* Constructors of parent can be invoked from child
* Object is superclass of each and every class
* Is-A is inheritance and Has-a is composition/aggregation
* Cons,static and Instance block are not inherited
* Static are inherited by subclass
* Animal a = new Dog() – this is called **upcasting** where we assign object of child class to reference of parent class

Cat cat = new Cat();

AnimalUtility.printname(cat);

Here java automatically performs upcasting

Printname(Animal animal){

}

* Dog d = (Dog)anm – downcasting

If(anm instanceOf Cat){ - check to avoid exception

}

* Private,static and final(methods/variabes) use static binding
* MOD doesn’t work on private and static
* Return type of child method can be same or subclass of return type in parent method.
* Overridden method can’t introduce new checked exception or can introduce checked exception that is descendant of checked exception used in parent
* MOL – applicable to static/non-static also
* IF static method is defined in subclass, that concept is called method-hiding
* Vehicle v = new car();

Here static method of parent vehicle is invoked, as java always refers to the reference incase of static.

* Field hiding also always refers to object reference that its hiding
* Same field name is enough to do field hiding, irrespective of datatypes
* Java takes care of initializing default constructor of super class
* If Def constr is not present in super-class, dev shld invoke super-class parm constructor using super keyword.
* this/super cant be used in static area
* abstract can’t have private, static ,private const,
* Java 17

public sealed class Person permits Employee, Student{

}

sealed interface Animal permits Dog, Cat{}

The sealed keyword is used to declare the class as sealed.

The permits clause lists the classes that are allowed to extend the sealed class.

The permitted subclasses must be located in the same module or package as the sealed class.

Sealed classes can also be abstract or interfaces.

Permitted subclasses must be either final, sealed, or non-sealed.

 Sealed classes provide a mechanism to restrict the inheritance hierarchy for classes and interfaces. This means that you can control which classes can extend or implement a sealed class, thus offering more control over your codebase's architecture.

* **Improved type safety:** The compiler can check that all possible subclasses are handled in a switch statement, reducing the need for default cases.
* **Controlled inheritance:** You can prevent unintended extensions of your classes.
* **Clearer code:** The inheritance hierarchy becomes more explicit and easier to understand.
* Interface is similar to class, that can contain static fields ,abstract, default, private and static methods

Ads methods are public by default.

* Interface shld always have public and default
* SuperHero ironMan = new Ironman();

invokeSuperHero(ironMan );

* SuperHero spiderMan = new Ironman();

invokeSuperHero(spiderMan);

invokeSuperHero(SuperHero superhero){

sout(superHero.usePower());

sout(superHero.stopVillain(“N”));

}

* constants defined in interface are all Public, Static and final
* A and DSP (default, static and private since java 8) methods are allowed in interfaces
* All methods in Interface are abstract and public, unless specified as static/default
* In interface, abs methods can have parameters ,return type and throws clause
* Default methods are used in interface for future enhancement
* Private methods are mainly for commonly used default methods implementation logic.
* Static methods are added inside interface to define some utility functionality, they are public by default.
* Solution to avoid conflicts between default and abstract methods is

a) override conflicting abstract methods with same abstract method or ignore it

b) override conflicting method with default method and provide new implementation

c) override conflicting method with default method and call one of the methods of super-interfaces

* In case of conflict between superclass and super-interface, always super-class takes preference
* In case of conflicts, most-specific super Interface wins
* Marker/tagging interfaces doesn’t contain any methods and empty like cloneable, serializable
* Interfaces with SAM(single abstract method) is called as functional interface, but it can have any no of default/static methods
* Array copy can be done by loop,System.arraycopy,Arrays.copyOf()

System.arraycopy(sourceArray,0,destinationArray,0,sourceArray.length());

Arrays.copyOf(sourceArray,sourceArray.length+5);

* The **Arrays.sort()** method is part of the **java.util.Arrays** class and allows you to sort arrays of primitive types (such as **int**, **double**, **char**, etc.) and objects that implement the **Comparable** interface. It uses a highly optimized version of the quicksort algorithm for primitive types and mergesort for objects.
* Int []scores = new int[10];

Arrays.fill(scores,100);

Arrays.setAll(scores, i->i);

* Arrays.deepToString() needs to be used to convert nested array to String object

Streams

* PrintStream
* Private static final PrintStream out = null;
* Int num = System.in.read();
* For inputStream we have BufferedReader and Scanner class
* BufferedReader br = new BufferedReade(new InputStreamReader(system.in)

Br has default buffer size of 8kb and we can extend also

* Scanner has default buffer size of 1kb and cant be extended

Scanner s = new Scanner(System.in);

String str = s.next();

int age = s.nextInt();

* **Scanner** is not synchronized, which means it is not thread safe. If multiple threads access a **Scanner** concurrently, there may be unexpected behavior or data corruption. To ensure safe concurrent access, it's recommended to synchronize access to the **Scanner** instance manually or use alternative thread-safe mechanisms such as creating separate **Scanner** instances for each thread or using **Scanner** within a synchronized block. This aspect is crucial to consider when designing multi-threaded applications that involve input processing with **Scanner**.
* **SEVERE,WARNING,INFO,CONFIG,FINE,FINER,FINEST**
* From java 7 exceptions can be combined using pipe symbol |
* Try{Scanner scanner = new Scanner(System.in){

}

Try with resources

Here scanner is final and we can’t assign it again

Finally(scanner.close); - this is not needed,as we are using try with resources, so JVM takes care of closing resource

Try with resources can have multiple resources assigned and since java7 TWR need not have catch & finally blocks

* Throw used only with CE and propagate only UCE
* throws is used with CE and UCE and propagates only CE

**Object class**

* Object obj = new Person();

Person p = new Object()- **compile error**

Final methods – getClass,notify,notifyAll and wait

Non-final. – toString, equals, hashcode, clone and finalize

* HASH-Code is calculated by default using memory address of the object
* JAVA 7 HAS new java.lang.objects to calculate hash-code using Objects. Hash() method for computing hash code of an object
* Equals in object class compares obj references for equality i.e. it checks if 2 objects being compared refer to the same memory location
* Cloneable interface need to be implemented in class, where we are trying to clone.
* In shallow cloning, only references are copied, but not objects
* Deep cloning creates new object and also creates copies of objects referenced by original objects ,ensures nested objects are copied recursively and they implement cloneable.
* Record class are replacement of immutable class and we can create static fields ,initializers, methods etc.., except INSTANCE fields, initializers.

* Record class can’t be extended, as they are final by default
* getRecordComponents() and isRecord() are useful methods inside a Record
* String s = “hello world” to var message = “hello world” and any code that uses var as variable ,method or package is not affected ,but class/interface is affected and needs to be changed, var is a reserved identifier but not a keyword
* Var can’t be used for identifiers like class,interface,record,enum and annotation
* Var helps in DRY(don’t repeat yourself)
* In JS there are no type definitions for variables,
* Collections are maintained in java.util.package
* Primitive types cant be stored, only objects/reference types can be stored
* Wrap the primitive data-type before storing in collection and unwrap after retrieving it.
* boxing & unboxing- manual process

Int a = 10;

Integer integer1 = new Integer(a) – this auto-boxing is deprecated since java9.

Integer a = Integer.valueOf(num);

Integer b = Integer.valueOf(“10”) – both are valid ways to autobox

Wrapper classes have many utility methods that are useful ,like parsing which is converting String repr of primitive data-type value into its actual primitive data-type value.i.e

int a = Integer.parseInt(“18”);

Unboxing

int num1 = integer1.intValue();

* Auto-boxing & unboxing- automatic process by java-compiler since java5

int num = 16;

//

Integer numObj = num; -autoboxing

Integer myObj = 16;

unboxing

int num = myObj;

Auto-boxing & unboxing should be done carefully for null values