Integrated Development Environment and Eclipse

Instructor: Yongjie Zheng

February 11, 2016

CS 490MT/5555
Software Methods and Tools

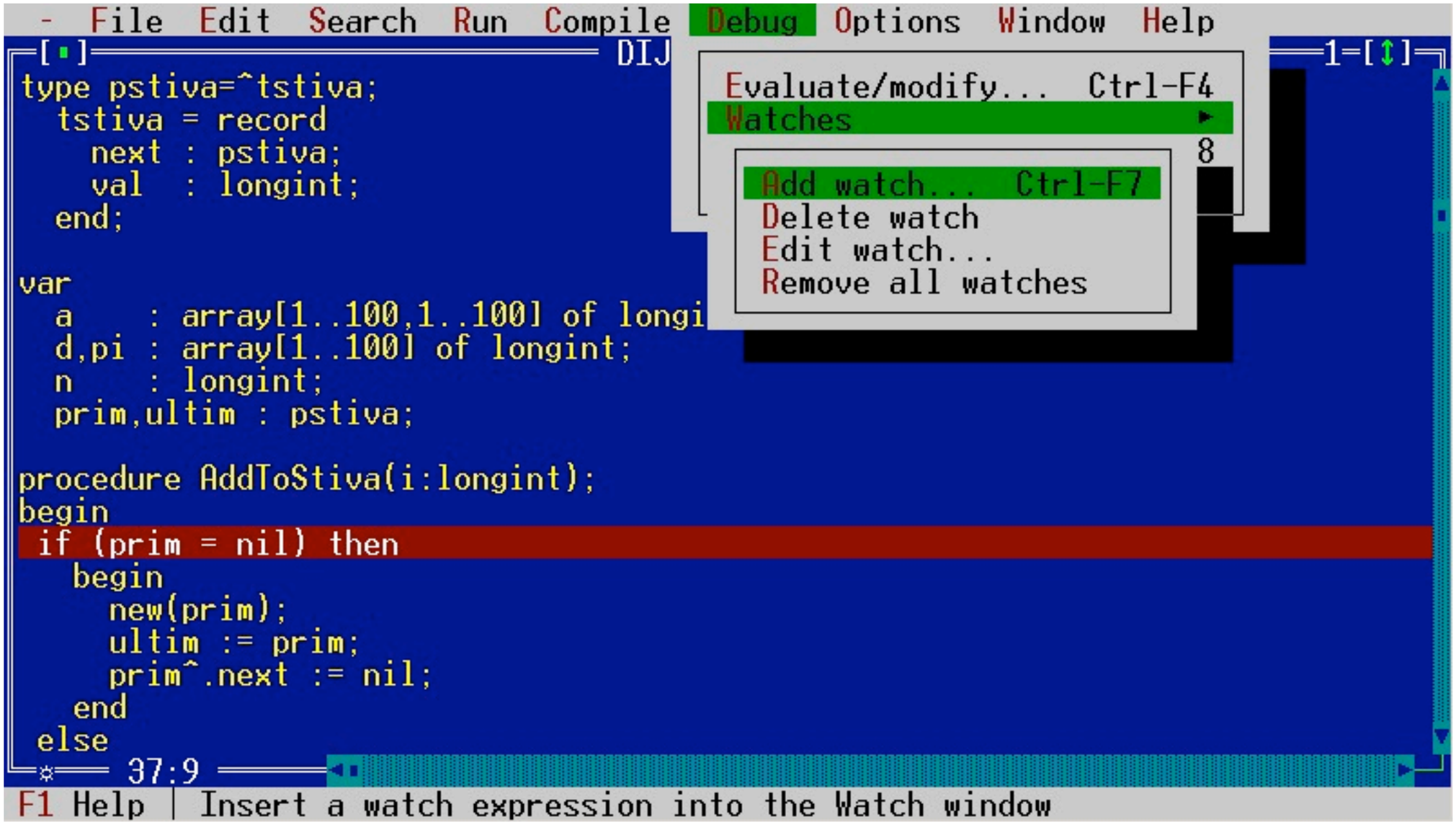
Integrated Development Environment (IDE)

- An IDE is a software application that provides comprehensive facilities to computer programmers for software development. [Wikipedia]
- Typically, an IDE consists of
  - A source code editor
  - A debugger
  - A compiler/interpreter
  - Other build automation tools

Examples of IDE

- Turbo Pascal (Borland)
- Eclipse (open-source, from IBM)
- Netbeans (open-source, from Sun/Oracle)
- Microsoft Visual Studio
- Xcode (Apple)
- Emacs (open-source, from the GNU project)
- ...

A screenshot of Turbo Pascal from online



Considerations when choosing an IDE

- Programming languages
- Operating systems
- Learning curve
- Extensibility
- Open-source
- Expenses
- Organizational restrictions
- Personal preference

Eclipse

Eclipse is a free, open, and extensible IDE that is widely used to develop applications in Java and other programming languages including C, C++, and Perl.

History of Eclipse

- Developed by the Object Technology International (OTI) group of IBM.
- Based on VisualAge Java that was written in Smalltalk.
- Eclipse is written in Java.
- First released in 2001.
- The Eclipse foundation was created in 2004.
- Eclipse 3.0 was released in 2004. This was the first version of Eclipse based the OSGi framework.
- Currently, a new version is released annually.

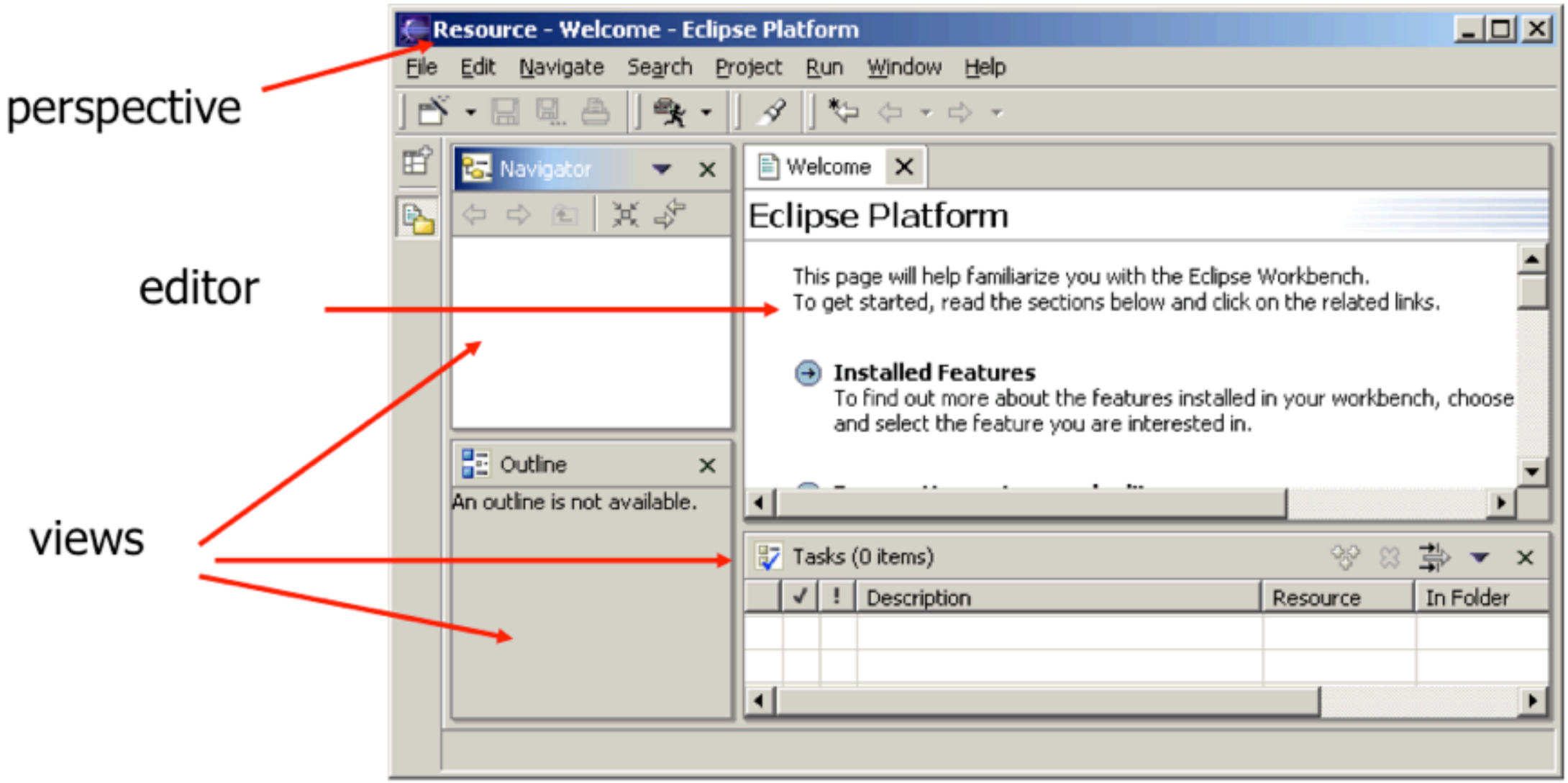
Download and Installation

- Eclipse does not include the Java Runtime Environment (JRE) and Java Development Kit (JDK)
  - To check if JRE is installed on your machine, run: java -version
  - JDK (which includes JRE) can be downloaded from http://www.oracle.com/technetwork/java/javase/downloads/index.html
- Getting Eclipse
  - http://www.eclipse.org/downloads/
  - Please download and install Eclipse Standard.
  - The current version is 4.5 (i.e. Mars).

Eclipse Workbench

- Eclipse Workbench is the desktop development environment in which all of your work will be done.
- The workbench consists of a main menu bar, toolbar, and a number of tiled panes known as *views* and e*ditors*.
- The welcome *view* opens automatically the first time that Eclipse is launched.
- Each workbench window contains one or more *perspectives*.

Perspectives, Views, and Editors



Perspectives

- Perspective: the combination of various views and editors visible within the Eclipse workbench.
- A perspective can be thought of as one page within the Eclipse workbench.
- A perspective defines the initial set and layout of views in the Workbench window.
- Perspectives control what appears in certain menus and toolbars.
- Examples: Java perspective, resource perspective, debug perspective, etc.

Views and Editors

- Views: used to navigate resources and modify properties of a resource.
  - Any changes made within a view are saved immediately.
  - Only a single instance of a view can be open in a given perspective.
- Editors: used to view or modify a specify resource.
  - Editors follow the open-save-close model.
  - Any number of editors of the same type can be open at one time.

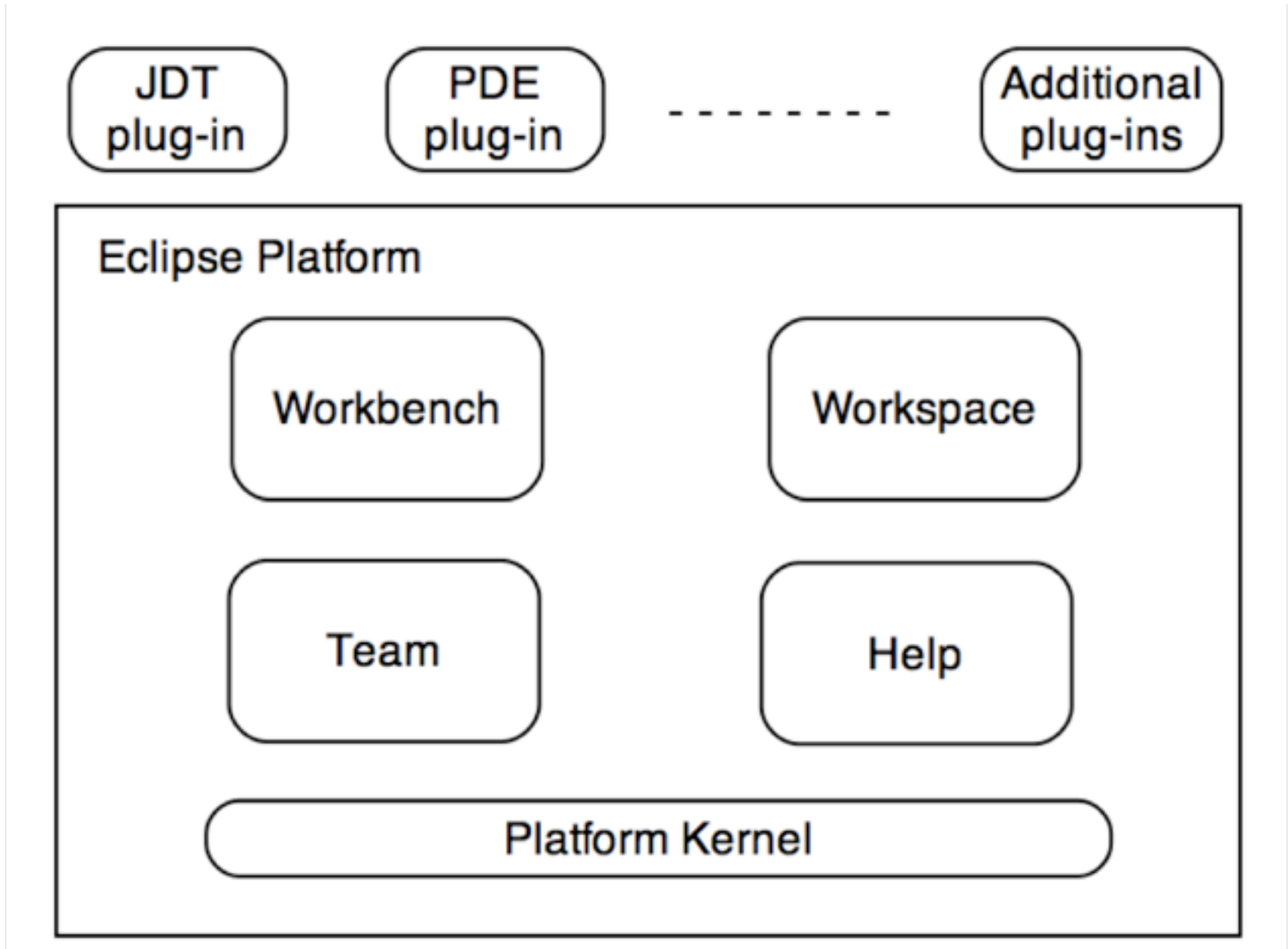
Basic Operations

- Setting up the environment
  - Customizing the current perspective
  - Customizing the Eclipse environment
- Creating a project
  - A general project
  - A Java project
  - An Eclipse Plug-in project
- Searching
- Navigating
- Writing code, running applications, debugging, ...

Eclipse as a Tool Integration Environment

- Eclipse is tool integration platform that includes
  - Kernel
  - Workbench
  - Workspace
  - Team
  - Help
- Eclipse Software Development Kit (SDK) ships with two additional plug-ins
  - Java Development Tools (JDT)
  - Plug-in Development Environment (PDE)
- Eclipse RCP (Rich Client Platform) is a subset of the Eclipse platform that primarily consists of kernel and workbench.

Eclipse Infrastructure



More about Eclipse Infrastructure

- In essence, Eclipse is a small kernel surrounded by hundreds (and potentially thousands) of *plug-ins*.
- The kernel (runtime system) is based on *Equinox*, an implementation of the *OSGi* specification.
- Even the entire Eclipse user interface is built in the form of plug-ins.
  - Based on the Standard Widget Toolkit (SWT) (instead of Java AWT or SWING).
  - JFace viewers are used to present object-oriented data.

Why do we need OSGi?

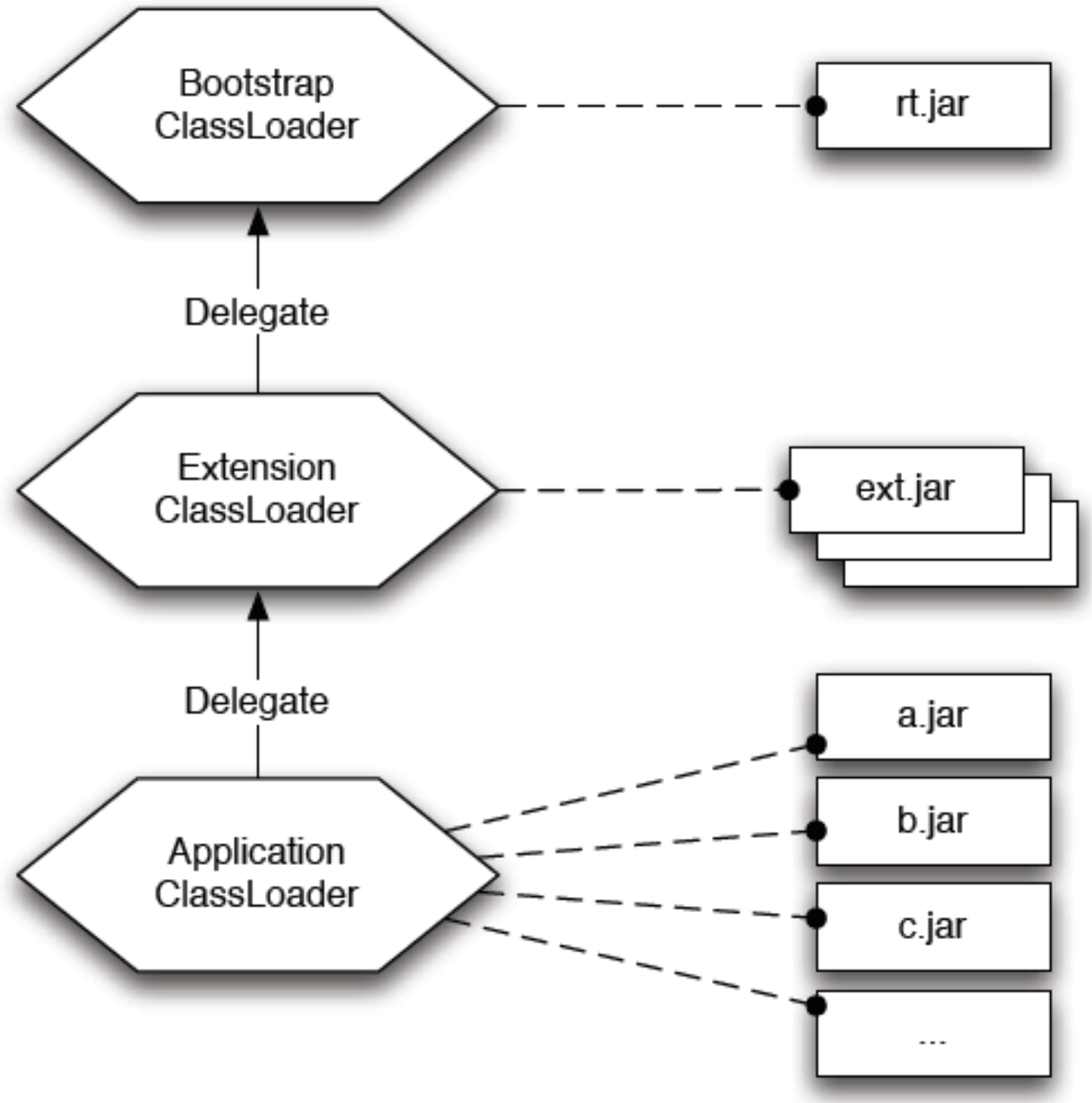
- To build a Java application as complex as Eclipse, modularity is very important.
  - Modularity: encapsulation, highly cohesive, loosely coupled.
- Java's existing mechanisms are not sufficient to build modular applications.
- OSGi is a module system based on which we can build modular applications (e.g. Eclipse) in Java.

Java's current support of modularity

- A large or enterprise Java application is often deployed as a number of JAR files, and each JAR contains a list of class files.
  - Is a JAR file really a module?
- All the JAR files share the same classpath and are dumped into a flat list when JVM runs.
  - E.g. java -classpath a.jar;b.jar;c.jar HelloWorld
- Major limitations of JARs
  - **Mutual dependency**, **Version Information**, **Information Hiding**.

Java's Class Loading Procedure

- Parent-first delegation (tree-based class loading)
  - Bootstrap class loader: loading all the classes in the base JRE library, e.g. everything with a package name beginning with java, javax, etc.
  - Extension class loader: loading classes from the "extension" libraries.
  - Application class loader: loading from the "classpath".



The standard Java class loader hierarchy.

Specific steps of loading a class (e.g. java -classpath log4j.jar;classes org.example.HelloWorld)

1. The JRE asks the application class loader to load a class.
2. The application class loader asks the extension class loader to load the class.
3. The extension class loader asks the bootstrap class loader to load the class.
4. The bootstrap class loader fails to find the class, so the extension class loader tries to find it.
5. The extension class loader fails to find the class, so the application class loader tries to find it, looking first in log4j.jar.
6. The class is not in log4j.jar so the class loader looks in the classes directory.
7. The class is found and loaded, which may trigger the loading of further classes — for each of these we go back to step 1.

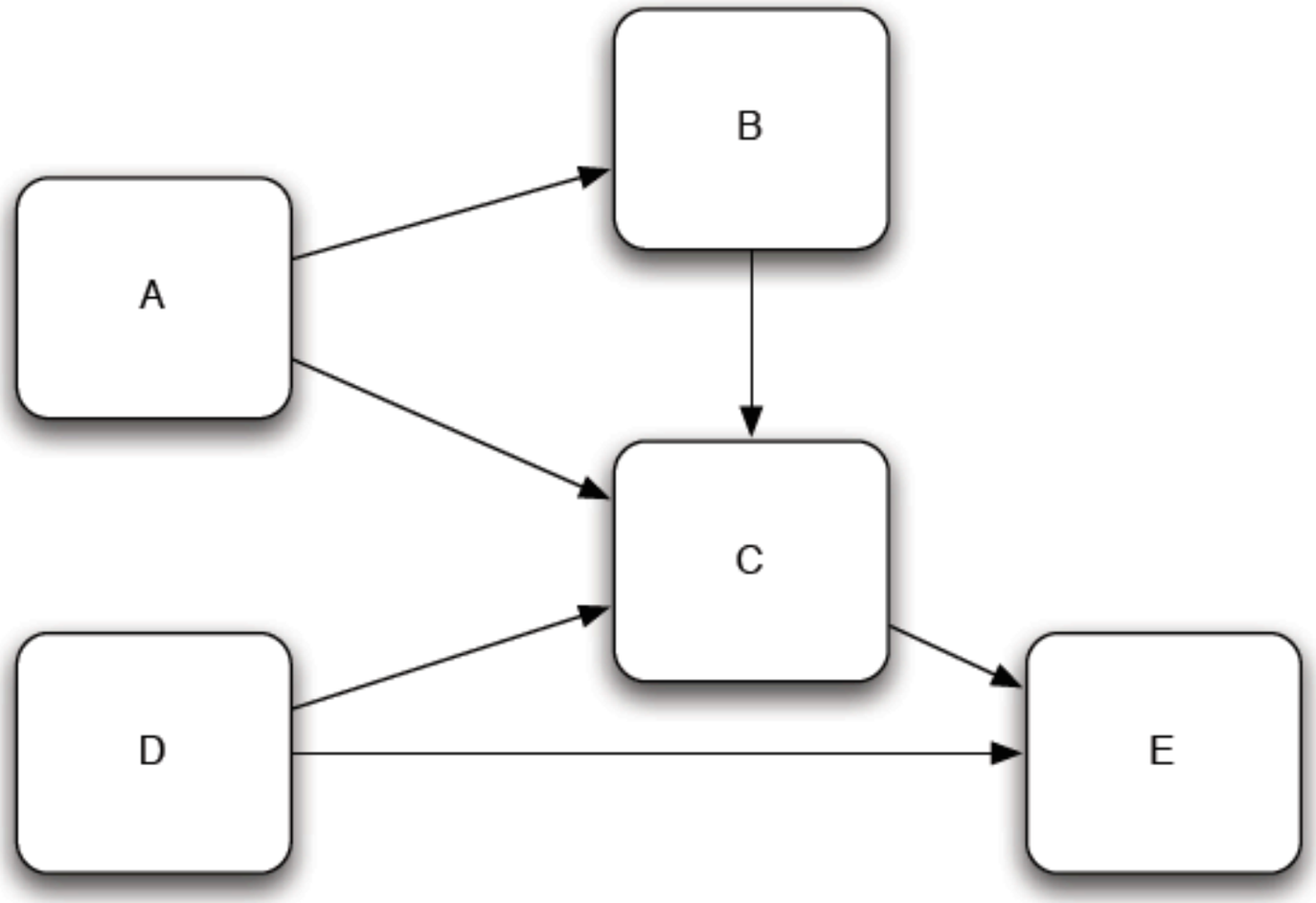
Potential Problems

- Lack of explicit dependency: one JAR may depend on the functionality provided by other JARs
- Lack of version information: multiple versions of the same JAR file may exist.
- Lack of information hiding: everything in a JAR is completely visible to all other JARs.
    - Conflicting classes: two classes of the same name may exist in two different JAR files.

The *JAR Hell* Problem

Example: java -classpath a.jar;b.jar;c.jar HelloWorld

Library A (in a.jar) works with Version 2 of library B (in b.jar), but library C (in c.jar) can only work with Version 3 of B . In a standard Java application, A and C must use the same version of B.



The OSGi class loader graph

Open Services Gateway initiative (OSGi)

- Each module has its own classpath.
- In OSGi, modules are referred to as bundles.
- Physically, a bundle is just JAR file. In addition, it contains a file named MANIFEST.MF that specifies: the *name* of the bundle, the *version* of the bundle, the list of *imports* and *exports*, etc.
- The OSGi framework will take responsibility for matching up the import with a matching export.
- The class loading procedure follows a **graph**, instead of a **tree**.

OSGi, cont.

OSGi hides everything in a JAR (bundle) unless explicitly exported. A bundle that wants to use another JAR must explicitly import the parts it needs. By default, there is no sharing.

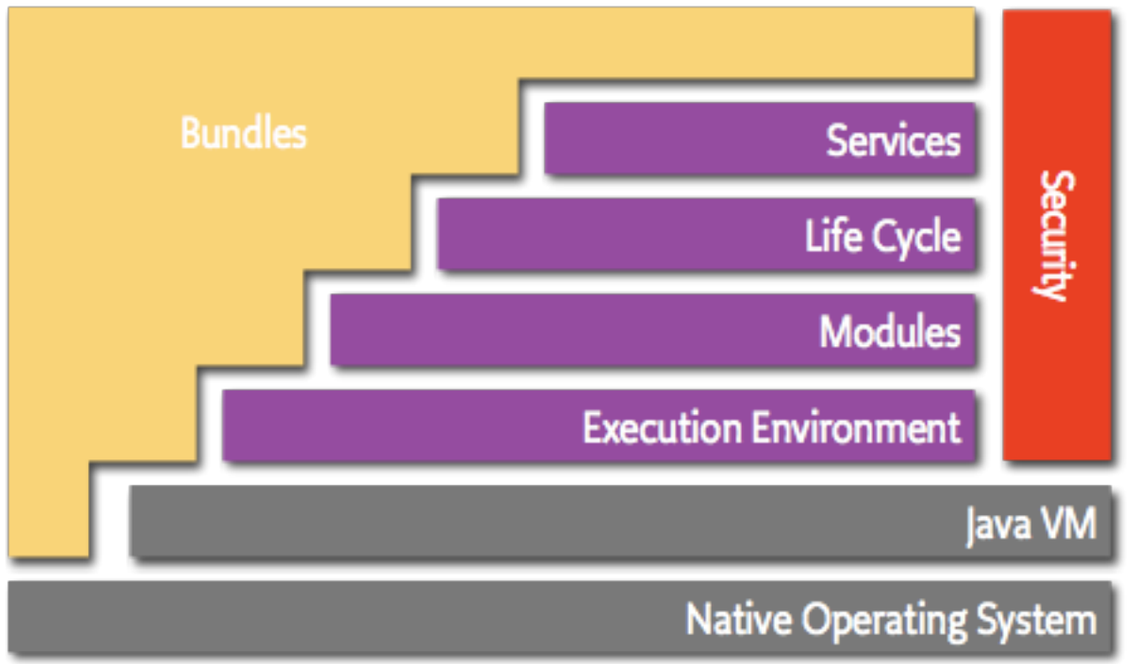
OSGi, cont.

One of the key advantages of OSGi is its **class loader model**, which uses the metadata in the manifest file. OSGi does not have a global class path. When **bundles** are installed into the OSGi Framework, the metadata is processed by the module layer and the declared **external dependencies** are reconciled against the **versioned** exports declared by other installed modules. The OSGi Framework determines the dependencies based on the manifest, and calculates the independent required class path for each bundle. This approach resolves the shortcomings of plain Java class loading by ensuring that the following requirements are met:

- Only packages explicitly exported by a particular bundle, through the metadata, are visible to other bundles for import.
- Each package can be resolved to specific versions.
- Multiple versions of a package can be available concurrently to different clients

From: http://pic.dhe.ibm.com/infocenter/rsahelp/v8/index.jsp?topic=/com.ibm.osgi.common.doc/topics/cosgiarchitecture.html

OSGi, cont.



The OSGi architecture from http://www.osgi.org/Technology/WhatIsOSGi

Reference

- Clayberg, Eric, and Dan Rubel. Eclipse Plug-ins. Addison-Wesley Professional, ISBN-13: 978-0321553461, 2009.
- OSGi In Practice by Neil Bartlett. - most of the materials about OSGi of this lecture are from the first chapter of this book.