(32) Varargs and new vs newInstance()

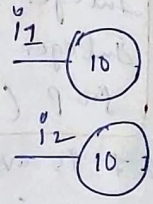
↳ 23rd Nov live class
↳ Nitin Sir

→ A topic w.r.t to wrapper classes which deals with Autoboxing, Widening (implicit typeating) and Var-arg approach.

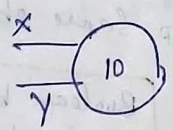
→ Wrapper class
1. Autoboxing
2. widening (implicit type)
3. Var-arg approach.

Snippets

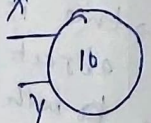
```
Integer i1 = new Integer(10);
Integer i2 = new Integer(10);
S.o.p(i1 == i2); // false
```




```
Integer x = 10;
Integer y = 10;
S.o.p(i1 == i2); // true
```




```
Integer x = Integer.valueOf(10);
Integer y = Integer.valueOf(10);
S.o.p(i1 == i2); // true
```




```
Integer i1 = 10;
Integer i2 = Integer.valueOf(10);
S.o.p(i1 == i2); // true.
```

→ Note:- When compare with Constructors it is recommended to use to use valueOf() Method to Create object.

① Var-args In Java:-

The Varargs allows the Method to accept zero or Multiple arguments.

→ JDK-1.0 version :- overloading Concept

→ Method overloading → Method name is same but different no. of arguments.

→ Compiler only binds Method Calls.

→ Compile time polymorphism.

→ Method overloading Concept is available from JDK 1.0 Version

② JDK 1.4V :-

from JDK 1.4 version if the arguments are changing you need to write different Methods to handle that requirement.

→ change in ~~arguments~~ (or) no. of arguments need to write new Methods.

→ if a Method handle two arguments if we give one more argument it is difficult to handle.

Eg) Public Void add(int a, int b){

S.o.p(a+b);

}

→ here add Method taken only two inputs.

→ third is input is not allowed to give. to add() Method.

→ if we want a Method with 3 arguments we have to write new Method.

```
Public Void add(int a, int b, int c){
        S.o.p(a+b+c);
}
```

→ here length of code increases when we write new Method.

→ Readibility of code will be difficult.

→ this was the problem till JDK 1.4 Version.

③ JDK 1.5 Version :-

→ from JDK 1.5 version we can write Single Method which can handle any type of arguments given to it, but the Expectation of that Method arguments type should be same.

→ if arguments are same type don't write multiple Methods.

① Var-args (Variable arguments) :-

The Varargs allows the Method to accept zero or multiple arguments. Before Varargs Either we use overloaded Method (or take array as the Method parameter but it was not considered good because it leads to the maintainance problem.

→ if we don't know how many arguments we will have to pass in the Method, Var-args is better approach.

Advantage of Varargs

→ we don't have to provide overloaded Methods so less code.

→ In Java language, if we have a Variable no. of arguments, then Compulsorily new Method has to written till JDK 1.4 Version.

→ But JDK 1.5 Version we can write Single Method which can handle Variable no. of arguments (but all of them should be of same type).

Syntax:- Methodone(dataType ... Variablename)

→ ~~it also called~~

→ it stands for Ellipse.

→ we have to write three dots after data type of arguments.

Method(dataType ... Variable name)

→ JDK 1.5 version or Var-arg Concept is available.

→ it is also called Ellipse approach.

→ Single Method accepts multiple arguments of same type. we can pass any number of arguments.

→ JVM internally uses array representation to hold the values of x.

Code:-

```
Class Demo2
    Public Void add(int... x){
        S.o.p("Var-args approach");
    }
}
Public Class Test2
    Public static Void main(String[] args){
        Demo2 d = new Demo2();
        d.add(10);
        d.add(10,20);
        d.add(10,20,30);
    }
}
```

O/P:- Var-args approach
Var-args approach
Var-args approach

→ JVM Internally uses Array representation to hold the values of x.

d.add() → new int[]{}
d.add(10); → new int[]{10} x → [10]
d.add(10,20); → new int[]{10,20} x → [10|20]
d.add(10,20,30); → new int[]{10,20,30} x → [10|20|30]

Case-①;

(i) Valid Signatures

① Public Void Method (int... x); (Valid) ✓
② Public Void Method (int... x); Valid ✓
③ Public Void Method (int ...x); Valid ✓

(ii) Invalid Signature

① Public Void Method(int x...); (Invalid) ✗
② Public Void method(int. x..); (Invalid) ✗
③ Public Void method (int. ..x); (Invalid) ✗

Case-②:- We can mix Normal argument with Var argument.

Eg:-
```
Public Void method (int x, int... y);
Public Void method (String s, int... x);
```

Case-③:- While Mixing Var-arg with normal argument Var-arg should be always last.

→ first argument should be normal argument but second one will be Var-arg.

Eg:-
```
m1 (int... x, int y);   // Invalid ✗
m2 (int x, int... y);   // Valid. ✓
```

Case-④:-

→ In an argument list there should be only one Var argument.

→ In a method Call Var-argument should always be one. We Cannot have two.

Eg:-
```
m1(int... a, int... b);  // Invalid (✗)
m1(int... a);            // Valid (✓)
```

Case-⑤ - Program → here arguments are String and int.

```
Class Demo3 {
    Public Void m1(String data, int... x) {
        S.O.P("Var-args");
    }
}
Public class Varargs3 {
    Public Static Void main(String[] args) {
        Demo3 d = new Demo();
        d.m1("Sachin");
        d.m1("Kohli", 10);
        d.m1("dhoni", 7, 8);
    }
}
```

Case-5 :- We can overload Var arg Method, but Var arg Method will get a call only if none of the Matches are found.

→ Just like default statement of switch Case.

```
Class Demo {
    Public Void m1(int... x) // Var-arg Method
    {
        S.o.p("Var-arg Method");
    }
    Public Void m1(int x) // int-Method.
    {
        S.o.p("int Method");
    }
}
Public Class Test {
    Public Static Void main(String[] args)
    {
        Demo d = new Demo();
        d.m1();      ——— ①
        d.m1(10,20); ——— ②
        d.m1(10);    ——— ③
    }
}
```

At ① :- d.m1()

↳ it calls Var-arg Method.

↳ d.m1();

↳ Behind the Scenes

d.m1(new int[]{});

→ No Method accepts Zero arguments.

Var-arg get a chance.

At ② :- d.m1(10,20)

↳ no Method accepts two arguments.

Var-arg get a chance.

d.m1(10,20); → d.m1(new int[]{10,20});

↳ behind the Scenes.

At ③ :- Here we have Method Specifically Catch one argument (int method)

→ Compiler try to bind for Exact match.

→ here it calls for int Method.

d.m1(10); // Exact Match

d.m1(10,20,30) = d.m1(new int[]{10,20,30})

code:-

```
Class Demo {
    Public Void m1(int... x){
        s.o.p("Var-args method");
    }
}
Class Test
{
  Public static Void main(String[] args)
  {
    Demo d = new Demo();
    d.m1(10,20,30);
    d.m1(new int[]{10,20,30});
  }
}
```

→ d.m1(10,20,30) and d.m1(new int[]{10,20,30}) are same.

Case-② Public Void Method(int... x) it is replace with Public Void method(int[] x):

```
class Demo{
    Public Void m1(int[] x)
    {
        s.o.p("int[] Method");
    }
}
Public class Test {
    Public static Void main(String[] args)
    {
        Demo d = new Demo();
        d.m1(new int[]{10,20,30});
    }
}
```

→ In this Case we have Pass only array as an argument.

Case-③:- we Cannot have two Methods with same Signature

Eg:

```
Public Void m1(int... x){ ——①
}
Public Void m1(int[] x){ ——②
}
```

→ ① & ② are Methods with same Signature.

① Public Void m1(int... x) {

}

→ This Method accepts arguments as direct Values and array also.

Eg: d.m1(10,20,30);
d.m1(new int[] {10,20,30});

② Public Void m1(int[] x) {

}

→ this method accepts only ~~int~~ array as an argument.

② Single Dimension Array Vs Var-Arg Method

① Whenever Single dimension Array is present we can replace it with Var-Args

Eg: Public static Void main(String[] args)
⇓
Public static Void main(String... args)

② Whenever Var arg is present we cannot replace it with Single Dimension Array.

Eg: Public static Void Main(String... args)
⇓
Public static Void Main(String[] args) ✗
↳ (Invalid)

→ m1(int... x)

↳ we can call to this method by group of int values and x will become 1D array (int[] x);

→ m1(int[] x)

↳ we can ~~this~~ call this Method by passing 1D array only.

Note:-

① Method(int... x)

↳ we can call this Method by passing a group int values, so it becomes 1-D array.

② Method(int[]... x)

↳ we can call this Method by passing a group of 1D int[], so it becomes 2-D array.

Example ⇒ next page.

Code:-

→ giving 2 arrays as input

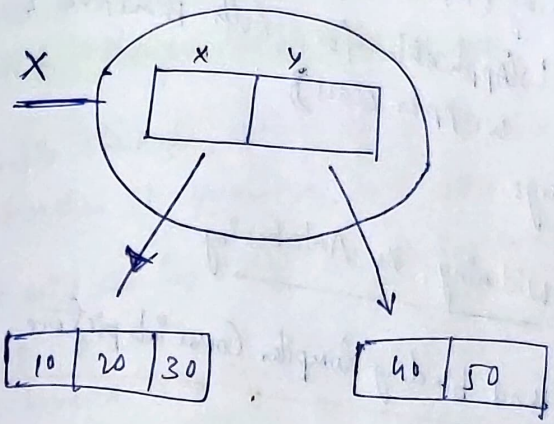
↳ x Collects array of 1D → (int[]... x)

↳ if we give x and y 1D array it becomes 2D.

```
Class Demo {

    Public Void m1(int[]... x){

        S.o.p(x); // [[I@ ---- → 2D array.
    // Prints Values of Arrays
        for(int[] a:x){
            S.o.p(a);
            for(int Elem : a){
                S.o.p(Elem);
            }
        }
}

Public class Test{
        Public Static Void main(String[] args)
        {
            int X[] = {10,20,30};
            int Y[] = {40,50};
            Demo d = new Demo;
            d.(x,y);
        }
}
```

Public Void method (int[]... x);

→ x is array of 2D



printing 10

③ Wrapper Class

① AutoBoxing
② Widening (implicit Type Casting done by compiler)
↳ (Applicable for both Primitive & wrapper class)

③ Var-Args.

Case-①: Widening Vs Autoboxing

→ for method binding (Compiler Comes into picture)

```
Class Demo1
    Public Static Void methodOne(long l){ — I
        S.o.p("widening"); // o/p: Widening
    }
    Public Static Void methodOne(Integer i){ — II
        S.o.p("autoboxing");
    }
}
Public class Test1
    Public Static Void main(String[] args)
    {
        int x=10;
        Demo d=new Demo();
        d.methodOne(x);
    }
}
```

x (int) → typ casting → long (found)

→ overloading - Compile time polymorphism.
→ Compiler check for Exact Match.
→ if Exact Match is not available Compiler do these things behind the Scenes.

① Compile check for nature of x (variable) Wheather it primitive (or) wrapper.
② it will do implicit type casting w.r.t int.
→ here x is primitive type (int)

implicit typcasting:
int → long

→ Complier found the match After implicit type casting.
→ In above Code there is a method which accepts long as argument. at I
→ if it find long then x is given to method which have long as argument.
→ long (binding happens by compiler).

Typecasting :- implicit type casting →
byte → short → int → long → float → double

(2) Case-2 :- Widening Vs Var-args Method

```
Class Demo {
    Public Static Void main methodOne(long l)
    {
     S.o.p("Widening"); // o/p: Widening
    }
    Public static Void main methOne(int... i)
    {
        S.o.p("Var-arg method");
    }
}
Public class Test {
    Public static Void main(String[] args)
    {
     int x = 10;
     Demo d = new Demo();
     d.methodOne(x);
    }
}
```

→ here x is Primitive type (Int)

(x) int → implicit typcasting → long (match)

(3) Case (3) :- AutoBoxing Vs Var-arg method

```
Class Demo {
    Public static Void main (Intege i)
    {
      S.o.p("AutoBoxing");
    }
    Public Static Void main (int... i)
    {
      S.o.p("Var-arg method");
    }
}
Public class Test {
    Public static Void main(String[] args)
    {
     int x = 10;
     Demo d = new Demo;
     d.methodOne(x);
    }
}
```

o/p = AutoBoxing

→ Compiler check nature of x and it found Primitive
→ ithen it do implicit type conversion

① int (x) → implicit typ casting → long (match not found)

→ Then Compiler do Autoboxing.

① Int (x) → AutoBoxing → Integer (match found)

① Compiler first check for nature of x.
→ if it is primitive it will do implicit type casting.

int (Primitive) → implicit → long, float, double
(x) type casting

→ ~~after there~~ ~~there~~.
→ here no match (methods) found which accepts long, float and double arguments.

② then Compiler do autoboxing
int → AutoBoxing → ~~long~~ Integer (match found)

→ Then x is given to the method accepting integer as argument.
→ here matic is found.

③ if there is no method accepting then it goes with method accepting Var-arg.

Case-①

```
class Demo {
    public static void methodone (long e){
        s.o.p ("long");
    }
}
public class Test {
    public ~~class~~ static void main (String[] args){
        int x = 10;
        Demo d = new Demo();
        d.methodone (x); // Compilation Error Can't find Method.
    }
}
```

① compiler do implicit type casting
int → typecasting → long, float, double (Match not found)

② her Compiler do Autoboxing.
int → AutoBoxing → Integer (match not found)

→ here Match not found
→ Compiler do widening on Integer

③ Integer → Widening → Number, Object (Parent of Integer)
(or (type casting).

→ ~~here~~ Compiler search for number and object (Match not found)

→ here Call will not be ~~resolved~~ resolved because match not found.

→ it gives Compilation Error

CE: Can't find the method.

Widening: Child reference can be collected by parent type which is called as "implicit type casting".

→ Note:- Widening followed by Autoboxing is not allowed in Java, but Autoboxing followed by widening is allowed.

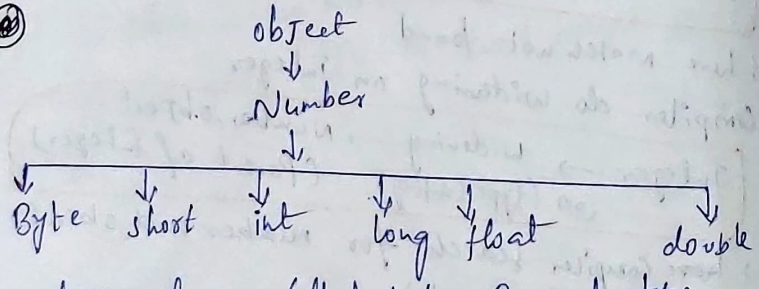
→ Whenever a Compiler tries to bind method call first preference is given to autoboxing then widening.

---

① Implicit type Casting :-

byte → short → int → long → float → double

②



→ Child reference collected by Parent type. We call as implicit type conversion.

Case-⑤ :-

```
class Demo {
    public static void methodOne(Object o)
    {
        s.o.p("Object");
    }
    public static void methodOne(Number n)
    {
        s.o.p("Number");
    }
}
Public class Varargs {
    Public static void main(String[] args)
    {
        int x = 10;
        Demo d = new Demo();
        d.methodOne(x);
    }
}
```

output: Number.

Explanation.

① first Compiler do autoboxing than it do type Casting.

1. int → auto boxing → Integer
2. Integer → Widening → Number, Object.

Which of the following declaration are Valid?

1. `int i = 10;` (Valid)
2. `Integer I = 10;` Autoboxing (Valid)
   int → autoboxing → Integer [ValueOf()]
3. `int i = 10L;` (invalid)
4. `Long l = 10L;` (Valid) Autoboxing (ValueOf())
5. `Long l = 10;` (invalid)
   ↳ Autoboxing (int → Integer)
   ↳ Widening (Integer → Number, Object)
6. `long l = 10;` (Valid)
   ↳ Int sits in long
7. `Object O = 10;` (Valid)
   ↳ int → autoboxing → Integer
   ↳ Integer → Widening → Number (or) Object (type casting)
8. `double d = 10;` (Valid)
   ↳ int go and sit in double
9. `Double d = 10;` (invalid)
   ↳ int → autoboxing → Integer
   ↳ Integer → Widening → number (or) Object (type casting)
10. `Number n = 10;` (Valid)
    ↳ int → autoboxing → Integer
    ↳ Integer → widening → number (or) Object

① New -

new is an operator to Create a object, if we know class name at the beginning then we can create an object by using new operator.

Eg :-

```
Student {
    String name;
    int RollNo;
}
Public class Test {
    Public Void static Void main(String[] args)
    {
        Student Std = new Student();
    }
}
```

Create object of Student class by using new keyword.

→ We Create object by new keyword.
↳ the information of class known to JVM and Compiler

Behind the Scenes When We Create Object

→ these actions happen when we initialize objects.

① new will Create memory on heap area.
② JVM will search for Student .class file in Current working directory.
↳ if found load the .class file data int method area.
③ During the loading of .class file
   ⓐ Static Variables get memory set with default value.
   ⓑ Static block get Executed.
④ in the heap area, for the required object memory for instance Variables is given. JVM will set the default value it.
   ⓐ Execute the instance block if available.
   ⓑ Call the Constructor to set the Meaningfull Values to the instance Variables.
⑤ JVM will give address of the object to hashing algorithm.
→ which generates the hashcode for the object and that hashcode will be returned as the reference to the programmer.

② newInstance() :- it is Method presenting class "Class", which can be used to Create object.

→ if we don't know the class name at the beginning and its available dynamically newInstance() Method.

Eg
```
Public class Test {
    Public static Void main (String[] args)
    throws ClassNotFoundException,
    InstantiationException, IllegalAccessException {

        String className = args[0];          ——①
        Class c = Class.forName(className);  ——②
        object obj = c.newInstance();        ——③
        Student std = (Student) obj;         ——④
        S.o.p(std);
    }
}
```

```
Class Student {
    //static block Executes at time of class loading.
    static {
        S.o.p("static block");
    }
    Public Student() {
    } S.o.p(Student Constructor);
}
```

Code Explaination :-

① Create object by using newInstance() Method
↳ take the input of the className for which object has to created at the runtime.
→ int arg[0] the class for which object has to be created available.

```
String className = args[0];
```

② load the class file explicitly by forName() Method ..
↳ for name method available in class "Class".

```
Class c = Class.forName(className);
```

③ for the loaded class, object is created using Zero parameter Constructor only.
→ if Zero parameter is not available in class it throws InstantiationException.
→ hence forth in the loaded class always Zero Constructor should be there.

```
object obj = c.newInstance();
```

→ newInstance Constructor always Excepts Zero Parameters.
→ We will see newInstance method in class.
→ type in cmd → java.lang.Class
→ we see public T newInstance();

④ Performing type casting to get student object.

```
Student Std = (Student) obj;
S.o.p(Std);
```

→ for Executing above code in cmd

javac Test.java
java Test Student → giving class as an argument.

→ output:

static block
Student Constructor.
Student@2ad46ca6
↳ hashcode send by hashing algorithm.

⑤ There is chance of Three Exceptions

① ClassNotFoundException:- When unknown class Name given as argument it throughs ClassNotfound exception.

② InstantiationException:-
when there is no constructor in class with Zero parameter it throws this Exception.

→ hence forth in the loaded classes always Zero Constructor should be present.

Eg:

```
Class Student {
    int a;
    Public Student(int a)
    {
      this.a = a;
    }
}
```

→ here student Contains Single parameter Constructor.

③ IllegalAccessException:-
When Constructor is private it throws IllegalAccessException.

Eg:

```
Private Student()
{
  S.o.p("Student Constructor");
}
```

↳ check Codes in github
new2.Java
new3.Java
new4.Java.

④ ClassCastException:- When String class given as argument it throughs ClassCast-Exception.

→ Eg: javac Test.java
java Test java.lang.String.

String → Student
↳ no relation b/w string & Student it throws ClassCastException.

④ Class Cast Exception :-

→ When we give String Class as String it gives corresponding Class Cast Exception.

→ During type casting, if there is no relationship between 2 classes as parent to child then it would result in "Class Cast Exception".

Eg :-

```
class student {
}
Public class Test {
  Public static Void main(String [] args)
  {
  object obj = class.forName(args[0]).newInstance();
  S.o.p(obj.getClass().getName());
  }
}
```

→ See code in github
↳ new1.java

Execution

javac Test.java
java Test java.lang.String
↳ it gives ClassCastException.

String → student type
→ no relation b/w String & student So it throws ClassCastException.

⇒ Difference b/w ClassNotFound Exception & NoClassDefFoundError :-

① NoClassDefFoundError :-

↳ it occurs at runtime. we get this Error when the class is not available in the program at runtime.

→ it is an unchecked Exception which a program throws when the requested class is not present at run time.

↳ check new5.java github.

→ for hard code class names at runtime in the corresponding .class files not available we will get NoClassDefFoundError.

② Class NotFound Exception :-

↳ the ClassNotFoundException occurs when the classpath doesn't get update with the JAR files

↳ for Dynamically provided class names at Runtime, if Corresponding .class file is not available then we will get the Runtime Exception saying "ClassNotFoundExcpt".

Eg: object o = class.forName("className").newInstance();

Difference between ClassNotFoundException & NoClassDefFoundError :

1. For hard coded class names at Runtime in the corresponding .class files not available we will get NoClassDefFoundError , which is unchecked

```
Test t = new Test( );
```
In Runtime Test.class file is not available then we will get NoClassDefFoundError

2. For Dynamically provided class names at Runtime , If the corresponding .classfiles is not available then we will get the RuntimeException saying "ClassNotFoundException".

```
Ex : Object o=Class.forname("Test").newInstance( );
```
At Runtime if Test.class file not available then we will get the ClassNotFoundException , which is checked exception.

1. new => requried class details known to compiler but not available at jvm then it would result in "NoClassDefFoundError"

2. newInstance() => requried class details not available at jvm then it would result in "ClassNotFoundException"

```java
//NoClassDefFoundError- Example
//new reqired class details known to compiler but not available at jvm.
//the it would result in NoClassDefFoundError

//The NoClassDefFoundError is an error when the required
//class definition is not available at runtime.

//new => requried class details known to compiler but not available at jvm then it
//would result in "NoClassDefFoundError"
//
//newInstance() => requried class details not available at jvm then it would result
//in "ClassNotFoundException"

class Student5
{

    static {
        System.out.println("Student .class file is loading");

    }
    public Student5()
    {

        System.out.println("Student constructor is called");
    }
}
public class new5 {

    static {
      System.out.print("Test.class file");
    }

    public static void main(String[] args) {
        Student5 std=new Student5();
        System.out.println(std);

    }

}

// 1. Both Student class and Test class is loaded
// C:\Users\Ashishpaul\Desktop\java>javac new5.java

// C:\Users\Ashishpaul\Desktop\java>java new5

// Test.class fileStudent .class file is loading
// Student constructor is called
// Student5@2c7b84de

// 2. after  compiling and deleting student class file we get  ClassNotFoundException

// C:\Users\Ashishpaul\Desktop\java>java new5
// Test.class fileException in thread "main" java.lang.NoClassDefFoundError: Student5
//         at new5.main(new5.java:21)
// Caused by: java.lang.ClassNotFoundException: Student5
//         at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:641)
//         at
java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.java:188)
//         at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:521)
//         ... 1 more
```

```java
// when we give unkown class as argument it throws ClassNotFoundException

class Student6
{

    static {
        System.out.println("Student .class file is loading");

    }
    public Student6()
    {

        System.out.println("Student constructor is called");
    }
}
public class new6 {

    static {
        System.out.print("Test.class file");
    }


    public static void main(String[] args) throws InstantiationException, IllegalAccessException,
ClassNotFoundException {
        String className=args[0];
        Object obj=Class.forName(args[0]).newInstance();
        System.out.println(obj.getClass().getName());

    }

}

// 1. when we give student class as argument
// C:\Users\Ashishpaul\Desktop\java>javac new6.java
// Note: new6.java uses or overrides a deprecated API.
// Note: Recompile with -Xlint:deprecation for details.

// C:\Users\Ashishpaul\Desktop\java>java new6 Student6
// Test.class fileStudent .class file is loading
// Student constructor is called
// Student6

// 2. when we give unkown class as argument it throws ClassNotFoundException

// Demo .class file not known to jvm and compiler
// C:\Users\Ashishpaul\Desktop\java>java new6 Demo
// Test.class fileException in thread "main" java.lang.ClassNotFoundException: Demo
//         at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:641)
//         at
java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.java:188)
//         at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:521)
//         at java.base/java.lang.Class.forName0(Native Method)
//         at java.base/java.lang.Class.forName(Class.java:383)
//         at java.base/java.lang.Class.forName(Class.java:376)
//         at new6.main(new6.java:23)
```