Resource Encoding For Java Modules

Draft

14 Pages

Text in Red is here to help you. Delete it when you have followed the instructions.
The <RFC Title> can be set from the File>Properties:User Defined menu. To update it onscreen, press F9. To update all of the fields in the document Select All (CTRL-A), then hit F9. Set the release level by selecting one from: Draft, Final Draft, Release. The date is set automatically when the document is saved.

Abstract

Since Java 9 the fixed set of system packages was replaced by a mutable set of JPMS modules. This set can be defined at build time via static linking or introspected at runtime through reflection. This opens opportunities for OSGi to customize the JRE and to represent accurately the JRE within the OSGi runtime. This also means some application logic may come as JPMS modules not under the control of the OSGi developer. This RFC describes the requirements needed to represent the JPMS modules are OSGi Resources. This will make the OSGi technology applicable to them. One prominent use case is the application of the OSGi Resolver in the construction of complete OSGi plus JPMS runtimes.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

The OSGi Alliance hereby grants you a limited copyright license to copy and display this document (the “Distribution”) in any medium without fee or royalty. This Distribution license is exclusively for the purpose of reviewing and providing feedback to the OSGi Alliance. You agree not to modify the Distribution in any way and further agree to not participate in any way in the making of derivative works thereof, other than as a necessary result of reviewing and providing feedback to the Distribution. You also agree to cause this notice, along with the accompanying consent, to be included on all copies (or portions thereof) of the Distribution. The OSGi Alliance also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights, to create and/or distribute an implementation of the Distribution that: (i) fully implements the Distribution including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the OSGi Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the OSGi Name Space other than those required and authorized by the Distribution. An implementation that does not satisfy limitations (i)-(ii) is not considered an implementation of the Distribution, does not receive the benefits of this license, and must not be described as an implementation of the Distribution. "OSGi Name Space" shall mean the public class or interface declarations whose names begin with "org.osgi" or any recognized successors or replacements thereof. The OSGi Alliance expressly reserves all rights not granted pursuant to these limited copyright licenses including termination of the license at will at any time.

EXCEPT FOR THE LIMITED COPYRIGHT LICENSES GRANTED ABOVE, THE OSGi ALLIANCE DOES NOT GRANT, EITHER EXPRESSLY OR IMPLIEDLY, A LICENSE TO ANY INTELLECTUAL PROPERTY IT, OR ANY THIRD PARTIES, OWN OR CONTROL. Title to the copyright in the Distribution will at all times remain with the OSGi Alliance. The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted therein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

THE DISTRIBUTION IS PROVIDED "AS IS," AND THE OSGi ALLIANCE (INCLUDING ANY THIRD PARTIES THAT HAVE CONTRIBUTED TO THE DISTRIBUTION) MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DISTRIBUTION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
NEITHER THE OSGi ALLIANCE NOR ANY THIRD PARTY WILL BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE DISTRIBUTION.

Implementation of certain elements of this Distribution may be subject to third party intellectual property rights, including without limitation, patent rights (such a third party may or may not be a member of the OSGi Alliance). The OSGi Alliance is not responsible and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

The Distribution is a draft. As a result, the final product may change substantially by the time of final publication, and you are cautioned against relying on the content of this Distribution. You are encouraged to update any implementation of the Distribution if and when such Distribution becomes a final specification.

The OSGi Alliance is willing to receive input, suggestions and other feedback (“Feedback”) on the Distribution. By providing such Feedback to the OSGi Alliance, you grant to the OSGi Alliance and all its Members a non-exclusive, non-transferable,

worldwide, perpetual, irrevocable, royalty-free copyright license to copy, publish, license, modify, sublicense or otherwise distribute and exploit your Feedback for any purpose. Likewise, if incorporation of your Feedback would cause an implementation of the Distribution, including as it may be modified, amended, or published at any point in the future ("Future Specification"), to necessarily infringe a patent or patent application that you own or control, you hereby commit to grant to all implementers of such Distribution or Future Specification an irrevocable, worldwide, sublicenseable, royalty free license under such patent or patent application to make, have made, use, sell, offer for sale, import and export products or services that implement such Distribution or Future Specification. You warrant that (a) to the best of your knowledge you have the right to provide this Feedback, and if you are providing Feedback on behalf of a company, you have the rights to provide Feedback on behalf of your company; (b) the Feedback is not confidential to you and does not violate the copyright or trade secret interests of another; and (c) to the best of your knowledge, use of the Feedback would not cause an implementation of the Distribution or a Future Specification to necessarily infringe any third-party patent or patent application known to you. You also acknowledge that the OSGi Alliance is not required to incorporate your Feedback into any version of the Distribution or a Future Specification.

I HEREBY ACKNOWLEDGE AND AGREE TO THE TERMS AND CONDITIONS DELINEATED ABOVE.

0.2 Trademarks

OSGi™ is a trademark, registered trademark, or service mark of the OSGi Alliance in the US and other countries. Java is a trademark, registered trademark, or service mark of Oracle Corporation in the US and other countries. All other trademarks, registered trademarks, or service marks used in this document are the property of their respective owners and are hereby recognized.

0.3 Feedback

This document can be downloaded from the OSGi Alliance design repository at https://github.com/osgi/design The public can provide feedback about this document by opening a bug at https://www.osgi.org/bugzilla/.

0.4 Table of Contents

0.5 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 10.1.

`Source code is shown in this typeface.`

0.6 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	16 Sep 2019	Todor Boev, initial content copied from RFP
0.1	*12 Dec 2019*	*Updates from the EG call at 20 Nov 2019*

1 Introduction

In today's world it has become increasingly important to create self-contained Java applications with minimal footprint. Even though we can resolve the minimal required set of bundles to include in an application often the total footprint of the application is dominated by the JRE itself.

With the introduction of the Java module system it became possible to statically link a JRE that reflects the needs of a concrete application. One problem that still hinders the creation of minimal OSGi runtimes is the lack of tools that can discover which JPMS modules are required by which OSGi bundles as well as the dependencies between the JPMS modules themselves.

This RFC introduces the requirements for a standard way to encode JPMS modules as OSGi Resources. Among other things this will enable the use of the OSGi Resolver service to discover the dependencies between OSGi bundles and JPMS modules. In this way a complete description of the Java runtime can be obtained

automatically. This should serve as the basis for tools that make the building of minimal OSGi applications practical. At runtime the accurate representation of the JRE as Resources can enable new kinds of application logic such as the application of extenders to modules.

2 Application Domain

In order to compete in the modern world web applications face unprecedented requirements for high availability and responsiveness at internet scale. For various reasons, outside of the scope of this RFP, this has lead to architectures where the application is decomposed into large number if independent simple processes or services. These architectures demand certain liquidity from their building blocks. Their binary images must be easy to move, their instances must be able to saturate a dynamically changing computational capacity, it must be possible to rapidly create and destroy large number of instances. This trend has progressed all the way to the "Function As A Service" (FAAS) architecture where instances must be used to service a single request and then be destroyed. In order to achieve this liquidity it is important for the services to have the smallest possible footprint both in storage and memory. The Java based services face increased competition from language runtimes like Node.js and Go which were built with these requirements in mind and often perform better in this respect.

On the opposite end of the spectrum when users deploy applications on their devices they expect a hassle-free experience. One way to achieve this is to deploy self-contained images. Again it is important for these images to be as small as possible in order to have quick download times and small footprint on the user device. Java applications however often require the users to first install a JRE, which must be of considerable size in order to support all future Java applications, rather than just the one being installed.

To address these needs Java SE 9 introduced the Java Platform Modular System (JPMS) which broke the JRE down into modules and provided a static linker tool which can build a smaller JRE from a subset of these modules. The modular system can also be used by application developers. The JPMS is inherently incompatible with OSGi and what's more both systems have independent dependency models which do not correlate by default. This leads to OSGi bundles being hosted in their own space within the JRE. These bundles however must still be supported by one or more modules from the JRE, which means they have hidden dependencies to those modules.

2.1 Terminology + Abbreviations

- **Bundle**
  For brevity the OSGi bundles will be referred to simply as "bundles".

- **Resource**
  A resource according to the Core OSGi specification *[needs reference]*. Resources are generic and can for example represent any component of an application. The bundles are a kind of Resource specific to OSGi applications.

- **Resolver**
  The Resolver service as defined by the Core OSGi specification *[needs reference]*. Given a pool of Resources the resolver can discover a consistent sub-set where all Resource requirements are satisfied.

- **Module**
  For brevity the JPMS modules will be referred to simply as “modules”.

- **Boot layer**
  The set of modules with which the JRE was started. The boot layer replaces the boot classpath in Java 9+ JREs.

- **Package**
  A java package or just “package”.

- ~~**Bundle packages**~~
  ~~Bundles provide packages to other bundles through the `Export-Package` manifest header. Bundles consume individual packages from other bundles through the `Import-Package` manifest header. This RFP is not concerned with bundles consuming packages through the `Require-Bundle` manifest header.~~

- ~~**Module packages**~~
  ~~Modules provide packages to other modules through the `exports` declaration in `module-info.java`. In contrast to bundles, modules consume packages by requiring entire modules through the `requires` declaration in `module-info.java`.~~

- **System packages**
  Bundles can consume packages provided by modules from the boot layer if those packages are exported by the system bundle.

- **Module Service**
  Modules can register classes with the `java.util.ServiceLoader` utility through the `provides` declaration in `module-info.java`.
  Modules can declare they wish to use `java.util.ServiceLoader` to obtain all classes registered under a given interface through the `uses` declaration in `module-info.java`.
  Bundles can use `java.util.ServiceLoader` to load classes provided by modules from the boot layer.

3 Problem Description

At present bundles only depend on the boot layer as a flat list of system packages. OSGi R7 specifies that this list is discovered dynamically at runtime by the OSGi framework by introspection of the boot layer. Even if this discovery is made at build time this will at best allow the build to make sure the set of bundles fit the boot layer. This does not fix the reverse problem of building the boot layer to fit a set of bundles. To solve this problem the modules must be visible to the build tools. Without automated help the developers must perform the reverse discovery manually through trial and error. This makes the creation of self-contained Java applications hard and sometimes impractical. For example when a large set of small self-contained services has to be maintained or when it must be easy to merge and split bundles into multiple runtimes.

This RFC defines the encoding of JPMS modules as OSGi Resources by mapping some of the JPMS module metadata to OSGi metadata. Having this mapping will make the hidden dependencies bundles have on modules visible.

Standardizing the encoding should enable open collaboration between tools that handle modules and bundles at different stages in the software production pipeline. For example one tool may publish the module metadata while another may read it to build a Java runtime, while yet another may read it to present to a developer. Such tooling will enable developers to achieve the use cases described in this RFC. Also making the JRE visible at runtime as

a set of Resources enables new kinds of application logic that can introspect the modules and the connections they have to bundles and act accordingly.

4 Requirements

4.1 General

- G-1: **Must not** preclude the use of the Resolver as the sole mechanism to build a consistent set of modules and bundles. Note that this does not mean the Resolver specification must not be extended.
- G-2: **Must not** force bundles to explicitly reference modules in order for their requirements to resolve to modules. For example bundles may choose to, but must not be forced to, require a module by ID or include the module ID in the filter of another requirement.
- G-3: **Should** handle the case where a bundle is also a module. For example publish it to a repository as two separate resources pointing to the same artifact.
- G-4: **Should** handle the case of the OSGi framework, which is the only bundle that can also be a module **at runtime.** For example allow the framework to have both module and bundle metadata.

4.2 Resolving Dependencies

- D-1: **Must** define how the identity capability of modules is derived from the module metadata. This **must** include best effort to map the module version to bundle version.
- D-2: **Must** be able to resolve a bundle's package import to a package exported by a module.
- D-3: **Must not** allow a bundle's package import to be resolved to a module's package export when that module exports the package only to a restricted set of other modules. For example such exports may be omitted from the module encoding into a resource.
- D-4: **Must not** allow a module's requirement for another module to be resolved to a bundle.
- D-5: **Must** be able to resolve a module's requirement to another module. For example this helps to understand the complete footprint of an application from the output of the Resolver alone since it will include not only the modules that are directly required by bundles, but also their transitive dependencies.

4.3 Module Services

- S-1: **May** represent a module's provided services as a resource capabilities and a module's used services as resource requirements. This for example will enable developers to discover missing implementations they should include in their Java runtime.
- S-2: **May resolve** a bundle requirement in the `osgi.serviceloader` namespace to a module service.

- S-3: **Must not** resolve a module's service requirement to bundle capability in the `osgi.serviceloader` namespace should MS-2 be implemented.

4.4 Third Party Modules

- T-1: **Must** allow modules not provided by the JRE to participate in the resolution process as equal peers to the JRE modules.

4.5 Module Validation

These requirements are useful only in the context of provisioning since at runtime the JVM will load and validate the modules with OSGi merely representing them in it's environment. These requirements are about ways to use the OSGi Resolver to build sets of modules that can later be loaded successfully by the JVM or the jlink tool.

- V-1: **Must not** allow two modules with the same name to simultaneously participate in the resolution result.
- V-2: **May not** allow for two modules to export the same package and be part of the resolution.
- V-3: **May not** allow modules participating in the solution to form a dependency cycle.

4.6 Module Metadata Enhancement

- E-1: **Should** allow module requirements for other modules to be enriched with version ranges. For example if the metadata of module `A` says it was build against module `B version V` this can map to a resource requirement with version range `[V, infinity)`.
- E-2: **May** allow bundle annotations *[needs reference]* to be used on Java modules to handle dependencies that can not be captured by `module-info.java`

4.7 Modules at Runtime

- R-1: **Should** model the boot layer at runtime as Resources. In other words the `org.osgi.framework.system.packages` and ~~`org.osgi.framework.system.packages.extra`~~ should be removed or set to empty and their content be distributed between the Resources that represent the modules from the boot layer. Note that this does not require that the OSGi bundles are treated as modules.

4.8 Compatibility with Other Specifications

- C-1: **Must** be possible to implement the runtime representation of java modules with the facilities provided by ~~agree with the technical solution of~~ *RFC-243: Connect [needs reference]*

5 Technical Solution

First give an architectural overview of the solution so the reader is gently introduced in the solution (Javadoc is not considered gently). What are the different modules? How do the modules relate? How do they interact? Where do they come from? This section should contain a class diagram. Then describe the different modules in detail. This should contain descriptions, Java code, UML class diagrams, state diagrams and interaction diagrams. This section should be sufficient to implement the solution assuming a skilled person.

Strictly use the terminology a defined in the Problem Context.

On each level, list the limitations of the solutions and any rationales for design decisions. Almost every decision is a trade off so explain what those trade offs are and why a specific trade off is made.

Address what security mechanisms are implemented and how they should be used.

The overall approach is to represent java modules as a special kind of bundle, rather than to introduce a new type of resource.

5.1 Module Identity

The identity of the bundle, which will be derived from the java module is determined as follows:

- The bundle name is equal to the java module name.
- The bundle version is determined from the java module version if present on a best effort basis. This is so because in JPMS versions have no specific format beyond a dot-separated list of numbers with an alpha-numeric suffix. There is no semantic meaning assigned to the numbers comprising the version, there are just rules on how versions are ordered. One simple approach is to pick the first three numbers from the java module version and perhaps convert the rest into a qualifier. If a version is not present use 0.0.0.

5.1.1 Identity Capability

The identity capability of a java module follows the pattern:

```
osgi.identity;
    osgi.identity=<module name>;
    version=<resolved module version>;
    type="osgi.bundle~~java.module~~";
    singleton:="true"
```

The <module name> and <resolved module version> tokens are the name and version of the module as described in the section above.

5.2 Module Dependencies

The java module dependencies have a natural representation in the “osgi.wiring.bundle” namespace

5.2.1 Module bundle wiring capability

The bundle wiring capability follows the pattern:

```
osgi.wiring.bundle;
    osgi.wiring.bundle=<module name>;
    bundle-version=<resolved module version>;
    connect=”java.module|java.classpath|frobnicator” // Delegate to RFC-243: Connect
singleton:=”true” - (?) not used in practice so don’t be so restrictive.
```

The <module name> and <resolved module version> tokens are the name and version of the module as described in the Module Identity section.

The bundle must be marked as a singleton to reflect the JPMS restriction to load multiple versions of a module. Even though it is possible to achieve multiple versions, this can not be implemented for the boot JPMS layer which is an important requirement for the automated construction of minimal JRE images.

The BundleNamespace is extended with a new attribute “bundle-type” which has the values from the “type” attribute from the IdentityNamespace plus the new “java.module” type.

The new attribute is needed in order to construct requirement filters in the “osgi.wiring.bundle” that can only match other java modules. The requirements preclude a module to be wired to a bundle since such a combination is generally impossible at runtime.

5.2.2 Module bundle wiring requirement

A requirement of one module for another is mapped to a follows the pattern:

```
osgi.wiring.bundle;
    filter:=”(&
       (osgi.wiring.bundle=<dependence module name>)
       (bundle-version=<dependence module version>)
       (connectbundle-type=java.module))”;
    resolution:=<”optional” if module dependence is “static”>;
    visibility:=<”reexport” if module dependence is “transitive”>
```

The <dependence module name> is the name of the required module. It can be directly extracted from the module metadata via reflection.

The <dependence module version> is determined on a best-effort basis following the rules for deriving the version of the module identity. The version is recorded at module compile time and is equal to the version of the required module against which the current module is being built. If the required module has no version use ‘0.0.0’

The requirement is of “optional” resolution if the module dependence has the “static” modifier. When a dependence is “static” it is only required at compile time, but not at runtime. The closest analog of this in the OSGi world is to have an optional dependency.

The requirement is of visibility “reexport” if the module dependence has the “transitive” modifier. When a dependence is “transitive” any modules that require the current module can also implicitly “read” the dependence module. The closest analog of this in the OSGi world is visibility “reexport”.

5.2.3 Module Fragment host capability

The module must not provide an “osgi.wiring.host” capability since fragments have no meaning to JPMS. This is equivalent to a bundle that has a “fragment-attachement:=never” directive in it’s Bundle-SymbolicName manifest header.

5.2.4 Module execution environment requirement

The module must have an “osgi.ee” requirement for “JavaSE” with version 9 or greater:

```
osgi.ee; filter:=”(&(osgi.ee=JavaSE)(version=9)”
```

Modules imported from a JRE (as opposed to modules provided by third parties) must use the version of that OSGi variant of the version of that JRE. E.g. JavaSE-12.

5.3 Module Exported Packages

The packages exported by a module are mapped to “osgi.wiring.package” capabilities following the pattern:

```
osgi.wiring.package;
    osgi.wiring.package=<module package name>;
    version=”0.0.0”;
    bundle-symbolic-name=<module name>;
    bundle-version=<resolved module version>;
    java-module-targets=<target module names which can see the package>;
    java-module-access=<”static” or “reflection”>
```

The <module package name> is the name of the java package exported by the module.

The package version is always “0.0.0” since it is very hard to derive a semantic version for the package.

The <module name> and <resolved module version> are the name and version of the module as described in the Identity section.

A new attribute named “java-module-targets” is introduced in the “osgi.wiring.package” namespace to capture the feature of the java module system to let modules export packages to a select set of named modules. Restricting package visibility like this can be implemented in the ResolveContext. The value of this attribute is dubious since module only require other modules and corner cases where a bundle imports a package, but we would like to make the package visible only to other modules don’t seem to warrant the increase in complexity. Perhaps if OSGi decides to introduce “friend bundles” for whatever reason the module “to” attribute can be mapped to a potential “friend bundles” attribute. Perhaps RFC-234 may change these considerations and make this attribute more useful.

A new attribute named “java-module-access” is introduces in the “osgi.wiring.package” namespace to capture the feature of the java module system to open packages for either reflection or both linkage and reflection access. The kind of access is enforced by the JVM itself, but only for modules rather than for legacy class loaded code. This makes the introduction of this feature of even more dubious value since at present OSGi can not make use of it. Perhaps RFC-234 may change these considerations and make this attribute more useful.

5.4 Module services (ServiceLoader)

The concept of providing and using services through the java.util.ServiceLoader can be supported by respective requirements and capabilities in the “osgi.serviceloader” namespace.

The goal is to use the service loader “registrar” to let modules provide services into the OSGi service registry. No use case to use the mediator.

5.5 Modules at runtime

The runtime behavior of modules must be implemented with in the capabilities and restrictions of RFC-243-Connect. I.e. this section describes the behavior of a ConnectFactory that loads modules into the OSGi framework.

5.5.1 Module lifecycle

The ConnectFactory must install all modules it finds, start them persistently at the default start level.

5.6 Dual purpose jars

Tom: you can have a module that has non-class loading caps/reqs.

When a jar has both bundle and module metadata it must be left to the user to determine whether they will use it as a bundle or as a module.

To what extent the bundle MANIFEST.MF and the module-info.java aggree with each other is out of the scope of this specification.

Tom: at least force the module name to be equal to the bundle symbolic name.

Todor: maybe just have the bundle MANIFEST take precedence in all cases.

5.6.1 Dual purpose jars at provisioning time

Publish the jar in an artifact repo as two separate resources that use the same underlying artifact: type=”osgi.bundle” and type=”java.module”. If needed the users can explicitly lock down either of them in the list of provisioning roots, or leave the automated provisioning runtime pick the best option.

5.6.2 Dual purpose jars at runtime

At runtime the ConnectFactory will just use the bundle MANIFEST.MF.

6 Data Transfer Objects

RFC 185 defines Data Transfer Objects as a generic means for management solutions to interact with runtime entities in an OSGi Framework. DTOs provides a common, easily serializable representation of the technology.

For all new functionality added to the OSGi Framework the question should be asked: would this feature benefit from a DTO? The expectation is that in most cases it would.

The DTOs for the design in this RFC should be described here and if there are no DTOs being defined an explanation should be given explaining why this is not applicable in this case.

This section is optional and could also be provided in a separate RFC.

7 Javadoc

Please include Javadoc of any new APIs here, once the design has matured. Instructions on how to export Javadoc for inclusion in the RFC can be found here: https://www.osgi.org/members/RFC/Javadoc

8 Considered Alternatives

For posterity, record the design alternatives that were considered but rejected along with the reason for rejection. This is especially important for external/earlier solutions that were deemed not applicable.

9 Security Considerations

Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.

10 Document Support

10.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

Add references simply by adding new items. You can then cross-refer to them by chosing <Insert><Cross Reference><Numbered Item> and then selecting the paragraph. ***STATIC REFERENCES (I.E. BODGED) ARE NOT ACCEPTABLE, SOMEONE WILL HAVE TO UPDATE THEM LATER, SO DO IT PROPERLY NOW.***

10.2 Author's Address

Name	Todor Boev
Company	Software AG
Address	
Voice	
e-mail	todor.boev@softwareag.com

10.3 Acronyms and Abbreviations

10.4 End of Document