JAVA

Definition – importance – Usage

Programming Foundations with Java

Section 1 _____

Java

Java is a high-level, compiled, strongly typed object-oriented programming (OOP) language. The different editions of Java are Standard Edition (SE), Enterprise Edition (EE), and Micro Edition (ME).

Java advantages

- Platform independence – every machine has its own JRE and JVM for Java to write once run anywhere
- C-language inspired syntax – those who are familiar with C will find it easy to understand and learn Java
- Automatic memory management – java automatically manages memory and takes out garbage data
- An extensive built-in runtime library – The Java Application Programming Interface (API) provides the programmer with a built-in library of functions that the programmer does not have to re-create.
- Support from the Oracle corporation
- A rich open-source community

Java Language Specification

- Syntax (Grammar)
- Semantics (Meaning)

OOP - Object-Oriented Programming

Programming with the constructs of classes and objects. An object in code can represent a real-world entity, or a conceptual entity. Classes are the blueprints for how to create objects that contain a certain state - which is represented by fields (variables) - and behavior - which is defined via methods.

Java Compilation

Compilation means to transform a program written in a high-level programming language from source code into object code. Programmers write programs in a form called source code, which must go through several steps before it becomes an executable program. The first step is to pass the source code through a compiler, which translates the high-level language instructions into object code.

The final step in producing an executable program — after the compiler has produced object code — is to pass the object code through a linker. The linker combines modules and gives real values to all symbolic addresses, thereby producing machine code.

Java, being a platform-independent programming language, doesn't work on the one-step compilation. Instead, it involves a two-step execution, first through an OS-independent compiler; and second, in a virtual machine (JVM) which is custom-built for every operating system.

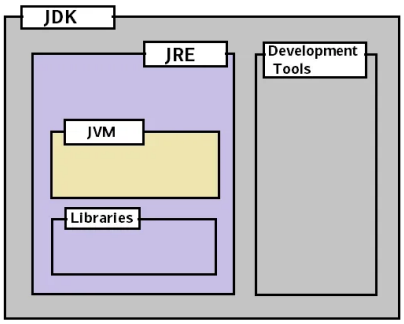
JDK - Java Development Kit

- **Development Tools**
  - **Compiler** – compiles source code into bytecode
  - Debugger
  - Documentation tools

- Other command-line utilities.

- **JRE - Java Runtime Environment**
  - Runtime Libraries
  - **JVM** – executes machine code / instructions
    - **JIT** - Just-In-Time Compiler - turns bytecode into machine code / instructions, in most instances on a line-by-line basis.

Step 1: The developer would write the source code which the JDK would compile into bytecode.
Step 2: The JRE processes the bytecode line by line using the JIT compiler
Step 3: JIT compiler turns the bytecode into machine code/instruction line by line which JVM executes.



Phases of the Process

Compile time: This is when source code is converted at one time to byte code.
Runtime: This is when Java uses the JIT. The bytecode that the developer wrote is then turned into instructions for the JVM to execute.

First Program: HelloWorld

"Hello World!" programs have been implemented in nearly every programming language.

- To illustrate how the process of coding works, especially for introductory programmers,
- As a "sanity test" sometimes, by a more experienced programmer to make sure the components of a language have been correctly installed.
- To compare the executable file sizes of different programmers.

Entities of Java

1. **Class**: The blueprint for creating objects (naming convention: PascalCase)
   - **Object**: Instance of a particular class, consists of-
     i. State (variables of the object)
     ii. Behavior (Methods of the object)
2. **Variables**: Entities that allow us to store data (naming convention: camelCase)
3. **Methods**: Blocks of reusable code that can be invoked again and again (naming convention: camelCase)

Section 2 _____

Method

Methods are a block of reusable code that can be invoked as many times as we would like. The core parts of a method are:

1. **Method Name** - a unique name to identify a method from other methods
2. **Method Parameters** - variables declared inside of the parenthesis of a method which can be utilized inside of the method. Values to these variables are passed as arguments inside parentheses when the method is invoked.
3. **Return Type** - The datatype of the value that is going to be returned from the method. If nothing is needed to return from a method, the "void" keyword is used as the return type.

Anytime we can identify repetitive code in an application, abstracting that logic into a method is generally the best solution to keep the program manageable.

Data Types in Java

Java is a strongly typed language, when a variable is declared in Java, the type must be specified.

- **Primitive types** - the data types defined by the language itself.
- **Reference types** - the data types defined by classes in the Java application Programming Interface (API) or by classes we create rather than by the language itself.

Primitive type vs Reference type

- The memory location associated with a primitive-type variable contains the actual value of the variable. As a result, primitive types are sometimes called value types.
- The memory location associated with a reference-type variable contains an address (called a pointer) that indicates the memory location of the actual object.

Primitive Types of Java

Primitive type	Size (Byte)	(bit)	Datatype Usage
boolean	.1 B	1b	true and false values
byte	1 B	8 b	numerical values
char	2 B	16 b	1 character
short	2 B	16 b	numerical values
int	4 B	32 b	numerical values
float	4 B	32 b	floating point value
long	8 B	64 b	numerical values
double	8 B	64 b	floating point value

Operator precedence

The operators below are listed in order of precedence. Operators with higher precedence will be evaluated before operators with lower precedence.
When operators of equal precedence appear in the same line of code: All binary operators with the exception of assignment operators are evaluated from left to right. Assignment operators are evaluated from right to left.

Operators	Precedence
postfix	expr++ expr--
unary	++expr --expr +expr -expr ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
Bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	?:
assignment	= += -= *= /= %= &= ^=

String

Strings are a commonly used reference type in java to hold multiple characters (text).

String Pool

When Strings are created, they are placed in a special location within the heap called the String Pool.
When String literals are created, if there is an existing String that matches in the String Pool, the reference variable will point to the existing value.
Duplicates will not exist in the String Pool. This is important because Strings take up a lot of memory. Being able to reuse the same value throughout your application is advantageous.

One way to circumvent the above process is to use the new keyword along with the String constructor, which will explicitly create a new String object in memory, even if one already exists in the String Pool.

String Methods

The String API consists of the following:

- toUpperCase() -Converts all the characters of a string to upper case.
- toLowerCase()-Converts all the characters of a string to lower case
- charAt(int index) -This returns the indexed character of a string, where the index of the initial character is 0
- concat(String s) -This returns a new string consisting of which has the old string + s
- equals(String s) -Checks if two strings are equal
- equalsIgnoreCase(String s) -This is like equals(), but it ignores the case (Ex: 'Hello' and 'hello' are equal)
- length( ) -Returns the number of characters in the current string.
- replace(char old, char new) -This returns a new string, generated by replacing every occurrence of old with new.
- trim() -Returns the string that results from removing white space characters from the beginning and ending of the current string.

Stack Trace

Each JVM thread (a path of execution) is associated with a stack that's created when the thread is created.
A stack trace (also known as a stack backtrace) is a report of the active stack frames at a certain point in time during a thread's execution.

Scopes of variables

- **Instance, or object, scope** - The variable is attached to individual objects created from the class.
- **Class, or static, scope**- Resides on the class definition itself, declared with the static keyword.
- **Method scope** - Declared within a method block; only available within the method in which they are declared.
- **Block scope** - Only exist within the specific control flow block ({}, if/else, while, for etc.)

Memory

To run an application in an optimal way, JVM divides memory into stack and heap memory. Whenever we declare new variables and objects, call a new method, declare a String, or perform similar operations, JVM designates memory to these operations from either Stack Memory or Heap Space.

Parameter	Stack Memory	Heap Space
Application	Stack is used in parts, one at a time during execution of a thread	The entire application uses Heap space during runtime
Size	Stack has size limits depending upon OS, and is usually smaller than Heap	There is no size limit on Heap
Storage	Stores only primitive variables and references to objects that are created in Heap Space	All the newly created objects are stored here
Order	It's accessed using Last-in First-out (LIFO) memory allocation system	This memory is accessed via complex memory management techniques that include Young Generation, Old or Tenured Generation, and Permanent Generation.
Life	Stack memory only exists as long as the current method is running	Heap space exists as long as the application runs
Efficiency	Much faster to allocate when compared to heap	Slower to allocate when compared to stack
Allocation / Deallocation	This Memory is automatically allocated and deallocated when a method is called and returned, respectively,	Heap space is allocated when new objects are created and deallocated by Garbage Collector when they're no longer referenced

Custom Class

A custom class adds flexibility to standard mechanism provided by Designer for defining message processing. For example, you can define validation rules, mapping rules and processing rules using the inbuilt formula language. You can also define these rules using custom classes. A custom class is more flexible compared to the formula language. For example, it can connect to a remote object to retrieve a value that can be used in validating a field. To use a custom class, you have to write a Java class that implements the corresponding interface and plug it to the item to be customized

Wrapper Class

Wrapper classes are classes that let you treat primitives as Objects. This is necessary - for example - for certain methods which only accept objects and not primitives. Boxing is the process of converting a primitive to its wrapper class. Java has

a feature called autoboxing which will automatically convert primitives to wrapper classes implicitly. Unboxing is the reverse - converting a wrapper class to its primitive. Below the wrapper classes are listed:

Primitive Type	Wrapper Class
boolean	Boolean
byte	Byte
short	Short
char	Character
int	Integer
long	Long
float	Float
double	Double

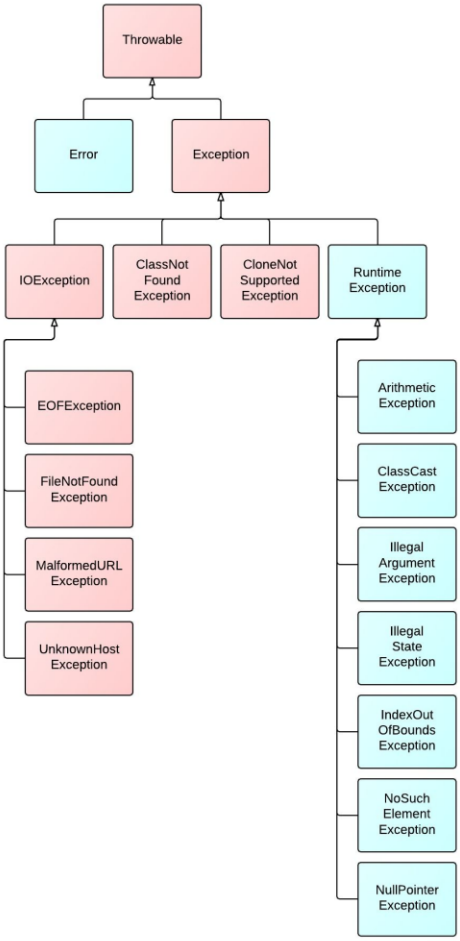
Errors, Exceptions, and Compilation Errors

Exceptions and errors are thrown at runtime.

Compilation errors prevent the code from compiling. Recall that in the java compilation process, the compiler checks that any checked exceptions that could be thrown are handled (via either try/catch block or the throws declaration on the method signature). If this is not the case, the compiler cannot compile the code. This is an example of a compilation error - not an exception. Compilation errors generally occur due to improper syntax, like calling a method that doesn't exist, forgetting a semicolon on a line, using the wrong data type, using a reserved keyword incorrectly, and as we've shown, not handling checked exceptions properly.

Exceptions can only be thrown when the code is executing (running).
The table below summarizes the difference between Errors, Exceptions, and Compilation Errors.

Name	Description	Occurs At	Can be caught?	Must be handled?	Example
Compilation Error	The compiler cannot compile the source code	Compile-time	N/A	N/A	SyntaxError
Error	Severe problem with the program	Runtime	Yes	No	OutOfMemoryError
Checked Exception	Any exception not derived from RuntimeException class	Runtime	Yes	Yes	IOException
Unchecked Exception	Any exception derived from RuntimeException class	Runtime	Yes	No	NullPointerException



Functional Interfaces

Interfaces that have only one abstract method. This method is what lambdas are implementing when they are declared - the parameter types and return types of the lambda must match the functional interface method declaration. The Java 8 JDK comes with many built-in functional interfaces.

```
interface MyFunctionalInt {
  int doMath(int number);
}

public class Execute {
  public static void main(String[] args) {
    MyFunctionalInt doubleIt = n -> n * 2;
    System.out.println(doubleIt.doMath(2)); // 4
  }
}
```

Lambda Expressions

Can be used as the argument in .forEach method of Iterable interface and to implement functional interfaces.

```
parameter(s) -> expression
stringList.forEach(string -> System.out.println(string));
integerArrayList.forEach(n -> { if (n%2 == 0) System.out.println(n); });
```

Method Reference

A special type of lambda expressions. They're often used to create simple lambda expressions by referencing existing methods. There are four kinds:

- Static methods
  - `ContainingClass::staticMethodName`
  - `stringList.forEach(StringUtils::capitalize);`
- Instance methods of objects
  - `containingObject::instanceMethodName`
  - myApp::appendStrings2
- Instance methods of an arbitrary object of a particular type
  - ContainingType::methodName
  - String::concat
- Constructor
  - ClassName::new
  - HashSet::new

Collections and Algorithms

Generics are constructs introduced in Java 5 which enforce compile time safety by allowing you to use parameterized types.

Generics can be declared on a class (generic types), method parameters (generic methods), or return types.

With generics, we can restrict a class to only accepting objects of a given type and the compiler will prevent us from using any other type.

To make a class (or interface) generic, use the angle brackets when declaring it, and use an arbitrary "generic type" which is determined by the invoking code. The generic type can then be reused throughout the class to enforce type safety.

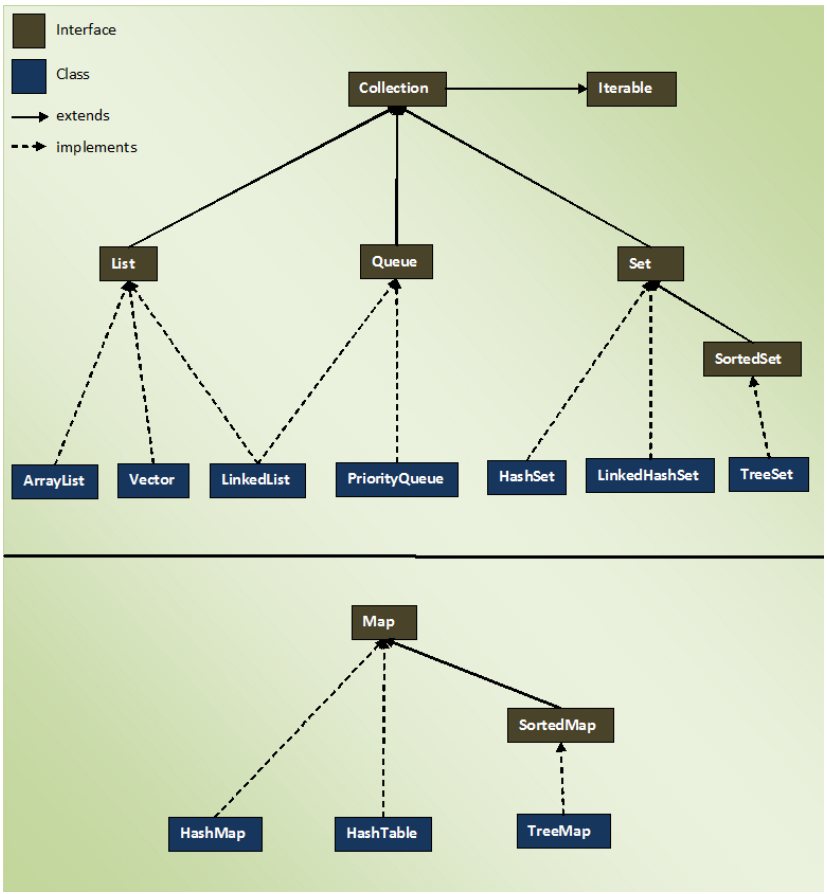
Naming Convention for Generics

Technically, type parameters can be named anything you want. The convention is to use single, uppercase letters to make it obvious that they are not real class names.

- E => Element
- K => Map Key
- V => Map Value
- N => Number
- T => Generic data type
- S, U, V, and so on => For multiple generic data types

The Collections framework

A set of classes and interfaces that implement commonly used data structures. A collection is a single object which acts as a container for other objects.



The important interfaces in the Collections API are:

- **Iterable** - guarantees the collection can be iterated over
- **List** - an ordered collection
  - ArrayList - contains an array within it but can resize dynamically. Traversal is fast (constant time) Insertion or removal of elements is slow (linear time)
  - Vector - a thread-safe implementation of an ArrayList
  - LinkedList - implements both the List and Queue, composed of nodes internally, each with a reference to the previous node and the next node - i.e. a doubly-linked list. Insertion or removal of elements is fast (no risk to resize, just change the nearest references), but traversal is slow for an arbitrary index.
- **Queue** - a collection that operates on a first-in-first-out (FIFO) basis
  - LinkedList - implements both the List and Queue
- **Set** - a collection does not contain duplicates
- **Map** - contains key/value pairs. Does not extend Iterable.

The Set interface

Contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.
Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ.

- HashSet - implements Set and is backed by a HashMap.
- TreeSet - a Set whose elements maintain sorted order when inserted. Internally, it is backed by a Sorted Tree.

Methods:

- Union: setA ∪ setB setA.addAll(setB);
- Intersection: setA ∩ setB setA.retainAll(setB);
- Symmetric difference: setA - setB setA.removeAll(setB);

The Queue Interface

Places objects on a “waiting list”, typically based on First-In-First-Out (FIFO).

- Useful for storing objects prior to processing.

- Elements are added to the tail of the queue.
- Elements can be "popped" off the front of the queue

The Deque Interface extends the Queue interface

- Short for “double-ended queue”
- Pronounced “deck”
- Supports element insertion and removal from both ends of the queue
- Can be used to implement a stack, with Last-In-First-Out (LIFO) behavior

Queue implementations include:

- LinkedList
- ArrayDeque - Resizable-array implementation of Deque interface
- PriorityQueue - Elements in the queue are ordered by priority based on their natural ordering (or a Comparator)
- ArrayBlockingQueue
    - Array-backed, implements BlockingQueue interface which supports operations that wait on the queue to contain an element or for space to become available in the queue
    - Solution to the producer-consumer problem, where a producer thread inputs elements into the array while a consumer thread removes them for processing

Core Methods

The Queue declares several methods that need to be coded by all implementing classes. Let's outline a few of the more important ones now:

- offer() – Inserts a new element onto the Queue
- poll() – Removes an element from the front of the Queue
- peek() – Inspects the element at the front of the Queue, without removing it

Map Interface

It is used to identify a value by a key, and each entry in a map is a key-value pair. Does not implement the Collection interface, however it is part of the Collections framework.

- It is used to identify a value by a key, and each entry in a map is a key-value pair.
- Because it does not implement Iterable, Maps cannot be iterated over directly. Instead, one must either:
    - Use the entrySet() method to iterate over the set of Map.Entry
    - Use the keySet() method to iterate over the set of keys
    - Use the values() method to return a Collection of values which can then be iterated over
- Implemenations of Map include:
    - HashMap
    - TreeMap
    - HashTable, which is an older, thread-safe implementation of a HashMap. It does not allow null keys or null values.

Algorithm

A process, or set of rules to be followed in calculations or some other problem-solving operations. Typically, the size of the input will often affect the overall runtime of the algorithm.

Time Complexity

"asymptotic" = "the tendency in the long run, as the size of the input is increased."
Central to asymptotic analysis is Big-Oh notation. Using this notation, we might say, for example, that an algorithm has a running time that is O(n^2) or O(n) or O(log(n)).
Common Big O notations:

- O(1) : Constant time
- O(N) : Linear time
- O(N^2) : Quadratic time
- O(log(n)) : Logarithmic time

Linear Search

A sequential search is done for all items one by one.

- Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.
- generally not the most efficient method of searching an array
- it is the best method for searching an unsorted array.

Binary Search

Binary search is a method for searching for a given item in a sorted array.

- If you are searching for an item in a sorted list, then it is possible to eliminate half of the items in the list by inspecting a single item. After eliminating half of the initial list, the same process can be repeated to continually cut the list in half until the routine completes by either finding the item or concluding that the item is not in the list.
- The time complexity is O(log n).
- The space complexity of the binary search algorithm depends on the implementation of the algorithm.
    - for iterative method - O(1).
    - for recursive method - O(log n).

Bubble Sort

A simple sorting algorithm that repeatedly steps through the array to be sorted, compares adjacent elements and swaps them if they are in the wrong order.

- The time complexity of bubble sort is O(n^2)

Merge Sort

A sorting algorithm utilizes the divide-and-conquer approach.

- The time complexity of this sort is O(nlog(n))

Optional Class

By using Optional, we can specify alternate values to return or alternate code to run.

```
Optional.ofNullable(nullName).orElseGet(() -> "john");
```

Stream Interface

Streams do not store data, they simply define operations like filtering, mapping, or reducing, and can be combined with other operations and then executed.
Streams are divided into intermediate and terminal operations.

- Intermediate streams return a new stream and are always lazy - they don't actually execute until a terminal operation is called.
- Terminal operations trigger the execution of the stream pipeline, which allows efficiency by perfoming all operations in a single pass over the data.

Finally, reduction operations take a sequence of elements and combine them into a single result.
Stream classes have the reduce() and collect() methods for this purpose, with many built-in operations defined in the Collectors class.

```
List<Student> students = new ArrayList<>();
// add students...
List<Double> grades = students.stream()
                        .filter(s -> s.isAttending())
                                                .mapToDouble(s -> s.getGrade)
                                                .collect(Collectors.toList());
```

Reflection API

- Reflection allows an executing Java program to examine or "introspect" upon itself, and manipulate internal properties of the program
- For example, it's possible for a Java class to obtain the names of all its members and display them.
- The ability to examine and manipulate a Java class from within itself may not sound like very much, but in other programming languages this feature simply doesn't exist.

- For example, there is no way in a Pascal, C, or C++ program to obtain information about the functions defined within that program.
- One tangible use of reflection is in JavaBeans, where software components can be manipulated visually via a builder tool. The tool uses reflection to obtain the properties of Java components (classes) as they are dynamically loaded.

THREADS

Concurrency

Refers to breaking up a task or piece of computation into different parts that can be executed independently, out of order, or in partial order without affecting the outcome. One way - but not the only way - of achieving concurrency is by using multiple threads in the same program. Operating systems use concurrency to manage the many different programs that run on them.

Multi-core Processing

On systems which have multiple cores or CPUs, different processes can be run on different CPUs entirely which enables true parallelization and is a key benefit of writing multithreaded programs.

Time Splicing

On systems without multiple cores or CPUs, operating systems can still achieve concurrency by running one process for a short time, then switching to another, and back very rapidly. This ensures that no process or application is completely blocked.

Thread

A subset of a process that is also an independent sequence of execution, but threads of the main process run in the same memory space, managed independently by a scheduler. Every thread that is created in a program is given its own call stack where it stores local variables references. However, all threads share the same heap, where the objects live in memory.

Multithreading

extends the idea of multitasking into applications where you can subdivide operations in a single application into individual, parallel threads. The OS divides processing time not just with applications, but between threads.
reading is achieved via the Thread class and/or the Runnable interface.

Thread methods

A few important methods in the Thread class include:

- **getters** and **setters** for id, name, and priority
- **interrupt() -** explicitly interrupt the thread
- **isAlive**(), **isInterrupted()** and **isDaemon()** - test the state of the thread
- **join() -** wait for the thread to finish execution
- **start()** - begin thread execution after instantiation

A few important static methods are also defined:

- **Thread.currentThread()** - returns the thread that is currently executing
- **Thread.sleep(long millisecond)** - causes the currently executing thread to temporarily stop for a specified number of milliseconds

Thread Priorities

Priorities signify which order threads are to be run. The Thread class contains a few static variables for priority:
MIN_PRIORITY = 1
NORM_PRIORITY = 5, default
MAX_PRIORITY = 10

States of a Thread

New: newly created thread that has not started executing
Runnable: either running or ready for execution but waiting for its resource allocation
Blocked: waiting to acquire a monitor lock to enter or re-enter a synchronized block/method

Waiting: waiting for some other thread to perform an action without any time limit
Timed_Waiting: waiting for some other thread to perform a specific action for a specified time period
Terminated: has completed its execution

Synchronization

Synchronization is the capability to control the access of multiple threads to any shared resource.

Synchronized keyword

In a multithreaded environment, a race condition occurs when 2 or more threads attempt to access the same resource. Using the synchronized keyword on a piece of logic enforces that only one thread can access the resource at any given time. synchronized blocks or methods can be created using the keyword. Also, one way a class can be "thread-safe" is if all its methods are synchronized.

Deadlock

Deadlock is a condition when two or more threads try to access the same resources at the same time. Then these threads can never access the resource and eventually go into the waiting state forever.
So, the deadlock condition arises when there are more than two threads and two or more than two resources. Basically, a deadlock occurs when multiple threads request for the same resource but they are received in a different order. Eventually, they get stuck for an infinite period of time and cause a deadlock.

Detecting Deadlock in Java

Though it is not possible to completely get rid of the deadlock problem, we can take precautions to avoid such deadlock conditions. These preventive measures are as follows:

- By avoiding Nested Locks - Nested locks mean we try to provide access to resources to multiple threads. If we have already assigned one lock to a thread then we should avoid giving it to the another thread
- By avoiding unnecessary locks - We should only provide the lock to the important threads and avoid using unnecessary locks.

Livelock

Two or more threads keep on transferring states between one another instead of waiting infinitely as we saw in the deadlock example. Consequently, the threads are not able to perform their respective tasks.
A great example of livelock is a messaging system where, when an exception occurs, the message consumer rolls back the transaction and puts the message back to the head of the queue. Then the same message is repeatedly read from the queue, only to cause another exception and be put back on the queue. The consumer will never pick up any other message from the queue.

Producer-Consumer Problem

The Producer-Consumer problem is a classic example of a multi-process synchronization problem. Here, we have a fixed-size buffer and two classes of threads - producers and consumers. Producers produces the data to the queue and Consumers consume the data from the queue. Both producer and consumer shares the same fixed-size buffer as a queue.
Problem - The producer should produce data only when the queue is not full. If the queue is full, then the producer shouldn't be allowed to put any data into the queue. The consumer should consume data only when the queue is not empty. If the queue is empty, then the consumer shouldn't be allowed to take any data from the queue.
We can solve the Producer-Consumer problem by using wait() & notify()methods to communicate between producer and consumer threads. The wait() method to pause the producer or consumer thread depending on the queue size. The notify() method sends a notification to the waiting thread.
Producer thread will keep on producing data for Consumer to consume. It will use wait() method when Queue is full and use notify() method to send notification to Consumer thread once data is added to the queue.

```
public synchronized void produce() {
        while (queue.size() == MAX_SIZE) {
                //Queue is full, Producer thread waiting for consumer to take data from the queue
                wait();
        }
        /* When queue has space, Producer produces the data and adds them into the queue.
        *  After that, Producer sends the notification to the Consumer.
        */
        //producing data
        queue.add(data);
```

```
        notify();
}

public synchronized String consume() {
        while (messages.isEmpty()) {
                //Queue is empty, Consumer thread waiting for producer to put data to the queue
                wait();
        }
                /* When queue has data, Consumer consumes the data and removes it from the queue.
                *  After that, Consumer sends the notification to the Producer.
                */
                //consuming data
                queue.remove(data);
                notify()
}
```