If any file contains more than 1 class, we can name it anything.
e.g. File Test.java can have class A and class B.

If any class is specified as public then the file name should match with the class.
public class A, class B → Class name must be A.java only.

Any file can have only 0 or 1 public class.

How to use any collection class without an import statement?

java.util.List<String> list = new java.util.ArrayList<>();

There are two types of import, explicit import and implicit import.
Explicit import: java.util.ArrayList
Implicit import: java.util.*

Explicit import is recommended as it provides better code readability.

Import statement is not required for the same package and lang package.

Import statement imports classes and interfaces available in that package only, it does not include sub packages.
import java.* will not import ArrayList as it is under the java.util package.

---

package must be the first non comment line in any java file.

Class level modifier:

- public: Accessible from anywhere
- protected:
- default (or without modifier): Accessible within same package only
- abstract: Object creation is not allowed
- static: Restrict to only one instance
- final: Child class creation not possible
- strictfp: Force strict floating point behaviour

Allowed modifiers for top level class:

1. public
2. default
3. abstract
4. final
5. strictfp

Allowed modifiers for Inner class:

1. All modifiers applicable for top level class
2. protected
3. private
4. static

I want to extend one abstract class which contains 2 abstract methods, but I want to provide implementation for only one class. How can I handle this?

In this case, we can create a child class, implement one abstract method and define that class itself as abstract. Since we do not know the implementation of all methods, we cannot create concrete class.

Member modifiers:

- public: Accessible from anywhere
- default: Accessible within same package
- private: Accessible in that class only
- protected: Accessible within same package + Child class from other packages

Protected method and variables from Child class in other package:

A.java:

```
package pack1;
public class A {
	public int v1;
	protected int v2;
	protected void m1() {
		System.out.println("In M1");
	}
	public void m2() {
		System.out.println("In M2");
	}
}
```

B.java:

```
package pack2;
import pack1.A;
public class B extends A {
	public static void main(String[] args) {
		A a = new A();
		a.m1();
	}
}
```

O/P: Compile time error
We cannot access protected members and variables of parent class in other packages with parent class reference.

```
public static void main(String[] args) {
	B b = new B();
	b.m1();
}
```

This code will work fine, as we are referencing a child class.

Interface:

- Service requirement specification (SRS)
- Contract between client and service provider
- Earlier definition was 100% pure abstract class, as all methods inside interface are public and abstract by default. But since from Java8 onwards, we can have default and static methods and from Java9 onwards we can have private methods inside the interface, we cannot consider it as a 100% pure abstract class.

Can we have a public method in parent class/interface and override with default modifier in child class?

No, while overriding methods we cannot reduce access scope.

OOPS:

- Data hiding:
  - Hiding of data
  - Securing data from outside world
  - Can be implemented by declaring member variable as private and specifying public method to access that member
- Abstraction:
  - Hiding internal implementation
  - Highlight list of offered services
  - Security, Enhancement, Maintainability / Modularity
  - We can enhance application without affecting end users

- Encapsulation:
    - Grouping related member variables and methods in single unit
    - Example is class which contains variables and methods
    - Any component which follows data hiding and abstraction and said to be encapsulated
- Tightly encapsulated:
    - When all member of any class is private, we can call that class tightly encapsulated
- Inheritance:
    - Is-A relationship
    - Main benefit is to achieve run time polymorphism
    - Code reusability (We should not extend from any class just to reuse code, class must be of that type only.)
    - Types of Inheritance:
        - Single level: class B extends A
        - Multiple inheritance: class C extends A, class B (Not supported by Java)
        - Multilevel: class C extends B, class B extends A
        - Hierarchical: class C extends A, class B extends A (Multiple classes extends same class)
        - Hybrid: Combination of all above (not supported by Java due to multiple inheritance)
        - Cyclic: Inheritance in cyclic fashion. class B extends A, class C extends B, class A extends C (not supported by Java).

Why is multiple inheritance not supported by Java?

To eliminate ambiguity problems. Cosider class C extends A and B, both have the m1 method, now when we call m1 with C object reference, it fails to identify which m1 to call as it is there in both classes A and B.
This is also called the Diamond access problem.

What are the 3 pillars of oops concept?

The 3 pillars of oops are:

- Encapsulation
- Inheritance
- Polymorphism

How does Python support multiple inheritance?

class C (A, B):
If both A and B contain the m1 method and we call m1 with C class reference then it will call A class m1 method as A is specified first while declaring class.

As all Java classes extend from Object class, if class B extends class A explicitly, should this be considered as multiple inheritance?

No, if class B extends class A then B is not a direct child class of Object. Compiler adds extends to class which does not extend from any other class. So in this case the compiler will treat class B extends class A and class A extends class Object.

Why is cyclic inheritance not supported by Java?

Because it is not required, there is no practical requirement to achieve this.

What is method signature?

Combination of method name and arguments (in same order) is called method signature.

public void m1 (String s, int i, float f) {}
Here m1 (String, int, float) is method signature.
m1(String, float, int) will be considered as a different method.

What is overloading?

Same method name with different argument type.
m1 (String, int, float)
m1(String, float, int)
m1(String)
m1(int)

Why is overloading called compile time polymorphism?

Because method resolution takes place at compile time by compiler based on reference type.

This is also called static polymorphism and early binding.

What will be the output of below code?

```
public class MultiInt {
   public static void main(String[] args) {
      A a = new A();
      a.m1('6');
```

```
        a.m1('a');
    }
}
class A {
    public void m1 (long i) {
        System.out.println("In long: " + i);
    }
    public void m1 (byte b) {
        System.out.println("In byte: " + b);
    }
    public void m1 (char c) {
        System.out.println("In char: " + c);
    }
}
```

O/P:
In char: 6
In char: a

What will be the output of below code? and why?

```
public class MultiInt {
    public static void main(String[] args) {
        A a = new A();
        a.m1('6');
        a.m1('a');
    }
}
class A {
    public void m1 (long i) {
        System.out.println("In long: " + i);
    }
    public void m1 (byte b) {
        System.out.println("In byte: " + b);
    }
}
```

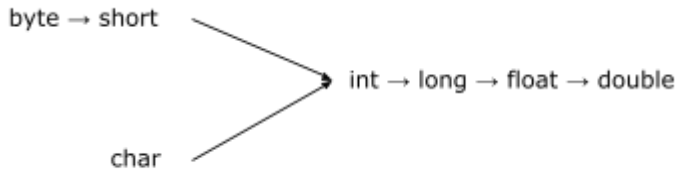
O/P:
In long: 54
In long: 97

Here compiler will follow automatic promotion and perform below operations:

- Find method with char
- Promote char to int and find method with int
- Promote int to long and find method with long
- Method with long is available so call that method

While converting char to int, the compiler considers ascii value of passed character.

What is an automatic promotion sequence?



Primitive type → Respective class → Object

What will be the output of below code? and why?

```
public class MultiInt {
    public static void main(String[] args) {
        A a = new A();
        a.m1('6');
        a.m1('a');
    }
}
class A {
    public void m1 (Object i) {
        System.out.println("In Object: " + o);
    }
    public void m1 (String s) {
        System.out.println("In String: " + s);
    }
}
```

O/P:
In Object: 6
In Object: a

Here char will promote to Char, and Char will promote to Object.

What will be the output of below code? and why?

```
public class MultiInt {
    public static void main(String[] args) {
        A a = new A();
        a.m1(null);
    }
}
class A {
    public void m1 (Object i) {
```

```
        System.out.println("In Object: " + o);
    }
    public void m1 (String s) {
        System.out.println("In String: " + s);
    }
}
```

O/P:
In String: null

Compiler checks for method in below sequence:

- Check for exact match
- If an exact match is not found and both parent and child can handle that argument, get it done with child method.

What will be the output of below code? and why?

```
public class MultiInt {
    public static void main(String[] args) {
        A a = new A();
        a.m1(null);
    }
}
class A {
    public void m1 (StringBuilder sb) {
        System.out.println("In StringBuilder: " + sb);
    }
    public void m1 (String s) {
        System.out.println("In String: " + s);
    }
}
```

Compile time error “The method m1(StringBuilder) is ambiguous for the type A”.
Since both methods are of the same level (no parent child relationship), there is no way for the compiler to identify which one to call.

What will be the output of below code? and why?

```
public class MultiInt {
    public static void main(String[] args) {
        A a = new A();
        StringBuilder sb = null;
        a.m1(sb);
    }
}
class A {
    public void m1 (StringBuilder sb) {
```

```
        System.out.println("In StringBuilder: " + sb);
    }
    public void m1 (String s) {
        System.out.println("In String: " + s);
    }
}
```

O/P:
In StringBuilder: null

Though we are passing null, it is of type StringBuilder, so it calls its respective method only.

What will be the output of below code? and why?

```
public class MultiInt {
    public static void main(String[] args) {
        A a = new A();
        a.m1();
        a.m1(10);
        a.m1(10,20);
    }
}
class A {
    public void m1 (int i) {
        System.out.println("In int: " + i);
    }
    public void m1 (int... i) {
        System.out.println("In var-arg int: " + i);
    }
}
```

O/P:
In var-arg int: [I@70dea4e
In int: 10
In var-arg int: [I@5c647e05

If for any argument both the general method and var-args method applicable then general method gets preference.

What is runtime polymorphism? Why is it called so?

Method overriding is called as runtime polymorphism, as with method overriding, JVM checks run time object and calls its respective method.

```
Parent p = new Child();
p.print();
```

Here the compiler will check existence of print method at compile time, but if print method is overridden in Child, at runtime Child class print method will be called.

This is also called dynamic polymorphism and late binding.

What is the covariant return type in overriding?

```
public class MultiInt {
   public static void main(String[] args) {
      Parent p1 = new Parent();
      p1.m1(10);
      Parent p2 = new Child();
      p2.m1(10);
   }
}

class Parent {
   public Object m1 (int i) {
      System.out.println("In Parent m1: " + i);
      return new Object();
   }
}

class Child extends Parent {
   @Override
   public String m1 (int i) {
      System.out.println("In Child m1: " + i);
      return "ABC";
   }
}
```

O/P:
In Parent m1: 10
In Child m1: 10

Here from Parent class m1 we are returning Object, but from Child class m1 returning String. With overriding, method signature must be the same but return type can be of covariant type, meaning it allows child objects as return type in child class.

Covariant concept introduced in Java 1.5.

What will be the output of below code and why?

```
public class MultiInt {
   public static void main(String[] args) {
      Parent p = new Child();
      p.m1(10);
```

```
    }
}

class Parent {
    private Object m1 (int i) {
        System.out.println("In Parent m1: " + i);
        return new Object();
    }
}

class Child extends Parent {
    public String m1 (int i) {
        System.out.println("In Child m1: " + i);
        return "ABC";
    }
}
```

Compile time error.
Here though m1 method of Child class is public, compiler will first check visibility of m1 method from Parent class, since m1 of Parent is private, it will throw compile time error.

What are some important method overriding rules?

Some important method overriding rules are:

1. Method declared as final cannot be overridden in child classes.
2. Methods defined as private wont take part in method overriding, we can have methods with the same signature in child class but that will not be considered as method overriding.
3. We cannot reduce the scope of methods. The public method of parent class cannot be override as protected method in child class, but reverse is possible.
4. If a child class throws any checked exception then the parent must throw the same or any of its parent exception. This is applicable only for checked exception.
5. Method declared as static cannot be specified as non-static and vice-versa.

What is method hiding?

Method defined as static in parent and child class also specifying method with same method signature as static, then from child class reference we can never use parent class method, this is called method hiding.

What will be the output of below code and why?

```
public class MultiInt {
    public static void main(String[] args) {
```

```
        Parent p = new Child();
        p.m1(10);
    }
}

class Parent {
    public void m1 (int... i) {
        System.out.println("In Parent m1: " + i);
    }
}

class Child extends Parent {
    public void m1 (int i) {
        System.out.println("In Child m1: " + i);
    }
}
```

O/P:
In Parent m1: [I@70dea4e

Here the method signature of m1 is not the same, so it is overloading not overriding. In case of overloading compiler decides which method to call based on the reference type.

What is variable hiding or shadowing?

Defining some variables in child which are present in the parent is called variable hiding or shadowing.

What will be the output of below code and why?

```
public class MultiInt {
    public static void main(String[] args) {
        Parent p = new Child();
        System.out.println(p.str);
    }
}

class Parent {
    public String str = "Parent";
}

class Child extends Parent {
    public String str = "Child";
}
```

O/P:
Parent

Variables are resolved at compile time only, here compiler will consider reference type and get its variable value.

Comparison between method overloading and overriding:

Property	**Overloading**	**Overriding**
Method name	Must be same	Must be same
Argument type	Must be different (or order must be different)	Must be same
private / final / static	Can be overloaded	Cannot be overridden
Return type	No restriction	Till 1.4: Must be same From 1.5: Must be of covariant type
Throws clause	No restriction	If child throws checked exception, parent must throw same of any of its parent.
Method resolution	Compiler resolves based on reference type	JVM resolves based on runtime object
Other names	- Compile time polymorphism - Static polymorphism - Early binding	- Run time polymorphism - Dynamic polymorphism - Late binding

---

What will be the output of below code and why?

```
public class MultiInt {
    public static void main(String[] args) {
        Parent p = new Child1();
        Child2 c2 = (Child3) p;
        System.out.println(c2.str);
    }
}

class Parent {
    public String str = "Parent";
}

class Child1 extends Parent {
```

```
    public String str = "Child";
}

class Child2 extends Parent {
    public String str = "Child";
}

class Child3 extends Child2 {
    public String str = "Child";
}
```

O/P:
Runtime exception: Child1 cannot be cast to Child2

Reason: It does not satisfy runtime rule of object casting which says that runtime object must be of same type or Child of casted class.

What are the rules for object type casting?
Parent p = new Child1();
Child2 c2 = (Child3) p;

There are 3 rules to type cast object:

1. Compile time "(Child3) p": There must be some relation between p and Child3 (parent/child, child/parent, same type).
2. Compile time "Child2 c2 = (Child3)": Child2 must be of the same type or parent of Child3.
3. Run time "Child2 c2 = (Child3) p": Child2 must be of the same type or parent of Runtime object p.

When we type cast any object and create a new object, does Java create a new object?

No, it refers to the same old object.

```
public class MultiInt {
    public static void main(String[] args) {
        Parent p = new Child();
        Child c = (Child) p;
        System.out.println(c == p);
    }
}

class Parent {
    public String str = "Parent";
}

class Child extends Parent {
```

```
    public String str = "Child";
}
```

O/P:
true

Adapter class: Abstract class without abstract method?
HttpServlet: Abstract class without abstract method?