JSON
(JavaScript Object Notation)

JSON (JavaScript Object Notation)

- A lightweight data-interchange format
- A subset of the object literal notation of JavaScript
- Completely language independent
- Easy to understand, manipulate and generate

JSON

- JSON is **NOT**
  - Overly Complex
  - A “document” format
  - A markup language
  - A programming language

JSON Feature

- Straightforward syntax
- Easy to create and manipulate
- Supported by all major JavaScript frameworks
- Supported by most backend technologies

JSON Syntax

- Unordered sets of name/value pairs
- Begins with **{** (left brace)
- Ends with **}** (right brace)
- Each name is followed by **:** (colon)
- Name/value pairs are separated by **,** (comma)

Sample JSON Data

```
var employeeData = {
    "employee_id": 1234567,
    "name": “Ramesh",
    "hire_date": "1/1/2013",
    "location": “Chennai",
    "consultant": false
};
```

Arrays in JSON

- An ordered collection of values
- Begins with **[** (left bracket)
- Ends with **]** (right bracket)
- Name/value pairs are separated by **,** (comma)

JSON Array

```
var employeeData = {
    "employee_id": 1236937,
    "name": “Ramesh",
    "hire_date": "1/1/2013",
    "location": “Chennai",
    "consultant": false,
  "random_nums": [ 24,65,12,94 ]
};
```

Data Types: Objects & Arrays

- Objects: Unordered key/value pairs wrapped in { }
- Arrays: Ordered key/value pairs wrapped in [ ]
- Array Objects

- data = [{ id: '101', name: "Samsung" }, { id: '102', name: "Lenovo" }]

JSON Data

```
lanPlayerList = [
    {
        player: {
            firstName: 'Sangakara',
            lastName: 'Kumar'
        },
        runs: 3420,
        iplteam: "Chennai Super kings",
    }
]
```

JSON Parse

- The **JSON.parse()** method parses a string as JSON

- var city='{"location": "chennai"}';
- loc =JSON.parse(city);
- alert(loc.location);

JSON Stringify

- Need a JSON parser or a function, **`stringify()`**, to convert between JavaScript objects and JSON encoded data.

- var emp={1:"Ramesh"};

- var strEmp= JSON.stringify(emp);

- alert(strEmp.slice(6,9));

GSON

- A Java library that can be used to convert Java Objects into their JSON representation.
- Can also be used to convert a JSON string to an equivalent Java object.
- https://mvnrepository.com/artifact/com.google.code.gson/gson/2.8.6

```
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.6</version>
</dependency>
```

Converting Java Object to jSON

```
Gson gson = new GsonBuilder().setPrettyPrinting().create();


Student student = Student(101,”Ramesh);


String jsonStr = gson.toJson(student);

System.out.println(jsonStr);
```

JSON to Java Object

- Gson gson = new Gson();
- Student student = gson.fromJson(jsonStr, Student.class);
- System.out.println(student.toString());

Logging with Log4j

Logging

- Prints useful statements in files or consoles
- Used to debug the application in case error is caught.

- Important in debugging standpoint.
  - Commonly used Debug tools and options are not be available in actual production environments.

- Logs can be referenced anytime in future as the data is stored

- Logging gives following Details :
  - complete error stack trace
  - root cause,
  - method details in which error is caught
  - Line number date and time error occurred

Log4j

- A logging library for Java applications and is by Apache.
- Available Since 1996
- A Reliable, Fast and Flexible Logging Framework (APIs)
- Highly configurable through external configuration files at runtime

Log4j Advantages

- Provide high level configurability and various logging levels which is not possible with System logs.
- Support for various logging levels, using simple configuration
- No need to use binaries even though debug statements are there in code.
- Several appenders to write the log statements to files, console, database, email etc.
- Thread safe and supports internationalization.
- Layouts can be used to change message format

Log4J Installation

- Can be downloaded from http://logging.apache.org/log4j/.
- Added to the Class Path of the System or Eclipse Build Path
- Can also be added as Maven Dependency in pom.xml

```
<dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
```

Component of Log4j

- **Loggers:**
  - The top level layer is Logger which provides Logger object.
  - The Logger object is responsible for capturing logging information and they are stored in a namespace hierarchy.

- **appenders :**
  - Responsible for publishing logging information to various preferred destinations.
    - database,
    - file,
    - console,
    - UNIX Syslog

Component of Log4j

- **layouts:**
  - To format logging information in different styles
  - provides support to appender objects before publishing logging information.
  - publishing logging information in a way that is human-readable and reusable.

Logging Level

- Defines the granularity and priority of any logging information.
- There are seven levels of logging defined within the API:
- ***From smaller to greater :***

  ALL, DEBUG, INFO, WARN, ERROR, FATAL, OFF
- When a logging level is set, only *messages belonging to that level or greater levels are printed*.

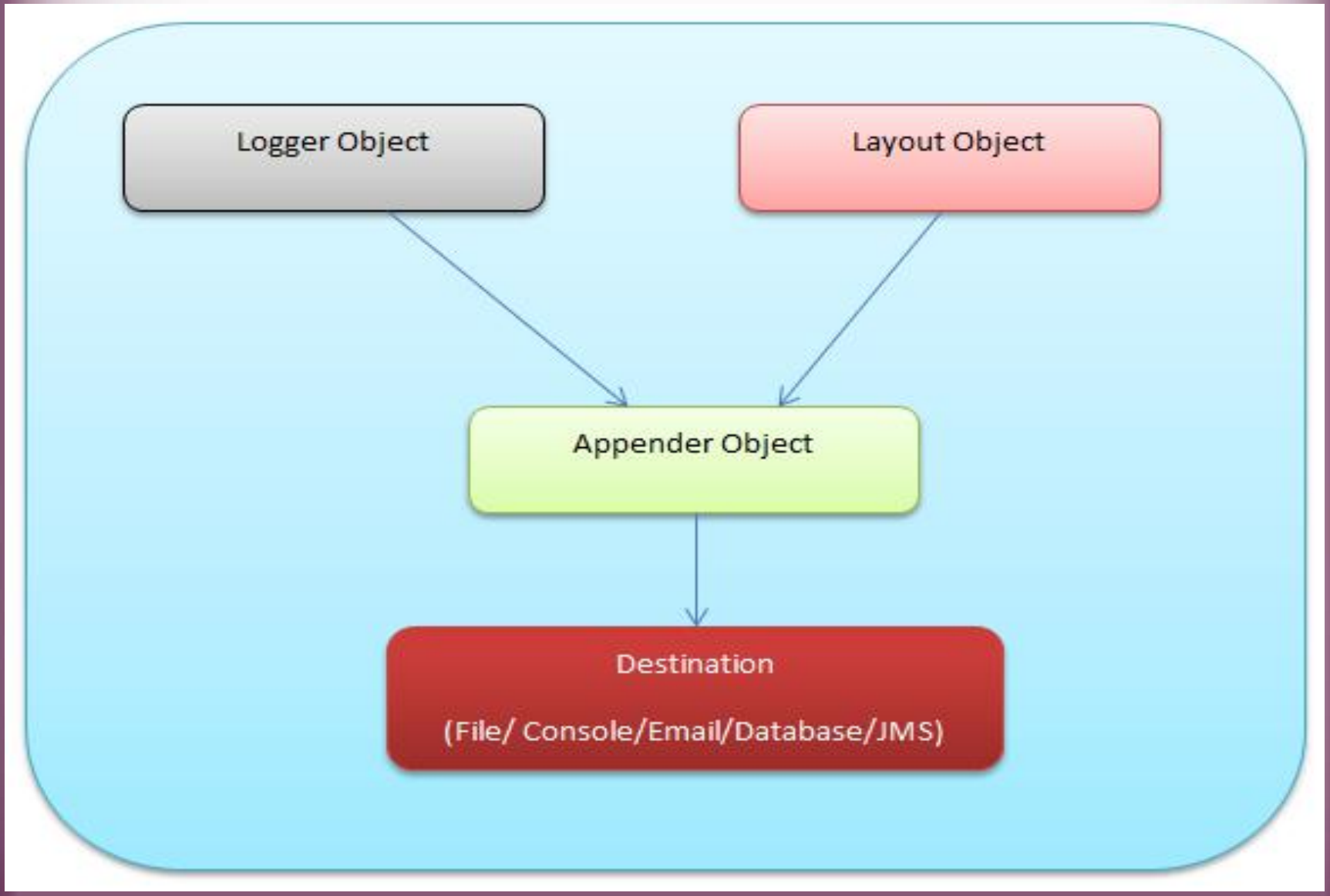
- *DEBUG => DEBUG,INFO,WARN,ERROR,FATAL*
- *INFO => INFO,WARN,ERROR*

Log Level

- Level class provides Level for Messages
- **ALL**
  - All levels including custom levels.
- **DEBUG**
  - Designates fine-grained informational events that are most useful to debug an application.
- **ERROR**
  - Designates error events that might still allow the application to continue running.
- **FATAL**
  - Designates very severe error events that will presumably lead the application to abort.

  - **To Set Log Level for Log4j -Dlog4j.debug=true**

Log4j Components



Root Logger Class

- **public static Logger getRootLogger();**
- Method returns the application instance's root logger and does not have a name.
- The root logger resides at the top of the logger hierarchy. It is exceptional in two ways:
  1. it always exists,
  2. it cannot be retrieved by name.

Logger

- Factory methods to get Logger
  - **Logger.getLogger(Class c)**
  - **Logger.getLogger(String s)**

- Method used to log message
  - trace(), debug(), info(), warn(), error(), fatal()
  - Details
    - **void debug(java.lang.Object message)**
    - **void debug(java.lang.Object message, java.lang.Throwable t)**
  - Generic Log method
    - **void log(Priority priority, Object message)**
    - **void log(Priority priority, Object message, Throwable t)**

Appenders

- Used to printing logging messages to consoles, files etc;
- Has different properties associated with it

- **Layout**
  - Used with conversion pattern to format the logging information.

- **target**
  - console, a file, or another item depending on the appender.

- **Level**
  - The level is required to control the filteration of the log messages.

Appenders

- Appender object are added to a Logger
- **log4j.logger.[logger-name]=level, appender1,appender..n**

- **ConsoleAppender**
  - appends log events to System.out or System.err using a layout specified by the user.
  - The default target is System.out.
- **FileAppender**.
  - Writes the statement to a file
- **Other Appenders**
  - ConsoleAppender
  - DailyRollingFileAppender
  - FileAppender
  - JDBCAppender
  - JMSAppender

File Appender

- *org.apache.log4j.FileAppender.*

- **Filename**
  - The name of the log file.

- **fileAppend**
  - This is by default set to true, which mean the logging information being appended to the end of the same file.

- **bufferSize**
  - Idicates the buffer size. By default is set to 8kb.

Layout

- Layout are used with patterns
- Available Layout are:

- **PatternLayout**
  - To generate logging information in a particular format based on a patternDateLayout

- HTMLLayout
- XMLLayout

- HTMLLayout and XMLLayout
  - Generate log in HTML and in XML format

Pattern Layout

- Log Messages based on a pattern
- Pattern Layout extends the abstract Layout class
- **conversionPattern**
  - Sets the conversion pattern.

- **Default is %r [%t] %p %c %x - %m%n**

- c - The calling class name,
- m - The logging message.
- n - The platform dependent line separator.
- p - The logging Level.
- r - The relative date in millisecond since application start.
- t - The invoking thread.
- [x|X] - the Message Diagnostic (MDC) information.

Layout Patterns

- **C** - The fully qualified class name.
- **d** - The date of the logging request, can be formatted with java.text.SimpleDateFormat, i.e. %d{yyyy-MM-dd}
- **F** - The name of the calling File.
- **l** - The location information, same as %C%M(%F:%L). This can be slow.
- **L** - The calling class line number
- **M** - The calling class method.

- **[%p] %d{MM-dd-yyyy HH:mm:ss} %c %M - %m%n**
- **[%p] %d{DATE} %c %M - %m%n**

Example –With Properties Files

- log4j.rootLogger=ALL, appendToConsole
- log4j.appender. appendToConsole =
  org.apache.log4j.ConsoleAppender
- log4j.appender. appendToConsole.layout=
  org.apache.log4j.PatternLayout
- log4j.appender. appendToConsole.layout.ConversionPattern=
  %-4r [%t] %-5p %c %x - %m%n

File Appender

```
log4j.rootLogger = INFO, File, stdout


log4j.appender.File=org.apache.log4j.FileAppender

log4j.appender.File.File=C:\\logs\\logs.log

log4j.appender.File.layout=org.apache.log4j.PatternLayout

log4j.appender.File.layout.conversionPattern=%d{dd/MMM/yyyy
HH:mm:ss,SSS}- %c{1}: %m%n
```

Example

```
import org.apache.log4j.Logger;
 public class Example {

    static Logger log = Logger.getLogger(this.getClass().getName());

   public static void main(String[] args) {
        System.out.println("Enter  a number from 0 to 100 ");
         Scanner scanner = new Scanner(System.in);
          int number = scanner.nextInt()
             log.info("You inserted the number:"+number);
        if(number > 100) {
            log.error("You entered a wrong number!");
            throw new RunTimeException("Wrong Number");
        } else {
            log.debug("Number is smaller than 100, -correct!");
        }
    }
}
```

Annotations

- allows developers
  - to define custom *annotation types*
  - to *annotate fields, methods, classes, etc. with annotations corresponding to these types*

- allow tools to read and process the annotations
  - no direct effect on semantics of a program
  - e.g. tool can produce additional Java source files or XML documents related to the annotated program

Retention

- **SOURCE:**
  - discarded after compilation
- **CLASS:**
  - recorded in the class file as signature attributes
  - not retained until run time
- **RUNTIME:**
  - recorded in the class file *and retained by the VM at run time*
  - may be read reflectively

Annotation type

- Every annotation has an *annotation type*
  - takes the form of a highly restricted interface declaration
  - new "keyword" @interface
  - a *default value may be specified for an annotation type* member
  - permitted return types include primitive types, String, Class

Meta annotations

- **@Target(ElementType[])**
  - indicates the program elements to which an annotation type can be applied
  - TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR,LOCAL_VARIABLE
  - default: applicable to *all program elements*

- **@Retention(RetentionPolicy)**
  - indicates how long annotations are to be retained
  - values: SOURCE, CLASS, RUNTIME
  - default: CLASS

Annotation

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface MyTable {

 String tableName() ;
}
```

Annotations

```
import javax.annotation.Resource;

@MyTable(tableName="BookData")
public class Book {
    private int bookNumber;
}
```

- Annotation Processor

```
Book ac=new Book();

Class  c =ac.getClass();

MyTabble ano = (MyTable) c.getAnnotation(MyTable.class);

 System.out.println(ano.tableName());
```