|  |  |  |  |
| --- | --- | --- | --- |
| **CS2030S: Programming Methodology II**  **© Isabel Chong 2022**  **Variable and Types**   * Java is a statically-typed (Variables have to be declared with specific type) and strongly-typed (Stricter, allow typecasting only if it makes sense). * Primitive: Stores actual value. * byte<:short <:int <:long <:float <:double * char<:int * Reference Types: Stores ref to the object. Two ref types can store the same value (aliasing).   + null: default value b4 instantiation.   **Functions**   * Advantages:  1. Isolate complexity within its body. Reduces dependencies between variables. 2. Reduces need to change (function is hidden by implementor) as software evolves reducing chance of bugs. 3. Reduce repetition and chance of bugs.  * Abstraction Barrier: Keep code between implementer and client themselves. * public final static void main(String[] args) {} * Method Signature: Method name + Number + Type + Order of parameters. * Method Descriptor: Method Sig. + Return Type.   **OOP (Pillars)**   * Something that instantiates objects of different classes and orchestrates their interactions with each other by calling each other’s methods. * Encapsulation (Class, Object, Composite Data): encapsulate fields and methods into a class. * Composition: HAS-A Relationship. * Inheritance: IS-A Relationship with extends keyword. \*Make sure it preserves subtyping. * Overriding: @Override. Use super to access overridden methods. * Overloading: Changing method signature of the same method. \*\*Polymorphism   + Tick outlinecontains(double) ≠ contains(point)   **Data Hiding**   * Access Modifiers: private and public. * public static final type * Constructor: method that initialises an object. * Constants/Static fields: public static final   **Wildcards and PECS**   * Upperbound is covariant   S <: T, A<? extends **S**> <: A<? extends **T**>  A<S> <: A<? extends S>   * Lowerbound is contravariant   S <: T, A<? super **T**> <: A<? super **S**>  A<S> <: A<? super S>   * Producer extends, Consumer Super * Unbounded <?>: Cannot type check * Differences:  1. Array<?> is an arr of some specific but unknown type. 2. Array<Object> is an arr of Object, with type checking. 3. Array is an arr of Object instances without type checking (Don’t use)   **Type Inferencing (Refer to Green Card)**  **Immutability**   1. Ease of understanding objects 2. Enable safe sharing of objects and internals 3. Enable safe concurrent execution (reduce bug)   **Nested Classes**     * Can access private fields and methods of the container class. * Rmb to private it if not useful outside.     **Anonymous Class** | **Tell, Don’t Ask**   * Getters and Setters: Breaks AB * Define methods from within the class itself and call it from the outside.   **Stack and Heap**     * Stack holds all instantiation of all variables. * Heap holds the object instantiations.   **Run-Time Type**   * CC c = new C(p, 0); // error given CC <: C * C is the compile type, CC is the runtime type.   **Polymorphism (Dynamic Binding)**   * Change code behaviour. Method runs on compile and run time type.   **Method Invocation**   * Compile Time: Chooses most specific method of compile type (method descriptor is stored). * Run Time: Looks for accessible method with run type. If not, go up class hierarchy until we reach Object..   **Liskov Substitution Principle (LSP)**  "Let ϕ(x) be a property provable about objects x of type T. Then ϕ(y) should be true for objects y of type S where S<:T."   * For Inheritance and Overriding * To prevent Inheritance: final class Circle {} * To prevent Overriding: final public int m(int x) {}   **Side Effect-Free Programming**   * Functions: f(x) does not change the value of x and if f(x) = a, they are always equivalent. * Pure Function does not cause side effects and is deterministic (same result every time which ensures *referential transparency*)   + Modifying a field, printing, write to files, throws exceptions, modify argument values.   Eg. Pure Functions    Eg. Non-Pure Functions    **Functional Interface (#30)**   * Interface with a single abstract method. * @FunctionalInterface annotation.   Eg1.      Eg2.        Lambda Examples: | **Abstract Classes**   * Something so general that should not be instantiated. * *Just one* abstract method makes it abstract. * abstract class keyword.   **Interface (Also a type)**   * CAN-DO behaviour. * Even more general than abstract classes. * interface keyword with -able suffix. * Class implements interface = class can do smt * Class <: interface * Impure Interfaces: abstract and default mthds.   **Wrapper Types**  Integer i = 4; int j = i;   * Auto-boxing: primitive value 4 converted to instance of Integer. * Unboxing: takes 4 out and stores into int j. * Still less efficient than primitive types.   **Variance** (where C(S) is a complex type like arrays)   * Covariant: S<:T implies C(S) <: C(T) * Contravariant: implies C(T) <: C(S) * Invariant: Neither Covariant nor Contravariant. * Java array is covariant.   + Integer <: Object = Integer[] <: Object[] * Can have run-time errors as compiler compiles based on compile time type.       **Generics**   * Reduces duplication of code and improve type safety. * Takes in other types as type parameters. * Specify <Type> during class instantiation and object instantiation.   + class Pair<S, T>   + Pair<String, Integer> p = new Pair<String, Integer>(“Hi”, 4); * Generic Methods also takes in type para to constrain the type parameters used.   + public <T> boolean contains   + A.<String>contains(…)   **Lazy Evaluation (Memoization) (#32)**   * Ensures functions which only produce a certain output are only computed once.   **Infinite List (#33)**          **Streams**   * Stream is lazy. * forEach: terminal operations that takes in a stream and applies a lambda (Consumer) to each expression.      * limit: takes in int n and returns stream containing first n elements. * takeWhile: takes a predicate and returns stream containing elements of stream until it becomes false. * peek: takes in consumer to peek at stream.      * reduce: takes identity (eg.0) and accumulation function (eg. (x, y) -> x + y) and returns a single value. | **Exceptions**    which throws an exception     * How to throw exception in method?      * Unchecked: IllegalArgumentException, NullPointerException, ClassCastException * Checked: FileNotFoundException * Unchecked are subclasses of RuntimeException and causes run time error, caused by programmer’s error. * Checked is not under programmer’s control. Must be handled or else code will not compile. * Good Practices:  1. Clean up in finally block 2. No Pokemon Exception Handling (Catch and do nothing about it) 3. No Overreacting (System.exit(0) after catching) 4. Do not break AB! 5. Don’t use it to control flow mechanism  * Error Class: Terminates as there is no way to recover from the error.   **Type Erasure**     * Replaces parameter with Object or whatever it implements or extends. * Don’t mix generics and arrays! * *Illegal*: Pair<String, String>[], new Pair<S, T>[2]. * *But Legal*: new Pair<?, ?>[10] since no info lost.   **@SuppressWarnings**   * Use to most limited scope. * Use only if you are sure it will not cause a type error. * Raw types should not be used (just Pair~~<S, T>~~)   **Monad (#36)**   * Maybe<T>, Lazy<T>, Loggable<T> * InfiniteList<T> no flatmap = not monad! * Left Identity Law:   Monad.of(x).flatMap(x -> f(x)) = f(x)   * Right Identity Law:   mon.flatMap(x -> Monad.of(x)) = mon   * Associative Law:   mon.flatMap(x -> f(x)).flatMap(x ->g(x)) =  mon.flatMap(x -> f(x).flatMap(y -> g(y)))  **Functors**   * BooleanCondition<T>, Transformer<T>, Lazy<T>, Maybe<T>, InfiniteList<T>, etc. * Abstraction in Functional-Style Progamming that supports map. * Simpler construction than monad. * Only ensures lambdas can be applies sequentially to the value without worrying about side information. * Preserving Identity:   functor.map(x -> x) = functor   * Preserving Composition:   functor.map(x -> f(x)).map(x -> g(x)) =  functor.map(x -> g(f(x))  **Threads**   * Synchronous Programming: Method is blocked until it returns. Not very efficient especially with frequent method calls. * Thread is a single flow of execution.      * Thread.currentThread().getName() to get name of the current working thread. * Thread.sleep(int n) to cuase current execution thread to pause execusiont immediately until sleep time is over. * Thread.isAlive() to check is another thread is still running. * Program only exits after all threads are run to completion. |