CS2030S Midterm Cheatsheet	CS2030S
Compiler (Tombstone)	Definition > T-shaped blocks that convert rightwards > Compilers (javac) convert written code to bytecode language (virtual machine language) > Compilers themselves are written in a certain language, which needs to be interpreted to the hardware instructions
Interpreter (Tombstone)	Definition > Rectangle blocks that convert downwards > Interpreters (java) convert bytecode language to hardware instructions > Interpreters (jshell) can also convert written code directly to hardware instructions (line by line)
Executable (Tombstone)	Definition > Compilers (javac program.java) will create a bytecode script (.class) from the written script (.java) > Interpreters (java program.class) will run the code
Method Signature	Definition > Method Name > Number of Arguments > Type of Arguments > Order of Arguments
Composition	Definition > Models the relationship between object types using boxes and arrows Topics ?> Has A Relationship ?> Is A Relationship ?> Can Do Relationship
Java List	Usage > Stores elements in an expanding fashion ><JV import java.util.* JV> Types ?> Java_ArrayList ?> Java_LinkedList Methods ><JV List(len) JV> ><JV add() JV> ><JV get() JV> ><JV addAll(list) JV> ><JV size() JV> ><JV isEmpty() JV> ><JV contains(obj) JV> ><JV remove(obj) JV> ><JV clear() JV> ><JV set(id,obj) JV>
Java Super	Usage > Can super() to parent construct > Can super.method() to use parent method > Can super.field to use parent class field
Dynamic Binding	Usage > Figures out which method implementation is called Properties > Class methods cannot be dynamically bound Process ?> Compile_Time ?> Run_Time
Compile Time	Topics ?> Compile_Time_Type Process > Identify the /Compile_Time_Type of the parameters > Check the compile-time class for method > Identify the most specific implementation of the method to use > Store method descriptor in bytecode > Then return that method's signature (name, *args) > /Type_Erasure
Run Time	Topics ?> Run_Time_Type Properties > Usage :- /Run_Time_Type of target, /Compile_Time_Type of arguments Process > Get method descriptor from /Compile_Time > Starting from the /Run_Time_Type, go up the /Class_Heirarchy until you find the first matching method
Liskov Substitution Principle	Definition > A subclass should not break the expectations set by the superclass > You can do extra, but not less
Final Modifier	Definition > Data is immutable > Classes cannot be extended > Methods cannot be overridden
Stack Heap Diagram	Usage > Understand what happens when we make method calls Topics ?> Stack__Stack_Heap_Diagram_ ?> Heap__Stack_Heap_Diagram_ ?> Meta_Space ?> Stack_Frame Process > Initialise with variables / stack frames{}, including the frame for main(){} > Draw boxes for object instances and draw arrow to point from variable to object > If there is a new method call, create a new stack frame on top of the stack, and point arrows wherever > Once done with method, remove stack frame and clear all reference arrows > Once done with main(), remove all object frames with nothing referencing it
Stack (Stack Heap Diagram)	Definition > Contains variables, stack frames{} for constructors / methods Properties > Frames can only point to the /Heap__Stack_Heap_Diagram_ > Variable names have no types, but rather types are based on what they point to Examples > Block containing c1, c2, p
Heap (Stack Heap Diagram)	Definition > Instances of objects Properties > Frames can point to each other Examples > Circle frame with center and radius
Compile Time Type	Definition > Fixed as what the variable was defined as > Compiler will tell you what will go wrong Examples ><JV Circle c = new ColouredCircle(); JV> - gives Circle

Run Time Type	Definition > Check what object the variable is pointing to > No compiler can tell you there will be a mistake there Examples ><JV Circle c = new ColouredCircle(); JV> - gives ColouredCircle
Checked Exception	Definition > Programmer has no control > User might give an incompatible type or do something unexpected
Generics	Usage > Systematically create specific classes for many occasions Properties > Cannot mix with array :- Pair<String, Integer>[2] can't work as it will convert it into Object, and it is not type safe Topics ?> Generic_Type ?> Generic_Method ?> Type_Erasure ?> Wildcard__Generics_ ?> Java_Equals__Generics_ ?> Type_Inference__Generics_ ?> Valid_Generics
Valid Generics	Examples ><JV Pair p = new Pair(2,3) JV> ><JV Pair<Integer,Integer> p = new Pair<>(2,3) JV> ><JV Pair p = new Pair<Integer,Integer>(2,3) JV> ><JV Pair p = new Pair<>(2,3) JV> ><JV List<?> L = new List<String>() JV> ><JV Pair<?,?> p = new Pair<Integer,Integer>() JV>
Invalid Generics	Examples ><JV Pair<> p = new Pair<>(2,3) JV> ><JV Pair<Integer,Integer> p = new Pair<?,?>() JV> ><JV List<?> L = new List<?>() JV>
Type Inference (Generics)	Definition > Compiler will list out all possible types for the type parameter T, based on the bounds provided > Inference can only happen in the rightwards direction

Generic Type	Purpose
	> Class where you can pass a type as a variable T
	> Compiler will make sure the types are correct
	
	Types
	?> Raw_Type
	?> Bounded_Type
	?> Unbounded_Type
	?> Unchecked_Type
	
	Topics
	?> Type_Invariance
	
	Usage
	> Treats T and S as variables which can be used later
	> Cannot directly create an array of T[], can be typecasted with <JV T[] arr = (T[]) new Object[n]} JV>
	> javac -Xlint:unchecked file.java to compile with unchecked
	> javac -Xlint:rawtypes file.java to compile with rawtypes
	
	Examples <JV
	class Pair<T, S> {
	  public Pair(T obj1, S obj2) {}
	}
	Pair<String, Integer> p = new Pair<String, Integer>()
	
	class Pair<T extends Comparable<T>, S> implements Comparable<Pair<T, S>> {}
	JV>

Type Erasure	Usage > Once the code is compiled, all the generic types are removed, and values are typecasted > Changes all type parameters to Object if unbounded (T → Object), or specific type if bounded (T extends Comparable<T> → Comparable) ?> Code_Sharing Examples <JV Pair<Integer, String> p = new Pair<Integer, String>(1, "one"); Integer i = p.getFirst(); JV> Becomes <JV Pair p = new Pair(1, "one"); Integer i = (Integer) p.getFirst(); JV>
Raw Type	Usage > Only use it for /Java_Instance_Of Disadvantages - Compiler cannot catch mistakes
Producer Extends	Definition > Take information out of the object Usage > /Parameter_Type > <? extends T> Code ><JV static <? extends T> T foo(T){} JV>
Consumer Super	Definition > Insert information into the object Usage > /Return_Type > <? super T> Code ><JV static <T> T foo(Array<? super T>){} JV>