# Part 1: Intro

- Everything belongs in a class. The code flow is initiated from the main method of the Main class. (typically) It is convention to name methods with pascalCase and classes with CamelCase.

- Although everything is a class, there are 8 static datatypes interconvertible with their class equivalent. These are byte, short, int, long, float, double, character and boolean. Apart from the last 2, these fall under the parent class Number.

- System.out connects to the stdout stream. System.console() can read from stdin. Scanner(System.in) also does the same in a cleaner fashion.

- Semicolon acts as an expression delimiter. Newlines and spaces do not matter. Braces denote scope.

- Basic Syntax is very similar to C++:

- if(){} else if (){} else{}

- while(){}

- do{} while()

- for(int i=0; i<n; i++){}.

- for has 2 types: 1. Classic for, identical to while in functionality 2. for(int x : arr){} It’s possible to use a predeclared i for the classic for, but the second version only works with a locally scoped variable.

- switch case, no expressions can be used as a case argument, only constants are allowed. Control flows down without stopping if not broken explicitly.

char x='\\u03C0'; // Greek letter Pi

final float pi = 3.14159265f;

# Part 2: Classes

1. Classes contain 2 things: variables and methods.

- Variables (or fields), by default, belong to objects (instances).

- The static modifier makes a variable a class variable.

- The final modifier makes a variable impossible to change once instantiated.

- The private modifier makes a variable inaccessible outside the class. This means that class code can use private variables of all instances of the class.

- The protected modifier makes a variable accessible within its class and all its subclasses.

- The default behaviour of a variable is that it is accessible only within its package. With the public modifier it becomes openly accessible in any package.

2. Methods

- A method is described both by its name and the types and number of its arguments. This is called the signature of the function. A method can be defined multiple times with the same name but different signatures, so they are different entities. This is called overloading.

- The static modifier makes a method callable without an instance. (object)

- The final modifier makes a method impossible to override.

- If unspecified, a method is package level. It’s not possible to call it outside the package. Default (Package), Private, Protected and Public behaviour is same as variables.

3. Constructors

- Constructors are special methods that bear the name of the class. Their definition does not have a return type. Every constructor calls the default constructor of its parent class, if the constructor in question does not explicitly invoke a super constructor. The keyword this(args) is used to call alternate constructors of the same (self) class. This can be used to chain constructors. The constructor chain leads till Object.

4. Polymorphism and Subclassing

- A variable of static type of class A can be instantiated to be an object of any subclass of A.

- Static typing prevents the invocation of any method not available to a class A by an object of static type A, even if it may in actuality be a subclass. To call such a method, typecasting is done.

- If a method is invoked by an object A, then, regardless of the static type, the most specific version of the method available to the object A is used. This is called runtime polymorphism.

- When a method is called, it will execute successfully only if the arguments are of the correct types. It is acceptable for an argument of class A to be replaced by an object of a subclass of A.

- When a method is called, and there are multiple *overloaded* functions with the same number of arguments, fully compatible with this function call, that function which is most specific to the **static types** of the arguments is used. (Recall boolean equals(Object o))

- **A static method cannot be overridden.** It can, however, be redefined. If an object is used to call a static method (which it should not,) then the one belonging to the static type of the object is used. No polymorphism.

- All classes are subclasses of the class Object.

- runtime/functional polymorphism refers to how a collection of objects each know what they should do although a single common method is called on all of them. The single method is taking many forms.

- Structural polymorphism refers to how a piece of code only expects its arguments to conform to a certain structure, and if it does, then it can act upon the objects, regardless of what else they may be. The single method is acting on many objects, thus, in a way, possessing many forms, akin to overloaded functions.

# Part 3: Interfaces

1. Methods

- An unimplemented method can be declared in a class with the keyword abstract. This renders the class incapable of spawning any objects ever. The class must also be given modifier abstract.

- An interface is an unimplemented class with almost entirely abstract functions. Several updates to Java allowed more methods to be permitted in interfaces:

* Default methods are inherited by all who implement the interface. They need not be overridden.
* Static methods are also inherited. They *cannot* be overridden, only redefined.
* Private methods are merely for the purpose of encapsulating code to serve static and default methods. They are inaccessible outside the interface.

- Any non default/static function is automatically abstract in an interface, so the keyword is rarely used before the methods. Similarly, any non private method is public and this cannot be changed.

- The following is a stream of nonsense; pause for a second and the absurdity must be obvious: private default, default static, private abstract, static abstract, default abstract

2. Inheritance rules

- An interface can **extend** any number of interfaces.

- Any number of interfaces can be implemented by a class. All abstract functions in all interfaces **must**\* be implemented concretely in this class.

\* Caveat 1: If an interface A has an abstract function f, and another interface B has a *static* function f, then an a class C that implements both A and B does not have to provide an implementation for f since it inherits the static function from B.

- However, if instead of static, the method in interface B was default, class C must necessarily provide its own implementation and override the default method. (This can be seen as a consequence of default being created to be an “optional placeholder”.)

\* Caveat 2: If a class C that extends a class B and implements an interface A, inherits a function f from B which A has as abstract, then the implementation need not be done in C. However, this function in B must be public to avoid the implementation, as the interface specifically requests for a public function.

3. Conflict Resolution

- if a class C inherits default function f from both interfaces A and B, then C must implement f.

- if f were static, then there is no such requirement, since an interface’s static method can only be invoked with if\_name.f() and not with an object.

- if a class C inherits default function f from interface A and f of same signature from class B, then C happily uses the class B one, since default functions are made to be overridden.

- if f were static, then there is no issue. Object.static\_method (which is bad practice,) will call the class version, since interface static methods***cannot*** be invoked by instances.

- Note that this is comprehensive; (static, 2 interfaces) conflict and (static, class-interface) don’t matter; (default, 2 interfaces) needs implementing, and class wins in (default, class-interface).

4. Common uses of Interfaces

- Encapsulating the public methods of a private class, so that external code can interface with it.

- Acting as an umbrella data type that identifies a class to possess a particular functionality. This is key to implementing Structural Polymorphism.

(“Structural polymorphism refers to how a piece of code only expects its arguments to conform to a certain structure, and if it does, then it can act upon the objects, regardless of what else they may be.”)

- An interface with exactly one abstract function embodies exactly one functionality, so it is called a functional interface. “Anonymous” classes (similar to anonymous functions in JS) implementing functional interfaces can be created elegantly using lambda expressions.

- Notable examples: Iterator and Iterable interfaces; The Runnable interface and Thread class;

Addendum: If a class can be implemented internally in multiple ways, and each one has its own pros and cons, one would want multiple classes to choose between these implementations. However, this means that if one wants to swap implementations they must find every mention of this class and carefully change it. Instead, a common interface with all the expected properties can be defined, and that can be used everywhere for object static types or function return types, and then the instantiation alone needs to be changed to swap implementations.

In this sense, an interface is allowing you to interface with an abstract concept and adding a level of “indirection”. You can say that it will be implemented by someone, and write code on top of it. This example not only shows the broader picture of the first point I mentioned, but makes the keywords used a lot more intuitive.

# Part 4: Generics

- Methods and classes defined purely with Object class arguments in order to be general potentially result in runtime errors. Relationships between arguments cannot be enforced, and objects of only class Object can be returned, causing irritating typecasting which could also cause errors at runtime.

- To solve both these problems Generic types were introduced.

- A parameter type can be declared along with a class, which makes the class a “parametrised type”.

- A parameter type can be declared along with a method, which allows a generic return type and input pattern along with convenient types in the function body. If the parameter shadows a class parameter, the method parameter is used, and the class parameter is forgotten.

- Internally, Generics are implemented by Type Erasure. This means that the class and method definitions’ parameters are all replaced with the type Object, (or the closest bound class,) and any returned objects are auto typecast. Due to erasure, parameter classes cannot call methods not present in their closest bound class, including constructors.

- Further, functions cannot be overloaded if their signature differs only in generics. printAll(LinkedList<Date>) and printAll(LinkedList<Money>) is not allowed, since there is only LinkledList<Object> at runtime, nothing else.

- Scalar datatypes are allowed to exist despite Java being an OOP language, but due to incompatibility with the rest of Java, wrapper class with Capitalisation are used in their place. These are autoboxed by Java for convenience. The class names need to be used when dealing with Generics. (eg: LinkedList<Number>)

# Part 5: Runtime

- Reflection is the ability of code to introspect at runtime and choose to make changes in the control flow based on that.

- myobj.getClass() returns an object of type Class. It has substructure comprised of Field, Method and Constructor<> objects. Some notable methods are:

* + myclassobj.getMethods()
  + myclassobj.getFields()
  + myclassobj.getConstructors()
  + mymethod.invoke(args)
  + myfield.get(myobj)
  + myconstructor.getParameterTypes()

- Exceptions are major errors in the code that can potentially be mitigated and accounted for. In case of a crash, they act as debug info with rich structure.

- All Exceptions extend the Throwable **class**. Notable fields of this class include cause and detailmessage, both of which are **private**.

- cause is a Throwable object, allowing a chain of errors to be analysed. This can be set using the initCause method, and accessed using the getCause method.

* + detailmessage is necessarily filled when a Throwable object is constructed. Common constructors include Throwable(message), (cause) and (message, cause). detailmessage is accessed with getMessage().

- An Exception invoked by the programmer is known as a checked exception, as opposed to RunTimeExceptions which just happen. If any exception is deliberately thrown in a method, that method’s header line must end with a list of all of them.

modifers returntype f(args) throws E1,E2...{}

- Exceptions can be caught with a try block, and they can be handled one at a time with catch blocks. These are checked top down, which must be kept in mind when dealing with Exceptions which are subclasses of other Exceptions. The equivalent of the bare except of python is catch(Exception e). An identifier for the exception must always be provided, unlike python.

- Assertions are used to flag fatal errors during testing phases. Assert lines do not execute unless a –ea flag is used when the file is executed from the CLI.

- Logs are used to flag and record non fatal concerns or notable information. Logs come in 7 levels: SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST. By default, the first 3 levels are logged.

- Logs are made by Loggers. If configured to do so, a logger simply writes a log to file (unconfirmed). All loggers except root forward the log message to their parent logger and their own handler(s), if they exist.

- A log statement will not cause a log if it does not clear the filter of the logger. It will not be printed to stdout if it doesn’t clear the handler’s filter.

- If a logger’s level is set to null, then it effectively has the same level as its first parent with a level, since it will be the first one to intercept the message. A handler can receive the message before this as well. (null lets all logs through and is the default value of the level of a newly created logger.

- Some useful commands:

Logger.getGlobal() to get the root logger.

Logger.getLogger(“name\_here”); in general (making new Loggers with the Logger(string\_name) constructor is not possible as that is a private constructor.)

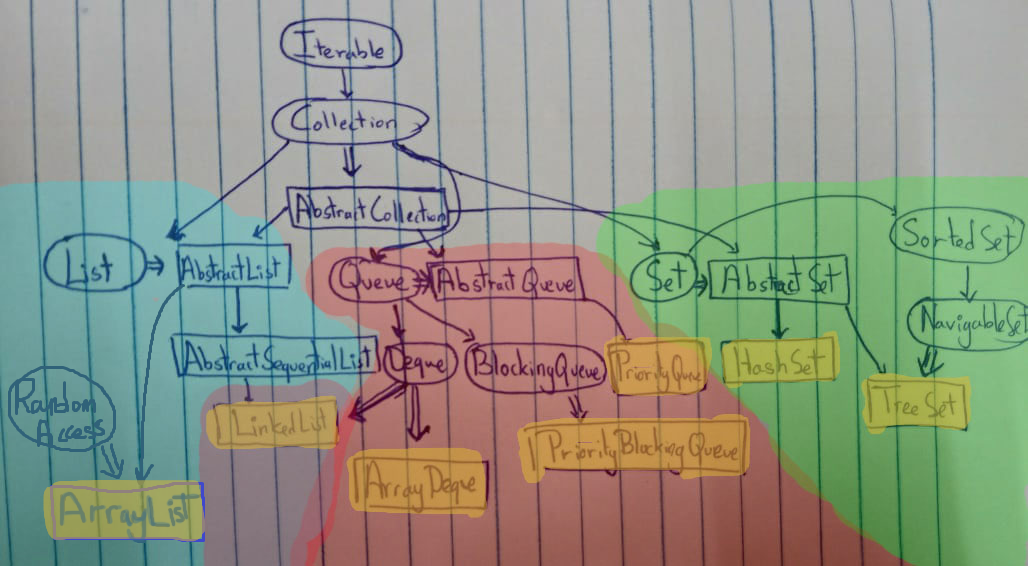
my\_logger.setLevel(Level.FINER);

my\_logger.getLevel() ;

my\_logger.getParent();

my\_logger.getHandlers();

# Part 6: Collections



(I) Collections

- 4 core methods: add, remove, contains, iterator. Tons of other crazy methods.

- 3 Great children : Set, List, Queue.

1. Lists

- can be RandomAccess or not (sequential access)

- has a separate listIterator method, which can traverse in both directions.

- Concrete random: ArrayList; Concrete Sequential: LinkedList, which is also a deque.

- Ironically, the standard array used does not fall under collections; it is a standalone class called Arrays.

2. Queue

- add, remove only work on tail and head, respectively. Gentler version that does not raise exception: offer and poll.

- Inspection of head is done with element, gentler version is peek.

- Deque has all these 6 methods for both ends.

- Priority Queue’s remove method always removes the highest priority item.

- Blocking Queue’s take and put methods wait until an object is available to take, or until room is available for adding, and then perform the operation. It operates on many threads at once, and is said to be “[Thread safe](https://learn.microsoft.com/en-us/archive/blogs/ericlippert/what-is-this-thing-you-call-thread-safe)”, in some sense.

3. Sets

- add works only if the element doesn’t already exist in the set. Set equality works without considering order. Element presence can be checked very rapidly.

- HashSet uses a hash table to implement set, the most common type.

- A sorted set maintains some ordering; it’s possible to construct a complex hashed set which has double pointers to other elements allowing ordered traversal. The concrete implementation of that is called a LinkedHashSet. One more concrete implementation is the TreeSet which uses a balanced binary search tree to implement SortedSet.

- Note that TreeSet uses the compareTo method (from the Comparable interface) between two elements to insert elements into itself, so the comparison equality is used to guarantee uniqueness. A HashSet on the other hand uses both the equals function and the hashCode function (both of these are defined in Object, and must be overridden) to check for equality: if the objects are equal and their hashes are equal, then an extra copy will be rejected.

(II) Maps

There are 10 main methods of interest:

- V put(K key, V value) : adds an entry to the dictionary, and returns null if it was successful. If the Key already exists, then the existing value V is returned instead.

- V get(Object key)

- boolean containsKey(Object Key)

- boolean containsValue(Object value)

These take Object type arguments for legacy reasons, and in order to support a broader definition of equality that can equate objects of type K with some other random class. This is a controversial design choice of Java which led to most implementations having to do many type checks in the implementations.

- V getOrDefault(Object key, V defaultValue) method. If the get key doesn’t exist in the map, then the default value is returned.

- V putIfAbsent(K key, V value). Puts only if key is absent in the map.

- V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)

If the key does not exist in the map, set value (argument 2) as the value. Else, use the given function (argument 3) to merge the existing value and the given value. In the generic type of the BiFunction, Arg1 is existing value, Arg2 is new value, and Arg3 is the return type.

- Set<K> keySet();

- Collection<V> values();

- Set<Map.Entry<K,V>> entrySet();

These are commonly used to iterate through the Map. Since all three of these objects have an iterator method in any implementation, a for each loop can be used on them.

The Map.Entry Class has 2 methods of interest: getKey and getValue.

There are 3 important implementations we must be aware of:

HashMap, TreeMap: the keySet of each of these return a HashSet, TreeSet respectively.

LinkedHashMap has two way pointers in the storage location, allowing ordered traversal in the order of insertion.

# Part 7: Features for Flair

1. Clone

- A clonable object must be tagged by the Cloneable interface. Any that implement it are expected to define clone, although not doing so raises only a warning. Any definition of clone is expected to call super.clone() although, again, not doing so only raises a warning.

- the original clone function that actually makes a bitwise copy returns an object of type Object. This must be typecast as needed in implementations of clone.

- the clone function can be overridden despite a new return datatype in its signature.

- any call to clone can raise a CloneNotSupportedException, and this means that the error must be caught in every call or the error must simply be thrown again in that method. This causes an irritating level of try catching during which one must reconsider the life decision of using clone.

- by default, a clone is a shallow copy. Care must be taken to avoid this.

- subclasses of a class that allows clone will most likely make shallow copies, as the most specific version of clone is that of a parent. To avoid this, it is good practice to override clone in the subclass with a method that just throws CloneNotSupportedException. But then again, it is best practice to just not use clone.

2. Type Inference

- only locally scoped variables, not object fields can be type inferred. To do so, simply use the var keyword instead of the static type. This is especially convenient in for each loops.

- A large amount of type inference can be used in the crafting of lambda functions.

3. Lambda Functions

- Often, methods need to take functions as arguments. However, this is not allowed, as only classes exist in Java. Thus, the function is housed in a “functional interface”.

Eg: Comparator<T> has one method, compare() which can be used to compare two objects. If a comparator is needed for a function, this will be the argument type.

- Essentially, we have a box, a function, and instead of passing around the box, we put it in a bigger box, called a class and then pass that around instead. Although on the level of principle there is no way to avoid it in Java, it can be averted in code.

- (arg1, arg2…) -> {yada yada return x yaay!} this represents a lambda function. It is a function that seemingly exists in isolation and has no predeclared return type. Such a function can be fed into a method that expects an object implementing a functional interface; the single function associated with that interface is the lambda function. Note that this means that the signatures must match.

- Explicitly, this is equivalent to declaring a class implementing the interface, overriding the abstract method with the function body, and then passing a dummy instantiation of said class into the function.

# Part 8: Streams & Optional

1. Making Streams

- Streams are means to handle sequences of arbitrary length, (even infinite) lazily.

- Stream<T> is the type of a stream of objects of type T.

- Stream.of(obj) is a means of converting any objection into a stream; Typically, this is an array. Alternatively, methods meant for streamifying within other classes can be used, such as the .stream method of List<>.

- Stream.generate takes a function argument, and uses the value of that function for each entry of the sequence. For instance, Stream.generate(math::random) is an infinite stream of random numbers.

- Stream.iterate() is used to create a dependent stream; It also takes a function argument, but this function takes one argument, which is the previous entry in the stream.

Eg: Stream.iterate(0, n ->n+2 ) is a sequence of even numbers. (0 represents the start value)

- Stream.iterate can also be used with 3 arguments, the second being a condition that determines when the stream must be terminated. This is a boolean function that takes n.

Eg: Stream.iterate(1, n -> n<100, n -> n+2) will give odd numbers upto 99. When a new number is generated, it must pass the check of n<100.

2. Modifiying streams

- the .filter method takes a function argument that takes in one argument and returns a boolean. Only elements that result in true are retained in the stream. Filter will go through every element in the stream.

Eg: Stream<String> longwords = wordlist.stream().filter(w -> w.length() > 10)

- the map function maps each entry to a new value.

Eg: Stream<Integer> squares = Stream.iterate(1, n-> n+1).filter(n -> n<100).map(n->n\*n)

Throws an error when it runs out of memory! Filter will look at every element; if you wanted to terminate the stream, use the 3 arg iterate, or equivalently, takeWhile(). Using a limit method will allow the code to actually finish executing, even when filter is used on an infinite stream.

- takeWhile() takes a function that takes one arg, returns bool; stream keeps evaluating elements till the function returns false. That false entry is discarded. dropWhile does the opposite.

- limit(n) takes only the first n entries of the stream.

- max and min take a Comparator function and return a single value from the stream.

- flatmap is an alternative to map that collapses list brackets, and returns a flat stream instead of a stream of lists or something of the sort. (this makes sense only if the function given to map returns a collection)

- any stream that has been filtered can potentially be empty. An empty stream is still a valid stream, but if a function like max is called on it, the result may be null. To account for this, an “Optional” type must be used for the return value.

3. Collecting values from Streams

- Stream is iterable, and thus, mystream.forEach(x -> func)

- To get an array out of a stream, use mystream.toArray(). This will return an array of objects by default, and to change the type of the element in the array, pass an array constructor into the toArray function, for example, String[]::new.

- mystream.collect(factory function) makes a collection out of a stream. Any collection of choice can be made with an appropriate factory method passed into it. Such a method will implicitly create an object and populate it with the entries in mystream. Eg: Collectors.toList(); More generally, mystream.collect(Collectors.toCollection(MyCollection::new)) can be used.

- It’s possible to get all the descriptive statistics about a stream in one go using summarizingInt. It’s used in a somewhat peculiar way: mystream.collect(Collectors.summarizingInt(x -> func))

The function is meant to convert the given stream into an integer stream, so as to use summarizingInt on the stream. This returns an object of type IntSummaryStatistics, which has the min, max, avg, count and sum, all accessible with getters.

Similarly, there exists the Double and LongSummaryStatistics classes with the corresponding summarizingDouble and Long methods in Collectors.

- groupingby partitioningby and toMap with a merge function

Optional Types

- Languages tend to handle scenarios where an optional type is needed with a null pointer. Null can be considered implicitly to be of any type, so any object might end up being null. This is true in java as well, and calling regular methods on null will result in a NullPointerException on *runtime*.

- Java also offers a much cleaner fashion of dealing with nulls, which is the Optional<> Type. It’s a box of type T, which can either have something in it, or just be an empty box. An empty box is very different from null; we can do a lot of cool things with it, like put something in it.

- .orElse(value) returns value in box. if box is empty, the value argument is returned instead.

- .orElseGet(function) returns value if present, else calls function to generate the returned value.

- .orElseThrow(throwable object *Supplier*) Eg: ExceptionNameHere::new; when the function is invoked it returns a pointer to an object of a class that extends Throwable.

- .ifPresent(v -> do something with v) Eg: add v to a list optval.ifPresent(results::add)

- .ifPresentOrElse(v -> do something with v, () -> do something in absence of v)

- .or(Optional *Supplier*) if value is not present, return the optional obtained from the supplier. The optional on which the method is invoked is unchanged, as is the case in most of the above methods.

- .map((v) -> func) the value is mapped, if no value, the optional remains empty. The outcome of map is an optional.

- .flatmap(func). If the return value of the function is in itself an Optional, map will store an Optional containing an Optional, a rather dumb thing to do. Flatmap removes the extra layer, and stores only the value of the optional returned by the function.

Making Optional Types

- When a functions return type is specified to be Optional<Something>, returning Optional.empty miraculously auto types to the correct type.

- Optional.of(value) returns the box with the item.

- Optional.ofNullable(value) transforms a null value into the empty optional.

# Part 9: Concurrent Programming

1. Java allows for multithreading, which, if used frivolously can lead to unexpected behaviour or inconsistent states. The cause of the problem is the fact that some critical components are meant to be accessed by exactly one process at once.

2. Java resolves this with the introduction of monitors; multiple methods can be tagged with the keyword synchronous, and java will see to it that no two of them ever execute simultaneously; They are made to wait their turn and execute one after another.

3. From a theoretical point of view, there are algorithmic solutions to the Mutual Exclusion (Mutex) problem: Peterson’s Algorithm for 2 processes, and Lamport’s Bakery Algorithm for n processes. These are very interesting to learn of and make no demands on the system, and as a consequence are not the most powerful things practically usable. It is possible to guarantee an atomic test and set operation by working at a very low level, and once this is done, Semaphores can be used for mutex.

4. Java allows the use of a Semaphore class, which is much more flexible: For instance, if we may wish for critical sections to be segments of code rather than entire methods; monitors ask that we just separate out the critical segments into their own methods, which may not always be convenient.

5. Another Java class that can establish mutex is ReentrantLock, and it is very similar to Semaphore, but the two have slightly different use cases.

6. wait(), notify() and notifyall() are used to implement an internal queue in Java.

7. BlockingQueue and ConcurrentMap are powerful thread safe data structures.