Table of Contents

[Features 1](#_Toc41899559)

[Method/Class Syntax 1](#_Toc41899560)

[Declaring 1](#_Toc41899561)

[Comparison and Operators 1](#_Toc41899562)

[Loops 1](#_Toc41899563)

[Flow Control 1](#_Toc41899564)

[Strings 1](#_Toc41899565)

[ASCII Values 1](#_Toc41899566)

[Threads 1](#_Toc41899567)

[Concurrency 1](#_Toc41899568)

[Debegging 1](#_Toc41899569)

[Unit Testing 1](#_Toc41899570)

[Exceptions Handling 1](#_Toc41899571)

[Log4J 1](#_Toc41899572)

[JDBC 1](#_Toc41899573)

# Features

PMD (Programming Mistake Detector) – Static code analysis. Detects if source code conforms to coding standards

Junit – Test cases. Use annotations to denote which cases to run.

EclEmma – Code Coverage

Log4j API – Logging events, used more in front end development

* Source -> Compilter(Javac) -> Byte Code(.class)

JVM -> class loader, Byte Code verifier, Interpreter(Byte to machine code), JIT compiles reusable byte code, Runtime,OS,Hardware

* Java is platform independent – compiles/JVM different on each OS but JVM
* Abstraction is a process of hiding the implementation details from the user. Оnly the functionality will be provided to the user.
* Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.

Pillars of Object Oriented Programming

* Encapsulation – Process of grouping together groups of related data.
* Abstraction – Exposing relavent details. Ex: Dashboard of a car.
* Inheritance – Specialized classification from a more general one.
* Polymorphism – Have an object behave differently depending on the context.

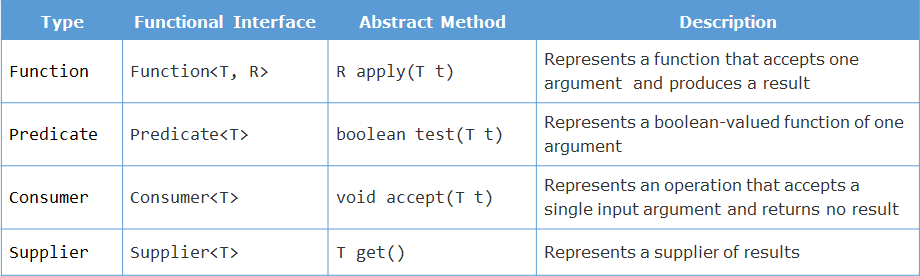
Comments:

* / -- one lone
* /\* \*/ -- Multiline comment
* /\*\* \*\*/ -- Java document comment

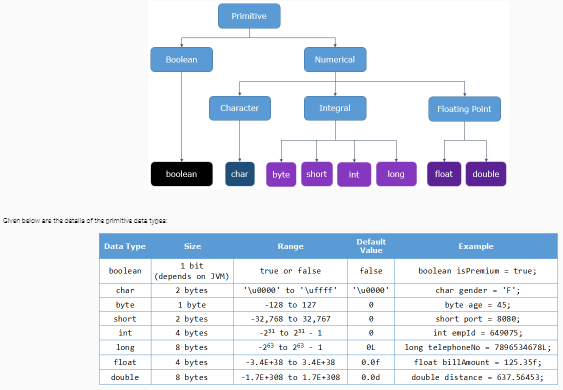
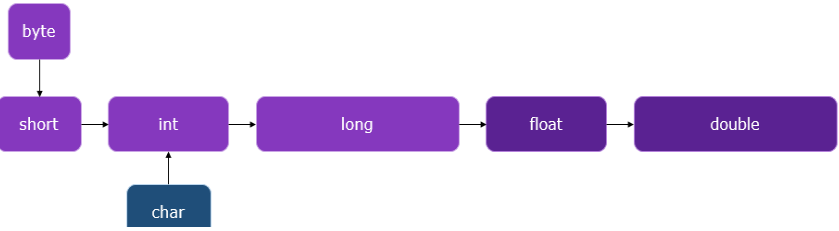
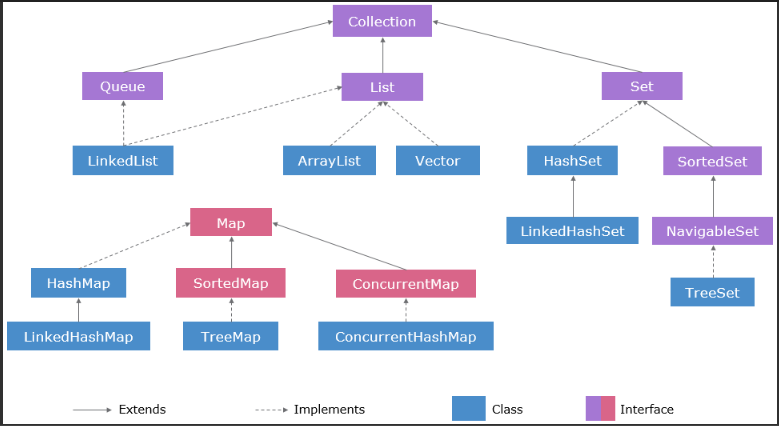
# Method/Class Syntax

* MAIN: public static void main(String[] args) { }
* METHOD: <return type> methodName(byte y, float z) { }
  + Overloading: 2+ methods with same function name but different parameters. They have different signature.
    - Order does not matter, number of parameter types cannot match.
  + Varargs – a parameter can be named (DataType… var1) instead of array brackets. Must be the last parameter in method signature.
* CLASS: class className { //body }
  + INSTANCE VARIABLES: <Access> <DataType> VarName
    - Initialized to default values automatically. Unlike regular local variables.
  + ‘this’ – identifies instance of object
    - Abilitiy: this(constructor args,…)
  + METHOD: <Access> <ReturnType> <MethodName>(datatype var1,…) { }
  + Get/Set: RClick🡪Source🡪Generate Getters/Setters in Enclipse.
  + **Accessors**:
    - Public – Accessed anywhere
    - Protected – accessed in same package and subclasses in different packages
    - Default – accessed in same package
    - Private – Accessible only in own class.
  + Final
    - variable – cannot be change once initialized
    - method – cannot be overridden
    - class – cannot be subclassed
  + Java can inherit multiple interfaces. Inherit only one concrete class.
    - extends – class keyword for inheritance. child extends parent.
    - Access must be reduceable to parent access.
  + Abstract class – A class meant to be inherited. To be implemented by those inheriting. If there’s at least one Abstract method, the class must be Abstract.
    - Abstract method – a declared method with no implementation.
  + High cohesion, Low Coupling – High cohesion: does what it is meant to do by itself. Low Coupling: class uses few outside sources to keep its identity.’
  + @Override – Annotation above method signature, class, instance variable. Will explicitely replace the functionality of parent method of same name.
  + Interface class – all methods are public and abstract. Keyword ‘implement’ on class signature for classes using an interface. An interface extends to another interface.
  + **Enumeration** – Keyword: enum class specifier.
    - Syntax: enum PizzaSize {SMALL,MEDIUM,LARGE}
    - enum fields are static and final implicitly
  + **Lambda** – (argument) -> (body) Syntax below
    - transactionList.sort((x, y) -> -x.getTime().compareTo(y.getTime()));
    - or
    - Comparator<Transaction> TransactionTimeComparatorDesc = (Transaction x, Transaction y) -> -x.getTime().compareTo(y.getTime());

transactionList.sort(TransactionTimeComparatorDesc);

* + - * Negative sign for descending order.
    - Lambda type: Implements a functional interface like Comparator in sort methods.
      * Functional Interfaces have *one* abstract method. @FunctionalInterface can be used to make it explicit.
    - Can be used with List.forEach method: listObj.forEach(x -> System.out.print(x + " "));
    - Predicate – a predefined interface that allows multiple filtering criteria with lambda expressions.
      * Declared: Predicate<DataType> varName;
      * Method: predicateVar.test(listObject);
        + Test returns true or false based on the passed Predicate argument
      * Usage: filterTransactions(transactionList, (t) -> t.getAmount() <= 1000);
  + Anonymous Class – A class define within the scope of brackets. Example:
    - transactionList.sort(new Comparator<Transaction>() { //method here } );
  + **Stream** - represents a sequence of elements from a source. supports various data processing operations. Example Below:
    - transactionList.stream().filter(t -> t.getAmount() > 10000) //Filter uses Predicate
    - .map(t -> t.getTransationId()) // Map uses Function
    - .sorted() //Sort uses Comparator
    - .forEach(id -> System.out.println(id)); //forEach uses Consumer
    - Streams are part of collections interface. They execute in a pipeline. Streams do not execute until they’re required to. This saves storage in memory.
    - Stream actions can be Intermediate or Terminal
      * Intermediate – Returns another stream and can be chained together: filter,sort,map,limit,distinct,…
      * Terminal – Produces results from pipeline: forEach, collect, count. Non-stream objects.
    - A stream is closed after a terminal operation is used. Streams work in parallel
  + In other words, it provides an abstraction over an existing collection

# Declaring

* <data type> <identifier> = <value>;
  + Multiple variables: Int sum = 0, count = 0
* Array:
  + DECLARE:
    - Int[] arrName = {val1,val2,…};
    - Int[] arrName = new datatype[size];
    - Int[][] 2dArr = new datatype[row][col] { {data1,…},{} };
    - Int[][] 2dArr = new datatype[row][]
  + Access: arrName[index]
* List
  + Declare: as ArrayList<E>,LinkedList<E>
    - List<String> arrList = new ArrayList<String>();
    - List<String> arrList = new ArrayList<>();
      * Elements must be of declared type.
    - Old raw type: List list = new ArrayList();
    - More than one type for class or interface:
      * Class MyGenericClass<K,V>{ }
  + Initialize option: *List<String> (list of strings)*
  + Methods: add(index,e),remove(e),get(index)
    - equals(returns true if elements in same order)
    - LinkedList: addFirst(),getLast(),removeLast()
  + Iterator: ListIterator<E> listIterator()
* Collection
  + Methods: sort(),reverse(),max(),freq()
* Conversion:(datatype) var1
* Float/Long - Declare with f or typecast
  + = 45.5f or (float) var1
  + = 45.5L or (long) var1
  + Implicit Type Casting: Ex. Int and short can be cast to long type.
  + Throwable Hierarchy
* Primitives – CHAR, INT, BOOLEAN, BYTE, SHORT, LONG, FLOAT, DOUBLE
* Sizes: Float:4,Double:8,Char:2,Byte:1,Short:2,Int:4,Long:8,Bye:Udf
* DateTime
  + from String:
  + String pattern = "dd-MMM-yyyy";
  + LocalDate.parse(date, DateTimeFormatter.ofPattern(pattern));
  + To String:
  + String dateString = localDateObject.format(DateTimeFormatter.ofPattern("dd-MMM-yyyy"));

# Comparison and Operators

* Bitwise Logical Operators (&, |, ^(XOR), ~(NOT), <<(LEFT SHIFT), >>, >>>(Zero fill right-shift))
* Short-Circuit Logical Operators (&&, ||, !)
* Assignment Operators (=, +=, -=, \*=, /=, %=)
* Ternary operator:
  + <condition> ? <statement if true> : <statement if false>;
* Prefix vs Postfix:
  + B = ++A 🡪 A+=1 then B=A
  + B = A++ 🡪 B=A then A+=1
* Order of Operations:
  + Unary operators, Binary

Loops

* For loop
  + For (declare; comparison; increment) {}
    - A (continue;) will execute increment, then comparison
* Foreach
  + For( datatype var : arr) { }

Flow Control

* IF:
  + if (<Condition1>) { }

else if (<Condition2> { }

else { }

* SWITCH: //Used to replace nested if/else
  + switch(var1) { //var1: byte, short, char, int or String

case val1: <Statement>;

break; //from innermost switch or loop

continue; //continue to next loop iteration of loop

return; // exit current method to last call

default: <Statement>; //Execute when no case matched or no break/continue/return from a case. }

* WHILE:
  + while(<condition>) { }
  + do{ } while(<condition>) { }

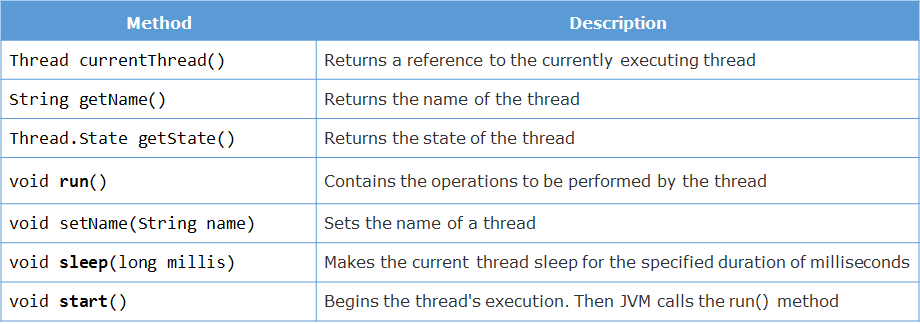
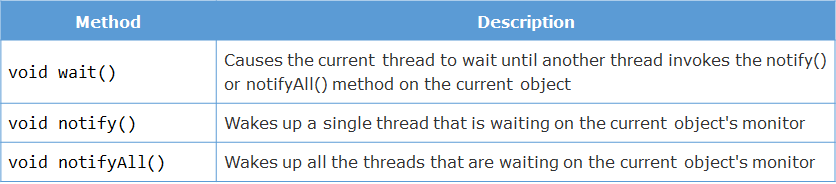
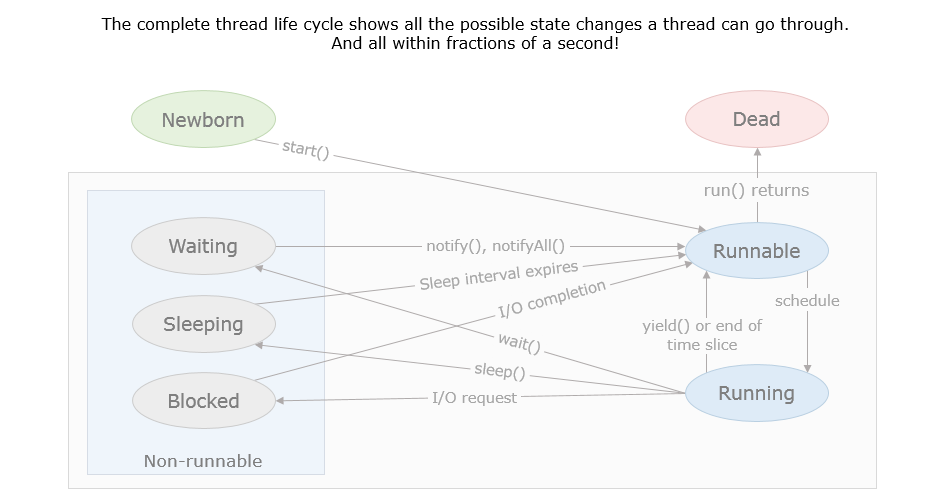
# Strings

* Concatenate
  + StringVar + “String” //overloaded ‘+’ behavior
* String.length() -- other types have .length data field.
* String.format(string,args)
* StringBuffer – thread-safe and mutable strings. Faster and take less memory during manipulations.
  + Useful: append(),insert(),delete(),reverse()
* StringBuilder – Non-synchronized. Faster but not thread-safe.
* Character – Wrapper class for char.
  + Character wrappedChar = ‘C’; //unboxing: avoids constructor call

# ASCII Values

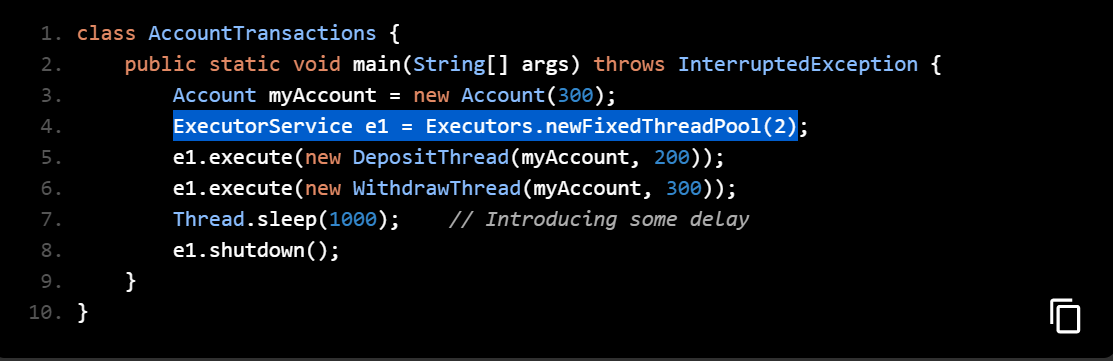
* 97 – lowercase a
* 65 – uppercase A
* 48 - 0

# Threads

* Extend java.lang.Thread or Implement java.lang.Runnable
  + Override run() – contains thread operations
  + Example execution of custom thread:
    - CustomerThread thread1 = new CustomerThread("Customer 1");
    - thread1.start(); //Operating system schedules thread.
  + Locking a resource to a thread until its job is finished can also be achieved by using the **synchronized** keyword.
  + Methods or code blocks that should be accessed by only one thread at a time should be marked as **synchronized**
* Inter-Thread communication:
  + Methods only called in Syncrhonized context
* A **ReentrantLock** - if a thread tries to acquire a lock that it already holds for a different block (in the same shared resource), the request succeeds, i.e. a thread can acquire the lock multiple times without blocking on itself.
  + It means that locks are acquired on a per thread basis rather than per invocation basis. It is implemented by associating with each lock a counter and an owning thread.
  + Here's how it works:
    - When the counter is zero, the lock is unheld
    - When a thread acquires a previously unheld lock, the counter is set to one
    - If the same thread acquires the lock again, the counter is incremented by one
    - When the owning thread exits the synchronized block, the counter is decremented by one
    - When the counter reaches zero, the lock is released

## Concurrency

* Use java.util.concurrent.ExecutorService and Executors to execute concurrent threads.
* Example Usage:



* A **Future** object represents the value that will be returned by a thread in the future.
  + This value can be retrieved using the get() method of the Future object.
  + If the result is ready, it will be returned. If not, the calling thread will be blocked

# Debegging

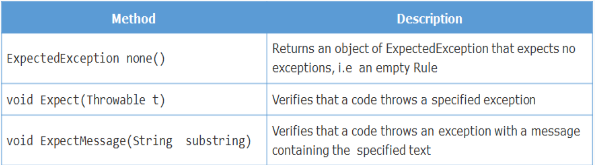
* *Step Into*
  + Executes current line and goes to next line. Will go to called methods
* *Step Over*
  + Will execute the method without going into the method

## JUnit

* Import org.junit.Assert;
* Import org.junit.Test;
* @Test – annotation for method that contains testing.
* Syntax: Assert.assertTrue( True\_Method\_Call\_or\_Operation)
  + AssertFalse(), AssertEquals(), AssertNull(), AssertNotNull()
* Right click on class 🡪 Run As 🡪 Junit Test
* Creating test suites:
  + @RunWith(Suite.class) – A class with this annotation becomes a test suits class. It’s invoked by Junit to run the test classes, instead of the built in runner.
  + @Suite.SuiteClasses – Specifies the test classes to be grouped together and executed when their test suite class runs.
  + Example:

@RunWith(Suite.class)

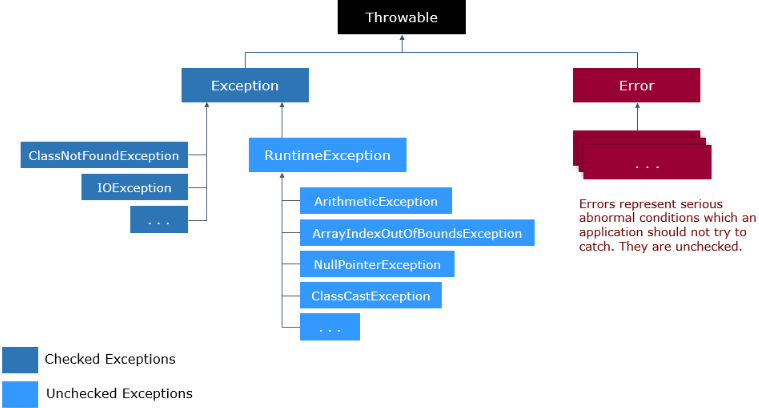
@Suite.SuiteClasses({UserTest.class, ProductTest.class, OrderTest.class})

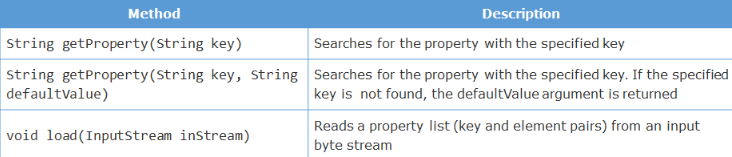
public class TestSuiteDemo { ...} }

* Code Coverage – Jcov, CaCoCo, Cobertura, EclEmma, Emma
  + EclEmma used in this course.
  + Usage: RClick -> Coverage As -> JUnit Test
* Exceptions:
  + Use TestRule class. Example code pictured at end.

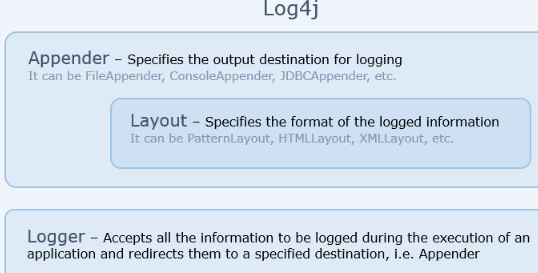
# Exceptions Handling

Try {}

Catch (Exception e) { }

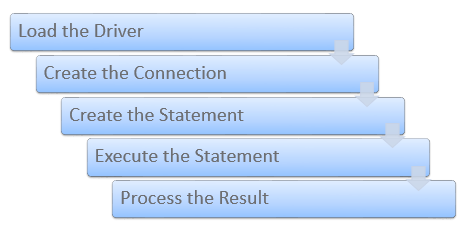
* Catches a method that throws an exception
  + ‘throws Exception’ after method signature.
  + Propogate throws to create exceptions.
  + Throw a new error:
    - Throw new Exception(“Error Message Here”);
    - Or: Exception e = new Exception(); throw e;
    - Optimal to propogate throw in catch blocks
  + e.getmessage() – outputs error message without stack trace.
* Properties Files – maintains all the messages error codes output.
  + Standardization of messages (user friendly messages)
  + Configuration of enterprise applications
  + Internationalization or localization
  + Uses **java.util.Properties** class. See EMart part-2 example.
    - Look at the *configuration.properties* file in resources package of your project
    - Utilize key-value pairs in this manner:
      * AppConfig.PROPERTIES.getProperty(“OUT\_OF\_STOCK”);
      * *AppConfig.java* file present in the *resources*package helps reading and loading key-value pairs in properties file.
      * PROPERTIES.load(inputStream);

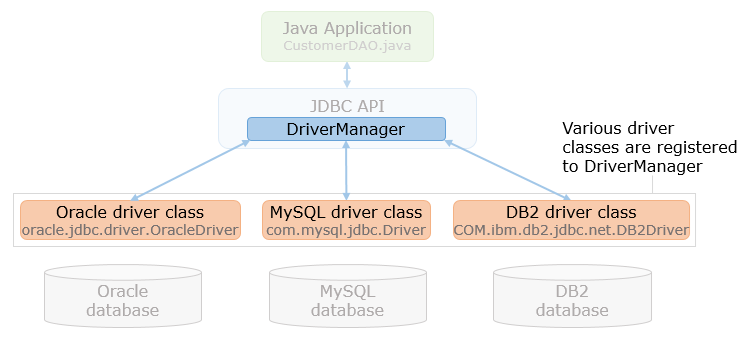
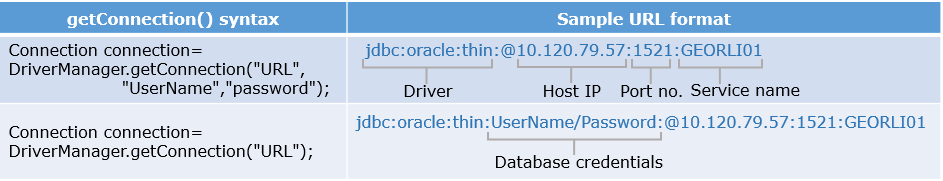
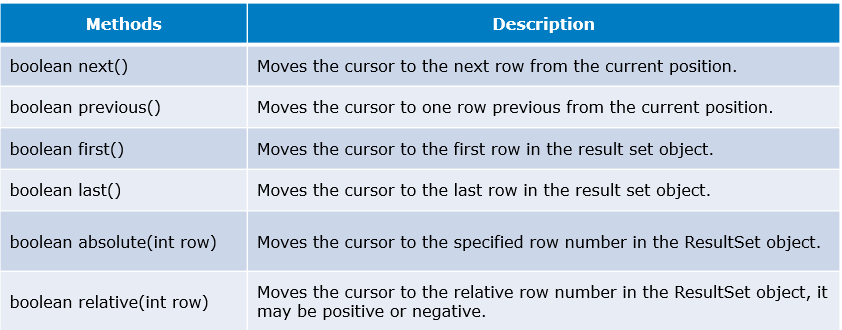
# Log4J

* Error Logging. Springboot also has this ability that can’t be ran at same time.
* Popular logging API: JDK Logging API, Apache Log4j, Commons Logging API
* Log4j uses XML (Extensible Markup Language).
  + Document Type Definition (DTD) or Schema file – Defines limits to elements and attributes.
* LEX: <https://lex.infosysapps.com/viewer/web-module/lex_27724529293303360000?collectionId=lex_19179202633144730000&collectionType=Course>

# JDBC

Java Database Connectivity – Is a data persistence API to connect a Java application and a database. It is part of the Java Core API

* JDBC can access databases such as Oracle, MS Access, MySQL, SQL Server, …
* Functions of JDBC are in java.sql and javax.sql packages
* Uses database specific drivers to translate instructions for a database
  + A drive is used to translate between two dissimilar devices.
  + Specific databases implement the java.sql.Driver interface to build drivers
  + Types of JDBC Drivers:
    - Type 1 (JDBC-ODBC bridge driver)
    - Type 2 (JDBC native-API driver)
    - Type 3 (network-protocol or middleware driver)
    - Type 4 (database-protocol or pure Java driver)

1. Load the Driver
   1. Driver class for that database should be first loaded in build path
      1. Example: Oracle done with OracleDriver.java from ojdbc7.jar
   2. DriverManager.getConnection()
      1. Detects oracle.jdbc.driver.OracleDriver
   3. Multiple Databases connected are detailed in the image
2. Create the Connection
   1. Syntax: Connection connection = DriverManager.getConnection(DBConnectionURL,DBUserName, DBPassword);
   2. Accpts the formats detailed in the image.
3. Create the Statement
   1. Java.sql.preparedStatement is required to serialize sql string to the database command.
4. String sqlString="insert into ChillNGrill\_Orders values(?,?,?,?,?,?)";
5. try(Connection connection=DriverManager.getConnection(DBConnectionURL, DBUserName, DBPassword);
6. PreparedStatement prepareStatement = connection.prepareStatement(sqlString);)
7. {
8. prepareStatement.setInt(1, order.getOrderId());
9. prepareStatement.setString(2, order.getCustomerName());
10. prepareStatement.setString(3, order.getOrderedItem());
11. prepareStatement.setDouble(4, order.getTotalBillAmount());
12. prepareStatement.setLong(5, order.getContactNo());
13. prepareStatement.setString(6,order.getOrderedBranch();}
    1. Syntax shown in image.
14. Execute the statement
    1. preparedStatement.executeUpdate()
       1. This will execute insert query
       2. Returns rows effected.
    2. preparedStatement.executeQuery()
       1. This will retrieve rows as ResultSet
       2. Iterate with resulSet.next()
15. Process results with the ResultSet methods found in the image below.
    1. Package java.sql.ResultSet
    2. A ruleset object can retrive a column field with: getByte, getDouble, getInt, getString
16. Close the connection
    1. Prepared statements and connections done in Try blocks so if something goes wrong it automatically closes
    2. To explicitely close use preparedStatement.close() and connection.close() in a finally block

* ACID Properties
  + Atomic
  + Consistent – commiting changes so they’re perminant.
  + Connection.setAutoCommit(false)
    - Turn off automatic commit for JDBC
    - Connection.commit() to manually commit
    - Connection.rollback() to manually rollback changes done.