Week1

- When an integer is divided by another integer, result is always an integer (quotient). If the dividend < divisor, output will be 0. When one of the operands is float/double, the output gets *promoted* to this type.
- Activation records are added into the stack for every called function. Each AR consists of local variables, control link and result link. Control link points to the activation record of the parent. Result link points to the address from where the original call was made.
- Dynamic lookup is a process in programming where the specific function or method to be executed is determined at runtime based on the actual type or class of an object, allowing for flexibility and late binding of functionality.
- Each .java file can have only one public class, no more, no less.
- Entry point to a java app is the main() method.
- Operator precedence doesn't work in Java like in Python. This results in a difference between the operations: *b * a / b* and *a / b * b*. As per BODMAS, *b * a / b* is evaluated as *b * (a / b)*. This is how Python works. But, in Java, *b * a / b* is evaluated as *(b * a)/b*.

  Normally, this shouldn't make a difference, and both should result in *a*, but if a and b were integers and b is not a factor of a, *a / b* would result in loss of precision during division operation (or result in 0, if *a* < *b*) and hence the outputs would be different for these expressions.

  Hence, the following Java program would result in *0.0, 10.0*

```
class FClass{
    public static void main(String[] args) {
        int i1 = 10, i2 = 29;
        double d;
        d = i1 / i2 * i2;
        System.out.print(d + " ");
        d = i2 * i1 / i2;
        System.out.print(d + " ");
    }
}
```

Week2

- Size of data types don't depend on the JVM architecture. Following table represents the number of bytes each datatype takes in Java.

Type	Size in bytes
int	4
long	8
short	2
byte	1
float	4
double	8
char	2
boolean	1

  NOTE: *char* datatype takes 2 bytes (instead of 1), due to the necessity to support Unicode characters.
- String is not a primitive type and is a reference type (object).
- Strings are represented in double-quotes, while characters are represented in single-quotes.
- Boolean values are *true* and *false* – in all lowercase.
- Variables can be initialized during the time of declaration.
- Data is treated as constant, if it's prefixed with the *final* keyword, during declaration.
- *float* supports 6 digits after the decimal point, whereas *double* supports 15 digits after the decimal point.
- To mark data as float, use suffix *f*. For example, *float x = 1.3456f*
- When the operands of a division operation are integers, the output will use integer division.
- Java doesn't support an exponentiation operator, and hence needs to use *Math.pow()*
- *String* is not a array of characters. Hence, indexing a *string* is not possible, unlike Python. Instead use *charAt()* method to index.
- Slicing of *string* is not possible, unlike Python. Instead, use *substring* method of *String* class.
- To initialize an array of 3 strings, use *String[] arr = new String[3]*. Note that each element is initialized with null, if the array contains objects.
- To find the number of elements in an array, use length as a property. In the case of a string, use *length()* to find the number of characters.
- *Integer* class is a wrapper class for the primitive type *int*
- Variable of a lower type can be assigned to a variable of higher type without explicit typecasting. Thus, *int* gets auto-converted to *double* when an integer value is assigned to a variable *of double* type. However, to assign a *double* to an *int* variable, it must be cast to an *int*. Note that *double > float > long > int > char > short > byte*.
- To convert *Integer* to *Double*, use *doubleValue()* method on the *Integer* variable.
- Class constructor should have the same name as the class. Constructors don't return anything, but shouldn't be marked void. Constructor function should be marked public.
- If the class constructor isn't defined, a *default* constructor provided by the language will be used. In this case, the instance variables will be initialized to *sensible* defaults. Thus, an numeric variable gets 0, Boolean variable gets a false and a string variable gets a null.
- Multiple constructors could exist for the same class, each of which differ from the other by the number/type of parameters. Note that the default constructor will NOT be available only if one has been explicitly defined in the code.
- It's possible to *call* one constructor from another using *this* keyword.

- When the class constructor use an object of the same class as its parameter, it's called *copy constructor*. Here is an example usage.

```
public class Date {
  private int day, month, year;

  public Date(Date d){
    this.day = d.day;
    this.month = d.month;
    this.year = d.yeary;
  }
}
```

```
public void UseDate() {
  Date d1,d2;
  d1 = new Date(12,4,1954);
  d2 = new.Date(d1);
}
```

  Note that this is an example of a deep-copy, rather than a shallow-copy done with *d1 = d2.*
- When mutable types (like arrays) are passed into a function, they're passed as reference. Call by reference have side-effects, and any changes made inside the function is reflected outside too.
- It's not possible to call a non-static method from within a static method. See following example.

```
public class Test1{
    public void fun() {
        System.out.println("hello");
    }
    public static void main(String args[]) {
        fun();
    }
}
```

  For this example to work, *fun()* should be defined as a static method of *Test1* class.

  Alternatively, create an object of Test1 inside main() method and invoke its fun() method, like so.

```
public class Test1{
    public void fun1() {
        System.out.println("hello");
    }
    public void fun2() {
        fun1();
        System.out.println("all");
    }
    public static void main(String args[]) {
        Test1 obj  = new Test1();
        obj.fun2();
    }
}
```

- There are two forms of for loop – one of which is a C-style loop, and another is a python-style (for-each) loop. In the first style, a variable can be *optionally* defined and limited to the loop body by defining as part of for syntax. E.g. *for (int i = 0; i < 10; i++)*. In the second style, this is compulsory though. E.g. *for (int i : arr)*
- *Switch* statement must use a *break* statement after each *case*, unless you want to execute both blocks for the same *case*.
- If a Java function doesn't return anything, it must be defined as *void.*
- When instantiating an array of objects, you can skip the () after the class name. For example, in order to create one object of App class, use *App a = new App()*. In order to create 5 App objects, use *App[] a = new App[5]*
- In the case of objects (like String), == operator checks if LHS and RHS points to the same object. In the case of non-objects (primitive types), == internally uses *equals()* method and typically checks for the equality of values. Note also that you could overload *equals()* method in the class definition.

Week3

- Java doesn't support multiple inheritance.
- When a class is inherited from another, while creating the child class object the super() constructor gets called implicitly. It needs to be called explicitly, only if there is no default constructor (but an overloaded constructor is available) in the parent class. In this case, if there's no explicit call to super(), you will get a compile-time error. Refer *GA9_Modified.java*
- Method overloading overloads the same method with different signatures, within the same class. It's also called compile-time polymorphism.
- Methods can be overloaded either by the number of parameters, types of parameters or even order of parameters, but not by the return type.
- When the parent class is inherited by a child class, the parent method can be *over-ridden* in the child class. In this case, both parent method and child method must have the same signature. This is called run-time polymorphism.
- Object of a child class can be assigned to a parent reference (*upcasting*), but object of a parent class cannot be assigned to a child reference, which will result in compiler error. When it's cast to the child class (*downcasting*) before assigning to the child reference, it could work, but can result in a runtime error if the actual object is not an instance of the child class. Thus, while *downcasting* use the *instanceof* operator to check if the actual type of the object matches the child class.
- https://discourse.onlinedegree.iitm.ac.in/t/week-3-pa-question-9/106083/4
- In the case of run-time polymorphism, a child object can be referred using parent type. In this case, If the method is called with respect to parent object, parent method gets called. If the method is called with respect to child object, child method gets called. Note that , if the method signature is not available in the parent, but only the child, it will give a compile error, unless the reference to the object is *typecast* to child type. Note that the object itself cannot be typecast, and will raise a compiler error.

  Refer *GA1.java*. Here, #15 will not work because display function accepts a *String* parameter only in class B, not in class A. Thus, an object declared as type class A must be typecast to class B, for the display function call to work.

```
1  class GA1_A {
2      public void display() {
3          System.out.print(s:"Hii ");
4      }
5  }
6
7  public class GA1 extends GA1_A {
8      public void display(String s) {
9          display();
10         System.out.println(s);
11     }
12
       Run | Debug
13     public static void main(String[] args) {
14         GA1_A a = new GA1();
15         //a.display("Ram");
16         ((GA1)a).display(s:"Ram");
17     }
18 }
```

  Another example

```java
class Test_A {
    void f() {
        System.out.println(x:"Test_A::f");
    }
}
class Test_B extends Test_A {
    void f() {
        System.out.println(x:"Test_B::f");
    }
}

public class Test {
    Run | Debug
    public static void main(String[] args) {
        Test_A a1 = new Test_A();
        Test_B b1 = new Test_B();
        a1 = b1; //upcasting works always
        a1.f(); //Test_B::f, polymorphism in action.

        Test_A a2 = new Test_A();
        Test_B b2 = new Test_B();
        if (a2 instanceof Test_B)
            //downcasting works only if the
            //object is an instance of subclass.
            b2 = (Test_B)a2;
        b2.f(); //Test_B::f
    }
}
```

- When a class has the *final* access specifier, it cannot be inherited. Likewise, if a method in the parent class has the *final* access specifier, it cannot be overridden in the child class.
- If two classes are in the same file (and thus, in the same package), they can access each other's protected members. However, a class in a different package cannot access another class's protected members, unless it is a subclass.
- Access specifiers have a certain precedence, when it comes to inheritance - *public, protected, package and private* in the same order. Thus, a method declared in a class using one of these access specifiers cannot be overridden by a child class method which has a *lower* access specifier. For example, public method in the parent cannot be overridden in the child by a private method, though the method signature remains same. Watch this live session.
- Similarly, when overriding a method in the parent class A in the child class B, return type of B's method can be a sub-class of the A's method.
- It's not possible to override a static method. Thus, even when the static method in parent is redefined in child class, only the static method in parent gets called. https://discourse.onlinedegree.iitm.ac.in/t/pa-q-14-how-can-a-static-method-be-called-on-an-object/105818
- If the child class redefines the private variable defined in the parent, it results in variable hiding.
- Following table summarizes the access modifiers and their visibility.

Specifier	Same class	Different class same package	Different package subclass	Different package non-subclass
private	Y	n	n	n
package	Y	Y	n	n
protected	Y	Y	Y	n
public	Y	Y	Y	Y

Week4

- Demonstrating use of abstract class (Week4/Demo3.java)

```java
abstract class FoodOrder {
    public abstract void order();
}

class Zomato extends FoodOrder{
    public void order() {
        System.out.println(x:"Zomato Order");
    }
}

class Swiggy extends FoodOrder{
    public void order() {
        System.out.println(x:"Swiggy Order");
    }
}

class Person {
    public void foodOrder(FoodOrder obj) {
        obj.order();
    }
}
public class Demo3 {
    Run | Debug
    public static void main(String[] args) {
        Person p = new Person();
        p.foodOrder(new Zomato());
        p.foodOrder(new Swiggy());
    }
}
```

  Here, FoodOrder is an *abstract* class with an *abstract order* method. It's implemented by Swiggy and Zomato class. The *order* method has been overridden in both of them. Thus, it can be used as *<Swiggy object>.order* or *<Zomato object>.order*. Note that when the *<Person object>* places an order, it's called using the *abstract <FoodOrder object>* and its *order* method. Remember, *FoodOrder* can represent *Zomato* and *Swiggy* in terms of capabilities.
- It's mandatory that all *abstract* methods in an *abstract* class are implemented in the child class, else compiler will throw an error. Alternatively, declare the child class also as *abstract*.
- *Abstract* class can't be instantiated. Only concrete implementations (extended from the abstract class) can be.
- *Abstract* classes can have constructors, which is used when its concrete subclass is initialized.
- *Interface* cannot define constructors of its own; only its implementations can.
- Its not necessary that abstract class have any abstract methods at all. It's possible that all methods are concrete.
- Interfaces can only have static, default or abstract methods defined in it. No concrete methods are allowed.

```java
interface Interface {
        abstract void concrete_method();
        static void static_method() {
                System.out.println(x:"static method in interface");
        }
        default void default_method(){
                System.out.println(x:"Default method in interface");
        }
}

class Concrete implements Interface {
        public void concrete_method() {
                System.out.println(x:"Concrete method in interface");
        }
}
public class Test{
        Run | Debug
        public static void main(String[] args) {
                Interface.static_method();
                Concrete c = new Concrete();
                c.concrete_method();
                c.default_method();
        }
}
```

- In order to use a *static* method defined in an *abstract* class or *interface*, it's not necessary to sub-class.
- Class can be declared as *private* only inside *public or package* classes.
- Abstract methods cannot have *final* access specifier. This is because, *abstract* method must be overridden, and *final* prohibits that.
- All methods of an *interface* are abstract by default. So, no need to specify that during the definition.

- All methods in an *interface* must be redefined in the child class, unlike abstract class where some methods might already have implementations. However, it's not necessary to implement *default* methods defined in the interface. If re-implemented in the child class, it'll be used, else the *default* implementation of the interface is used.
- If the child class *implements* two interfaces A and B, both of which have the same *default* method signature, it must be reimplemented in the child class. This is because, there would be ambiguity as to which method to use, otherwise. In the reimplemented child class method, you could use the *default* method from A, using *A.super.<method>*. Similarly, to use the *default* method from B, use *B.super.<method>.*
- The implementation of an *abstract* class/interface can have more methods than in the original *abstract* class/*interface* itself.
- All data members of an *interface* are *final* by default, and hence can't be altered during implementation. So, either do not declare any variables in the *interface*, or initialize them when they're declared. This is not true with *abstract class* – data members are not *final* in *abstract class* and can be modified in the child class.
- Interface can be *extended* to create another interface. Interface can be *implemented* to create a class.
- Use a private class if the interaction needs to be controlled. For example, if the *getStatus* method needs to be available only for the logged-in users, in the *login* method return an object of a private *QueryObject* class, iff the user is successfully logged in. Note that *QueryObject* class has a method *getStatus,* thus making it available only to those users who can create *QueryObject* object, which in turn is possible only for those who could successfully login using *login* method, not to everyone.

```
public interface QIF{
  public abstract int
    getStatus(int trainno, Date d);
}

public class RailwayBooking {
  private BookingDB railwaydb;
  public QIF login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
       qobj = new QueryObject();
       return(qobj);
    }
  }
  private class QueryObject implements QIF {
    public int getStatus(int trainno, Date d){
     ...
    }
  }
}
```

- *QueryObject* class is typically implemented from an interface – in this case *QIF.* This is required such that the user knows about the existence of this capability.
- To expose the method of private class *Inner* to outside its containing class *Outer*, define an interface *A* and ensure that that class *Inner* implements it. Consider the following code, where *obj.fun()* in the last line works only because class *Inner* is implementing the interface *A*. If that was not the case, *obj.fun()* would give compiler error. Note that *fun2* is not available outside the *Outer* class, because it's not part of the interface *A*.

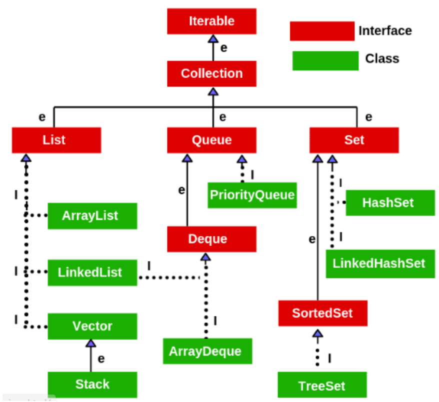
```java
package test;
interface A{
    public abstract void fun();

}
class Outer{
    private class Inner implements A{
        public void fun() {
            System.out.println("Hello");
        }
        public void fun2() {

        }
    }
    public A getreference() {
        return new Inner();
    }
}
public class Test8 {
    public static void main(String args[]) {
        A obj = new Outer().getreference();
        obj.fun();
    }
}
```

Week5

- In Java, you can convert an integer to a string using the *Integer.toString(int)* method or by using *String.valueOf(int)*
- Generic arrays cannot be instantiated directly, but only through a method.
- Generics are NOT covariant. Assigning an integer array into a reference variable for object array works fine. But, not if the array class is generic. In this case, Array<Integer> can only hold Integer values; in order to hold String values, Array<String> needs to be constructed separately.
- Use of generics in a class must not clash with the method defined in it. Thus, <T> used at the class level shouldn't be used again to make a method generic.
- Use of <T> generalizes the class or method to Object. If you want to bound to a specific class (say *Number*), use *<T extends Number>.* If you want to bound to a specific interface, usage remains the same (there's no *implements* keyword in generics)
- However, in the above example, the sub-class information is used only during compile-time to check if the object indeed belongs to a sub-class of *Number*, but it's always promoted to upper-bound during the run-time. This feature is called *type erasure.* This means that using code such as *<obj> instanceof T* will fail to work.
- Wild-card (?) can be used as a generic type in method definitions, if the type doesn't need to be used inside the function body. However, wild-cards cannot be used with class generics.
- In the case of a generic, *getClass* always returns the original class.
- To get the class given the class name, use *Class.forName(<classname>)*
- *getMethods* returns the list of all public methods of the class, including those from the ancestors. To include private methods into the list, use *getDeclaredMethods.*
- *getFields* returns the list of all public fields of the class, including those from the ancestors. To include private fields into the list, use *getDeclaredFields.*
- *getConstructors* returns the list of all public constructors of the class, including those from the ancestors. To include private constructors into the list, use *getDeclaredConstructors.*

Week6

- Queue interface has a poll (returns and remove the first element in a queue) and peek (return the first element in a queue, without removing it) functionalities.
- *java.util* contains the classes and interfaces of the Collection framework.
- Interface adds a level of indirection.
- Interface can be used to choose between multiple concrete implementations.
- List, Queue and Stack are examples of abstract data types.
- An abstract data type separates the interface from its implementation.
- Inappropriate choice of abstract data types can make the program inflexible and unproductive.
- Numerous data structure are available, so programmers can choose the appropriate data structure that suits the requirement.
- Shown below is the availability of interfaces and concrete implementations in *collection* hierarchy.



- *Collection* interface has sub-interfaces *List, Set and Queue.* Similarly, *AbstractCollection* has implementations *AbstractList, AbstractSet and AbstractQueue.*
- *AbstractSequentialList* extends *AbstractList. LinkedList* is a concrete implementation of *AbstractSequentialList.* Note that the *get()* method of *AbstractSequentialList* isn't efficient.
- *ArrayList* is an implementation of *AbstractList.*
- *ArrayList* can grow and shrink in size, similar to Python lists.
- *ArrayList* takes only objects. Thus, in order to create an *ArrayList* of integers, use *Integer* wrapper.
- *ArrayList* provides an add() method, but *Array* do not.
- *Collection* interface comes with a lot of methods, and needs all of them to be implemented in a concrete class. Hence, often *AbstractCollection* is used to implement collections. *AbstractCollection* is a skeletal implementation of the *Collection* interface. Note that *AbstractCollection* comes with its own implementation for many methods, but *iterator()* and *size()* are not provided and hence must be implemented in the concrete implementation using an *iterator* interface.
- An *Iterator* uses *hasNext()* and *next()* methods to iterate over a *Collection* object.
- A *ListIterator (extended from Iterator)* uses *hasPrevious()* and *previous()* methods, in addition to the *hasNext()* and *next()* methods to iterate over a *Collection* object.
- *for-each* loop can be used to avoid explicit iterator.
- Using *remove()* on an iterator created from an underlying *Collection* object, also removes it from the object, provided *next()* is called prior.
- *Vector* class is by default synchronized, but not *ArrayList* class.
- Non-generic *ArrayList* can hold any type of objects.

- *ArrayList* has two versions of *add()*
    - appends (part of *Collection* interface) . Returns a Boolean indicating if the operation is successful.
    - inserts (part of *ListIterator* interface). Returns a void.
- *Set* is unordered, and stores its contents using hash functions. It's similar to *AbstractCollection* interface. It allows for efficient membership tests, and returns *false* upon using *add(*) method
- *HashSet, TreeSet, LinkedHashSet* are implementations of a *Set*.
    - *HashSet* doesn't guarantee order of its items.
    - *TreeSet* keeps a comparable order of the objects added to it. Objects added to a *TreeSet* need to implement *Comparable* interface.
    - If we need to order the elements of a *HashMap* in insertion/access order, use a *LinkedHashSet*.
- Elements could be duplicated in *List and Array*, but unique in *Set*.
- Elements could be added anywhere in a *List, Array or Set*, but in *Queue* could be added only at the start or end.
- *Queue* interface supports *add()* (or *offer()*),*remove()* (or *poll()*) and *peek()*. *offer()* returns a Boolean true if the addition succeeds, false otherwise. *remove()* returns null, if operation fails, else returns the removed element. *peek()* returns null if there's no element in the head.
- *ArrayList* implements a queue*; ArrayDeque* implements a deque*.*
- Shown below is the availability of interfaces and concrete implementations in *Map* hierarchy*.*

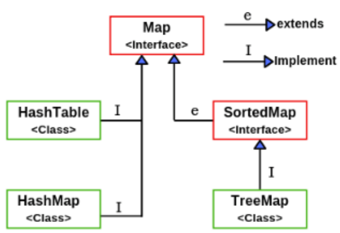


Fig: Map Hierarchy

- *Map* interface has the following definition.

```
public interface Map<K,V>{
  V get(Object key);
  V put(K key, V Value);

  boolean containsKey(Object key);
  boolean containsValue(Object value);
  ...
}
```

  Note that the *put()* requires *key* of type K and *value* of type V. But, rest of the functions in the interface accept Objects.
- *put()* returns V, because it returns the existing value, in case it's being overwritten.
- *Map* interface provides a *getordefault()* in addition to *get()* method. If the specified key is not present in the map, then it returns the default value provided as the second argument. Works similar to the Python dictionary's *get()* method. This is a typical usage of the method*.*

```
scores.put(bat,
  scores.getOrDefault(bat,0)+newscore);
  // Add newscore to value of bat
```

  Alternatively, use *putIfAbsent(bat, 0)* method, followed by *get(bat)+newscore*

- Following methods exist to extract the key and values from a *Map*. All of them are referred to as views. Note that the last one return *Map.Entry* objects, which provides *getKey()* and *getValue()* methods.

```
Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K, V>> entrySet()
```

- *HashMap, TreeMap and LinkedHashMap* are implementations of *Map* interface.
    - *HashMap* stores the key and value at a position, because of the clashes that can occur as a result of the hash function. It's not guaranteed that the *HashMap* maintains the order of key-value pairs.
    - *TreeMap keeps a comparable order of the objects added to it*. Objects added to a *TreeMap* need to implement *Comparable* interface.
    - If we need to order the elements of a *HashMap* in insertion/access order, use a *LinkedHashMap*.
- It's possible for a *HashMap* key to be null, unlike in Python. However, only one of the keys can be null. Note that *TreeMap* doesn't allow null keys.
- It's possible for a *HashMap* key to be a set or a list. It's not necessary for keys to be immutable, unlike in Python.

Week7

- *Throwable* is the root class for *Exceptions*. *Errors* also inherit *Throwable*.
- *Exceptions* can be Checked or Unchecked. Unchecked exceptions are called *RuntimeExceptions*.
- Checked exceptions are conditions that a well-written application should anticipate and recover from. By forcing the programmer to deal with the possibility that the exception will be thrown, the Java language helps to ensure that these conditions do not cause the program to fail unexpectedly at runtime.
- Following are some examples of Checked exceptions
  IOException, FileNotFoundException, ClassNotFoundException, InterruptedException, NoSuchMethodException, NoSuchFieldException
- Unchecked exceptions, on the other hand, are exceptions that occur at runtime and that you cannot reasonably be expected to recover from. They are usually identified as a programming error, such as logic errors or improper use of an API.
- Following are some examples of Unchecked exceptions
  NullPointerException, IllegalArgumentException, ArrayIndexOutOfBoundsException, ClassCastException, ArithmeticException, NumberFormatException
- *Errors* indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions.
- Given below are examples of Errors
  OutOfMemoryError, StackOverflowError, NoClassDefFoundError, AssertionError, InternalError
- Functions, in their signature, should indicate if they're capable of throwing exceptions. The exception type used in the method signature could be a super-class of all exceptions it can throw.
- The caller of the function must handle the exception by using a try-catch block or passing it to its parent by declaring so in its own signature. Note that this is mandated for all *checked exceptions* by the compiler.

- If checked exceptions are left uncaught in code (or the containing method *throws* the exception in its signature), compilation will fail. If unchecked exceptions are left uncaught in the code, it will crash the program.
- When an exception is thrown, it can use the initCause() method of the *throwable* class to include the details of the exception. When the caller of the function catches this exception, it can use getCause() method to get the details.
- *Try* block could have multiple *catch* blocks, which are executed in sequence. Each *catch* block could be associated with a specific Exception type. Alternatively, multiple Exceptions could be handled in the same *catch* block by using '|' character between them. If the generic exception is handled before the specific exception, it'll reach unreachable code, and compiler will throw a warning about it.
- *Finally* block always gets executed in a try-catch scenario, unless there's a call to *System.exit()* in any of the preceding blocks.
- If the *try* or *catch* block have return statements, the *finally* block gets executed before either of them.
- It's possible to define a *try* block without *catch* block, in which case a *finally* block can occur immediately after *try* block. But, note that it's not legal to have a *catch* block after the *finally* block. The order is always *try-catch-finally.*
- All methods and variables defined inside a public class is visible to all. All methods and variables that don't have an access specifier are visible within the same package (same containing directory). All methods and variables defined as protected are visible to all in the same inheritance tree.
- The package definition should appear in the first line of the Java program, else compilaton will fail.
- Using a reserved name for the package (eg. java) will not raise a compile-time error, but *SecurityException* during runtime, while trying to access the package classes/methods.
- Assertions are a run-time capability of Java to check if the given predicate is true. If false, it fatally exits the program.
- *AssertionError* is a subclass of Error.
- *assert(x < 100):"Illegal value";* prints *"Assertion Error: Illegal value"* and exits the program if *x >= 100*. Note that none of the code written after the assert line is executed, in this case. However, if the *assert* inside a *try* block fails, the *finally* block still gets executed.
- Assertions are disabled by default, and can remain in code, unlike exceptions. They can be enabled while running using *-ea* switch. It can also be enabled for specific classes only (*java -ea:Myclass*) or specific packages only (*ea:com.mypackage*). To disable assertions, use *-da* switch. Use *-esa* switch to enable assertion for all system classes.
- Using assert statements as replacement for conditional statements is a bad practice because an assert statement may or may not be executed at run time.
- Logging is a capability used to log diagnostic/audit information while the program runs, using *Logger.getGlobal().info(<Log message>).*
  To suppress logging, use *Logger.getGlobal().setLevel(Level.OFF)*
- There are 7 levels of logging - SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST.
- By default, the first 3 levels (SEVERE, WARNING, INFO) are logged.
- Create a custom logger *myLog* using
  private static final Logger myLogger = Logger.getLogger("top.next.bottom")

Note that in this example, *top* is the *highest* class of *mylogger*, *next* inherits *top*, *bottom* inherits *next*. THus, if *top* package's logger is set to INFO, then *top.next* and *top.next.bottom* are set to INFO by default. However, it can be changed by using *setLevel* method.

- *Logger.getGlobal().setLevel(Level.FINE)* will log everything above *FINE* level logged into global log. However, logging anything less than INFO will happen only if separate handler is defined. By default, console handler is set.
- Here's how to set the custom logger *logger* to FINE (follow the same process for any value less than INFO)

```
Logger logger = Logger.getLogger(MyClass.class.getName());
MyHandler handler = new MyHandler();
logger.setLevel(Level.FINE);
handler.setLevel(Level.FINE);
logger.addHandler(handler);
```

  Note that handler's level also needs to be set to FINE or lower to see FINE level messages at *logger*. Further, the parent logger's level must also allow FINE messages.
- The log level settings can be changed external through configuration files.

Week8

- The *Cloneable* interface in Java is a marker interface, which means it does not contain any methods.
- The *clone()* method, which is used to create the clone of an object, is actually a method of the *Object* class. A class must implement the *Cloneable* interface and must define *public object clone()* if we want to create the clone of an object of the class.
- If a class doesn't support the *Cloneable* interface, then the clone() method generates the *CloneNotSupportedException*. Hence, the method signature must have *throws CloneNotSupportedException*.
- Java compiler (Java 9+) infers the types of local variables (including objects and non-objects), when declared using *var* keyword.
- Type inference does not work when used in the following contexts - instance variables, method parameters, return values of methods.
- Type inference works only when the initialization is done in the same line as declaration. Thus, *var a; a = 2;* fails to work.
- In the case of objects, the inferred type is always most constrained type. Thus, if *class Manager extends Employee*, *var e = new Manager()* infers *e* to be *Manager*, and not *Employee*.
- Type inference will not work when the variable is initialized to *null.*
- A functional interface is an interface that has exactly one abstract method in it. THere could be more than one default/static methods in it, but as far as the interface has exactly one abstract method, it's a functional interface.
- Higher order functions are functions that take other functions as parameters. There are a few standard techniques used to define such functions.
  - Define a functional interface, and implement it in a class. Create an object of this class and pass it to the higher-order function as argument.
  - Create a lambda expression, wrap it in a functional interface.   Now, it can be used instead of the function defined in the interface.

```java
//This is an example of lambda expression
//used in a higher order function.
interface X {
    void message();
}

public class Test {
    Run | Debug
    public static void main(String arg[]) {
        X ref = () -> {
            System.out.println(x:"Interface X");
        };
        ref.message(); //Interface X
    }
}
```

- A slightly more complicated example

```java
interface ArrOperations<T extends Number> {
    public abstract void display(T[] arr);
}

public class Test {
    Run | Debug
    public static void main(String[] args) {
        Integer[] a = new Integer[3];
        a[0] = 12;
        a[1] = 13;
        a[2] = 14;
        ArrOperations<Integer> arr = (s) -> {
            for (int i = 0; i < s.length; i++) {
                System.out.print(s[i] + 2 + " ");
            }
        };
        arr.display(a);
    }
}
```

- Use a method reference (<Class>::<method>), wrap it in a functional interface. Now, it can be used instead of the function defined in the interface.

```java
//This is an example of method reference.
//THis is possible because AQ3_5_Example::show()
//has the same signature as AQ3_5_X.display()
interface X{
    boolean display(int a);
}
class Y{
    public static boolean show(int b){
        return b>10;
    }
}
public class Test{
    Run | Debug
    public static void main(String[] args){
    X ref=Y::show;
    System.out.println(ref.display(a:1)); //false
    }
}
```

- Here is an example of how to use lambda expressions to define *checkEligibility* for two unrelated classes. (NOTE: Being unrelated, it won't make sense to inherit one from the other or even an interface).

```java
interface Eligible<T>{
    boolean checkEligibility(T obj);
}
class Voter{
    int age;
    Voter(int x){
        age = x;
    }
}
class Account{
    int overdraftlimit;
    Account(int odl){
        overdraftlimit = odl;
    }
}

public class test15 {

    public static void main(String args[]) {
        Voter v1 = new Voter(23);        // eligible bcoz > 18
        Account a1 = new Account(12000); // not eligible bcoz > 10000

        Eligible<Voter> fn1 = (obj) -> {return obj.age > 18;};
        Eligible<Account> fn2 = (obj) -> {return obj.overdraftlimit < 10000;};

        System.out.println(fn1.checkEligibility(v1));
        System.out.println(fn2.checkEligibility(a1));

    }
}
```

- Lambda expressions are anonymous functions that take the form - (p1, p2,…) -> {<function body>}.
- A stream can be created out of any collection object, using its *stream()* method
- *stream* has a method called forEach, which acts on each item in the stream.
- A stream can be transformed using one of *filter(), map()*. Transformation returns a stream.
- Now, the final step of a stream pipeline is an aggregation, like *count(), reduce(), max(), min()* etc.
- At any of the steps, the output can be empty, which is indicated by the type *Optional* or *Optional<T>*
- To convert a stream of items in array into arraylist, use *Stream.of(<Array>)*
- *takeWhile()* method of stream continues to allow elements in the stream as long as the predicate is true, but stops as soon as the predicate is false.
- *dropWhile()* method of stream starts to allow elements in the stream, only when the predicate is true. Once it starts, it continues till the stream ends.

Week9

-

Week10

- There are two ways to implement threads in Java.
  - Here, *AQ1_2_Example* class extends *Thread* and defines *run()* method. Start the thread from the *main()* method with the *start()* method.

```java
class AQ1_2_Example extends Thread{
    public void run(){
      for(int i=1;i<3;i++){
          System.out.print(i+" ");
       }
    }
 }
 class AQ1_2{
     Run | Debug
     public static void main(String[] args){
       AQ1_2_Example e=new AQ1_2_Example();
       e.start();
       for(int i=3;i<=5;i++){
          System.out.print(i+" ");
       }
    }
 }
```

  - Here *Demo1_Example* class implements *Runnable* and defines *run()* method. Create a thread using the *Demo1_Example* object, like ***Thread t=new Thread(e);*** Use ***t.start();*** in order to start thread *t*.

```java
class Demo1_Example implements Runnable {
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println("Child.." + i);
        }
    }
}

public class Demo1 {
    Run | Debug
    public static void main(String[] args){
        Demo1_Example e = new Demo1_Example();
        Thread t = new Thread(e);
        t.start();
        for (int i = 3; i <= 5; i++) {
            System.out.println("Main.." + i);
        }
    }
}
```

- In both examples, ***start()*** method starts the thread execution. Once the thread starts execution, it interleaves itself with other threads or the main thread. Thus, the programs could result in many possible combinations of numbers 1-5. Note that, it's guaranteed that the 1 will be printed before 2, and 3 will be printed before 4 and 5. THis will result in 5!/(2! * 3!) combinations.

- Calling *start()* on a thread more than once will result in *IllegalThreadStateException*
- To use *Thread.sleep(<millis>)* method, the containing method must handle or throw *InterruptedExecution*.
- Unless provided by the user, *getName()* method on the first thread will return Thread-0. SImilarly, unless set by the user, *getPriority()* method will return the default priority of 5.
- Priorities of threads range from 1 to 10, 1 being the lowest priority and can be set using *setPriority(<priority>)* method.
- When working with two (or more) threads with shared variables between them, Concurrent update of a shared variable that may lead to data inconsistency. This is called a race condition.
- To avoid race conditions, we must define critical sections in the code. However, this could lead to starvation (where the second thread is not releasing the request to run) or a deadlock situation (where both threads are waiting for each other).
- To deal with these issues, there're at least two algorithms (Peterson's and Lamport Bakery), but generalizing them and implementing them with more than 2 threads is not easy. Hence, we'll look for *test and set* atomic logic built-into the programming languages.
- Semaphores ensure that an exclusive *test (pass) and set* (release) is available. Semaphore is essentially an integer used to flag the operation state.
- In pseudo code, P(S) is

```
if (S > 0)
  decrement S;
else
  wait for S to become positive;
```

  and V(S) is

```
if (there are threads waiting
    for S to become positive)
  wake one of them up;
    //choice is nondeterministic
else
  increment S;
```

- Semaphores guarantee mutual exclusion between threads, no starvation and no deadlock, but a low-level solution to race conditions and is prone to programming errors.
- Monitors are built into Java and ensure inherent mutual exclusivity of its functions.
- To start with, all threads wait in an external queue. At any point in time, one of them (say *Thread-1*) is selected at random and granted access to the *monitor*. If it has to wait for a certain condition to be met (by running a different thread, say *Thread-2*), it must wait on its *internal* queue and allow the new thread access to the monitor. Once *Thread-2* has done its job, it must notify *Thread-1* on *its* internal queue. Once *Thread-1* steps in, *Thread-2* releases its access. *Thread-1* will continue, if the condition it was waiting for is met, else goes back to the queue again and *Thread-2* will continue.

Week10

- In Java, *synchronized* keyword is used to indicate a critical section. It can be used in the context of a method in a class, or an object (external or current object).
- In Java, when a method is declared with the *synchronized* keyword, it means that an intrinsic lock (also known as a monitor lock) is acquired on the object that the method belongs to, or on the Class object if the method is static. Only one thread can hold the lock on an object or Class object at a time. If another thread tries to enter a *synchronized* method of the same object (or the same Class

object in the case of static methods), it will have to wait until the first thread releases the lock. Thus, if M1 and M2 are synchronized methods of the same object (or are static methods of the same class), then T1 and T2 cannot enter M1 and M2 at the same time.

- When used in the context of an object, if the code blocks B1 and B2 are synchronized on the same object, then T1 and T2 cannot enter B1 and B2 at the same time
- However, it's important to note that while no other thread can access a *synchronized* method or block of the same object, they can access *non-synchronized* methods of the object. Also, other threads can still access *synchronized* methods or blocks of different objects.
- Java threads can be in any of six states –
    - New - created, but not yet started
    - Runnable - Started, but not scheduled
    - Blocked - waiting for lock
    - Waiting - suspended by wait, unblocked by notify
    - Timed wait - Thread.sleep
    - Dead – Thread terminated.

  If *t* is a thread, *t.getstate()* gets the state *t* is currently in.

- *t.interrupt()* interrupts thread *t* using *InterruptedException.* Client must use *interrupted()* to check if it's interrupted by another thread, but clears the interruption as soon as it checks it. Alternatively, use *t.isInterrupted()* to check if *t* is interrupted without clearing the interruption. In addition to this *interrupted* flag, *run()* must be wrapped in a try block and catch *InterruptedException*
- *InterruptedException* can be raised by *wait()* and *sleep()*
- *t.join()* can be used to wait until *t* finishes. If you call *t.join()* inside another thread *t1*, then *t1* is blocked until *t* finishes.
- Java provides thread safe collections that allow locks at a low-level. *BlockingQueue* is one such thread-safe collection that ensures none of the individual updates are lost. It will block the queue, if the queue is full and the thread is trying to add a new item, or when the queue is empty and the thread is read/remove an item from it. Note that the serializability of the transactions must still be ensured manually by the programmer.
- *ConcurrentHashMap* provides safe access to data structure without explicit synchronization.
- Use *Collections.synchronizedList(arraylistobject)* to synchronize ArrayList class externally.
- Use *Collections.synchronizedMap(hashmapobject)* to synchronize HashMap class externally.
- Use *Collections.synchronizedSet(setobject)* to synchronize HashSet class externally.

[OPPE Reference]