Chapter 1

Reader, Meet Eclipse; Eclipse, Meet the Reader

In This Chapter

- How I learned to love Eclipse
- How the Eclipse project is organized
- How Eclipse puts widgets on your screen

The little hamlet of Somerset, New Jersey, is home to an official Sun Microsystems sales office. Once a month, that office hosts a meeting of the world-renowned Central Jersey Java Users' Group.

At one month's meeting, group members were discussing their favorite Java development environments. "I prefer JBlipper," said one of the members. "My favorite is Javoorta Pro," said another. Then one fellow (Faizan was his name) turned to the group and said, "What about Eclipse? It's pretty sweet."

Of course, Faizan's remark touched off an argument. Everyone in the group is attached to his or her favorite Java development tools. "Does Javoorta do refactoring?" "Does JBlipper support Enterprise JavaBeans?" "Does Eclipse run on a Mac?" "How can you say that your development environment is better?" "And what about good old UNIX `vi`?"

Then someone asked Faizan to demonstrate Eclipse at the next users' group meeting. Faizan agreed, so I ended the discussion by suggesting that we go out for a beer. "I don't drink," said one of the group members. "I don't either," I said. So we went out for pizza.

At the next meeting, Faizan demonstrated the Eclipse development environment. After Faizan's presentation, peoples' objections to Eclipse were more muted. "Are you sure that Eclipse runs well under Linux?" "Can you really extend Eclipse so easily?" "How does the open source community create such good software for free?"

A few months later, I ran into a group member at a local Linux conference. "Does Javoorta Pro run under Linux?" I asked. "I don't use Javoorta Pro anymore. I've switched to Eclipse," he said. "That's interesting," I said. "Hey, let's go out for a beer."

An Integrated Development Environment

An *integrated development environment* (IDE) is an all-in-one tool for writing, editing, compiling, and running computer programs. And Eclipse is an excellent integrated development environment. In a sense, that's all ye need to know.

Of course, what you absolutely need to know and what's good for you to know may be two different things. You can learn all kinds of things about Java and Eclipse, and still benefit by learning more. So with that in mind, I've put together this chapter full of facts. I call it my "useful things to know about Eclipse" (my "uttkaE") chapter.

A Little Bit of History (Not Too Much)

In November 2001, IBM released $40 million worth of software tools into the public domain. Starting with this collection of tools, several organizations created a consortium of IDE providers. They called this consortium the Eclipse Foundation, Inc. Eclipse was to be "a universal tool platform — an open extensible IDE for anything and nothing in particular."* (I know, it sounds a little like Seinfeld's "nothing." But don't be lead astray. Eclipse and Seinfeld have very little in common.)

This talk about "anything and nothing in particular" reflects Eclipse's ingenious plug-in architecture. At its heart, Eclipse isn't really a Java development environment. Eclipse is just a vessel — a holder for a bunch of add-ons that form a kick-butt Java, C++, or even a COBOL development environment. Each add-on is called a *plug-in,* and the Eclipse that you normally use is composed of more than 80 useful plug-ins.

While the Eclipse Foundation was shifting into high gear, several other things were happening in the world of integrated development environments. IBM was building WebSphere Studio Application Developer (WSAD) — a big Java development environment based on Eclipse. And Sun Microsystems was

**Quoted from the eclipse.org Web site:* `www.eclipse.org`.

promoting NetBeans. Like Eclipse, NetBeans is a set of building blocks for creating Java development environments. But unlike Eclipse, NetBeans is pure Java. So a few years ago, war broke out between Eclipse people and NetBeans people. And the war continues to this day.

In 2004, the Eclipse Foundation turned itself from an industry consortium to an independent not-for-profit organization. Among other things, this meant having an Executive Director — Mike Milinkovich, formerly of Oracle Corporation. Apparently, Milinkovich is the Eclipse Foundation's only paid employee. Everybody else donates his or her time to create Eclipse — the world's most popular Java development environment.

The Grand Scheme of Things in Eclipse

The Eclipse Foundation divides its work into projects and subprojects. The projects you may hear about the most are the Eclipse project, the Eclipse Tools project, and the Eclipse Technology project.

Sure, these project names can be confusing. The "Eclipse project" is only one part of the Eclipse Foundation's work, and the "Eclipse project" is different from the "Eclipse Tools project." But bear with me. This section gives you some background on all these different projects.

And why would you ever want to know about the Eclipse Foundation's projects? Why should I bother you with details about the Foundation's administrative organization? Well, when you read about the Foundation's projects, you get a sense of the way the Eclipse software is organized. You have a better understanding of where you are and what you're doing when you use Eclipse.

The Eclipse project

The *Eclipse project* is the Eclipse Foundation's major backbone. This big Eclipse project has three subprojects — the Platform subproject, the Java Development Tools subproject, and the Plug-in Development subproject.

The Platform subproject

The *Platform subproject* deals with things that are common to all aspects of Eclipse — things such as text editing, searching, help pages, debugging, and versioning.

At the very center of the Platform subproject is the platform *core.* The core consists of the barebones necessities — the code for starting and running Eclipse, the creation and management of plug-ins, and the management of other basic program resources.

In addition, the Platform subproject defines the general look and feel of Eclipse's user interface. This user interface is based on two technologies — one that's controversial, and another that's not so controversial. The controversial technology is called SWT — the *Standard Widget Toolkit.* The not-so-controversial technology is called *JFace.*

- The Standard Widget Toolkit is a collection of basic graphical interface classes and methods, including things such as buttons, menus, labels, and events.

  For more chitchat about the Standard Widget Toolkit (and to find out why the Toolkit is so controversial), see the second half of this chapter.

- JFace is a set of higher-level graphical interface tools, including things such as wizards, viewers, and text formatters. JFace builds on the work that the Standard Widget Toolkit starts.

The Java Development Tools (JDT) subproject

The word "Java" appears more than 700 times in this book. (Yes, I counted.) In many people's minds, Eclipse is nothing more than an integrated development environment for Java. Heck, if you start running Eclipse you see the Java perspective, Java projects, Java search tools, and a bunch of other Java-specific things.

But Java is only part of the Eclipse picture. In reality, Eclipse is a language-neutral platform that happens to house a mature Java development environment. That Java development environment is a separate subproject. It's called the *Java Development Tools* (JDT) subproject. The subproject includes things like the Java compiler, Java editor enhancements, an integrated debugger, and more.

When Eclipse documentation refers to the "core," it can be referring to a number of different things. The Platform subproject has a core, and the JDT subproject has a core of its own. Before you jump to one core or another in search of information, check to see what the word "core" means in context.

The Plug-in Development Environment (PDE) subproject

Eclipse is very modular. Eclipse is nothing but a bony frame on which dozens of plug-ins have been added. Each plug-in creates a bit of functionality, and together the plug-ins make a very rich integrated development environment.

But wait! A plug-in is a piece of code, and the people who create plug-ins use development environments, too. For these plug-in creators, Eclipse is both a tool and a target. These people use Eclipse in order to write plug-ins for Eclipse.

So wouldn't it be nice to have some specialized tools for creating Eclipse plug-ins? That way, a programmer can seamlessly use Eclipse while writing code for Eclipse.

Well, whadaya' know? Someone's already thought up this specialized tools idea. They call it PDE — the *Plug-in Development Environment* — and they have an entire subproject devoted to this Plug-in Development Environment.

The Eclipse Tools project

Compared with the main Eclipse project, the *Eclipse Tools* project houses subprojects that are a bit farther from Eclipse's center. Here are some examples.

The Visual Editor subproject

If you're familiar with products like Visual Basic, then you've seen some handy drag-and-drop tools. With these tools you drag buttons, text fields, and other goodies from a palette onto a user form. To create an application's user interface, you don't describe the interface with cryptic code. Instead you draw the interface with your mouse.

In Eclipse 3.0, these drag-and-drop capabilities still aren't integrated into the main Eclipse bundle. Instead, they're a separate download. They're housed in the *Visual Editor* (VE) — a subproject of the Eclipse Tools Project.

The CDT and COBOL IDE subprojects

The *C/C++ Development Tools* (CDT) subproject develops an IDE for the C/C++ family of languages. So after downloading a plug-in, you can use Eclipse to write C++ programs.

As if the CDT isn't far enough from Java, the *COBOL IDE* subproject has its own Eclipse-based integrated development environment. (COBOL programs don't look anything like Java programs. After using Eclipse for a few years to develop Java programs, I feel really strange staring at a COBOL program in Eclipse's editor.)

The UML2 subproject

The *Unified Modeling Language* (UML) is a very popular methodology for modeling software processes. With UML diagrams, you can plan a large programming endeavor, and work your way thoughtfully from the plan to the actual code. The tricks for any integrated development environment are to help you create models, and to provide automated pathways between the models and the code. That's what *UML2* (another subproject of the Eclipse Tools project) is all about.

The Eclipse Technology project

The *Eclipse Technology project* is all about outreach — helping the rest of the world become involved in Eclipse and its endeavors. The Technology project

fosters research, educates the masses, and acts as a home for ideas that are on their way to becoming major subprojects.

As of 2004, this project's emerging technologies include *Voice Tools* — tools to work effectively with speech recognition, pronunciation, and the control of voice-driven user interfaces.

Another cool item in the Eclipse Technology project is *AspectJ.* The name AspectJ comes from two terms — aspect-oriented programming and Java. In AspectJ, you can connect similar parts of a programming project even though the parts live in separate regions of your code. AspectJ is an up-and-coming extension to the Java programming language.

What's the Best Way to Create a Window?

According to Sun Microsystems, Java is a "Write Once, Run Anywhere" programming language. This means that a Java program written on a Macintosh runs effortlessly on a Microsoft Windows or UNIX computer. That's fine for programs that deal exclusively with text, but what about windows, buttons, text fields, and all that good stuff? When it comes to using graphical interfaces, the "Write Once, Run Anywhere" philosophy comes up against some serious obstacles.

Each operating system (Windows, UNIX, or whatever) has its own idiosyncratic way of creating graphical components. A call to select text in one operating system's text field may not work at all on another operating system's text field. And when you try to translate one operating system's calls to another operating system's calls, you run into trouble. There's no good English translation for the Yiddish word *schlemiel,* and there's no good Linux translation for Microsoft's object linking and embedding calls.

When Java was first created, it came with only one set of graphical interface classes. This set of classes is called the *Abstract Windowing Toolkit* (AWT). With the AWT, you can create windows, buttons, text fields, and other nice looking things. Like any of Java's "Write Once, Run Anywhere" libraries, the AWT runs on any operating system. But the AWT has an awkward relationship with each operating system's code.

The AWT uses something called *peers.* You don't have to know exactly how peers work. All you have to know is that a peer is an extra layer of code. It's an extra layer between the AWT and a particular operating system's graphical

interface code. On one computer, a peer lives between the AWT code and the UNIX code. On another computer, the peer lives between the AWT code and the Microsoft Windows code.

The AWT with its peer architecture has at least one big disadvantage. The AWT can't do anything that's not similar across all operating systems. If two operating systems do radically different things to display trees, then the AWT simply cannot display trees. Each of the AWT's capabilities belongs to the least common denominator — the set of things that every popular operating system can do.

Here comes Swing

Somewhere along the way, the people at Sun Microsystems agreed that the AWT isn't an ideal graphical interface library. So they created *Swing* — a newer alternative that doesn't rely on peers. In fact, Swing relies on almost nothing.

With the AWT, you write code that says "Microsoft Windows, please display a button for me." But with Swing you don't do this. With Swing you say "draw some lines, then fill in a rectangle, then put some text in the rectangle." Eventually you have all the lines and colors that make up a button. But Microsoft Windows doesn't know (or care) that you've drawn a button.

To use the official slogan, Swing is "pure Java." Swing draws everything on your screen from scratch. Sure, a Swing button may look like a UNIX button, a Macintosh button, or a Microsoft Windows button. But that's just because the Swing developers work hard to replicate each operating system's look and feel.

Here's the problem with Swing: Drawing windows and buttons from scratch can be very time consuming. In my experience, Swing applications tend to run slowly.* That's why people who develop Eclipse plug-ins don't use Java's Swing classes. Instead, they use classes from the Standard Widget Toolkit (SWT).

The Standard Widget Toolkit

The word "widget" comes from the play "Beggar on Horseback," written in the early 1920s by George Kaufman and Marc Connelly. (I first heard of widgets when I saw the 1963 James Garner movie *The Wheeler Dealers*.) In ordinary usage, the word "widget" means a vaguely described gadget —

**My friends at Sun Microsystems claim that Swing applications are lightning fast, but I can't tackle that debate in this book.*

a hypothetical product whose use and design is unimportant compared to its marketing potential.

In computing, the word "widget" represents a component in a graphical user interface — a button, a text field, a window, or whatever. That's why a group of developers coined the phrase *Standard Widget Toolkit* (SWT). These developers were people from Object Technology International and IBM. At first they created widgets for the language SmallTalk. Later they moved from SmallTalk to Java.

In contrast to Swing, Eclipse's SWT is very fast and efficient. When I run Eclipse under Linux, I don't wait and watch as my buttons appear on the screen. My SWT buttons appear very quickly — as quickly as my plain, old Linux buttons.

To achieve this speed, the SWT ignores Java's "Write Once, Run Anywhere" philosophy. Like the AWT, the SWT isn't pure Java. But unlike the AWT, the SWT has no peer layer.

The SWT isn't nearly as portable as Swing's pure Java code, and this lack of portability drives the "pure Java" advocates crazy. So the big debate is between Swing and the SWT. Sun's NetBeans IDE calls Swing classes to display its dialogs and editors, and Eclipse calls SWT classes. This difference between NetBeans and Eclipse has several important consequences.

- **Eclipse runs noticeably faster than NetBeans (unless you run NetBeans on a very powerful computer).**
- **Eclipse's graphical interface isn't merely an imitation of your computer's interface.**

  The button on a NetBeans panel may look like a Linux button or like a Microsoft Windows button, but it's not really one of these buttons. A NetBeans button is a drawing that's made to look like a Microsoft Windows button.

  In contrast, the button on an Eclipse panel is the real McCoy. When you run Eclipse on a Macintosh, you see real Macintosh buttons. When you run Eclipse in Windows, you see Bill Gates's own buttons.

  Do you want real buttons or simulated buttons? Believe it or not, you can see the difference.

- **Eclipse can use tools that are specific to each operating system.**

  If you run Eclipse under Microsoft Windows, you can take advantage of the functionality provided by Windows ActiveX components. But if you run Eclipse under Linux, then you can't use ActiveX components. That's why certain features of the Eclipse IDE are available in Windows, but not in Linux.

In stark contrast to the situation with Eclipse, NetBeans doesn't use ActiveX components. (Even on a computer that runs Microsoft Windows, NetBeans doesn't take advantage of any ActiveX functionality.)

- **In theory, Eclipse isn't as portable as NetBeans.**

  At `www.eclipse.org` you can download versions of Eclipse for Microsoft Windows, Linux, Solaris, QNX, UNIX, and Mac OS X. But if someone creates the MyNewOS operating system, then the NetBeans/Swing camp has a slight advantage over the Eclipse/SWT people.

All in all, I prefer Eclipse to NetBeans. And I'm not saying this only because I have a contract to write *Eclipse For Dummies.* For my money, the Eclipse development environment is simply a better tool than NetBeans.

Relax and Enjoy the Ride

As an Eclipse user, you wade through lots of written material about the SWT. That's why you want to know about the "SWT versus Swing" issue. But "knowing" doesn't mean "worrying." The war between the SWT and Swing has the greatest impact on people who write code for the Eclipse Foundation. The "SWT versus Swing" controversy comes alive when you try to enhance the Eclipse development environment. But as a plain, old Eclipse user, you can just sit back and watch other people argue.

Using Eclipse, you can write Swing, SWT, AWT, and text-based applications. You can just go about your business and write whatever Java code you're accustomed to writing. So don't be upset by this chapter's "SWT versus Swing" harangue. Just remember some of the issues whenever you read other peoples' stories about Eclipse.