**Object**: has a specific purpose & can be used by other objects; class = data type, object = actual thing; an object has state(variables and value) and behavior(methods). Behavior can change the state. Objects expose only necessary details to other objects.

**Interface**: public; abstract implementation (method signature); separate what(interface, abstract) and how(object, detail).

**Abstract class:** has one or more abstract methods(override in the subclasses).

**Encapsulation**: combine everything necessary for a cohesive entity into a class; hid all details in a capsule; classes represent self-complete entities**.**

**Information hiding:** expose only necessary info. *public*: to world; *protected*: to the package, all subclasses and classes; *Default*: visible to the package and class(abstract); *private*: to the class only(outer & nested). Constant immutable value: public static final

**(Hign)Cohesion:** only entity, one purpose and all functionality for this entity; Bad: no-sufficient role: methods change external data only. Multi-purposes: class mixes operations.

**(High)Decoupling:** objects should use each other only when needed.

**Final**: field: can’t change a given value(the constructor ensures it, only methods can change it, method NextValue() preserves evenness, there aren’t any other public methods) when a class represents a genuine value rather than something more stateful, immutability makes sense; class: can’t have subclass; method: can’t be override.

**String.format**("%d:%02d:%02d", hours, minutes, seconds): %d: the next parameter is an int, %2d: the number to be at least two characters long. %02d: Zero-pad the integer to make it two characters long.

**Overflow:** long->in (int) (inSeconds/3600) too big long would exceed the range of int;Check: > Integer.MAX\_VALUE

**Compare:** Compare(o1, o2) & o1.compareTo(o2)

**Hash**(x,y,z): for multiple objects ;HashCode(o) : single object

**Interface polymorphism**: CompactDuration implements Duration interface as well so that we don’t change the codes.

**Less kinds of constructors**: If not users would look up specific classes but they like too look interface

**Factory method pattern** (static): to deal with the problem of creating objects without having to specify the exact class of the object that will be created. In the abstract: protected Duration plus(Duration that) { return fromSeconds(this.inSeconds() + that.inSeconds()); In the DurationImpl: protected Duration fromSeconds(long s) { return new DurationImpl(s); In CompactDuration: protected Duration fromSeconds(long s) { return new CompactDuration(s)}

**Primitive types** : Boolean, byte, short(16), char(Character), int(32)(Integer), float, long(64), double;==, No null, no super

**Reference/Boxed:** Takes more space and types, nullable, have super type, equals(Object)

**Charater**: Character.isLetter(‘a’); Character.isDigit(‘8’); Character.isLowercase(‘8’); “abc”.charAt(0); “abcd”.indexOf(‘a’); char[] abc = {‘a’, ‘b’, ‘c’}; String.valueOf(‘ab’); ‘abc’.toCharArray();

**Enumerations**:A special kind of class;creates a finite set of named values; represents a small, fixed set of options(No need to extend). Enum Binop { Add, sub, Mul}; Binop.Mul; Binop.values() indicates all Binops;

**Switch:** swich(op) { case Add: return a + b;…default: throw new illegalArguementException(“ ”)/unrecognized()}. Don’t miss break. Return is fine.;otherwise, it will call everything below this; Only works on primitive types, enum and String. **Array** τ[ ]: efficient, mutable(they can change in content but not in length), *fixed-length,* constant-time-indexed, sequence of values of type τ(with a defined order), store both primitive and objects; Fully initialized: int[] array1 = {2,3,4,5}; uninitialized: new int[64];***Construction***: String[] array2 = new String[] {“a”, “b”};String[] array2 = new String[21];***Length***: array.length; ***Indexing***:arrayl.[1];***Aliasing***: if a1 =a3, a1 value changed, a3 does as well(Same address).***Copying***: int[] a3 = new int[a1.length]; *for(int i = 0; i< a1.length; i++) {a3[i] = a1[1];}*if a1 changed, a3 not.**Testing**: assertArrayEquals(); ***Varargs***: Use when you have unlimited or unknown numbers of variables to a method with same data type. Example: arbitrary length sequences: void setPlayers(String[] newPlayers){} Or void setPlayers(String…newPlayers){}*for (String player: newPlayers) {System.out.println(i)}};****SM****:* Arrays.asList(…),Arrays.binarySearch(int[] array, int key), Array.copyOfRange(T[ ] original, int from, int to), equals(Object[] a1, Object[] a2),deepEquals.

**ArrayList:** no need fixedrange, accept objects only, mores pace and less efficiently. List<> *list* = new ArrayList<>(Arrays.asList(item1, item2…));

**Equals**: shallow/intensional/nominal/physical: data simply refer to the same object. Deep/extensional/structural/logical: copy the fields by creating a new object. "=="compares the immediate contents of variables(for reference types this means memory locations). Default equal: physically. Need to override: if(this==obj)-> true, if(!(object instanceof Posn) -> return false, return this.x == that.x && this.y == that.y;

**Equiality rules**: reflexivity:x.equals(x); symmetry: x.equals(y) -> y.equals(x); Transitivity: x=y, y=z -> x=z;

**HashCode rules**: Compatibility: if x=y, they have same hashcode; Non-injectivity: if x, y have the same hashcode does not contract !x.equals(y). => override both hashcode and equal.

**Static versions of object methods:** T**.**compare(), T.toString(), T.max(), T.min().**instanceOf/null:** rarely use

**Exception**: checked: extends exception, possibly recoverable, must appear in throw clauses(in a method signature, not catch still catch though); unchecked: extends Error or RuntimeException, probably bail out, may appear in throw clauses.

**Static vs. Dynamic:** Enum and interfaces are always static**;**static(increase efficiency, compile time)belongs to a class while dynamic(run time) belongs to an object. ***Field(***use sparingly***)***: use when you want one variable for the whole class rather than one per object, outer and nested’s private names are visible to each other; ***Method***: class.method. Use when you want to associate some method with a class that doesn’t depend on having an instance. ***Classes***:Use when you want a helper class that;s strongly associated with the enclosing class, especially when the helper doesn’t make sense on its own. Nesting a helper class also allows the outer class to see its private members and vice versa. Example: collection & iteration.

**Generics:** class BinTree<T> { }; Generic methods: <T> void permute(List<T>); Need a type when create an instance; Java allows raw type but we not NEVER: BinTree tree = new BinTree() because it requires a type when use get()

**String**: Representing textual information, where the possible values are many or unlimited and not known ahead of time.

**Builder pattern:** A better solution for handling several optional arguments. Public static final class Builder How to use: if set width: ConnectNGame.builder().width(15).build(). Public PersonBuilder { Public Person build() { return new Person(age, name…)} public PersonBuilder age(int ag) {this.age = ag; return this;}…} Toset: pb.age(20).name().build()

**Class invariants**: A class invariant is a *logical statement* (a claim that is true or false: value is small) about *the instantaneous state of an object (*meaning the values in its fields at a point in time: value never decrease*)*that is *ensured by the constructors*(when a constructor finishes normally then the claim is true) and *preserved by the methods*(if the claim is true before a method is called, then it is true afterwards: value is non-negative). Wrong: small, never decrease, false

**Controller: *Synchronous*(**game**)** vs ***Asynchronous***(passive); ***Mock Object***: To test know what expect

**Comparable vs. Comparator:** comparable: 1. Implements to comparable<T>. 2. Override public in compareTo(T that) { return Long.compare(T1, T2)}3. Collections.sort(list). Comparator: step1: create a new class named CardComparator implements Comparator<Card> 2. Override public int compare(card1,card2) {} 3. Collections.sort(list, comparator).

**Controller:**

**public class** SimpleCalc6 {  
 **public static void** main(String[] args) {  
 **try** {  
 **new** Controller6(**new** InputStreamReader(System.***in***), System.***out***).go(**new** Calculator());  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 } } }  
**class** Controller6 **implements** CalcController { public class TestController6 {  
 **final** Readable **in**; @Test  
 **final** Appendable **out**; public void testGo() throws Exception {  
 Controller6(Readable in, Appendable out) { StringBuffer out = new StringBuffer();  
 **this**.**in** = in; Reader in = new StringReader(“+ 3 4 + 8 9   
 **this**.**out** = out; q”);  
 } CalcController controller6 = new Controller6(in,   
 **public void** go(Calculator calc) **throws** IOException { out);  
 Objects.*requireNonNull*(calc); try {Controller6.go(); } catch (Illegalment..Excep e){}  
 **int** num1, num2; *assertEquals*(**"7\n17\n"**, out.toString());  
 Scanner scan = **new** Scanner(**this**.**in**); }  
 **while** (**true**) { }  
 **switch** (scan.next()) {  
 **case "+"**:  
 num1 = scan.nextInt(); List<Card> deck = new ArrayList<Card>()  
 num2 = scan.nextInt(); for(Suit s: Suit.values()) {  
 **this**.**out**.append(String.*format*(**"%d\n"**, calc.add(num1, num2))); for(Value v: Value.values()) {  
 **break**; Card c = new Card(v,s);  
 **case "q"**: deck.add(c);   
 **return**;  
 } } } } throw new IllegalArgumentException(“ “)

**public enum** Suit {  
 @Test(expected = IllegalArgumentException.class)

public void test() {} ***CLUBS***(**"♣"**, 18),  
 ***DIAMONDS***(**"♦"**, 17),  
 assertFalse() ***HEARTS***(**"♥"**, 16),  
 assertTrue() ***SPADES***(**"♠"**, 15);  
 **public final** String **suitText**;  
 **private final int suitNum**;Suit(String suitText, **int** suitNum) {  
 **this**.**suitText** = suitText;  
 **this**.**suitNum** = suitNum;  
 }**public** String printSuitText() {  
 **return suitText**;  
 }  
 }