output **1991**, with the output **1983** below **1991**. You should see the same listing as below:



### Listing 6.3: INI.READV.TEST

```
1. SUBROUTINE (PASSER)
2. *** Version# 0.01[1] – 08/14/2001 – 7:06pm – JASONS - develop
3.
4. *** Subroutine : JTS.DYN.READ
5. *-----*
6. *** This subroutine will read in the year that the cars in record ID 1 and
7. *** record ID 4 from the JTS.MANUAL.TEST file (attribute 5) and
8. *** both years on the screen for 5 seconds. It will then close this window
9. *** and return control to the calling routine.
10. *-----*
11. *** PASSER – This argument is not used in this subroutine.
12. *-----*
13. *** COMMON VARIABLES:
14. *** None are in this routine.
15. *-----*
16. OPEN 'JTS.MANUAL.TEST' TO TEMPFILE ELSE RETURN
17.
18. WINDOW
19.
20. *** Read in the year the car in record ID 1 was made (attribute 5)
21. READV YEAR1 FROM TEMPFILE,1,5 ELSE YEAR = "
22. READV YEAR2 FROM TEMPFILE,4,5 ELSE YEAR2= "
23.
24. PRINT YEAR1
25. PRINT YEAR2
26.
27. SLEEP 5
28.
29. WINDOW.CLOSE
30.
31. RETURN
```

You now know two different ways to access data from a record in a file on disk:

1. READ command
2. READV command

Use the READ command if you are looking for more than two attributes from the record. Use the READV command if you are only looking for one or two attributes from the record. If a record is large and you need only a few pieces of information, use up to three READV commands in place of the READ command. If you used the READ memory access command, you would then have to access the information from the dynamic array.

## Using Dimensioned Arrays to Access Data in UniVerse

Dynamic arrays are strings with special delimiting characters to separate out data elements. With dynamic arrays, you do not need to estimate the size of the array; it can be as large or small as needed. As dynamic arrays become larger, drawing elements out of it becomes inefficient. Use dimensioned arrays to access elements of large records more efficiently.

Dimensioned arrays allow us to pre-allocate space in which to store elements so that access time becomes quicker. To use dimensioned arrays, indicate to UniVerse that a special variable of a defined size is needed. UniVerse pre-allocates pointers to each element of the dimensioned array. You can now access element 50 as quickly as element 1. The elements are stored in memory. They can include 50 or 150 elements in a dimensioned array and still be accessed quickly.

With dimensioned arrays, you should be cautious with the amount of space pre-allocated. If you pre-allocate too much space, you waste memory. If you pre-allocate too little space, you inadvertently place data in overflow. Using dimensioned arrays also makes accessing and writing to the disk longer. The system must parse each element, rather than placing the entire record into memory as with dynamic arrays.

Dimensioned arrays are commonly used for large records. Most of the dimensioned arrays are already defined in the users COMMON memory section.

To set up a dimensioned array, use the DIM statement followed by the variable name you want to assign the array. For example, for a dimensioned array named MY.DIM.ARRAY, use the following syntax:

```
DIM MY.DIM.ARRAY(ELEMENT COUNT)
```

The **ELEMENT COUNT** represents the number of elements to pre-allocate for the array.

To access a record from disk and place into a dimensioned array, use the MATREAD statement rather than the READ statement. To read data into the variable MY.DIM.ARRAY from the TEMPFILE handle record ID 1, use the following syntax:

```
MATREAD MY.DIM.ARRAY FROM TEMPFILE,1 ELSE  
  MAT MY.DIM.ARRAY = "  
END
```

Because you break the **ELSE** clause onto a second line, you need to place the **END** statement after the last command within the **ELSE** clause.



**Note:** For the **ELSE** clause, you cannot use the syntax **MY.DIM.ARRAY = "**. Instead you need to enter **MAT MY.DIM.ARRAY = "**. This syntax tells UniVerse that you want it to make every element of the dimensioned array an empty string if the record does not exist in the file. If you forget to include **MAT** and then try to compile this routine, the compiler forces you to indicate an element of that array. Because a dimensioned array is not a long string, you must always tell UV on which element to work.

## ***Exercise 6.5: Using INI.DIM.ACCESS to Access Elements through a Dimensioned Array***

1. Create a new subroutine named **INI.DIM.ACCESS**. Ensure that this array accepts only one argument **PASSER**, which will not be used.

This subroutine reads in record ID 1 from the INI.MANUAL.TEST file and finds what data is stored within attribute 2.

2. Make sure that the top comments are correct and complete.
3. Dimension an array named **AUTO.REC** to five elements. Use the following syntax:

**DIM AUTO.REC(5)**



***Note:** We used 5 in this case because we know that we have five attributes defined in the dictionary for this file layout. Always try to dimension arrays to the correct size and larger to avoid re-dimensioning them each time we add an attribute to the file structure. Do not make it too large or you waste memory.*

4. In the dimensioned array, read the data from record ID 1.
5. Access the second attribute into a variable named **MODEL**.
6. Close the window.
7. Return control to the calling routine.
8. Compile and run this routine.

You should see the same listing as below.

## Listing 6.4: INI.DIM.ACCESS

```
1      SUBROUTINE (PASSER)
2      ** Version# 0.01[3] - 09/14/2001 - 12:01pm - JASONS - develop
3
4      *** Subroutine: JTS.DIM.ACCESS
5      *-----*
6      *** This subroutine will read in record ID 1 from the JTS.MANUAL.TEST file
7      *** into a dimensioned array, access attribute #2 (Model) and display
8      *** it on a blank screen for 5 seconds.
9      *-----*
10     *** PASSER is not used in this routine.
11     *-----*
12     *** COMMON VARIABLES:
13     *** None are used in this routine.
14     *-----*
15     OPEN 'JTS.MANUAL.TEST' TO JTSFILE ELSE RETURN
16
17     *** Dimensioned array we will be putting the automobile information into.
18     DIM AUTO.REC(5)
19
20     *** Read in record ID 1 from the JTS.MANUAL.TEST file into the dimensioned
21     *** array AUTO.REC.
22     ID = 1
23     MATREAD AUTO.REC FROM JTSFILE,ID ELSE MAT AUTO.REC = "
24
25     *** Pull the model of the car from attribute #2 of this dynamic array.
26     MODEL = AUTO.REC(2)
27
28     *** Open a blank window and display what we got out of attribute 2.
29     WINDOW
30
31     PRINT @(10,2):MODEL
32     SLEEP 5
33
34     WINDOW.CLOSE
35
36     RETURN
37     !JASONS~09/14/01~12:01
```

In a dynamic array, in order to access this variable more than once, it is more efficient to place the data into the variable **MODEL**. Otherwise, the system must scan to that attribute for the special string.

With dimensioned arrays, it is not as efficient to place the data into the variable **MODEL**. In a dimensioned array, each element of the array has its own memory space. Accessing element 1 and element 50 takes the same amount of time, as does accessing a single variable. You use dimensioned arrays because they are more readable.

In the above exercise, you did not need to put the second element into the **MODEL** variable. Instead you could have displayed **AUTO.REC(2)** on the screen. Many cases exist where this small inefficiency makes little difference and the added readability improves the code. In these cases, it is better to put the element into a variable that is easier to read rather than programming to know what lives in element (2).

You should use the **;**\* commenting standard when using AUTO.REC(2) to keep the efficiency of the code and still let other programmers know what is stored in AUTO.REC(2). To use this commenting standard, follow the syntax below:

**PRINT@ (10,2):AUTO.REC(2)     ;\*Model of the automobile**

Place the semicolon (**;**) immediately in front of the asterisk (**\***) and place one or more **Tabs** between the 2 statements. Place the **Tab** to make it easier to reach this position if you need to change the comment. The **Tab** also allows you to line up all of your comments as you place equal (**=**) signs. If there is more than one line of code, you can quickly scan the comments. See the format below:

**PRINT AUTO.REC(2)             ;\*Model of the automobile**

**PRINT AUTO.REC(3)             ;\*Number of doors in this automobile**

**PRINT AUTO.REC(4)             ;\* Number of passengers in this automobile**

## Using Dynamic Arrays to Update Record Images on Disk from a Program

You have learned three different ways to access data from a file within an Eclipse program. Now you will learn about updating the data in a file from a program. As a programmer, you need to allow users to view and edit Eclipse applications, as well as place those edits back into the database. We must also ensure that, when users edit records, they are the only users allowed to make changes at that time.

To place a dynamic array back into the record/file on disk, use the **WRITE** statement. The **WRITE** statement is broken down similarly to the **READ** statement.

Segment	Definition
<b>Command</b>	The command itself.  For example: <b>WRITE</b>
<b>Dynamic Array</b>	The dynamic array you want applied to the record for this program.  For example: <b>AUTO.REC</b> .
<b>ON or TO Statement</b>	The file handle, a comma, and the Record ID.  For example: <b>ON TEMPFILE, 1</b>
<b>Possible Exception Clauses</b>	<ul style="list-style-type: none"> <li>• <b>THEN</b> is executed if the record lives in the file.</li> <li>• <b>ELSE</b> is executed if the record is not found in the file. If you do not include an <b>ELSE</b> statement and the record is not found in the file, an error message stating the variable is <b>Undefined</b> occurs at run-time.</li> </ul>

For example, use the following syntax to write out the AUTO.REC dynamic array:

**WRITE AUTO.RECON JTSFILE,ID**

This syntax takes the information in the AUTO.REC dynamic array and places it in the record ID of the JTSFILE.

### ***Exercise 6.6: Creating a New Record in the INI.MANUAL.TEST File from a Program***

1. Add a new automobile to our file. Make the record ID **11**. The car is a Jeep Wrangler, year 1997, 2 doors, and 4 passengers. Name this subroutine **INI.DWRITE.TEST**.
2. Open the file for use to the TEMPFILE file handle.
3. Initialize the dynamic array AUTO.REC to “” using the following syntax:  
**AUTO.REC = ‘’**
4. Load the data into the correct attributes for this dynamic array.
5. Use **WRITE** to put this data onto disk with record ID 11.
6. Return control to the calling subroutine.
7. Compile and run this routine.
8. Use **LIST.ITM** or **CT** and **TCL** to verify the data loaded correctly. Check the listing below.
9. If it did not load correctly, recheck steps 1-6 and see your training manager with any questions.

You should see the same results and listing as below.

```

11
001 Jeep
002 Wrangler
003 2
004 4
005 1997

```

### Listing 6.5: INLDWRITE.TEST

```

1      SUBROUTINE (PASSER)
2      ** Version# 1[1] - 10/03/2001 - 04:04pm - JASONS - develop
3
4      *** Subroutine : JTS.DWRITE.TEST
5      *-----*
6      *** This subroutine will create record ID "11" in the JTS.MANUAL.TEST file
7      *** which will contain the data for a 1997 Jeep Wrangler using a dynamic
8      *** array named AUTO.REC
9      *-----*
10     *** PASSER is not used in this routine
11     *-----*
12     *** Common Variables:
13     *** None are used in this routine.
14     *-----*
15     OPEN 'JTS.MANUAL.TEST' TO TEMPFILE ELSE RETURN
16
17     REC.ID = 11
18
19     AUTO.REC = "
20
21     *** Setup the data in our dynamic array to write it out to disk.
22     AUTO.REC<1> = 'Jeep'
23     AUTO.REC<2> = 'Wrangler'
24     AUTO.REC<3> = 2
25     AUTO.REC<4> = 4
26     AUTO.REC<5> = 1997
27
28     *** Save this new record out to disk on the JTS.MANUAL.TEST file record
11
29     WRITE AUTO.REC ON TEMPFILE,REC.ID
30
31     RETURN
32 !JASONS~10/03/01~16:04

```

Notice in line numbers 24-26 in the code above that you did not need to surround the numbers with quotes. PICK, like most other programming languages, does not let a variable start with a number. PICK assumes that anything starting with a number not surrounded by quotation marks is a numeric value.

There are still problems with this routine. If this routine updates a record in Eclipse, it is writing or attempting to write the record to the file whether or not someone else is using the record. To solve this problem, lock the record to the process while you are updating it. When a record is locked, UniVerse does not allow any other process to lock or any other update to occur until the lock is removed.

To lock a record, use **READU**, **READV4**, or **MATREADU** commands. They all work the same as the **READ** command, except they lock the record in the file until you write it out or use the **RELEASE** command.

To make our routine smart enough to lock the record while we update it, replace Line 19 with the following code:

```
      READU AUTO.REC FROM TEMPFILE LOCKED
      WINDOW
      PRINT 'Record is locked to a different user.'
      SLEEP 7
      WINDOW.CLOSE
      RETURN
    END ELSE
      AUTO.REC = ''
  END
```

Notice the new clause on the **READU** statement: **LOCKED**. This clause is valid only in the **READU**, **READV4**, or **MATREADU** commands. The system executes the **LOCKED** clause if the record is locked to another process when you try to lock it. If you omit the **LOCKED** clause on these statements, the **READU** waits until the other process releases the lock. When the subroutine reaches the **WRITE** statement to save this record off, it releases the lock and lets another process have the record for updates.

You will not use the **READU** command to lock records in Eclipse on a regular basis. If you need to use it, though, you now know how. Also, after a **READU** command if no **WRITE** statement exists, use the **RELEASE** statement to unlock the record. If you do not use the **RELEASE** statement, the lock stays on that ID until the process is terminated (the user logs out of Eclipse).

### ***Exercise 6.7: Updating INI.DWRITE.TEST to Lock the Record Before Updating***

1. In your routine, replace the **AUTO.REC = ''** logic with the **READU** logic.
2. Save and compile the routine.
3. Run the routine.

You should see the same listing as below.



### Listing 6.6: Revised INI.DWRITE.TEST for locking the record

```
1      SUBROUTINE (PASSER)
2      ** Version# 1[2] - 10/04/2001 - 02:37pm - JASONS - develop
3
4      *** Subroutine : JTS.DWRITE.TEST
5      *-----*
6      *** This subroutine will create record ID "11" in the JTS.MANUAL.TEST file
7      *** which will contain the data for a 1997 Jeep Wrangler using a dynamic
8      *** array named AUTO.REC
9      *-----*
10     *** PASSER is not used in this routine
11     *-----*
12     *** Common Variables:
13     *** None are used in this routine.
14     *-----*
15     OPEN 'JTS.MANUAL.TEST' TO TEMPFILE ELSE RETURN
16
17     REC.ID = 11
18
19     *** Make sure that another process is not trying to update this record.
20     *** If it is already locked by another process let the user know and exit.
21     READU AUTO.REC FROM TEMPFILE,REC.ID LOCKED
22     WINDOW
23     PRINT 'Record is locked by another process.'
24     SLEEP 5
25     WINDOW.CLOSE
26     RETURN
27     END ELSE
28     AUTO.REC = "
29     END
30
31     *** Setup the data in our dynamic array to write it out to disk.
32     AUTO.REC<1> = 'Jeep'
33     AUTO.REC<2> = 'Wrangler'
34     AUTO.REC<3> = 2
35     AUTO.REC<4> = 4
36     AUTO.REC<5> = 1997
37
38     *** Save this new record out to disk on the JTS.MANUAL.TEST file record 11
39     WRITE AUTO.REC ON TEMPFILE,REC.ID
40
41     RETURN
42 !JASONS~10/04/01~14:37
```



**Note:** In the above listing, you still need the *END ELSE* clause on the *READ* command. Using the *END ELSE* clause ensures that you set the dynamic array to the empty string if the record was not already in the *INI.MANUAL.TEST* file. *READ* does not set up the dynamic array without the record being there. You, as the programmer, must handle this case. Outside of the routine, if the record does not exist within the file, there may be times you need to generate an error message that indicate to stop processing.

## MATWRITE to Save Data from a Dimensioned Array

Use the MATWRITE function in place of the WRITE function when using a dimensioned array instead of a dynamic array. MATWRITE has the same clauses and structure as WRITE, except that in the array section, MATWRITE is a dimensioned array, as shown below:

```
MATWRITE MY.AUTO.REC ON TEMPFILE,ID
```

The above line of code saves the information in the dimensioned array MY.AUTO.REC to the TEMPFILE file in the record ID.

### ***Exercise 6.8: Creating INI.MWRITE.TEST***

1. Follow the same steps as in Exercises 6.6 and 6.7 to put a new auto record for a Dodge Caravan with 4 doors, 7 passengers, and year 1999. Make sure to lock the record while updating it.
2. Save this record under ID **12**.
3. Use the dimensioned array MY.AUTO.REC of 10 elements.

You should see the same results and listing as below.

```
LIST.ITEM JTS.MANUAL.TEST 12 04:05:43pm 03 Oct 2001 Page 1
    12
001 Dodge
002 Caravan
003 4
004 7
005 1999
;
```

## Listing 6.7: INI.MWRITE.TEST

```
1      SUBROUTINE (PASSER)
2      ** Version# 0.01[2] - 10/04/2001 - 02:46pm - JASONS – develop
3      ** Copied from BP JTS.DWRITE.TEST Version# 1[2] - 10/04/2001 - 02:37pm - JASONS - develop
4
5      *** Subroutine : JTS.DWRITE.TEST
6      *-----*
7      *** This subroutine will create record ID "12" in the JTS.MANUAL.TEST file
8      *** which will contain the data for a 1999 Dodge Caravan using a
9      *** dimensioned array named MY.AUTO.REC
10     *-----*
11     *** PASSER is not used in this routine
12     *-----*
13     *** Common Variables:
14     *** None are used in this routine.
15     *-----*
16     OPEN 'JTS.MANUAL.TEST' TO TEMPFILE ELSE RETURN
17
18     DIM MY.AUTO.REC(10)
19     REC.ID = 12
20
21     *** Make sure that another process is not trying to update this record.
22     *** If it is already locked by another process let the user know and exit.
23     MATREADU MY.AUTO.REC FROM TEMPFILE,REC.ID LOCKED
24     WINDOW
25     PRINT 'Record is locked by another process.'
26     SLEEP 5
27     WINDOW.CLOSE
28     RETURN
29     END ELSE
30     MAT MY.AUTO.REC = "
31     END
32
33     *** Setup the data in our dynamic array to write it out to disk.
34     MY.AUTO.REC(1) = 'Dodge'
35     MY.AUTO.REC(2) = 'Caravan'
36     MY.AUTO.REC(3) = 4
37     MY.AUTO.REC(4) = 7
38     MY.AUTO.REC(5) = 1999
39
40     *** Save this new record out to disk on the JTS.MANUAL.TEST file record 12
41     MATWRITE MY.AUTO.REC ON TEMPFILE,REC.ID
42
43     RETURN
44     !JASONS~10/04/01~14:46
```

## Using WRITEV to Update a Single Attribute of a Record

UniVerse also has a WRITEV function. With WRITEV, the program can update a single attribute of a file's record at a time. If you do not want to update any data besides the single attribute, use this function.

WRITEV is more efficient than the WRITE and MATWRITE statements. If you begin to simultaneously run more than two WRITEV functions in a program on the same record, stop and use the WRITE statement instead.

Exceptions to this rule exist: on large records, such as our LEDGER record, you can use as many as three WRITEV statements, which would take the same amount of time as one WRITE statement.

The WRITEV format is the same as the WRITE format, with the addition of the attribute after the record ID. This attribute allows you to indicate the record's attribute in which you want to place the data. A sample WRITEV statement is below:

**WRITEV NEW.YEAR ON TEMPFILE,REC.ID,ATTB**

Use the above statement to put information from the variable NEW.YEAR onto the TEMPFILE file, REC.ID record, and ATTB attribute. The WRITEV statement, as do the other two WRITE statements, supports THEN, ELSE, and LOCKED clauses.

### *Exercise 6.9: Using WRITEV to Update One Attribute of a Record*

In this exercise, use the WRITEV statement to update attribute 5 (year) to the year 2000 of record ID 12 in your INI.MANUAL.TEST file.

1. Create a new routine named **INI.WRITEV.TEST**.
2. Update the comments at the top to standards.
3. Open the INI.MANUAL.TEST file for use.
4. Lock record ID 12 for update.
5. Change the year from **1999** to **2000** using a **WRITEV** command.

You should see the same results as in Exercise 6.8, except the year should be **2000**.

You should see the same listing as below.

## Listing 6.8: Listing of INI.WRITEV.TEST

```
1      SUBROUTINE (PASSER)
2      ** Version# 0.01[1] - 10/04/2001 - 03:11pm - JASONS - develop
3
4      *** Subroutine: JTS.WRITEV.TEST
5      *-----*
6      *** This routine will change attribute 5 (YEAR) of record 12 (Dodge
7      *** Caravan) in the JTS.MANUAL.TEST file to 2000 using the WRITEV
8      *** statement.
9      *-----*
10     *** PASSER - Not used in this routine (IN)
11     *-----*
12     *** COMMON USED:
13     *** None are used in this routine
14     *-----*
15     OPEN 'JTS.MANUAL.TEST' TO JTSFILE ELSE RETURN
16
17     REC.ID = 12
18
19     *** Lock the record for exclusive use to this process
20     READVU YEAR FROM JTSFILE,REC.ID,5 LOCKED
21     WINDOW
22     PRINT 'Record is locked by another process.'
23     SLEEP 5
24     WINDOW.CLOSE
25     RETURN
26     END ELSE
27     YEAR = "
28     END
29
30     *** Change the variable to 2000
31     YEAR = 2000
32
33     *** Save these changes to disk
34     WRITEV YEAR ON JTSFILE,REC.ID,5
35
36     RETURN
37     !JASONS~10/04/01~15:11
```



**Note:** The READVU command locks the entire record for exclusive use.

## Summary

In this lesson, you have learned different ways to get data into a program from disk. You have also learned how to save changed data back out to a disk. For almost everything you do at Eclipse, you will need to know how to read and write data to the database.

We also introduced the multi-dimensional aspect of the UniVerse database. For this topic, we discussed how easy it is to add and remove data elements within our tables.

## Frequently Asked Questions

### Question:

Why do I have to OPEN the file before I can use the READ or WRITE statements? What does the OPEN statement actually do?

### Answer:

The OPEN statement goes out to the physical file that you indicated at the OS level to retrieve the header information for the following aspects:

- Type of file.
- Where the file lives.
- Permissions on the file.

It retrieves other information, as well. The READ and WRITE statements must have this information to be able to correctly hash groups for each record and to access this data on the disk.

### Question:

What are the advantages to using a dimensioned array when the READ and WRITE statements are slower? When should I use a dimensioned array instead of a dynamic array?

### Answer:

When you work with large records, the access time to pull out elements in the dynamic array becomes inefficient. The dimensional array allocates each element its own space in memory. Accessing attribute 300 is as fast as accessing attribute 1.

In Eclipse you should only use dimensioned arrays on the files that have pre-determined array names and sizes and that live in COMMON memory. If the file you are working on does not have a pre-determined dimensioned array, use a dynamic array.

## Assignments – Lesson 6

1. In your INI.MANUAL.TEST file of each record, create a subroutine that changes the YEAR attribute to be both a **start** and **end** year.

- Place the starting year in value position 1 and the end year in value position 2.
- Name this routine INI.MANUAL.YEARS.
- Create three internal subroutines using a dynamic array, one using a dimensioned array, and one using WRITEV. The years you will use should follow the below table:

Record	Model	Start Year	End Year
1	Diablo	1990	1999
2	F40	1991	2001
3	Countach	1975	1990
4	Testarosa	1976	1991
5	911	1978	2001
6	Esprit	1975	1991
7	Boxster	1995	2001
8	Viper	1993	2001
10	Mustang	1965	2001
11	Wrangler	1996	2001
12	Caravan	1985	2001

2. Using Eclipse dictionary maintenance, create two new dictionaries on your file:

- Create one for START.YEAR.
  - Create one for END.YEAR.
  - Make both D types, and point them to Attribute 5 and the corresponding value position.
-

# *Lesson 7*

## *Standard Eclipse Operating Environment*



### *Objectives*

**In this lesson, you will learn about:**

- Common memory
- Eclipse's use for Common
- Calling external functions or subroutines in Eclipse
- Opening files with UT.OPEN.FILE



# Overview

In this lesson, instead of focusing on Eclipse BASIC or Eclipse functions, you will be taking an in-depth look at Eclipse's standard operating environment. This includes discussions on Common memory (Common), including:

- Using Common
- The two types of Common
- Variables set in Common.

This lesson also discusses:

- Standard files
- The correct way to open files in Eclipse rather than using OPEN,

And reviews:

- Dynamic and dimensioned arrays
- READ and WRITE statements.

These discussions provide you with pertinent information for performing your job at Eclipse. We recommend that, regardless of your previous programming experience, you give this lesson your full attention.

## What is Common?

Common memory is a space in memory where developers can store variables of any kind. All programs running in the process can then access and change these variables. Despite this definition, though, Common is almost a philosophical concept to anyone who has worked with PICK BASIC for very long.

Common is much like the global variable concept in other programming languages. Professors and professional consultants may preach “global variables are bad,” but such statements oversimplify the facts. Common and global variables can be misused like anything else in software development (think of the GOTO statement). As a programmer, it is your job to determine what the true scope of a variable should be. When used correctly, Common and global variables can benefit any application.

For example, for variable that all programs need to access, such as the user ID that the system logs to the application, you have three options:

1. Read the user ID from a file record in every subroutine.
2. Pass the user ID through the argument list of every subroutine.
3. Use Common so routines needing to access this variable can do so straight from memory without reading it from disk.

Common is the best option to use in this scenario. Passing every possible global variable through every subroutine is cumbersome, and using OPEN and READ statements are the two most expensive functions a program can perform.



***Note:** In Eclipse, the Common variable name `USER.ID` identifies the user logged into the Eclipse session. `User.ID`'s common variable, `SECURITY`, contains all settings for this user from the `INITIAL` file.*

At Eclipse, Common is used for storing:

- Data that many routines need to access quickly, including routines that are often called in high profile places (such as Order Entry). For example: `USER.ID`.
- File handles for files that are repeatedly used. By using Common, routines do not have to open and re-open files. For example, in Eclipse, routines do not have to open the `LEDGER` file. Instead this file is opened at login and stored in the Common variable named `LEDFILE`.
- Dimensioned arrays of data that read from the standard files. For example, `LED` would be used for `LEDFILE` and `PRD` would be used for `PRODUCT` file. For a full listing of these arrays and the corresponding files, please see the Appendix.



***Note:** The dimensioned arrays above are the ones referred to in the previous lesson. They are the only dimensioned arrays used at Eclipse, unless your supervisor instructs you otherwise.*

To see a listing of everything defined in Eclipse's Common memory, pull up the file CC and the record COMMON in view-only mode in the program editor. You should see something similar to the listing below.

#### Listing 7.1: CC COMMON

```
1      COMMON /STDCOM/ COMDATA(150), SECURITY , MAIN.MEN, FILES(50) ,  
AOFILES(150), GLDATA(35)  
2      ** Version# 6[1] - 10/02/2001 - 04:55am - CHRISM - develop  
3  
4  
5      COMMON FLAGS(100)  
6      COMMON TCL.LEVEL., RECALL.FLAG, E.MESS.  
7      COMMON LED(200),OLED(200),LD(100),OLD.LD(100)  
8      COMMON PRD(200),PRDP(30),PRD.BR(70),PRDC.BR(30),PRDD.BR(30)  
9      COMMON PGRP(10),PLNE(30),PLNE.BR(10),PLNB(5),BL.BR(50)  
10     COMMON CUS(200),CUSS(200),MA(25)  
11     COMMON TAX(15),AR(50)  
12     COMMON SCROLL.VAR, SVIEW.DEF., SVIEW.DATA.(10)  
17  
14     $INCLUDE CC EQU.ESC.OBJECT  
15     $INCLUDE CC EQUATES  
16  
17     PROMPT CHAR(0)
```

This listing details how to use the COMMON function in order to tell UniVerse when to place a variable in Common. Much like the DIM statement, you need to indicate how large the array of Common information should be. Looking at lines 1 and 5, you can also see that Common accepts a name parameter. This parameter defines either named or unnamed Common. You will learn about these types of Common in the next section.

Technically, each subroutine that will access common variables must declare this at the top of the routine. However, our pre-compiler forces all subroutines to use the same Common in Eclipse. So the subroutine does not need to make this declaration. Pull up the OC version of our last subroutine, INI.WRITEV.TEST, to see results similar to the listing below.

## Listing 7.2: OC Version of INI.WRITEV.TEST

```
1      SUBROUTINE (PASSER)
2      $INCLUDE CC COMMON
3      *
4      *
5      *
6      *
7      *
8      *
9      *
10     *
11     *
12     *
13     *
14     *
15     OPEN 'JTS.MANUAL.TEST' TO JTSFILE ELSE RETURN
16     *
17     REC.ID = 12
18     *
19     *
20     READVU YEAR FROM JTSFILE,REC.ID,5 LOCKED
21     CALL WINDOW("","","","","")
22     PRINT 'Record is locked by another process.'
23     SLEEP 5
24     CALL WINDOW.CLOSE("")
25     RETURN
26     END ELSE
27     YEAR = "
28     END
29     *
30     *
31     YEAR = 2000
32     *
33     *
34     CALL EWRITEV(YEAR,JTSFILE,REC.ID,5,"UBP~JTS.WRITEV.TEST~34")
35     *
36     RETURN
37     * Compiled by JASONS on 10/04/01 15:11 from UBP:JTS.WRITEV.TEST
38     *** Version# 0.01[1] - 10/04/2001 - 03:11pm - JASONS – develop
```

Notice that the pre-compiler added the code **\$INCLUDE CC COMMON** at line 2 of this subroutine. When compiled at the Universe level (using the BASIC command you learned in Lesson 4), this code places data from the CC COMMON record at the top of this routine. Remember, you do not have to manually type in the BASIC command. The pre-compiler performs the BASIC on this subroutine once it generates the OC code.



**Caution:** Pay attention to this last piece of information. If you ever change what is in CC COMMON, you must re-compile everything in the OC using the BASIC command. If you do not re-compile, the system displays **Common size mismatch errors between the subroutines**. **Your supervisor must approve any changes to Common.**

## Defining Named Common, Unnamed Common, and Understanding Eclipse Menus as Subroutines

On-line UniVerse Help explains the difference between named and unnamed Common as follows:

*A common area can be either named or unnamed. An unnamed common area is lost when the program completes its execution and control returns to the UniVerse command level. A named common area remains available for as long as the user remains in the UniVerse environment.*

While this is a good definition of named and unnamed Common for most UniVerse developers, it does not define how unnamed Common works within Eclipse.

In Eclipse, the system never drops the user down to a UniVerse prompt. Instead it keeps users within Eclipse menus, and as discussed, everything that runs from an Eclipse menu is a subroutine. Unlike the traditional STOP function that ends the process and clears or protects the unnamed Common, when Eclipse uses an unnamed Common, it returns control to the subroutine before the end of the program.



**Note:** *The Common area name can be of any length, but only the first 31 characters are significant.*

When Eclipse users log on to the company's main server, the first process they see is the LOGON.ECLIPSE program. In UniVerse, VOC entries with the same name as the UniVerse account name that the session is initiated in will be automatically run. This is a unique feature of UniVerse.

In the next exercise, you will look in develop to determine why LOGON.ECLIPSE is the first process UniVerse runs at login.

## ***Exercise 7.1: Exploring Why LOGON.ECLIPSE is the First Program All Develop Users Run***

1. Press **F2-T** to access TCL.
2. Enter **CT VOC DEVELOP**.

You should see the same results as shown below.

```
<Esc>=Exit,<Alt>-P=Prev Cnds,<Alt>-L=Lists,<Alt>-C=Clear List,<Alt>-X=Phan Exec
;CT VOC Develop

      DEVELOP
0001 PQ
0002 HCLEARCOMMON
0003 HLOGON.ECLIPSE
0005 P
;
```

Attribute 1 of this record tells UniVerse that this VOC entry is a **proc** or **process**. Line 2 tells UniVerse to execute the CLEARCOMMON command. This command is used to avoid the **Common size mismatch** error when you switch between accounts (such as develop and support). The most important line in this VOC entry is Line 4. This line tells UniVerse to execute the program called LOGON.ECLIPSE. When a user starts a new UniVerse session in the account named Develop, UniVerse executes all of the commands set forth in the above VOC entry.

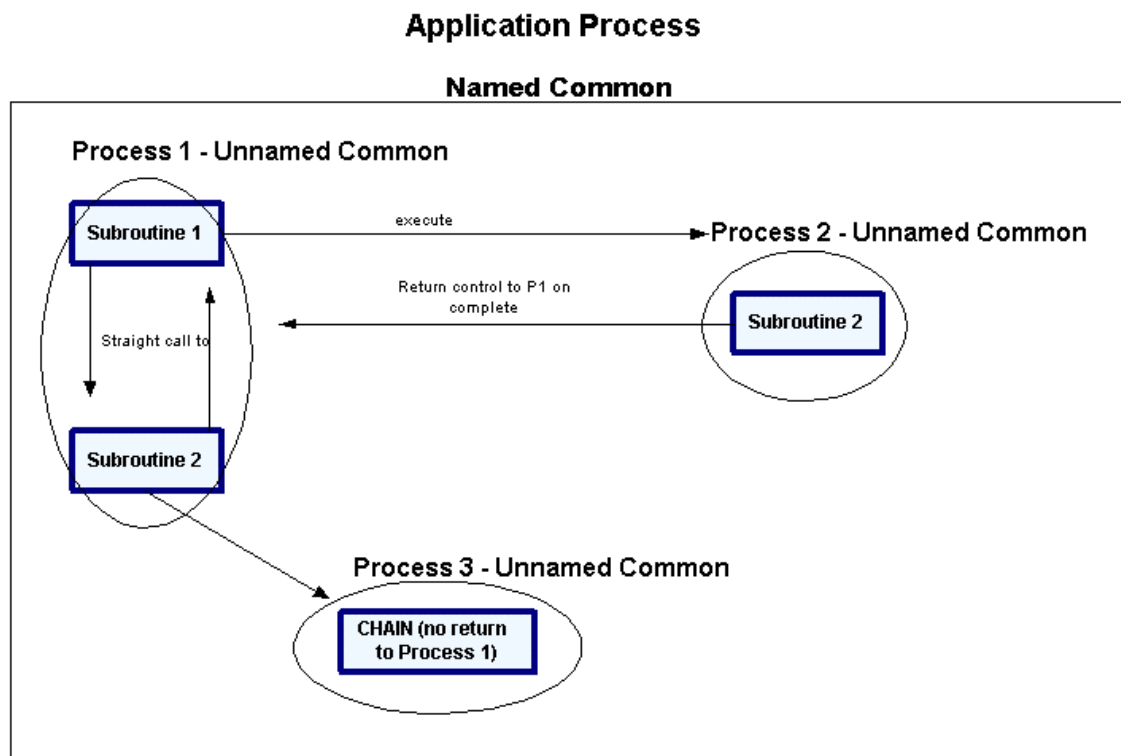
LOGON.ECLIPSE is not a complex routine. It checks a few environment variables and then calls the subroutine MENU.STARTER.

MENU.STARTER initializes our standard files and most of the common variables. It validates users' IDs and passwords when they log into Eclipse. Once Eclipse has validated the user ID, the remaining user specific common variables are initialized and then execute the MENU.DRIVER program using the CHAIN command. The CHAIN command terminates the MENU.STARTER process and continues it as the MENU.DRIVER.

The CHAIN command is the key to the differences between the named and unnamed Common at Eclipse. Every time a user pushes a menu at the root level, MENU.DRIVER runs another CHAIN command to terminate the old process and begin a new one. At this point, all unnamed Common is cleared and only named Common continues to the next process. If you use the CHAIN command, you cannot return to the current process, which is one drawback to using the command.

The EXECUTE command performs a UniVerse process in its own processing space and waits for that process to complete. Once the process completes (or dies), the current process picks up where it left off. Because the EXECUTE statement has its own process space, the process occurring under this statement is passed and the named Common variables (they are unprotected) and unnamed Commons are cleared (protected). The unnamed Common is cleared for the new process (inside the EXECUTE) but is still available to your original process. Your lower process sees only its own unnamed Common area.

The diagram below demonstrates how named and unnamed Common work together:



By using EXECUTE and CHAIN on the push levels, Eclipse protects the common variables in unnamed Common from being changed in the lower routines.



**Note:** As a programmer, you are responsible for understanding the statement above. Rather than saving common variables before calling another routine that may overwrite the variables, use the *EXECUTE* logic to protect the unnamed Common variables. If your called subroutine only takes one argument through the *EXECUTE*, use the internal Eclipse subroutine **EXEC.PGM** to run this lower routine. If the subroutine you need to call takes more than one argument, write a driver routine that can be called through the *EXEC.PGM* interface. These driver programs will be discussed in detail in later lessons.

When you add a variable to named or unnamed Common, you are defining the scope of that variable. All processes in Eclipse that are started from the menus of this user session can access and change data in named Common, while only the current level of processes can access and change the unnamed Common variables. This availability of named and unnamed Common variables determines the placing of variables like *USER.ID* into named Common and *LED* and *PRD* into unnamed Common.

Consider the following scenario:

If a user is in product file maintenance and pushes to a second screen with a different product, you do not want to overwrite the current product's information with the new product. You want them both to stay intact. By using *EXECUTE* and unnamed Common, you can accomplish this with no extra code.



**Note:** You must get your supervisor's permission before adding or deleting any common variables.



## Introduction to CC EQUATES and More Common Within Eclipse

At this point, you have learned about CC COMMON and should know the following:

- All Eclipse subroutines include the same Common variables.
- Use named Common variables when all processes need to access and change the data.
- Use unnamed Common variables when only the current process needs to access and change the data.
- CHAIN and EXECUTE protect the unnamed Common area from level to level.

You are probably still looking at CC COMMON and trying to find the USER.ID variable we keep referring to and do not see it defined. We will discuss why that is so in the next section.

## How Eclipse Defined Common

The architects of Eclipse defined Common using only arrays, rather than defining each variable in either unnamed or named Common. They then placed the arrays into named and unnamed Common variables. This may sound like a strange concept at first, but let's explore what makes this a good decision.

In the Program Editor, pull up CC EQUATES in view-only mode to understand how this definition worked. You should see a listing like the one below.

### Listing 7.3: First 50 Lines of CC EQUATES

```
1
2  ** Version# 112[2] - 09/20/2001 - 09:29am - DONS - develop
3  * G/L Auto Posting account number Equates
4  EQU GL.AUTO.INVTY   TO GLDATA(4)
5  EQU GL.AUTO.FGHT    TO GLDATA(5)
6  EQU GL.AUTO.HNDL    TO GLDATA(6)
7  EQU GL.AUTO.PURCH   TO GLDATA(7)
8  EQU GL.AUTO.CPTI    TO GLDATA(8)
9  EQU GL.AUTO.XFER.AR  TO GLDATA(9)
10 EQU GL.AUTO.XFER.AP  TO GLDATA(10)
11 EQU GL.AUTO.XCH      TO GLDATA(11)
12 EQU GL.AUTO.INVADJ   TO GLDATA(12)
13 EQU GL.AUTO.DISCG    TO GLDATA(13)<1,1>
14 EQU GL.AUTO.WOEDISCG$ TO GLDATA(13)<1,2>
15 EQU GL.AUTO.OSAP     TO GLDATA(14)
16 EQU GL.AUTO.GAS      TO GLDATA(15)
17 EQU GL.AUTO.AR       TO GLDATA(15)<1>
18 EQU GL.AUTO.AP       TO GLDATA(15)<2>
19 EQU GL.AUTO.UBAP     TO GLDATA(15)<3>
20 EQU GL.AUTO.CASH     TO GLDATA(16)
21
22 EQU AR.AGING.TYPE$   TO GLDATA(19)
23
24 EQU GL.POSTABLE.AR$  TO GLDATA(20)<1>
25 EQU GL.POSTABLE.AP$  TO GLDATA(20)<2>
26 EQU GL.POSTABLE.UBAP$ TO GLDATA(20)<3>
27 EQU GL.CONVERSION.DT$ TO GLDATA(20)<4>
28
29 * Equates preserved when levels are pushed
30 EQU USER.ID          TO COMDATA(1)
31 EQU PORT             TO COMDATA(2)<1>
32 EQU TERM.VERSION$    TO COMDATA(2)<2>
33 EQU DOWNLOAD.GDL$    TO COMDATA(2)<3>
34 EQU PASSER.COM       TO COMDATA(3)
35 EQU SCR.WIDTH        TO COMDATA(4)
36 EQU SCR.LENGTH       TO COMDATA(5)
37 EQU MENU.STACK       TO COMDATA(6)
38 EQU WINDOW.LEVEL     TO COMDATA(7)
39 EQU TTY.DATA         TO COMDATA(8)
40 EQU RELOG.NOW        TO COMDATA(9)
41 EQU TCL.LEVEL        TO COMDATA(10)
42 EQU PUSHMENU        TO COMDATA(11)
43 EQU LOCATION         TO COMDATA(12)
44 EQU GL.AUTO         TO COMDATA(13)
45 EQU EOMDS$          TO COMDATA(14)
46 EQU PRC.PRECI$      TO COMDATA(15)<1>
47 EQU PRC.ROUND$      TO COMDATA(15)<2>
48 EQU PRC.PRECI.SLS$  TO COMDATA(15)<3>
49 EQU LOG.OVRDS       TO COMDATA(16)
50 EQU TAG.COST.UPD$    TO COMDATA(17)
```



**Caution:** While many of the above variables do not end with a \$ symbol, Eclipse's standard is now to include the \$symbol at the end of all new common variables.

In CC COMMON, you should notice that there was actually an INCLUDE statement on this file (CC EQUATES), which places these lines of code into the subroutines at the time of compilation. Look at line 30: it has the variable USER.ID equated (or made to be a pointer) to the Common variable COMDATA(1). The USER.ID variable is only one part (element) of the COMDATA array. Yet when you write or read any code that needs to obtain a user ID, you see USER.ID referenced rather than the unreadable COMDATA(1). In Listing 7.1, line 1, COMDATA is placed in the named Common section. Because the array that we equated USER.ID to lives in names Common, USER.ID will also reside in named Common. You should also notice as you read further in the CC EQUATES that, had we put every variable in COMMON, the routine would become unbearable to read or maintain.

The most important feature of common data inside arrays is that the size of Common does not need to be increased when new Common variables are added. The new variable is equated to the next element in the current common array. This means you do not need to recompile every routine for the addition of one Common variable (as long as you used CC EQUATES and not CC COMMON). Instead you only have to compile the routine that references the new variable for everything to work.



**Caution:** *You must compile any routines that reference this new Common variable (from CC EQUATES) before they can be used. If you do not, the correct pointers will not be established within them to access this equated Common variable.*



**Note:** *When defining a new Common variable, you must pick the correct array to put it in. Some arrays live in named and others live in unnamed Common. You must decide what the scope of this new variable is before putting it into the CC EQUATES record.*



**Tip:** *One major drawback exists in putting Common variables as equates to large arrays. When using the RAID (UniVerse) debugging tool, you cannot find the value of an equated variable. Instead you must know the array and element within the array to find it. However, if you use the Eclipse SHOW function, this drawback can be minimized.*

## Standard Files, OPEN.STANDARD.FILES, and UT.OPEN.FILE

In this section, you will learn another way that Eclipse uses Common: the standard file handle. The architects created an array called FILES, which resides in the named Common area. This array contains the most commonly used files within Eclipse. These files are open at login time and remain open so the programs do not have to keep reopening them.

The most expensive thing a program can do is OPEN a file. If a file is not a Common file, it gets closed at the end of every routine. Common files remain accessible throughout routines. For files without Common, each subroutine would need to either open a lot of files at the top or pass file handles as parameters.

So, by picking the top 50 files and making them standard files, all programs can read or write to these files without the overhead of opening them. There is a hidden benefit in this approach: any program accessing any of these files always uses the same file handle. The PRODUCT file is always PRDFILE, the ENTITY file is always CUSFILE, and so on.

As a programmer, you can tell what a file is by the file handle. To find a list of all standard files, look in OPEN.STANDARD.FILES. The system displays a list of the files that are opened at login time.

If you ever need to access a file that is not in the standard file list, you should not use a direct OPEN statement to access it. Instead, use Eclipse's function UT.OPEN.FILE to open the file. UT.OPEN.FILE keeps the last 50 files open to this process in named Common. Therefore, if a user commonly hits a non-standard file, the file does not have to be reopened each time.



**Note:** *UT.OPEN.FILE replaces the older subroutine UT.OPEN.COMMON.FILE because it is an easier format to understand and use.*



**Tip:** *If you have developed in PICK BASIC, you should notice a difference in the way you call subroutines in ECLIPSE BASIC. You do not have to use the CALL statement in front of the routine, nor do you need to supply the arguments on the end of the subroutine if you are not going to use them. The pre-compiler populates these with the "" string. Be aware that when you change the number of arguments in a subroutine, you must recompile all routines that call this subroutine. If you do not, you will get an **Argument mismatch** error at run-time.*

Below is a table defining the parts of the following syntax that you want to use to open your INI.MANUAL.TEST file to TEMPFILE:

```
UT.OPEN.FILE 'INI.MANUAL.TEST',TEMPFILE,ERR.MSG  
IF ERR.MSG THEN RETURN
```

Parts	Description
Routine name	UT.OPEN.FILE
Argument list:	<ul style="list-style-type: none"> <li>• <b>FILENAME:</b> The actual file you want to open in this routine.</li> <li>• <b>FILE.HNDL:</b> Returned back to the calling routine as the handle to access above file.</li> <li>• <b>ERR.MSG:</b> Set to the error string if the file could not be opened.</li> <li>• <b>NO.ERR.DISP:</b> Tells UT.OPEN.FILE not to open a window in case it is being called from a phantom, web, or Java process that cannot close this window.</li> <li>• <b>HNDL.NUMBER:</b> Actual element within the FILES array where this file handle is living.</li> </ul>

Notice that you did not suppress the error message display. This process is live and you want the user to be notified if the file cannot be opened.



**Caution:** *UT.OPEN.FILE* does not display the error message when the file cannot be opened. If the file is a phantom, web, or Java process (or any other process for which you do not want a window opened or a message displayed), you must set the flag **NO.ERR.DISP** to **YES (1)** so the window does not display. You must also handle the **ERR.MSG** coming back into the program. If you do not, any access to this displays error messages: **with improper data type** error messages.

## Exercise 7.2: Using **UT.OPEN.FILE**

1. Pull up **INI.WRITEV.TEST** in edit mode.
2. Change the native **OPEN** statement to **UT.OPEN.FILE**.
3. Make sure to check **ERR.MSG** in case the file cannot be opened.
4. Save the routine and compile it.
5. Run the routine.

You should see the same results as in the previous lesson. The listing should be similar to the listing below.

### Listing 7.4: Updated **INI.WRITEV.TEST** to use **UT.OPEN.FILE**

```
1      SUBROUTINE (PASSER)
2      ** Version# 0.01[2] - 10/05/2001 - 07:33pm - JASONS - develop
3
4      *** Subroutine: JTS.WRITEV.TEST
5      *-----*
6      *** This routine will change attribute 5 (YEAR) of record 12 (Dodge
7      *** Caravan) in the JTS.MANUAL.TEST file to 2000 using the WRITEV
8      *** statement.
9      *-----*
10     *** PASSER - Not used in this routine (IN)
11     *-----*
12     *** COMMON USED:
13     *** None are used in this routine
14     *-----*
15     UT.OPEN.FILE 'JTS.MANUAL.TEST',JTSFILE,ERR.MSG
16     IF ERR.MSG THEN RETURN
17
18     REC.ID = 12
19
20     *** Lock the record for exclusive use to this process
21     READVU YEAR FROM JTSFILE,REC.ID,5 LOCKED
22     WINDOW
23     PRINT 'Record is locked by another process.'
24     SLEEP 5
25     WINDOW.CLOSE
26     RETURN
27     END ELSE
28     YEAR = "
29     END
30
31     *** Change the variable to 2000
32     YEAR = 2000
33
34     *** Save these changes to disk
35     WRITEV YEAR ON JTSFILE,REC.ID,5
36
37     RETURN
38     !JASONS~10/05/01~19:33
```

Notice that you did not need to supply the arguments at the end of the **UT.OPEN.FILE**. The pre-compiler fills these arguments with the “” string and passes it on.



**Note:** Because *UT.OPEN.FILE* assigns this file handle to a Common file handle in the *FILES-()* array, you do not need to use *UT.OPEN.FILE* in a stand-alone phantom routine created for one time use. However, we still recommend you use *UT.OPEN.FILE* for consistency.

## Summary

In this lesson, you have gone over the following:

- Common memory space
- Named versus unnamed Common
- How Eclipse uses Common
- How to call external functions or subroutines in Eclipse
- How to correctly open files in Eclipse using the `UT.OPEN.FILE` subroutine.

## Frequently Asked Questions

### Question:

How do I call an external subroutine, such as `UT.OPEN.FILE`, in the native UniVerse BASIC?

### Answer:

Use the **CALL** statement. If you look at the Listing 7.4, line 15 would become the following:

```
CALL UT.OPEN.FILE('JTS.MANUAL.TEST' ,JTSFILE,ERR.MSG,"")
```

You need to supply the final two arguments to UniVerse or you would receive the **argument size mismatch** run-time error message. Never call a subroutine in this manner.

In Eclipse, you can sometimes use the **CALL @** function, which you will learn about in a later lesson.

## Assignments – Lesson 7

1. Outline the steps to create a new common variable within Eclipse. Assume there is room in the current Common arrays. This Common variable will store the users' default-shipping branch for this session.

What would you name this variable? Should it live in named or unnamed Common? Why?

Hand this outline and information into your training supervisor.

2. Create a subroutine named `INI.UT.OPEN.FILE`. Use **UT.OPEN.FILE** to open your `INI.MANUAL.TEST` file and print out (in human-readable format) the information for the automobile in record ID 4 of the `INI.MANUAL.TEST` file. Make sure to give labels to each piece of data you display.



---

## ***Lesson 8***

# ***Screen Functionality: Calling, INP, F12, Esc, and Hotkeys***



### ***Objectives***

**In this lesson, you will learn about:**

- Screen design and screen calling
- Basic usage of INP
- Handling F12, Esc, and Hotkeys

### ***Chapter 8***

## **A. Screen Designer**

### **Screens:**

Screens are tools we use to display information to the user. We also use screens as a way to receive information from the user. Through the use of screens Eclipse creates a UI environment for the user to interact with. On these screens we can include mouse event handling and program hotkeys to allow the user to easily navigate their way through the Eclipse system.

### **Screen Designer:**

To design a screen for a particular subroutine we need to enter the program editor. After selecting the desired subroutine we then need to enter the key 'S' for "Screen". We are then taken to the Screen Designer. You will note that the first time entering the Screen Designer we are prompted with a blank screen. It is now our job as programmers to build a screen that follows Eclipse design protocol so that we maintain a uniform appearance.

### **Mouse Utilization:**

The Screen Designer strongly utilizes both right and left buttons on the mouse. Double clicking on the right mouse button will change the cursor to one of the six listed drawing tools: Standard (none), Single Vertical/Single Horizontal, Double Vertical/Single Horizontal, Single Vertical/Double Horizontal, Double Vertical/Double Horizontal, and Erase.

#### *Keyboard Controls*

- Standard (none) – When in standard mode no drawing will be done. This is the mode that we should be in if we are wanting to type text.
- Single Vertical/Single Horizontal – This drawing mode will draw a single line wherever the cursor is moved to.
- Double Vertical/Single Horizontal – When in this mode moving up and down will draw a double line, while moving left and right will draw a single line.
- Single Vertical/Double Horizontal – When in this mode moving up and down will draw a single line, while moving left and right will draw a double line.
- Double Vertical/Double Horizontal – When in this mode moving up, down, left, or right will draw a double line.

#### *Mouse Controls:*

- Note that when using any of the drawing modes a single left mouse click will reposition the cursor to any desired location on the screen.
- Horizontal and Vertical lines can be drawn using a single click of the right mouse button. Simply choose a location in which to start a line, single click the right mouse button. Then position the cursor to the ending position and perform another single right click. The Designer automatically knows to draw the selected style of line between the two chosen points.

## **Saving:**

When we are finished drawing and wish to return to the Editor press the Esc key. We are then taken to our program change log where we will note the changes and assign a tracker number.

\*When the screen is compiled it is saved under the same name as our subroutine, this will come into play when we want to call the screen.

## **Dump Screen:**

The DS command inside the Program Editor is very useful when programming a screen. When the DS command is executed the printer will print our screen with (x,y) coordinates, making it easy to locate specific points on our screen. Keep in mind that the printout begins at row and column 0, when assigning window sizes we must include these 0 positions.

## **B. SCREEN Command**

### **History:**

Up until now when we displayed a window by simply using the WINDOW command. Now that we have a screen that we want to show we are going to want to get more specific with our WINDOW statement.

***\*Note that if we type WINDOW followed by hitting the F10 key the compiler will automatically tell us the desired variables to include in the statement. This works for all Eclipse subroutines.***

WINDOW includes the following eight passable variables:

1. STCOLX – The starting column on the screen
2. STROWX – The starting row on the screen
3. WIDTH – How long the screen spans horizontally
4. LENGTH – How long the screen spans vertically
5. STYLE – Style of the window
  - 0 –single line border
  - 1 –single line vertical, double line horizontal
  - 2 –double line vertical, single line horizontal
  - 3 –double line vertical, double line horizontal
  - 9 –no border
6. SCRN – Screen ID
7. TITLE – Title to display to the top of the screen
8. REL – 1=> coordinates relative to current window

If we do not enter anything for the first two values the compiler automatically centers the window in the screen. In order for the screen to look professional we must be precise with our WIDTH and LENGTH pass. *This is one place where our DS printout will be helpful.*

## SCREEN:

After we have properly displayed a window it is time to call our screen. We can do this by using the SCREEN command. Early we learned that the screen is stored under the same name as the subroutine it is designed for, for that reason we need only to type SCREEN if we are in that screens subroutine. If for some reason we are not in the screen's subroutine we will need to reference the screen name like this →(SCREEN 'INI.MANUAL.TEST').

### Exercise 8.1

Let's build a screen that will have the capability to show all information in our *INI.MANUAL.TEST* file. The top line should be the automobile ID line. This ID line should be separated from the rest of the areas with a single horizontal line. (look below)

	Automobile ID:	
	Make :	
	Model :	

\*Note, we should always design our screens to match those already built by Eclipse. If you have any questions about format simply browse through the Eclipse system and mentally note the standards for each page. \*\*Or view our standards guide.

## C. **PRINT @ Command**

As we learned in previous chapters the PRINT @ command is useful when we want to print information to a desired position on a screen. Now that we have a background screen to print to and a DS printout, the PRINT @ command becomes that much more useful. If we created a screen for our INI.MANUAL.TEST information we would want to display the MAKE, MODEL, SEATS, DOORS, YEAR.START, and YEAR.END headings in our screen. With the PRINT @ command we can print information to the proper place on a screen. *PRINT @(COLUMN, ROW) \*\*relative to the screen displayed*

## D. **CLEAR.SCREEN**

If we ever want to clear the information on a screen without clearing what was written by our screen compiler we would use the CLEAR.SCREEN command. The CLEAR.SCREEN command basically prints the most recently called screen, in that window level, on top of the screen that is currently being displayed.

## ***E. Introduction to INP***

All we have learned so far is how to design a screen, display, and print out information that we already have in memory. Next we are going to learn how to take in information from the user. There are many ways of doing this but for now we will work with the INP command. All inputs should be written as internal subroutines. The subroutines should always begin with the letters INP, for example: "INP.AUTOMAKE:" This way everyone who looks at your code realizes that this subroutine is inputting data from the user.

A typical INP subroutine is made up of three pieces: The INP line, the QUIT line, and the ON MOVE line.

```
IN.AUTOMAKE:    INP AUTOMAKE, 12,15,18
                IF QUIT THEN GOTO FINISH
                ON MOVE+1 GOTO INP.AUTOMAKE, INP.AUTOMAKE,
INP.AUTOMAKE, INP.AUTOMAKE, INP.MODEL, INP.MODEL, INP.MODEL
```

### **The INP line**

The INP command has 12 passers listed below:

1. Variable – The name of the variable that the input should be stored in
2. Hor – The horizontal position that the input should be started at
3. Ver – The vertical position that the input should start at
4. Length – The maximum length that the user will be allowed to enter
5. CNVR – Conversion
6. STPOS – Starting position
7. AUTORETURN – Auto return from INP @ end of length
8. DFLT – Default value for this INP field
9. VERIFY – Criteria that the input must meet. This can be done using a D:, S:, F:, or C:
10. TERMCHARS –
11. HELPSCR –
12. OUTSTRING –

### **The QUIT line**

The variable QUIT is a common binary variable that is turned on if the user strikes the ESC or F12 key.

Example: IF QUIT THEN GOTO FINISH

This line checks to see if the user has struck the ESC or the F12 key, if so then take us directly to the FINISH internal subroutine. In the FINISH internal subroutine we will further check to see which key the user struck so that we can either exit with a save or exit without a save.

## The ON MOVE line

The ON MOVE+1 GOTO line has seven options. The order of these seven options is as follows: ONERROR, Left, Up, Right, Down, Tab, Enter. This means that if the user moves using any of these keys they will be directed to the subroutine that we have specified. Commit the movement order (Left, Up, Right, Down) to memory, this is not the only place this will be used.

### *\*FALLING OFF THE SIDEWALK*

Falling off the sidewalk is logic that we will be using when dealing with Input subroutines. The metaphor “falling off the sidewalk” is describing to us what happens when we complete all tasks in an internal subroutine and there is no MOVE or GOTO statement to guide us. Our program simply falls to the next executable line in our code. For example what will happen in the following subroutine if the user enters information in the IN.AUTOMAKE input zone and hits the enter key?

```
*-----*
IN.AUTOMAKE:  INP AUTOMAKE, 12,3,18
               IF QUIT THEN GOTO FINISH
               ON MOVE+1 GOTO IN.AUTOMAKE, IN.AUTOMAKE,
IN.AUTOMAKE, IN.AUTOMAKE, IN.MODEL
*-----*
IN.MODEL:     INP MODEL, 12,4,18
               IF QUIT THEN GOTO FINISH
               ON MOVE+1 GOTO INP.MODEL, INP.MODEL,
INP.AUTOMAKE, INP.MODEL, INP.DOORS
*-----*
```

That's right they are going to be taken to the INP.MODEL internal subroutine because there is nothing written in the ON MOVE statement dealing with the Enter key. Falling off the sidewalk is logic all programmers should get comfortable with.

Exercise 8.

In this exercise we should create inputs for every data point on the *INI.MANUAL.TEST* screen.

## Handling ESC and F12

The difference between hitting the ESC key and hitting the F12 key is that the ESC sets only the QUIT variable to true, whereas the F12 key sets the QUIT variable and the F12 (common Boolean variable) to true as well. In our FINISH subroutine that we will write we need to take care in which test we run first, the QUIT test or the F12 test.

*\*If the user hits the F12 key we want the user to exit from the current task without saving any changes (this will make more sense when we get into the UPDATE subroutine). When they hit ESC we want to make changes and return to the previous screen.*

In Eclipse we have a subroutine called CONFIRM.ABORT that will return a 1 if the user confirms, or a 0 if the user does not confirm. We use this whenever handling the F12 key. For example we would use it like this:

```
IF F12 THEN
    CONFIRM.ABORT ANS
    IF ANS = 0 THEN GOTO IN.AUTOMAKE
    IF ANS = 1 THEN GOTO FINISH
END
```

Since a F12 key strike sets both QUIT and F12 to true we need to check the F12 value first or we will never be able to differentiate which key strike the user made.

## UPDATE

For the next exercise we are going to write our own UPDATE subroutine.

Background: At Eclipse we are dealing with many users accessing the same records on a frequent basis. They both might want to change values in the same field at the same time. For this reason we need to write our UPDATE subroutine so that if we have user collisions we inform the user that an error has occurred and their record changes were not made.

*\*For example if we have two people, person A and person B. Person A comes along and reads record number 44 at exactly 10:00 AM. At 10:01 AM person B looks at the same number 44 record. Person A decides to change the MAKE and START.YEAR fields and UPDATES the record at 10:05 AM. Person B decides to make changes to the START.YEAR and END.YEAR at 10:06. We can see that if we are not careful the changes that person A made may be wiped out.*

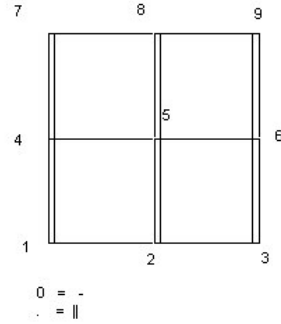
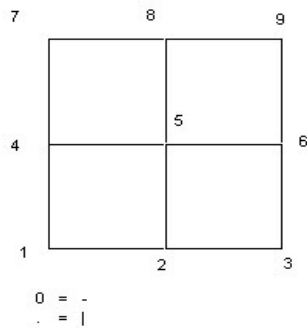
The five key ingredients that you will need to successfully complete this exercise are listed below:

- ❑ *Old Record - a copy of the record when we made the read*
- ❑ *New Record – a record that has the changes we made*
- ❑ *Current record – a current locked copy of the record in question*
- ❑ *A file handle*
- ❑ *A record ID*

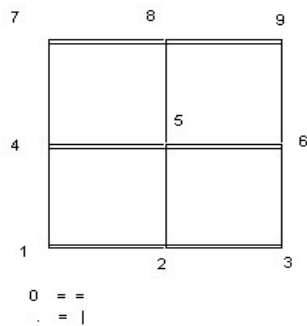
To make this easier break it down into steps. The first step is to read and lock a current copy of the record. The next step is to compare the current record to the old record. Is it the same? The next step would be to compare the current record to the new record. Is it different? The final step is to either write the new record on the current record or tell the user that there was an error and their changes were not made.

- Write an Update subroutine for the *INI.MANUAL.TEST* file.

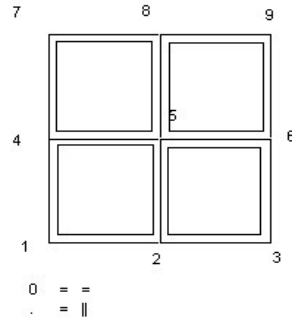
While holding down the ctrl key



While holding down the alt key



While holding down both the ctrl and alt keys



## Keyboard commands in the Screen Design

### F. Hotkeys!!!

We have noticed that on every screen in the Eclipse system we see hotkeys. Hotkeys are executed by holding down the ALT key and then hitting the highlighted character. What goes on behind the scenes? Well you are about to learn. Building hotkeys starts in the Screen Designer. It is uniform that all hotkeys are located at the bottom of the screen separated by a single horizontal. It is also uniform to have a space at both the front and back of the hotkey word. For example:

Exercise 8.

Include this at the bottom of your *INI.MANUAL.TEST* screen. Program them to load to proper places inside the subroutine. For example we do not want to allow the user to DELETE when they have not yet selected a record.

NEW	COPY	DELETE	ADDL
-----	------	--------	------

=====



The next part of programming a hotkey resides inside of our subroutine. We must find the proper location to initiate the MENU.LOAD command. We will need to perform a MENU.LOAD for each of the hotkeys that we have designed. The MENU.LOAD has five possible passes listed below:

- 1) HOR - This is the starting horizontal position of the hotkey word
- 2) VER - This is the starting vertical position of the hotkey word
- 3) ILEN - This is the length of the hotkey word
- 4) HPOS - This is the location of the character in the word that we wish to highlight.
- 5) HKEY- This is the actual character that is going to be highlighted

Here is an example of what the command would look like for the NEW hotkey listed above. MENU.LOAD 3,15,3,1,'N'

The next thing that we need to do is program the code for what happens when the hotkey is executed. We are going to call these hotkeys just like we would call any other internal subroutine, with a slight twist.

*\*It is very important to know the order in which these hotkeys were loaded. That order must be used in the SUBS: line described below.*

A SUBS: subroutine (line) must be created to act a distribution agent when a hotkey is used. When the user holds the ALT key and character the computer records this entry into a common variable named OPTION. Depending on the order in which the hotkeys were loaded determines the number that is passed into the OPTION variable. OPTION will then be used to direct us to the proper subroutine listed in our SUBS: line. Example below:

```
*-----*
SUBS:  ON OPTION GOTO NEW.REC,COPY.REC,DEL.REC,ADDL
*-----*
```

## GOTO vs. GOSUB

An important concept that programmers at Eclipse need to understand is the difference between the GOTO statement and the GOSUB statement. The GOTO statement does not require a RETURN statement whereas the GOSUB statement does require a RETURN statement. A good rule of thumb is “for every GOSUB there must be a return. When we use the GOSUB we are creating a stack. When we perform stacks it is important to unstack in the same order. If we fail to use these stacks correctly some interesting/bad things are going to happen to our subroutines. GOTO statements do not require a RETURN. A GOTO statement does not create a stack.

The RETURN statement by itself will send us back to the location that the GOSUB was called from. If we use the RETURN TO \_\_\_\_\_ statement then we will be returned to the specified location. Most importantly, with either of these RETURN's we have correctly returned to the previous stack.

# ***Lesson 9***

## ***More Eclipse Programming: VSCROLLS, INP's, and Mouse Event Handling***



### ***Objectives***

**In this lesson, you will learn about:**

- VSCROLLS
- INPWP, INP.WINDOW, INPWP.WINDOW, INP.MULTI
- Mouse event handling

### ***Chapter 9***

## A. VSCROLLS

### Overview:

In Eclipse we will use a VSCROLL whenever we want to display a list of records that we want the user to be able to select from. Like any other screen in the Eclipse system the VSCROLL building must begin in the Screen Designer. Build a screen that matches Eclipse uniform screen design. In this screen we will need a blank space in the center that is roughly 50 characters wide, and 12 lines long. We will call this screen (subroutine) *ini.auto.vscroll*.

### VSCROLL.DEFINE

This is the second step in our VSCROLL design. The VSCROLL.DEFINE has six possible passers listed below:

1. VIEW – A unique identifier by which this VSCROLL can be called
2. COL – The upper left column that the desired VSCROLL starts at
3. ROW – The upper left row that the desired VSCROLL starts at
4. WIDTH – The number of characters wide the VSCROLL region will be
5. ROWS – The maximum rows visible to the screen at one time
6. Screen.ID – The name of the current screen

### VSCROLL.SET

Once we have defined the region with the VSCROLL.DEFINE we can then load it using the SCROLL.SET. The VSCROLL.SET command has only one passer, it is the VIEW identifier that we set in the VSCROLL.DEFINE.

### VCLR

The VCLR is a clear vscroll command. The VCLR will wipe out anything that is written in the specified vscroll region by placing a blank copy over the top of the current copy. The VCLR has only one passer, it is the VIEW identifier that we set in the VSCROLL.DEFINE.

### VPRINT

The VPRINT command works similarly to the PRINT @ statement. It requires a desired location and a string to print. When we want to print information to a VSCROLL we will be using a VPRINT statement to do so. The VPRINT statement requires three passers listed below:

1. COL – This is the starting column position
2. ROW – This is the starting row position
3. STRING – This is the string that is to be printed to the specified area

### INPV

The INPV is the method we use to get an input from the user inside of a VSCROLL. The INPV is very similar to the INP method of retrieving information. Keep in mind we keep track of where we are inside of a VSCROLL by noting which LINE we are in. It is a good idea to set up a variable at the beginning of your code. Many things we do inside of the VSCROLL will need to change the value of LINE. We will later discuss how to move from one line to another, tracking our movements so that we can change the value of LINE.

*\*Helpful Hint*

Note that when we are printing information to the screen or to the VSCROLL we often want to print out an exact amount of spaces. We can do this by using the “(L or R depending on justification)#Desired number of spaces”. For example if we wanted to print out a string named *STRING*, and we wanted it to fill up exactly 25 spaces on the screen we would write the line:

```
VPRINT 3,1,STRING                “L#25”
```

This will print out the value stored in *STRING* and fill in the remaining area with spaces. The L or the R indicate whether we want to left or right justify.

## **PARSEMOVE**

The PARSEMOVE is how we are going to handle movement within the VSCROLL. The PARSEMOVE is very similar to the MOVE+1 that we worked with for INP statements. The PARSEMOVE is different than the MOVE statement because the PARSEMOVE actually returns values to let the program know where the user moved. For this reason we need to set up a few variables when we load our subroutine. The first needed variable will be named LINE. The LINE variable will hold which line the user is on. The other variable needed is the COL variable. This variable will hold the column that the user is in. If we do proper coding we can always determine where the user has the cursor within the VSCROLL. The PARSEMOVE has eight passers (five of which are required) that are listed below:

1. COL – This IN/OUT variable takes in where the user was and puts out where the user has gone.
2. L – This IN/OUT variable takes in the line the user was in and passes the line they have moved to.
3. COLS – This is the number of columns in the VSCROLL
4. LS – This is the number of lines in the VSCROLL
5. The size of a scroll page if the user does a page-up page-down
6. DNOK – Set to yes if the user can move up or down from the line they are on
7. NEWOK – Set to yes is the user can move to a blank line
8. BORDERMOVE – Records the direction that the user tried to exit from.

We should create a subroutine names MOVEMENT that we will contain a F12 and QUIT check, a PARSEMOVE, and an ON COL. The ON COL will tell us to move to a certain INPV statement if the user moves in that direction. The ON COL statement should look something like this: ON COL GOTO MAKE, MODEL, DOORS, PASSENGERS, START.YEAR, END.YEAR.

## **B. INPWP, INP.WINDOW, INPWP.WINDOW, INPMULTI**

### **INPWP      INPUT WORD PROCESSOR**

The INPWP acts as an input field that allows multiline entries. For example if we have a record that has an address attribute we might want to enter more than one line. The INPWP allows us to do just that. Not only will it allow us to enter multiple lines but it also puts a VM (Value Mark) at every CRLF (Carriage Return Line Feed). For example if we were to type the following text into a INPWP

Address: Eclipse, Inc.  
1909 26<sup>th</sup> St.

we would be storing an attribute that looked like this

Eclipse, Inc.<sup>2</sup>1909 26<sup>th</sup> st.

*\*Note that the value mark was already inserted for us.*

The INPWP has eight possible passers listed below:

1. ITEM – The variable to assign the input to
2. HOR – This is the horizontal starting position of the input
3. VER – This is the vertical starting position of the input
4. WIDTH – The number of characters wide the input field will be
5. LENGTH – The number of lines visible to the user at one time
6. LINES – The maximum allowable lines that the user can type to
7. MOVES – The moves that the user can make inside of the INPWP zone
8. HELPSCR – The help screen that should display for this input field

The MOVES variable should be a length of four 0's or 1's. Remember the move path of (left,up,right,down), that is the path we are going to set. The first value is left, the second up and so on. The value in the positions is either going to be 0 to not allow, or a 1 to allow. For example the following MOVES pass will not allow up down moves, but will allow left right moves. StrMoves = "1010"

Note that we put this value into a string, then when we call the INPWP we will call the string. An example of code might be this:

StrMoves = "1010"

ADDRESS: INPWP ADDR5,10,8,35,2,50,StrMoves

## **INP.WINDOW**

The INP.WINDOW is another way in which we can bring in information from the user. It works just like the standard INP command except it calls its own window to display the (information/input) field. With the INP.WINDOW we can bring in multiple values into a single attribute, separated by value marks. Also with the INP.WINDOW we have the ability to set the read-only flag to true, if we only want to display information, not receive information from the user. A good place to use the INP.WINDOW command is when we want to create a list of items from the user. The INP.WINDOW could be used to read a list of product for an order. The attribute field would then be made of a set a value mark delimited ID's.

## **INPWP.WINDOW**

The INPWP.WINDOW is just like the INP.WINDOW command except it allows for multi-line entries in the new input window. This would then break down the storage to the next level and put Value Marks in the place of the CRLF within a single entry. We might use this if we wanted to gain a list of addresses from the user.

## INP.MULTI

The INP.MULTI is a call that will allow the programmer to bring up a new window that has already been designed with three hotkeys listed next: CLEAR LIST, SAVE, and RECALL. This window is designed to bring in a list items from the user. INP.MULTI has seven passers listed below:

- IDS – The variable that the input will be stored in.
- Max.len – The maximum length of the input
- Justify – Where the cursor will justified on the pop up screen
- Verify – Input verification
- CNV – Conversion format subroutine
- Title – The title of the popup window
- FLCode – The name of the file that you wish to store the data in
- READONLY – is set to yes if it only a read only

Remember that the variable that is used for the IDS must be defined before it is called because Eclipse treats this as a dynamic array. For example:

```
CARTEST = ""  
INP.MULTI CARTEST, 7
```

This subroutine has the ability to write the list off to a file, to do this the programmer must assign a value to the FLCode.

## B. Mouse Event Handling

Mouse event handling is designed to create a pseudo GUI that will allow the user to click on portions of the screen to initiate designated events(subroutines). This internal subroutine basically records the X and Y positions when the user makes a click with the mouse. We can then evaluate the X and Y values and launch subroutines on that basis. To get a quick copy of a generic mouse event handling, hold down the alt key and hit 1. You can then copy this code and paste this code at the bottom of your subroutine.

```
MOUSE$:  GET.MOUSE.POSITION X,Y  
         Y -= 1  
         BEGIN CASE  
         END CASE  
         RETURN
```

In order to handle the mouse click we at Eclipse use CASE statements, as seen on the copy you made from the alt-1 command. The two things that need to be checked in the case statement are the x and y positions, for example if we are programming for the user to click on the hotkeys on the screen shown below:

```
=====Mouse Testing=====
=
=
=
=
=
=
=
=
=
=
=
=
=
=
=
=
=
=
=
=====
= Eagle = Albatross = Birdie = Par = Bogie =
=====
```

If the word Eagle begins on line 18 and column 3 we would handle the case statement like this:

```
    CASE Y = 17 AND X < 7 AND X > 1
      GOTO EAGLE
```

EAGLE is a subroutine that prints the message 'One under Par' to the screen.