**interface:**

1. by default variables are public static final.

2. by default methods are public abstract.

1**. default methods in interface can be accessible only in package**, they can’t be accessed outside the package.

2. we can extend only one abstract class, but we can implement any number of interfaces.

3. cannot create constructors in interfaces

4. Unlike in classes static methods are not involved in the inheritance, so only we can call static methods only with the interface it belongs to. (feature given in java 8)

**Abstract classes:**

**helps only with in the package or project.**

1. When we need both defined and declared and that too in local scope we go for abstract classes.

2**. methods will have local scope that is protected**. or you can make them public.

3. can create constructors but cannot be instantiated. even though it is called when subclass object is instantiated.

Note:

Using Super class reference which is holding subclass object, we cannot call the local methods of subclass.

**Access Modifiers:**

1. **Public**: anywhere

2. **private**: with in current class.

3. **default**: with in current package.

4. **protected**: with in current package anywhere and outside package only in the child class and that with child class reference and its sub class references (Child of child). We should not use the parent class reference.

Sc about protected:

A screenshot of a computer

Description generated with very high confidence

A screenshot of a social media post

Description generated with very high confidence

A screenshot of a computer

Description generated with very high confidence

A screenshot of a computer

Description generated with very high confidence

A screenshot of a computer

Description generated with very high confidence

Know about javap:

A screenshot of a social media post

Description generated with very high confidence

**Overloading:**

compiler decides which method to call known as **method resolution**. This takes place based on reference type so, this is also called compile time polymorphism or static polymorphism or early binding.

\*\* automatic promotion of arguments is available in overloaded methods in java

**Rules**:

1. Method name must be same, with different arguments

**Overriding:**

First step: Compiler checks if the reference variable has the method it is referring to, if not it gives CE.

Second step: At run time JVM executes the method based on runtime object.so overriding is also known as Runtime polymorphism or dynamic polymorphism or Late binding

**Rules**:

1. Method signature must be same (Just method name and parameter list).

2. It accepts Covariant return types: it means child class object's return type can be same as parent class objects return type or it can be its child class object.

3. cannot override method with final access modifier.

4.cannot decrease the scope of the method by assigning weaker access modifier.

Note: therefore, we keep public for implementing a method form interface.

5.If child class method throws any checked exception, then compulsory parent class method should throw same checked exception or its parent.

6.If child class method throws any unchecked exception, then there are no restrictions.

**Note**: Unchecked means all run time exceptions and its child classes such as AE or null pointer and Error and its child classes such as Stack Overflow error....

7. It is not possible to override static method as non-static or non-static as static. The only possible situation is to override static method is only with static, but this is not overriding, It is known as method hiding.

**Note**: In method hiding method resolution takes place by compiler based on reference type.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* **variable resolution always takes place by compiler based on reference type.** \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Modifiers allowed for constructor:**

public, private, protected, default.

**Prototype of Default constructor:**

1. no-arg constructor.

2. modifier is same as class modifier if it is only public or default.

3. only one line in the body: super();

\*\* the first line of constructor should be this() or super(). if not, compiler will generate super() in the first line of constructor.

4. If parent class constructor throws any checked exception, compulsory child class constructor should throw same checked exception or its parent.

\*you cannot handle this by using try catch because the first statement in the constructor should be super().

**Type Casting :**

A b = (C)d;

there are 3 rules we need to remember

**Rule-1(Compiler) :**

The type of 'd' and 'C' must have some relationship

(either parent to child or child to parent or same type )

If not CE: incompatible types.

**Rule-2(Compiler) :**

C must be same as A or child type of A.

If not CE: incompatible types.

**Rule-3(JVM) :**

Runtime object of 'd' must be either same as C or derived type of 'C'.

If not Runtime Exception CCE(class cast Exception)

**Exception Handling:**

Exception: which disturbs the normal flow of the program and leads to abnormal termination.

Handling: to achieve Graceful or normal termination of program and to define an alternative way.

**Runtime stack mechanism:**

For every thread JVM creates a stack and all methods calls related to that thread will be stored in that stack. Once all methods are executed stack frames are removed step by step and then JVM destroys the stack and thread will be terminated.

**\* Every entry in stack is known as stack frame or activation record**

**Default Exception Handling**:

If an exception raised in any method, by default that method is responsible to create exception object. Inside Exception object it has Name, Description and location (Stack Trace).

**Summary**: If an exception raised in the method JVM checks for any handling code inside method, if available it handles the exception. If there is no handling code JVM removes this method's stack frame from stack and checks for any handling code in caller method. same repeats here, finally if main method doesn't handle this exception then all methods stack frames are removed from stack and stack is removed. Now JVM is the caller method to main () so JVM handles the exception to default exception handler. This terminates the program abnormally and prints the details from exception object to the console.

**Hierarchy:**

**Unchecked Exceptions:** All run time and errors are unchecked exceptions.

**Partially checked vs fully checked:** If both parent class and child classes are checked then it is fully checked, if parent is checked and child is unchecked then it is partially checked.

Ex: only two classes are partially checked in java. Exceptions class because of Runtime exceptions and Throwable class because of Runtime and Errors classes.

Methods to print Exceptions :

1. e.printStackTrace()
2. e.getMessage().
3. e.toString().

**throw and throws:**

**throw** keyword is used to handover the explicitly created exception object to JVM.

**throws** keyword is used to delegate the responsibility of exception handling to the caller. (caller may be another method or JVM)

**Wrapper Class:**

1. To wrap primitives in to object.
2. To define utility methods for the primitives.

**Utility Methods**:

valueOf() : to convert primitive or string in to wrapper object.

xxxValue() ; to convert wrapper object to primitive.

parseXXX(): to convert String to primitive.

**Autoboxing:** from primitive to object.

**Auto unboxing:** from object to primitive.

**Lambda Expressions:**

1. To enable functional programming in java we use lambda (functional programming means to assign function to a variable and to pass function as argument in method).
2. Lambda expressions are just anonymous/ nameless function with no name, return type or modifiers.
3. Lambda expressions are also called as **closures**.
4. **We can use lambda expressions only for functional interfaces.** (FI are references to call lambda expressions)
5. **Marker Interface:** If the interface doesn’t contain any method and by implementing the interface the object gets the ability, then it is marker Interface.

**Ex:** Serializable, Random Access, SingleThreadModel, Cloneable.

1. **Functional Interface:** The interface that contains the single abstract method.

**Ex:** Runnable 🡺run();

Callable 🡺call();

Comparable 🡺 compareTo();

1. FI can have any number of static or default methods but should have only one Abstract method.
2. Annotation for Functional interface is **@FunctionalInterface**: by using this annotation compiler will validate all rules of FI.

**A screenshot of a survey

Description generated with very high confidence**

**Types of main method:**

* Order of modifiers is not important (public static or static public).
* We can use these modifiers (final, synchronized, strictfp).

Ex: valid main

Final static synchronized strictfp public void main(String… sai){

}

**Video : Demo on 29-01-2018**

A screenshot of a video game

Description generated with high confidence

A screenshot of a cell phone

Description generated with very high confidence

**We can specify literals for integral datatypes in 4 types:**

Decimal: int x =10;

Octal: int x = 010;

Hexadecimal: int x = 0x10;

Binary: int x = 0B1111;

**For floating point, we get only decimal**:

* we can’t give 0X or 0B to with the floating-point variable. Even though it accepts 0123.45 as input it don’t consider it as octal. It always prints decimal

**Operators and Assignments:**

**Video: Demo on 30-01-2018**

* can’t use increment and decrement operators for values, can be used only for references.

Ex: byte b3 = 0;

b3+=129; // implicit type casting

System.out.println(b3); // -127

b3++; //implicit type casting

System.out.println(b3);

// b3 = b1+b2; // **Error : addition of two shorts or two byte gives int.**

* Sop(10/0) 🡪 AE
* Sop(10/0.0) 🡪 Infinity
* Sop(0/0) 🡪 AE
* Sop(0/0.0) 🡪 Nan

**Bitwise operators: Applicable for Boolean and integral types**

&,|,^(X-OR)

cA screenshot of a cell phone

Description generated with very high confidence

**Bitwise complement operator (~):**

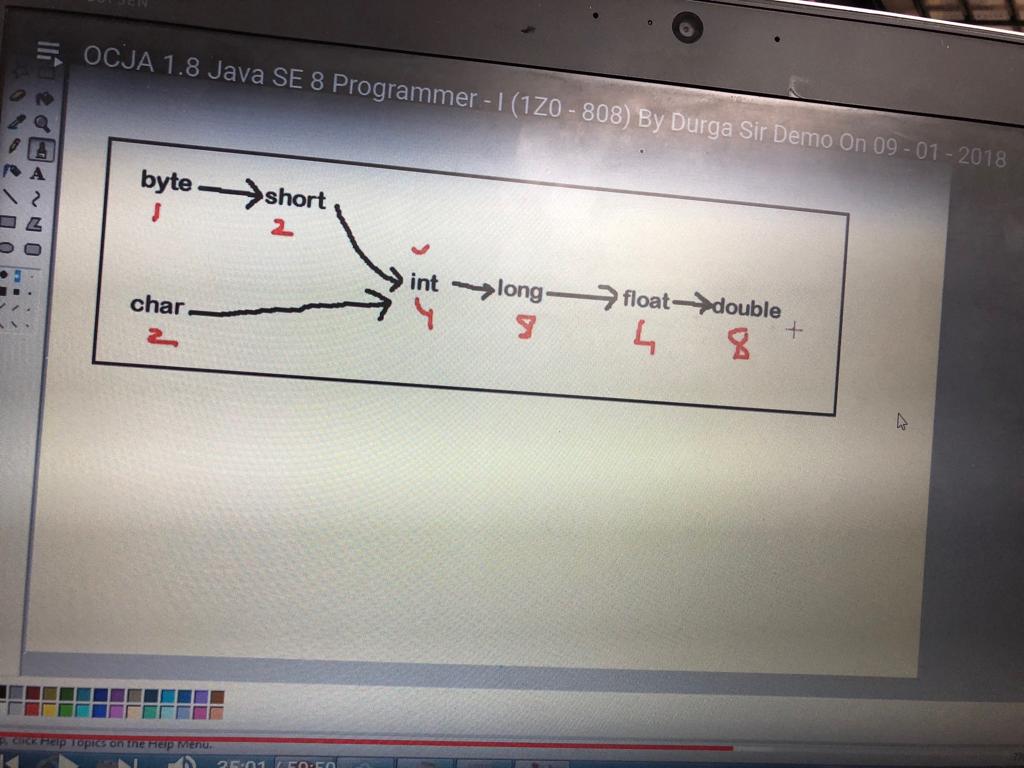
* Only for integral types not for Boolean.

**Short circuit operators:**

* &&, ||
* In case of bitwise & both are evaluated but in case of && if first is false it won’t check second one.

**switch:**

* Curly brass is mandatory.
* Only byte, short, int, char, enum, String and corresponding wrapper classes are allowed.
* Case labels should be compile time constants.

**Some Useful Images:A close up of text on a whiteboard

Description generated with high confidenceA screen shot of a computer

Description generated with very high confidenceA close up of text on a white background

Description generated with high confidenceA screen shot of a computer

Description generated with high confidenceA close up of a piece of paper

Description generated with high confidenceA screenshot of a computer

Description generated with very high confidenceA close up of text on a whiteboard

Description generated with high confidenceA screenshot of a computer screen

Description generated with very high confidenceA close up of text on a whiteboard

Description generated with high confidenceA close up of text on a whiteboard

Description generated with high confidenceA screen shot of a computer

Description generated with very high confidence**

**A screenshot of a social media post

Description generated with very high confidenceA screenshot of a cell phone

Description generated with very high confidenceA screenshot of a computer

Description generated with very high confidence**

## Java Operator Precedence Table

|  |  |  |  |
| --- | --- | --- | --- |
| **Precedence** | **Operator** | **Type** | **Associativity** |
| 15 | () [] · | Parentheses Array subscript Member selection | Left to Right |
| 14 | ++ -- | Unary post-increment Unary post-decrement | Right to left |
| 13 | ++ -- + - ! ~ ( *type* ) | Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast | Right to left |
| 12 | \*  /  % | Multiplication Division Modulus | Left to right |
| 11 | + - | Addition Subtraction | Left to right |
| 10 | << >> >>> | Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension | Left to right |
| 9 | < <= > >= instanceof | Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only) | Left to right |
| 8 | == != | Relational is equal to Relational is not equal to | Left to right |
| 7 | & | Bitwise AND | Left to right |
| 6 | ^ | Bitwise exclusive OR | Left to right |
| 5 | | | Bitwise inclusive OR | Left to right |
| 4 | && | Logical AND | Left to right |
| 3 | || | Logical OR | Left to right |
| 2 | ? : | Ternary conditional | Right to left |
| 1 | = += -= \*= /= %= | Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment | Right to left |