# operators

## assignment =

primitive assignment

assign a value or an expression to primitive type variables

Eight primitive Type: Boolean, char, bytes/short/int/long, float/double

|  |
| --- |
|  |
|  |

|  |
| --- |
| output: 5 |
| System.out.println(a=5); |

|  |
| --- |
| true  true |
| Test t1 = new Test();  Test t2 = t1;  System.out.println(t1==t2);  System.out.println(t1.equals(t2)); |

object assignment

assign object to a reference variable

|  |
| --- |
| c is reference variable |
| class Test{}  Test c = new Test(); |

## equal ==

The similar operators could be == or !=.

Operator == could compare value of primitive type and address of objects. If data type is different, compiler try convert type before comparison or throw compiling errors.

That can't compare two different data type or reference variables.

### primitive type

* For primitive type, two objects are equal if they are the same type and have identical value.
* If primitive type is compared with its wrapper, compiler first converts wrapper class type to primitive type and compare their value. If conversion failed, compiling error would be raised.
* Class casting is allowed before comparison of primitive type. Convert char/bytes/short to int type automatically. Or convert byte/short/int to int type. Or convert int type to float or double type.

Here is comparison between primitive types

|  |
| --- |
| false==false return true |
| boolean x = true;  boolean y = x = false;  System.out.println(x==y);//true |

Here is comparison between primitive type and its wrapper class

|  |
| --- |
|  |
| Integer i = 3;  int j =3;  System.out.println(i==j); //true  System.out.println(i.equals(j)); //true |

|  |
| --- |
| output: 2 |
| Integer x = 3;  int y = 0;  if(x == y) System.out.print("1");  else System.out.print("2"); |

Here are example casting for primitive type. Compare int type with double type

|  |
| --- |
| output: true |
| System.out.println('A'==65); //true  System.out.println(5==5.00); // true  System.out.println(5.0f==4); //false |

|  |
| --- |
| must explicitly cast type |
| Object a = 3;  System.out.println(3==(int) a); //true |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### String and StringBuilder

* == compare same memory between two Strings or two StringBuilders.
* equals() compare content of two String type. that is not working for StringBuilder.

Compare the difference of comparison in case of various String declaration approach.

|  |
| --- |
| "A" is a string object, stored in "string constant pool" created by JVM. a and b are reference variables that refer to "A" |
| String a = "A";  String b = "A";  **System.out.println(a==b);//true**  System.out.println(a.equals(b));//true |

|  |
| --- |
| Java create two String objects in memory. c and d are reference variables that refer to them. A literal "A" which is referred by the two String objects is stored in "string constant pool". "A" could be newly created or already exist. |
| String c = new String("A");  String d = new String("A");  **System.out.println(c==d);//false**  System.out.println(c.equals(d));//true |

|  |
| --- |
| compare two strings |
| // create one object  String a = "A";  String d = new String("A");  **System.out.println(a==d);//false**  System.out.println(a.equals(d));//true |

|  |
| --- |
|  |
| String n = new String("A");  String m = n;  **System.out.println(m==n);//true**  System.out.println(m.equals(n));//true |

Here are examples for StringBuilder comparison

|  |
| --- |
|  |
| StringBuilder s1 = new StringBuilder("neo");  StringBuilder s2 = new StringBuilder("neo");  StringBuilder s3 = s1;  if (s1.toString() == "neo")  System.out.println("1"); //false  if (s1.toString() == s2.toString())  System.out.println("2"); //false  if (s1 == s3)  System.out.println("3"); //true |

Note: StringBuilder is mutable but String is immutable

|  |
| --- |
| output: true |
| StringBuilder b = new StringBuilder("1");  StringBuilder c = b.append("2");  b.append("234");  c.deleteCharAt(1);  System.out.println(b == c); |

|  |
| --- |
| output: true true  Note: s1 is not updated |
| String s1 = "abc";  String s2 = s1;  System.out.println(s1 == s2); //true  **s1.substring(0,1);**  System.out.println(s1 == s2); //true |

|  |
| --- |
|  |
|  |

### object

== to check if two reference variables refer to the same object.

|  |
| --- |
| a1, a2, a3 are reference variables. |
| class A = {}  A a1 = new A();  A a2 = new A();  System.out.println(a1==a2);//false  **System.out.println(a1.equals(a2));//false**  A a3=a1;  System.out.println(a1==a3);//true  System.out.println(a1.equals(a3));//true |
| A a1 = new A();  Object a2 = a1;  System.out.println(a1==a2);//false  System.out.println(a1.equals(a2));//false |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## plus+

For primitive type, operator + is plus. but for String type, operator is concatenate. make sure that when operator + acts as plus operation or concatenate operation.

|  |
| --- |
| 7 |
| System.out.println(3+4); |

|  |
| --- |
| 173  char type ' ' is 82 |
| int a =9;  System.out.println(' '+ a + 100+' '); |

|  |
| --- |
| 42 |
| long x = 42L;  System.out.println(x); |

|  |
| --- |
| AB4C |
| System.out.println("AB"+4+'c'); |

|  |
| --- |
| AB:424 |
| x=42  System.out.println("AB:"+x+4); |

|  |
| --- |
| c4 |
| int a = 9;  System.out.println("c"+ a/2); |

|  |
| --- |
| 85ABC |
| int m=42, n=43;  String o = "ABC";  System.out.println(m+n+o); |

|  |
| --- |
| ABCfalse |
| String o = "ABC";  boolean c= false;  System.out.println(o+c); |

|  |
| --- |
|  |
| System.out.println(1 + 2 + "3" + 4 + 5); //==3345 |

|  |
| --- |
| output:5+6=11 |
| public class A {  private int i = 6;  private int j = i;  public A() {  i = 5;  }  public static void main(String[] args) {  A a = new A();  **System.out.println(a.i + a.j);**  } } |

|  |
| --- |
| output: equals  + act as concatenate. |
| String a = "";  a += 2;  a += 'c';  a += false;  if ( a == "2cfalse") System.out.println("==");  if ( a.equals("2cfalse")) System.out.println("equals"); |

|  |
| --- |
|  |
|  |

## shortcut operator

shortcut operator: &&, ||, ?..:..

<expression 1> && <expression 2> : If expression 1 is false then return false. Don't run expression 2.

<expression 1> || <expression 2> : If expression 1 is true then return true. Don't run expression 2.

<expression 1> ? <expression 2> : <expression 3> If expression 1 is true, then run expression 2, and don't run expression 3

|  |
| --- |
| output: ab |
| String s= "a";  boolean b = false;  if ( 4>3 || (b=true)) s += "b";  if ( b==true) s += "c";  System.out.println(s); |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

# Debugging

## Debug operations

### wrong literal

With names of variables in Java,

* It is ok to start with letters(A-Z, a-z), or $ or \_ is in the name. Other characters on the beginning is not allowed. For example, "\_\_x" or "$" is correct, but "2nd" is wrong.
* It is ok to consist of letters, digits, $ or \_ in names. Other characters are not allowed. For example, "com.my" or "good-year" is wrong.
* It is case sensitive. For example: Public and public are different.
* Don't use reserved identifier such public, static. but Public or Static is ok though that approach is not suggested.

### wrong declaration

duplicate declaration

|  |
| --- |
|  |
| int a, b=3;  int a=3; |

|  |
| --- |
| local variable age is declared duplicate. |
| void go(int age){  age++;  for(int age=3; age<6; age++)  ;  } |
|  |

not declaration

|  |
| --- |
| n can't be resolved to a variable |
| String m = n = new String("A"); |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

local variable is not initialized after declaration

|  |
| --- |
| The local variable i may not have been initialized |
| String i;  System.out.println(i==null); |

|  |
| --- |
| compiling error:  The local variable z is not initialized |
| int x = 10 % 2;  int y = 3 / 5 + ++x;  int z += 4 \* x;  System.out.print(x+","+y+","+z); |

### casting

data type is incompatible

|  |
| --- |
| Incompatible operand types int and String |
| int p =4;  String a = "A";  if (p == a) System.out.println("equal");  else System.out.println("not equal"); |
|  |

|  |
| --- |
| The operator == is undefined for the argument type(s) boolean, int |
| boolean z = true == 3; |

|  |
| --- |
|  |
|  |

### precedence

==

|  |
| --- |
| compiling error  assignment operator has the highest order. So the expression is equal to  final String statement = (250 ˃ 338 ? lion : tiger) = " is Bigger";  That is wrong. |
| String tiger = "Tiger";  String lion = "Lion";  final String statement = 250 ˃ 338 ? lion : tiger = " is Bigger";  System.out.println(statement); |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## Debug data type

* wrong literal: For example   
  int a = \_4; //can't put \_ at the beginning  
  float 3ab = 2.13; //only literal name should start with $, \_ or letters.
* repeat declaration: For example   
  int a =4; int a=3;
* Can't convert type in initialization : For example   
  int a = {1,2,3}; //can't convert array to int   
  int a = 3.4; //can't convert float type to int  
  StringBuilder a= "abc"; //convert convert string to stringbuilder
* No initialization: for example   
  int a[]; //array is not initialized by size  
  int a, b=3; a+=b; System.out.printlin(a); // a is not initialized
* Wrong methods: for example  
  String a="abc"; System.out.printlin(a.size()); //should be str.length()  
  char[] c = new char[2]; int length = c.length(); // should be arr.length

### Wrong literal

compiling error:

* use reserved keywords
* use illegal letters except $/leters/\_

|  |
| --- |
| class name can't be "StringBuilder" |
| public class StringBuilder{  public static void main(String[] args){  StringBuilder s1 = new StringBuilder();  StringBuilder s2 = new StringBuilder("abc");  }  } |

|  |
| --- |
| unclosed string literal.  should add escape operator "\\"+"a" -> "\a" |
| String s = "\"+"a"; |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### Wrong declaration

compiling error

* data type mismatch of two side of operator assignment =. These is frequent errors in Java.
* repeat declaration
* reassign array after initialization.
* Don't initialize stringbuilder like string type.
* .equals() is not used on Stringbuilder. Use sb1.toString().equals(sb2.toString())

|  |
| --- |
|  |
| int a = 100L; // long type  float b = 100; // double type  String c = 'abc'; //'a' is char, "a" is String  ArrayList<Integer> d = {1,2,3}; // arraylist can' be initialized as array.  ArrayList<int> e = New ArrayList<>(); //should be wrapper class Integer |

|  |
| --- |
| Wrong initialized size |
| int a = new int[]; // should declare size |
| int a[10] = {1,2,3}; //don't declare size |
| int[] a[10] = {1,2,3};//don't declare size |
| int[10] a = {1,2,3};//don't declare size |

|  |
| --- |
| Wrong class casting: can't convert Integer to Integer[] |
| Integer[] a = new Integer(); |
| int a = {1,2,3}; |

|  |
| --- |
| can't convert. {} is array which is static. Arraylist is dynamic type. |
| ArrayList a = new ArrayList({}); |

|  |
| --- |
| list is interface, which can't be instantiated directly. ArrayList is concrete class |
| List<String> a6 = new List<>(); |

|  |
| --- |
| can't convert list to arraylist |
| ArrayList<String> a7 = Arrays.asList("one", "two"); |

|  |
| --- |
| should use wrapper class Interger rather than primitive type int |
| ArrayList<int> b = new ArrayList<int>(10); |

|  |
| --- |
| repeat define int type |
| int a, b =10;  int a = 1; |

StringBuilder

|  |
| --- |
| compiling error: can't convert string to string builder |
| StringBuilder b = "rumble";  b.append(4).deleteCharAt(3).delete(3, b.length() - 1);  System.out.println(b); |

|  |
| --- |
| compiling error:  error: incomparable types: StringBuilder and String  The two StringBuilder objects are different objects and so are not equal using reference equality. |
| StringBuilder s1 = new StringBuilder("meow");  StringBuilder s2 = new StringBuilder("meow");  if (s1 == s2) System.out.println("one");  if (s1 == "meow") System.out.println("three"); |
| String s1 = "abc";  StringBuilder s2 = new StringBuilder(s1);  System.out.println(s1 == s2); |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### Wrong invoking

common errors

* array can't be reassigned after initialization.
* call attributes as instance method context. array.length is correct because that is not method but attribute . So array.length() is wrong.
* invoke wrong methods. str.equals() is correct. str.equal() or stringbuild.equals() is wrong because no such methods are defined.
* random access is wrong. For example, array[i][j] is correct. array[i,j] is wrong.

The size of array can't be changed after initialization.

|  |
| --- |
| wrong assignment.  c[0]=1;c[1]=2; c[2]=3; |
| int[] c= new int[10];  c={1,2,3}; |

|  |
| --- |
| length is not method but attribute of class int[] |
| int[] b;  System.out.println(b.length()); |

In java/c++/python, for 2D-array, random access should be arr[i][j]. In R, that could be arr[i, j].

|  |
| --- |
| should be times[i][j] |
| int[] times [] = new int[3][3];  for (int i = 0; i < times.length; i++)  for (int j = 0; j < times.length; j++)  times[i, j] = i\*j;  System.out.println(times[2, 2]); |

|  |
| --- |
| The method add(Integer) in the type ArrayList<Integer> is not applicable for the arguments (char) |
| ArrayList<Integer> a8 = new ArrayList<>();  a8.add('A');  a8.add('6');  a8.add("124"); |

|  |
| --- |
| str.concat() only accept char/String type. |
| s="a";  s=s.concat(true);  s=s.concat(4);  s=s.concat('c'); |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### No initialization

String is not initialized

|  |
| --- |
| string is null if that is not initialized after declaration. |
| String s;  System.out.println(s.length()); |

|  |
| --- |
| error: variable a might not have been initialized |
| int test(){  short a;  return a;  } |

|  |
| --- |
| variable b might not have been initialized  array should be initialized once it is declared |
| int[] a; |
| int[] a, b={1,2}; |
| int[] b;  b={1,2,3,4};  System.out.println(b); |

distinguish initialization pattern of local variable and class variables. Rather than class variables, local variables must be initialized once declaration.

|  |
| --- |
| one is local variable |
| public class Counts {  private boolean b;  public static void main(String[] args) {  Counts c = new Counts();  **int one, two = 0; //tricky declaration, variable one is not initialized**  if (c.b) {  System.out.println(one);  }  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### Runtime exceptions.

Compiling is ok. but exceptions will be traced.

* class casting exception
* out of bounds: do random access to array using index.
* Null pointer exceptions: object is no initialized

Compiling is ok, but out of bounds will cause runtime exceptions.

|  |
| --- |
| out of bounds |
| ArrayList<String> a = new ArrayList<String>();  a.add("one"); a.add("two"); a.add("three");  System.out.println(a.get(5));  System.out.println(a.get(-1)); |
| ArrayList<Integer> b = new ArrayList<Integer>(10);  System.out.println(b.get(3)); |

Runtime exceptions

|  |
| --- |
| false  java.lang.NullPointerException: Cannot invoke "Object.equals(Object)" because "<local1>" is null          at Test.main(Test.java:15) |
| A a1 = new A();  A a2 = a1;  a1 = null;  System.out.println(a1==a2);//false  System.out.println(a1.equals(a2)); |

|  |
| --- |
| there is tricky question. answer is G. |
| Which of the following is true?  A. A local variable of type boolean defaults to null.  B. A local variable of type float defaults to 0.  C. A local variable of type Object defaults to null.  D. A local variable of type boolean defaults to false.  E. A local variable of type boolean defaults to true.  F. A local variable of type float defaults to 0.0.  G. None of the above |

|  |
| --- |
| s[2]==null  NullPointerExceptions because i is out of bounds |
| String[] s = new String[3];  String[] a= {"a", "b"};  for(int i=0; i<s.length; i++){  s[i]=a[i];  System.out.println(s[i].toUpperCase());  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### Date and time

common errors:

* compiling error: apply wrong methods
* Forget reassignment.
* runtime exception: wrong arguments

|  |
| --- |
| compiling error: class LocalDate doesn't declared the method plusHours() |
| LocalDate date = LocalDate.parse("2018-04-30", DateTimeFormatter.ISO\_LOCAL\_DATE);  date.plusDays(2);  date.plusHours(3);  System.out.println(date.getYear() + " " + date.getMonth() + " "  + date.getDayOfMonth()); |

|  |
| --- |
| Runtime Exception |
| LocalDate date = LocalDate.of(2018, Month.APRIL, 40);  System.out.println(date.getYear() + " " + date.getMonth() + " "  + date.getDayOfMonth()); |

|  |
| --- |
| compiling is ok. but year is n't increased. |
| LocalDate date = LocalDate.of(2012,3,4);  **date.plusYear(3);**  System.out.println(date); |

|  |
| --- |
| runtimeException:  wrong type because f is dattimeformater but d is date object |
| LocalDate d = LocalDate.of(2015, 5,7);  Period p = Period.of(1, 2, 3);  d = d.minus(p);  DateTimeFormatter f = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);  System.out.println(f.format(d)); |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## Debug flow-control

### switch-case

switch-case doesn't support Boolean type

|  |
| --- |
|  |
| static void test(boolean val){  switch(val){  case true:  System.out.println("true");break;  case false:  System.out.println("false");break;  default:  System.out.println("nothing");break;  }  } |

In Switch-case statements, only case/default statement is acceptable. Other blocks would be ignored.

|  |
| --- |
| not compiling. |
| static void test4(int day){  switch(day){  if(day>3) {  System.out.print("good ");  }  case 'T':  System.out.print("Tuesday ");break;  default:  System.out.print("Nothing ");break;  }  System.out.print("\n");  } |

After break statement, the statement would never be invoked.

|  |
| --- |
| unreachable code |
| static void test4(int day){  switch(day){  case 'M':  System.out.print("Monday "); break;  System.out.print("good ");  case 'T':  System.out.print("Tuesday ");break;  default:  System.out.print("Nothing ");break;  }  System.out.print("\n");  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### out of scope

For if-else, for, while, the condition expression should be encompassed by a pair of parenthesis "()". Statement blocks should be encompassed by curly brackets "{}" if there are multiple statements in one block. Any variables within block is local variables.

|  |
| --- |
| output: a b c followed by exception of NullPointerException  the print statement is out of the scope of if-flow control, so b[3]==null can't be converted to upper case. {} should be added. |
| String[] a= {"a", "b", "c"};  String[] b= new String[4];  for(int i=0; i<b.length;i++){  **if(i<a.length)**  **b[i]=a[i];**  System.out.print(b[i].toUppercase() + " "); // this is out of if control  } |

|  |
| --- |
| s1 is local variable that work within 2nd if-block. |
| class Sparrow extends Bird { }  public abstract class Bird {  public static void main(String[] args) {  Bird b = new Sparrow();  Sparrow s = new Sparrow();  if (b == s) { System.out.println("=="); }  if (b.equals(s)) {  System.out.println("bird");  **Sparrow s1 = new Sparrow();**  }  if (b.equals(s1))  System.out.println("sparrow");  }  } |

|  |
| --- |
| compiling error  b is local variable separately define in if and else block |
| public class Test{  public static int count(int a) {  if (a != 3) { int b = 1;  } else { int b = 2; }  return a++ + b;  }  public static void main(String[] args) {  System.out.print(count(3));  System.out.print(count(9));  }  } |

|  |
| --- |
| s1 is local variable. |
| class Sparrow extends Bird { }  public abstract class Bird {  public static void main(String[] args) {  Bird b = new Sparrow();  Sparrow s = new Sparrow();  if (b == s) { System.out.println("=="); }  if (b.equals(s)) {  System.out.println("bird");  Sparrow s1 = new Sparrow();  }  if (b.equals(s1))  System.out.println("sparrow");  }  } |

### duplicate blocks

repeating some statement blocks is allowed. for example:

If-else if - else -if …

switch-case - case …

try-catch-catch ..

multiple initializer blocks.

But some cases are invalid. For example,

If-else-else

switch-catch-default-default

try-catch(Exception e)-catch(Exception e)

return; return;

duplicate method with identical signatures.

|  |
| --- |
| else statement should be at most one. |
| int x = 4;  long y = x \* 4 - x++;  if(y˂10) System.out.println("Too Low");  else System.out.println("Just right");  else System.out.println("Too High"); |

## Debug iteration

Common errors for For-loop/ while-loop

* condition expression isn’t encompassed by "()".
* Multiple statements are not encompassed by "{}".
* infinite loop
* local variable declared within loop is invoked outside.
* Data type is not covariant.

### wrong encompassing

For if-else, for, while, the condition expression should be encompassed by a pair of parenthesis "()". Statement blocks should be encompassed by curly brackets "{}" if there are multiple statements in one block.

|  |
| --- |
| compiling error  should be "while (x<10){y--; x++}" |
| int x = 1, y = 15;  while x ˂ 10  y--;  x++;  System.out.println(x+", "+y); |

|  |
| --- |
|  |
|  |

### infinite loop

1. Infinite loop may cause timeout. break point must be tested.

2. while(true){…} approach is suggested. Note: while(1){…} is not working in Java but works in C/C++.

Those infinite loop could be used if some break conditions are correctly configured.

|  |
| --- |
|  |
| for(,,){  …  } |
| while(true){  ….  } |
| do{  …  }while(true); |

here are examples of infinite loop

|  |
| --- |
| static void test3(){  int i=0;  while(i<3){  **i=i++;//i is always equal to 0.**  System.out.print(i);  }  } |
| static void test4(){  int i=-10; // wrong initial  while(i != 0){  i--;  }  } |
| public void sillyLoop( int i ){  while ( i != 0 )  i-- ;  } |

|  |
| --- |
| infinite loop |
| int x = 5;  while(x ˃= 0) {  int y = 3;  while (y ˃ 0) {  if(x˂2) continue;  x--; y--;  System.out.print(x\*y+" ");  }  } |

|  |
| --- |
|  |
|  |

### local variable

variable that is declared within loop block is local variable, which can't be invoked out of the scope.

Watch out any for or while loop without {}. It is more likely local variables declared in loop is out of scope or some statements are useless.

|  |
| --- |
| not compiling |
| static void test5(){  do{  int i =1;// i is local variable  System.out.println(i++);  }while(i<=10);  } |
| static void test1(){  int a; //variable is not initialized  while(a==10){  System.out.println("test1");  break;  }  } |

Another possibility is the curly bracket is missing.

|  |
| --- |
|  |
| static void test1(){  int a[] = {1,2,3};  for(int i = 0; i < a.length; i++)  System.out.print(a[i]);  a[i] += 10;  } |
|  |

|  |
| --- |
| bad initializer |
| static void test21(){  int a[] = {4,5,6};  int i;  for(i: a)  System.out.print(i);  System.out.println("");  } |

|  |
| --- |
| for loop only work on Line5. Line6 is out of for loop. The indentation may lure you confusion. |
| 3: String[] values = {"one","two","three"};  4: for(int index = 0; index ˂ values.length; index++)  5: System.out.print(values[index]);  6: System.out.print(index); |

|  |
| --- |
| y is local variable within do body statements |
| int x = 1;  do {  int y = 1;  System.out.print(y++ + " ");  } while(y ˂= 10); |

### Boolean expression

while/do-while loop should define correct Boolean expression

|  |
| --- |
| not compiling: can't convert int to boolean |
| static void test1(){  int a;  while(a=10){  System.out.println("test1");  break;  }  } |
| static void test1(){  int a;  while(1){  System.out.println("test1");  break;  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### out of bounds

Array is static data type. Runtime exceptions could be throwed if the index is out of bounds of array.

|  |
| --- |
| i would be 0 1 2 3. 3 is out of bounds |
| static void test2(){  int a[]= {1,2,3};  for(int i=0; i++<a.length;)  System.out.println(a[i]);  } |

|  |
| --- |
| i<a.length |
| int[] a= {1, 2, 3};  for(int i = 0; i<10; i++){  System.out.print(a[i]);  } |

|  |
| --- |
| throw NullPointerException  a[3] is null, which can't be converted. |
| static void test2(){  **String[] a= new String[4];**  String[] sb = {"red", "blue", "yellow"};  for(int i = 0; i<sb.length; i++){  a[i] = sb[i];  }  for(int j = 0; j<a.length; j++){  String s = a[j].toLowerCase();  System.out.print(s);  }  } |

|  |
| --- |
|  |
|  |

### Wrong data type

Data type should be covariant when iterating collections

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## Debug OOP

### class

|  |
| --- |
| This is correct class  "javac Duck.java"  "java Duck Duck Goose" |
| public class Duck {  public static void main(String[] args) {  for (int i = 1; i ˂ args.length; i++)  System.out.println(args[i]);  }  } |

Here are some examples of wrong coding that may appear in practice

|  |
| --- |
|  |
| public class Duck {  public void main(String[] args) { //main() should be static  for (int i = 1; i ˂ args.length; i++)  System.out.println(args[i]);  }  } |
| public class Duck {  public static void Main(String[] args) { // must be main()  for (int i = 1; i ˂ args.length; i++)  System.out.println(args[i]);  }  } |
| class Duck { // top-level class must be public with main() methods  public static void main(String[] args) {  for (int i = 1; i ˂ args.length; i++)  System.out.println(args[i]);  }  } |
| public class Duck {  public static void main(String[] args) {  for (int i = 1; i ˂= args.length; i++) // out of bounds  System.out.println(args[i]);  }  } |
| public class duck { // class name and file name of \*.java should be identical  public static void main(String[] args) {  for (int i = 1; i ˂ args.length; i++)  System.out.println(args[i]);  }  } |
| public class Duck {  public static main(String[] args) { // return type is usually void  for (int i = 1; i ˂ args.length; i++)  System.out.println(args[i]);  }  } |
| public class Duck {  static void main(String[] args) { // main() must be marked as "public static"  for (int i = 1; i ˂ args.length; i++)  System.out.println(args[i]);  }  } |
| public class Duck extends A{  public static void main(String[] args) {  for (int i = 1; i ˂ args.length; i++)  System.out.println(args[i]);  }  }  public class A{} // only one public class . class A is not top-level class. class A can't be public |
| public class Duck {  public static void main(String args) {// should be String array type  for (int i = 1; i ˂ args.length; i++)  System.out.println(args[i]);  }  } |
| public class Duck {  public static void main(String.. 23args) { // literal must start with letters/$/\_  for (int i = 1; i ˂ args.length; i++)  System.out.println(args[i]);  }  } |
| public class Duck {  public static void main(String[] args) {  for ( int i; i ˂ args.length; i++) // int type should be initialized before calling  System.out.println(args[i]);  }  } |
| public class Duck {  public static void main(String[] args) {  for (int i = 1; i ˂ args.length(); i++) // length is not method but attribute  System.out.println(args[i]);  }  } |

|  |
| --- |
|  |
|  |

### constructor

compiling errors:

* **this() or super() should be placed on the first statement in the constructor**
* Miss super() if the constructors in superclass need arguments.
* overloading: return type of overloading is not identical or repeat declaration
* inheritance: miss super() in subclass because conductor can't be overloaded
* miss data type in argument list. for example:  
  void test(int a, b) {…}
* instance variables is not declared once they are instantiated in constructor.
* Update final class variables in constructor
* circular instance: Instantiate itself within constructor

Errors that may not cause compiling errors:

* Declare local variables as instance variables wrongly
* Declare wrongly constructor as methods by adding return type
* abstraction: it is not allowed to define constructor in abstract class or interface

Runtime exceptions

* instance with wrong arguments
* ClassCasting exceptions

|  |
| --- |
| repeat define constructor |
| public class Test {  Test(){  System.out.println("constructor");  }  Test(){  System.out.println("constructor");  }  public static void main(String[] args) {  new Test();  }  } |
| error: constructor Animal in class Animal cannot be applied to given types;  constructor can't be overridden. "super(age)" refers two constructors of class Animal and Tiger. |
| public class Test extends Tiger{  Test(){ super(1); }  **Test(int age){ super(age); }**  public static void main(String[] args) {  Test t1 = new Test();  Test t2 = new Test(10);  System.out.println(t1.age);  System.out.println(t2.age);  }  }  class Animal{  int age;  Animal(int age){  this.age = age;  }  }  class Tiger extends Animal{  int age;  Tiger(int age){  this.age = age;  }  } |

|  |
| --- |
| compiling error:  this() should be at the first statement. |
| class Animal {  public Animal(int age) {  System.out.println("1");  this();  }  public Animal() { System.out.println("2"); }  }  public class Buffalo extends Animal {  public Buffalo() { System.out.println("3"); }  } |

wrong overloading

|  |
| --- |
| not compiling: constructor Animal(int) is already defined in class Animal |
| class Animal{  int age;  Animal(int age){  age = this.age;  }  Animal(int age){  age = 10;  }  } |

|  |
| --- |
| not compiling: variable should be declared before initialization |
| class Animal{  Animal(int age){  this.age = 10;  }  } |

constructor of abstract class can't be executed by instantiation of abstract class

|  |
| --- |
| Cannot instantiate the type Animal |
| public class ConstructorTest extends Animal{  public static void main(String[] args){  Animal c = new Animal(3);  System.out.println(c.age);  }  }  **abstract class Animal{**  int age;  Animal(int age){  age = this.age;  }  } |

wrong access identifier

|  |
| --- |
| not compiling: The constructor Animal() is undefined |
| public class ConstructorTest extends Animal{  public static void main(String[] args){  Animal c = new Animal();  System.out.println(c.age);  }  }  class Animal{  int age;  private Animal(int age){  this.age = age;  }  } |

|  |
| --- |
| wrong type of arguments. variable a is null |
| public class ConstructorTest{  int age;  ConstructorTest(int age){  this.age = age;  }  public static void main(String[] args){  int a;  ConstructorTest c = new ConstructorTest(a);  System.out.println(c.age);  }  } |

|  |
| --- |
| this() must be on the first line in the constructor |
| public class Cub {  private String name;  private double weight;  public Cub(double weight) {  this.weight = weight;  this("", weight);  }  public Cub(String name, double weight) {  weight = weight;  this.name = name;  }  public static void main(String[] args) {  Cub cub = new Cub(44);  System.out.println(cub.weight + "" + cub.name);  }  } |

instance methods can't be invoked as static context.

|  |
| --- |
|  |
| public class MethodInstance extends Animal{  public static void main(String[] args) {  int good = 1;  System.out.println(getAge(good, 3));  }  }  class Animal{  int getAge(int x, int y){ //instance method  return x+y;  }  } |
| //correct 1:  public class MethodInstance extends Animal{  public static void main(String[] args) {  int good = 1;  System.out.println(getAge(good, 3));  }  }  class Animal{  **static int getAge(int x, int y){ //static method**  return x+y;  }  } |
| //correct 2:  public class MethodInstance extends Animal{  public static void main(String[] args) {  int good = 1;  Animal c = new Animal(); //object reference  System.out.println(c.getAge(good, 3));  }  }  class Animal{  int getAge(int x, int y){  return x+y;  }  } |

|  |
| --- |
| Exception in thread "main" java.lang.StackOverflowError.  This is nested declaration and instantiation. |
| public class ClassTest {  int age;  ClassTest t1 = new ClassTest(1);  ClassTest(int a){  this.age = a;  }  public static void main(String[] args){  ClassTest t = new ClassTest(10);  }  } |

|  |
| --- |
| duplicate declare reference variable |
| public class Cardinal {  static int number;  Cardinal() { number++; }  public static void main(String[] args) {  Cardinal c1 = new Cardinal();  if (c1 == null) {  Cardinal c2 = new Cardinal();  } else {  Cardinal c2 = new Cardinal();  }  Cardinal c2 = new Cardinal();  System.out.println(c1.number);  } |

|  |
| --- |
| compiling error: circular instance till the program runs out of memory |
| public class Lion {  Lion l = new Lion();  static void public main(String[] args) {  new Lion();  }  public void roar() {  Lion l = new Lion();  if ( l == l) System.out.println("roar!");  }  } |
|  |
|  |

|  |
| --- |
|  |
|  |

### instance method

compiling errors:

* miss return type or wrong place when declaration
* miss return statement if return type is declared.
* unreachable code.
* invoke instance method in static context, or static method in instance context.

|  |
| --- |
| wrong return type |
| void test4(){  return 3;  } |
| int test4(){  return 3f;  } |

|  |
| --- |
| unreachable statement |
| int test4(){  return 5;  return 3;  } |
| int test4(){  return 5;  int a = 5;  } |

|  |
| --- |
| can't convert |
| int test4(){  return (int) "abc";  } |

|  |
| --- |
|  |
|  |

Any statements may be unreachable codes if they are after return statements, continue/break statements without conditions, or statements with throwed exceptions. Those statements would never be invoked though some of them may not raise compiling errors.

|  |
| --- |
| Compiling error the line is never reached |
| void test(){  return 3;  int a =3;  } |

|  |
| --- |
| The statement is useless though no compiling error is raised. |
| int test(){  if(true) return 3;  else return 0;  } |

|  |
| --- |
| repeat virtual methods |
| public class Test {  void Test(){  System.out.println("constructor");  }  protected void Test(){  System.out.println("constructor");  }  public static void main(String[] args) {  new Test();  }  } |

|  |
| --- |
|  |
|  |

Invoke methods with wrong arguments

|  |
| --- |
|  |
| public class MethodArgument {  void test1(){  System.out.println("print");  }  void test2(int a, float b){  System.out.println(a+b);  }  public static void main(String[] args){  MethodArgument C = new MethodArgument();  C.test1(45); //no arguments allowed  C.test2(3, 3.3); // the 2n argument should be float type  }    } |

|  |
| --- |
| If arguments is varargs pattern, .. |
| public int howMany(boolean b, boolean... b2) {  return b2.length;  }  howMany(true, {true, false}); |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### inheritance

compiling errors

* Java doesn't support multiple inheritance
* declare repeat default methods when extends multiple interface.
* Checked exceptions are not declared in subclass. Unchecked Exceptions have not such limits.
* Wrong literal: "class extends class", "class implements interface", or "interface extends interface" is correct. Any other combinations are wrong.
* miss super() or placed wrong place if constructor in super class needs arguments
* Invoke private methods or private variables in subclass

|  |
| --- |
| java doesn't support multiple inheritance |
| public class InitialOrder2 extends Tiger, Bird {  public static void main(String[] args){  new InitialOrder2();  }  }  class Tiger{  Tiger(){ System.out.print("tiger "); }  }  class Bird{  Bird(){ System.out.println("bird "); }  } |

|  |
| --- |
| error: types MyInterface and Test are incompatible;  the two methods getName() have the same signature. |
| public class InterfaceTest implements MyInterface, Test{  public static void main(String[] args){  InterfaceTest c = new InterfaceTest();  System.out.println(c.getName());  }  }  interface MyInterface{  default String getName() { return "inteface"; }  }  interface Test{  default String getName() { return "test"; }  } |

|  |
| --- |
| error: unreported exception Exception; must be caught or declared to be thrown  main() invoke inherited method getName(), which declared checked exceptions. Compatible exceptions should be declared in main(). |
| public class MethodInherit1 extends Test{  public static void main(String[] args) {  new MethodInherit1().getName();  }  }  class Test{  **void getName() throws Exception{**  System.out.println("super");  }  } |

|  |
| --- |
| can't omit super if super constructor need arguments.  put "super(age);" on the first line of constructor Mammal(). |
| class Mammal extends Animal{  String gender;  Mammal(int age, String gender){  this.gender = gender;  }  }  class Animal{  int age;  Animal(int age){ this.age = age; }  } |
| class Mammal extends Animal{  String gender;  Mammal(int age, String gender){  this.gender = gender;  super(age, gender);  }  }  class Animal{  int age;  Animal(int age){ this.age = age; }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### polymorphism

wrong casting will cause ClassCastException. **ClassCastException is not compiling exception but runtime exception.**

**When parent reference refer to child object, child method could be invoked only if there is corresponding method in parent.**

Parent reference could point to child object. That could call child methods only if parent methods have the same signature

|  |
| --- |
| should be " System.out.println(c.calRate(.15f));" |
| public class ClassMethodCast extends Test{  int age = 10;  **float calRate(){**  return (float) this.age \* 10;  }  public static void main(String[] args){  Test c = new Test();  System.out.println(c.calRate(.15f));    c = new ClassMethodCast();  System.out.println(c.calRate());  }  }  class Test{  int age = 10;  **float calRate(float rate){**  return (float) this.age \* rate;  }  } |

When parent reference refer to child object, child method could be invoked only if there is corresponding method in parent.

|  |
| --- |
|  |
| class Animal {  public int getAge() {return 10;}  }  class Mammal extends Animal {  protected int getAge(int input) {return 7;}  }  public class Sloth extends Mammal {  public boolean hasFur() {return true;}  public static void main(String[] args) {  **Mammal sloth = new Sloth(); //casting**  System.out.print(sloth.getAge());  System.out.print(sloth.getAge(2));  System.out.print(sloth.hasFur());  }  } |

|  |
| --- |
| ClassCastException  can't convert int array to String array. |
| int[] nums = new int[] { 1, 4, 6};  Object p = nums;  String[] two = (String[]) p; |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

**private access**

reference of abstract class type could points to object of concrete class, and employ methods defined in concrete class.

|  |
| --- |
| output is concrete. AbstractCast.fly() is overridden by Concrete.fly(). but if AbstractCast.fly() is private. AbstractCast.fly() can't be overridden but only be hidden. |
| public abstract class AbstractCast {  **void fly(){**  System.out.println("abstract");  }  public static void main(String[] args){  AbstractCast c = new Concrete();  c.fly();  }    }  class Concrete extends AbstractCast{  **void fly(){**  System.out.println("concrete");  }  } |
|  |

|  |
| --- |
| //output is abstract |
| public abstract class AbstractCast {  **private void fly(){**  **System.out.println("abstract");**  **}**  public static void main(String[] args){  AbstractCast c = new Concrete();  c.fly();  }    }  class Concrete extends AbstractCast{  **protected void fly(){**  System.out.println("concrete");  }  } |
| /not compiling  public abstract class AbstractCast {  **private void fly();**  public static void main(String[] args){  AbstractCast c = new Concrete();  c.fly();  }    }  class Concrete extends AbstractCast{  **protected void fly(){**  System.out.println("concrete");  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### overidding

overriding criteria:

* inheritance of instance methods
* identical method name and argument list

common errors:

* return type is not covariant
* throwed exceptions are not compatible
* access identifier are not compatible.
* final methods can't be overridden but hidden.
* private methods can't be overridden but hidden.
* Instance method can't override static methods
* constructor can't be overridden

can't override final instance methods

|  |
| --- |
| overridden method is final |
| public class MethodFinal extends Test{  void test(){  System.out.println("child");  }  public static void main(String[] args){  }    }  class Test{  final void test(){  System.out.println("child");  }  } |

can't override private instance methods

|  |
| --- |
| error: hasHorns() in Reindeer cannot override hasHorns() in Deer |
| public class Deer {  public Deer() {  System.out.print("Deer");  }  public Deer(int age) {  System.out.print("DeerAge");  }  boolean hasHorns() {  return false;  }  public static void main(String[] args) {  Deer deer = new Reindeer(5);  System.out.println(","+deer.hasHorns());  }  }  class Reindeer extends Deer {  public Reindeer(int age) {  System.out.print("Reindeer");  }  private boolean hasHorns() {  return true;  }  } |
|  |

Return type should be covariant. The java. lang.Number class is the superclass of classes BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short.

|  |
| --- |
| output:  The return type is incompatible with Rodent.chew()  Note: the throwed exception is ok. |
| class Rodent {  protected Integer chew() throws Exception {  System.out.println("Rodent is chewing");  return 1;  }  }  class Beaver extends Rodent {  public Number chew() throws RuntimeException {  System.out.println("Beaver is chewing on wood");  return 2;  }  } |

instance methods can't override static methods

|  |
| --- |
| output:  This instance method cannot override the static method from Rodent |
| class Rodent {  protected static Number chew() throws Exception {  System.out.println("Rodent is chewing");  return 1;  }  }  class Beaver extends Rodent {  public Integer chew() throws RuntimeException {  System.out.println("Beaver is chewing on wood");  return 2;  }  } |

Exceptions should be compatible.

|  |
| --- |
| output:  The method chew() from the type Beaver refers to the missing type IOException |
| class Rodent {  protected Number chew() throws Exception {  System.out.println("Rodent is chewing");  return 1;  }  }  class Beaver extends Rodent {  public Integer chew() throws IOException {  System.out.println("Beaver is chewing on wood");  return 2;  }  } |

|  |
| --- |
| output:  Cannot reduce the visibility of the inherited method from Rodent  the overriding method should be marked as protected or public |
| class Rodent {  protected Number chew(){  System.out.println("Rodent is chewing");  return 1;  }  }  class Beaver extends Rodent {  Integer chew(){  System.out.println("Beaver is chewing on wood");  return 2;  }  } |

|  |
| --- |
| return type should be int |
| interface Aquatic {  **public default int getNumberOfGills(int input) { return 2; }**  }  public class ClownFish implements Aquatic {  public String getNumberOfGills() { return "4"; }  public String getNumberOfGills(int input) { return "6"; }  public static void main(String[] args) {  System.out.println(new ClownFish().getNumberOfGills(-1));  }  } |

### final identifier

final variables could be reassigned once initialization is done.

|  |
| --- |
| cannot assign a value to final variable age |
| public class ClassTest {  final int age = 1;  ClassTest(int a){  this.age = a;  }  public static void main(String[] args){  ClassTest t = new ClassTest(1);  System.out.println(t.age);  ClassTest t2 = new ClassTest(10);  System.out.println(t2.age);  }  } |

|  |
| --- |
| error: hasHorns() in Reindeer cannot override hasHorns() in Deer |
| public class Deer {  public Deer() {  System.out.print("Deer");  }  public Deer(int age) {  System.out.print("DeerAge");  }  final boolean hasHorns() {  return false;  }  public static void main(String[] args) {  Deer deer = new Reindeer(5);  System.out.println(","+deer.hasHorns());  }  }  class Reindeer extends Deer {  public Reindeer(int age) {  System.out.print("Reindeer");  }  public boolean hasHorns() {  return true;  }  } |

|  |
| --- |
| modifier final not allowed here |
| public class Test {  final Test(){  System.out.println("constructor");  }  public static void main(String[] args) {  new Test();  }  } |

can't override final instance methods

|  |
| --- |
| overridden method is final |
| public class MethodFinal extends Test{  void test(){  System.out.println("child");  }  public static void main(String[] args){  }    }  class Test{  final void test(){  System.out.println("child");  }  } |

|  |
| --- |
| compiling error: final methods can't be overridden. |
| abstract class Reptile {  **public final void layEggs() {**  System.out.println("Reptile laying eggs");  }  public static void main(String[] args) {  Reptile reptile = new Lizard();  reptile.layEggs();  }  }  public class Lizard extends Reptile {  public void layEggs() {  System.out.println("Lizard laying eggs");  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
| cannot assign a value to final interface variable |
| interface MyVariable{  int weight=1;  }  public class InterfaceVariable implements MyVariable{  int weight = 10;  int height = 20;  public static void main(String[] args){  InterfaceVariable a = new InterfaceVariable();  //hide interface variable  a.weight = 21;  System.out.println(a.weight);  MyVariable.weight = 42;  System.out.println(MyVariable.weight);  }  } |

|  |
| --- |
|  |
|  |

### private identifier

common error:

* Runtime exception for classCastException: private instance methods in base class are wrongly taken as being overridden.
* Compiling error: can't reduce visibility of instance methods in
* Invoke private methods outside.

|  |
| --- |
| output:  Cannot reduce the visibility of the inherited method from Test  If the method test() in super class is marked as default, the method test() in subclass should be marked as default protected or public. |
| public class MethodPrivate extends Test{  private void test(){  System.out.println("child");  }  public static void main(String[] args){  new MethodPrivate().test();  }  }  class Test{  **void test(){**  System.out.println("parent");  }  } |

|  |
| --- |
| compiling error:  test() in MethodFinal cannot override test() in Test |
| public class MethodFinal extends Test{  **private void test(){**  System.out.println("child");  }  public static void main(String[] args){  **Test d = new MethodFinal();**  d.test();  }    }  class Test{  void test(){  System.out.println("parent");  }  } |

Runtime exception: ClassCastingException

|  |
| --- |
| Runtime exception:  The method test() from the type Test is not visible |
| public class MethodPrivate extends Test{  void test(){  System.out.println("child");  }  public static void main(String[] args){  Test t = new MethodPrivate();  t.test();  }  }  class Test{  **private void test(){**  System.out.println("parent");  }  } |

|  |
| --- |
| private methods can't be invoked within other class methods |
| public class Fish {  **private static String getColor() { return "Yellow"; }**  public static void main(String[] args) {  new Pufferfish().printDescription();  }  }  class Pufferfish extends Fish {  protected static String getColor() { return "Green"; }  public void printDescription() {  System.out.println(super.getColor()+","+this.getColor()+","+getColor());  }  } |

|  |
| --- |
|  |
|  |

### static

Instance initializers are not allowed for static variables.

|  |
| --- |
| public class Dino {  static final String species;  double weight;  { species = "Raptor"; }  public Dino(double weight) {  this.weight = weight;  }  public static void main(String[] args) {  Dino dino = new Dino(500);  System.out.println(dino.weight);  }  } |
| interface MyVariable{  static int height;  static {  height = 2;  }  } |
| interface MyVariable{  static int height;  {  height = 2;  }  } |
| interface Test{  public static final int age;  } |

|  |
| --- |
| super. is not working because interface variables are static |
| interface MyVariable{  int weight=1;  }  public class InterfaceVariable implements MyVariable{  int weight = 10;  int height = 20;  public static void main(String[] args){  InterfaceVariable a = new InterfaceVariable();  //hide interface variable  System.out.println(a.weight);  // implement interface variable  System.out.println(MyVariable.weight);  System.out.println(super.weight);  }  } |

|  |
| --- |
|  |
|  |

## Debug exceptions

do debugging following the below:

* Exception statements should follow one of patterns: try-catch, try-catch-finally, try-finally. Try statements itself cause compiling error.
* Checked exceptions should be declared before calling. For example, if "throw new Exception();" exists in certain method, this method and any methods which call this method should be marked with "throws Exception".
* Exceptions of catch statements in exception chain should in order from small to broader exceptions. For example, place ArithmeticException before RuntimeExceptions.
* java.lang.Exception must be placed after all other exceptions.
* For customary exceptions, exceptions of subclass must be compatible with those of superclass.
* if unreachable code exists. For example, some statements are place after "throw new Exceptions();". Those statements would never be executed.

### wrong try-catch-finally blocks

|  |
| --- |
| try-catch statements must be encompassed by curly brackets {} |
| static void test(int a){  try  System.out.println(10/a);  catch(Exception e)  System.out.println(e);  } |

|  |
| --- |
| try statement must be combined with try-catch or try-finally statement. |
| static void test(int a){  try{  System.out.println(10/a);  }  } |

|  |
| --- |
| catch and finally block is wrong order |
| try{  System.out.println(10/a);  }catch(Exception e){  System.out.println(e);  }finally{  System.out.println("final");  }catch(Exception e){  System.out.println(e);  } |

|  |
| --- |
| finally block should be place at the end |
| public static void main(String[] args) {  int i = 0;  try {  i += 1;  e();  i += 2;  } finally {  i += 8;  }catch (Exception e) {  i += 4;  }  System.out.print(i);  }  private static void e() {  throws new IllegalArgumentException();  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### Unreachable exceptions

|  |
| --- |
| Unreachable catch block for ArithmeticException. It is already handled by the catch block for Exception. ArithmeticException->RuntimeException->Exception |
| static void test(int a){  try{  System.out.println(10/a);  }catch(Exception e){  System.out.println(e);  }catch(ArithmeticException e){  System.out.println(e);  }finally{  System.out.println("final");  }  System.out.println("test");  } |

|  |
| --- |
| compiling error:  Exception in thread "main" java.lang.Error: Unresolved compilation problem:  Unreachable catch block for MyException. This exception is never thrown from the try statement body at ExceptionCustomary2.main(ExceptionCustomary2.java:6)  The statements in try block throws "ArithmeticException", which is sibling with customary exception MyException. "MyException" should be subject to "ArithmeticException".  correct: class MyException extends ArithmeticException{  or  class MyException extends RuntimeException{ |
| public class ExceptionCustomary2{  public static void main(String[] args){  int a=5, b=0;  try{  int res = a/b;  }catch(MyException e){  System.out.println(e);  }  }  }  class MyException extends Exception{  MyException(String msg){  super(msg);  }  } |

|  |
| --- |
| The 2nd catched exception could be Exception or RuntimeException. |
| class LimpException extends Exception{}  class HurtException extends LimpException{}  public void run() {  try {  split();  }catch(HurtException e) {  } catch(LimpException e) {  }  }  private void split() throws HurtException {} |

|  |
| --- |
|  |
|  |

### Unreachable code

|  |
| --- |
| compiling error: all statements after exceptions are thrown are unreachable code  The code marked as red is never reachable. |
| try{  throw new RuntimeException();  System.out.println("1");  }catch(RuntimeException e){  System.out.println("2");  } |
| public class ExceptionMask {  public static void main(String[] args){  try{  throw new RuntimeException();  }catch(RuntimeException e){  System.out.println("2");  }  System.out.println("4");  }  } |

|  |
| --- |
| The code snippet always cause compiling error because there is throw exceptions in catch blocks. Therefore, those statements marked by red would never be reached. |
| try{  throw new RuntimeException();  }catch(RuntimeException e){  throw new RuntimeException();  }  System.out.println("5"); |
| 1  2  3  4  Exception in thread "main" java.lang.RuntimeException  at ExceptionTest.test(ExceptionTest.java:10)  at ExceptionTest.main(ExceptionTest.java:17) |
| public class ExceptionTest {  void test(){  System.out.println("1");  try{  System.out.println("2");  throw new RuntimeException();  }catch(RuntimeException e){  System.out.println("3");  throw new RuntimeException();  }finally{  System.out.println("4");  }  }  public static void main(String[] args){  ExceptionTest c = new ExceptionTest();  c.test();  }  } |
|  |

|  |
| --- |
|  |
|  |

### Wrong exception chain

* Exception is not be placed before any other exceptions because Exception catch all exceptions..
* Rear exceptions in the exception chain should be wider than previous exceptions. For example, RuntimeException should follow other subclass of RuntimeException.

|  |  |
| --- | --- |
|  | correct |
| ArrayIndexOutOfBoundsException is subclass of RuntimeException | ArrayIndexOutOfBoundsException, IOException, RuntimeException |
|  | IOException, ArrayIndexOutOfBoundsException, Exception |

|  |  |
| --- | --- |
|  | wrong |
| Exception should be the end | ArrayIndexOutOfBoundsException, Exception, IOException |
| Exception, RuntimeException |
| ArrayIndexOutOfBoundsException should be after RuntimeException | RuntimeException, ArrayIndexOutOfBoundsException, Exception |

### unhandled exception type

|  |
| --- |
| The code snippet always cause compiling error because throw exceptions in finally block can't be handled |
| try{  throw new RuntimeException();  }catch(RuntimeException e){  throw new RuntimeException();  }finally{  throw new Exception();  } |

|  |
| --- |
| output:  ExceptionInherit.java:6: error: unreported exception myExceptions; must be caught or declared to be thrown          new ExceptionInherit().go();                                   ^  1 error  go() throw exceptions. the method main() should declare exceptions      public static void main(String[] args) throws myExceptions{ |
| public class ExceptionInherit extends myTest{  ExceptionInherit(){  System.out.println("parent");  }  public static void main(String[] args){  new ExceptionInherit().go();  }  }  class myExceptions extends Exception{  myExceptions(String msg){  super(msg);  }  }  class myTest{  myTest(){  System.out.println("child");  }  void go() throws myExceptions{  throw new myExceptions("child exceptions");  }  } |

The overridden methods should have the same signature including throw exceptions.

|  |
| --- |
| output:  ExceptionInherit2.java:5: error: go() in ExceptionInherit2 cannot override go() in myTest      void go() throws myExceptions{           ^    overridden method does not throw myExceptions  1 error |
| public class ExceptionInherit2 extends myTest{  ExceptionInherit2(){  System.out.println("parent");  }  void go() throws myExceptions{  throw new myExceptions("parent exceptions");  }  public static void main(String[] args) throws myExceptions{  new ExceptionInherit2().go();  }  }  class myExceptions extends Exception{  myExceptions(String msg){  super(msg);  }  }  class myTest{  myTest(){  System.out.println("child");  }  void go(){  throw new myExceptions("child exceptions");  }  } |

|  |
| --- |
|  |
|  |

### No declaration

|  |
| --- |
| Unhandled exception type Exception. exceptions should be declared before throwing checked exception |
| public class ExceptionThrows {  public static void main(String[] args){  throw new Exception("throw exception");  }  } |

|  |
| --- |
| Both main() and eat() should throws IOException |
| public class Supper {  **public static void eat() throws IOException {**  try {  System.out.print("1");  throw new IOException();  }catch(IOException e) {  System.out.println("2");  throw e;  }finally {  System.out.println("3");  }  }  public static void main(String [] args) {  eat(); //this line raise compiling error  System.out.println("4");  }  } |

|  |
| --- |
| compiling error  main() invoke eat(). main() must throws IOException |
| public class Supper {  public static void eat() throws IOException {  try {  System.out.print("1");  throw new IOException();  }catch(IOException e) {  System.out.println("2");  throw e;  }finally {  System.out.println("3");  }  }  public static void main(String [] args) {  eat();  System.out.println("4");  }  } |

### duplicate caught

|  |
| --- |
| HurtException is already caught by split() and noother LimpExceptions are thrown. That is unreachable exceptions. |
| public class ExceptionInherit1 {  private static void split() throws HurtException {  throw new HurtException();  }  static void run(){  try{ split();  }catch(LimpException e){ System.out.print("2 ");  }catch(HurtException e){ System.out.print("3 "); }  }  public static void main(String[] args) throws Exception {  run();  }  }  class LimpException extends Exception{  LimpException(){ System.out.print("limp "); }  }  class HurtException extends LimpException{  HurtException(){ System.out.print("hurt "); }  } |

## Debug abstraction

### debugging

|  |
| --- |
| can't do multiple inheritance |
| abstract class Life{  abstract int getSections();  }  abstract class Insect{  abstract int getLegs();  }  class Beetle extends Insect, Life{  int getLegs(){  System.out.println("number of legs");  return 10;  }  public int getSections(){  System.out.println("number of sections");  return 10;  }  } |

The first concrete class must implements all abstract methods defined in interface.

An interface can extend one or more interfaces, or be extended by another interface.

An interface can't extend any class.

An interface can't be instantiated or implement another interface or abstract class.

### abstract methods in abstract class

|  |
| --- |
| // not compiling: abstract methods can't defined in non-abstract class  class Inset {  abstract int getLegs();  } |

|  |
| --- |
| // abstract methods should be added "abstract"  abstract class Insect {  int getLegs();  } |

|  |
| --- |
| not compiling: No method body is allowed in abstract methods |
| abstract class Insect{  abstract int getLegs(){  System.out.println("number of legs");  return 10;  }  abstract void getSection(){}  } |

### concrete class extends abstract

|  |
| --- |
| not compiling: You can't instantiate abstract class |
| public class AbstractClass {  public static void main(String[] args){  Beetle b = new Beetle();  b.getLegs();  b.getSections();  }  }  abstract class Beetle {  int getLegs(){  System.out.println("number of legs");  return 10;  }  } |

|  |
| --- |
| not compiling: All abstract methods must be implemented in concrete class. The method getSections() should be concreted in class. |
| abstract class Insect {  abstract int getLegs();  abstract int getSections();  }  class Beetle extends Insect{  int getLegs(){  System.out.println("number of legs");  return 10;  }  } |
| The blow is ok because none abstract methods are defined. |
| class AbstractTest{  public static void main(String[] args){  Beetle c = new Beetle();  c.getLegs();  }  }  abstract class Insect{}  class Beetle extends Insect{  int getLegs(){  System.out.println("Number of legs is ");  return 1;  }  } |

|  |
| --- |
| not compiling: All abstract methods can't be marked as "final" or "static" |
| abstract class Insect{  abstract final int getLegs();  abstract static int getLegs();  }  class Beetle extends Insect{  int getLegs(){  System.out.println("Number of legs is ");  return 1;  }  } |

### keep being compatible on signatures

return type, method name, and arguments should be consistent

|  |
| --- |
| not compiling: |
| abstract class Insect{  abstract int getLegs();  }  class Beetle extends Insect{  int getLegs(int a){  System.out.println("number of legs");  return 10;  }  } |

|  |
| --- |
| not compiling: return type int is not compatible with void |
| abstract class Insect{  abstract void getLegs();  }  class Beetle extends Insect{  int getLegs(int a){  System.out.println("number of legs");  return 10;  }  } |

### Access identifier

Abstract class may not be marked as private, protected or final.

Default access of abstract method is public. Abstract method can't use private/final.

Can/t reduce access ability in concrete class.

|  |
| --- |
| not compiling: can't use private |
| abstract class Insect{  abstract private int getLegs();  }  class Beetle extends Insect{  int getLegs(){  System.out.println("Number of legs is ");  return 1;  }  } |

Access identifier of methods in concrete class has no such limits

|  |
| --- |
| compiling is ok. |
| abstract class Insect{  **abstract int getLegs();**  }  class Beetle extends Insect{  **protected int getLegs(){**  System.out.println("Number of legs is ");  return 1;  }  } |
| abstract class testAccess{  **private void fly();**  }  public class AbstractAccess extends testAccess{  **protected void fly(){**  System.out.println("abstract is flying");  }  public static void main(String[] args){  AbstractAccess d = new AbstractAccess();  d.fly();  }    } |

All interface and methods are public by default.

|  |
| --- |
| don't mark private identifier |
| private interface Test{ } |
| interface Test{  private void getName();  } |
| interface Test{  private String age = 10;  } |
| interface Test{  private default void getAge(){  return 10;  }  } |

**The concrete method must be marked as public.** Class methods is protected in default which is not allowed.

|  |
| --- |
| "public void print(){" |
| interface MyInterface{  default void print();  }  public class InterfaceDefine implements MyInterface{  void print(){  System.out.println("concrete1");  }  public static void main(String[] args){  InterfaceDefine c1 = new InterfaceDefine();  c1.print();  }  } |

### final identifier

|  |
| --- |
| Don't mark final identifier at declaration of interface or interface methods except variables. |
| final interface Test{ } |
| interface Test{  final void getName();  } |
| interface Test{  final static void getName(){;}  } |
| interface Test{  final default void getAge(){  return 10;  }  } |

|  |
| --- |
|  |
|  |

### method body

method body is not allowed in abstract methods except default or static methods. for default or static methods, they must be explicitly declared.

|  |
| --- |
| can't include method body |
| interface Test{  void getName(){ return; }  } |
| interface Test{  void getName(){}  } |
|  |
|  |

|  |
| --- |
| missing method body |
| interface Test{  default void getName();  } |
| interface Test{  static void getName();  } |

|  |
| --- |
|  |
|  |

### inheritance

concrete class must implement all abstract methods defined in interface. The rule excludes default or static methods.

Any methods in concrete class that concrete abstract methods or override default/static methods must be marked as "public".

|  |
| --- |
| abstract method test() is not implemented by concrete class. |
| public class InterfaceTest implements MyInterface{  public void print(){  System.out.println("concrete1");  }  public static void main(String[] args){  InterfaceTest c = new InterfaceTest();  c.print();  }  }  interface MyInterface{  void print();  **void test();**  **static void test2(){}**  } |

|  |
| --- |
| Exception: no interface expected here. should n't extends but implements |
| public class InterfaceTest extends MyInterface{  public void print(){  System.out.println("concrete1");  }  public static void main(String[] args){  InterfaceTest c = new InterfaceTest();  c.print();  }  }  interface MyInterface{  void print();  } |

|  |
| --- |
| interface can't extends class. |
| interface MyInterface extends Test{  void print();  }  abstract class Test{  abstract void test();  } |

|  |
| --- |
| abstract or concrete method in abstract class can't override or hide default method |
| abstract class Test implements Animal{  abstract String getName();  }  interface Animal{  default String getName(){  return "animal";  }  } |
| abstract class Test implements Animal{  String getName(){  return "test";  }  }  interface Animal{  default String getName(){  return "animal";  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### invoke

|  |
| --- |
|  |
|  |

|  |
| --- |
| can't find symbol.  FrogB.getF(32) |
| public class InterfaceStatic implements FrogB{  public static void main(String[] args) {  new InterfaceStatic().go();  }  void go(){  System.out.println(getF(32));  }  }  interface FrogB{  static int getF(int temp){  return temp\*9/5+32;  }  } |
| public class InterfaceStatic implements Frog{  public static void main(String[] args) {  new InterfaceStatic().go();  }  void go(){  InterfaceStatic d = new InterfaceStatic();  System.out.println(d.getF(32));  }  }  interface Frog{  static int getF(int temp){  return temp\*9/5+32;  }  } |

|  |
| --- |
| can't reference instance method in static context |
| public class InterfaceDefault implements Frog {  public static void main(String[] args){  System.out.println(hop());  }  }  interface Frog{  default String hop(){  return "hop";  }  } |

|  |
| --- |
| can't find symbol |
| public class InterfaceDefaultInherit implements Frog {  public static void main(String[] args){  new InterfaceDefaultInherit().go();  }  void go(){  System.out.println(super.hop());  }  public String hop(){  return "scrawl";  }  }  interface Frog{  default String hop(){  return "hop";  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## lambda

### wrong declaration

wrong arguments list or method body

|  |
| --- |
| if argument types are provided, then it should be provided for all arguments or none of them |
| (a, int b) -> a/b; |

|  |
| --- |
| if there are no parameters, then empty parentheses () must be provided |
| -> 7; |

|  |
| --- |
| return statement and curly brackets{} can't be omitted if multiple statements exist in method body |
| (x) -> {  int y =4;  x+y; //add return  } |
| (x) -> int y =4; return x+y; //add {} |
| (x) -> return x+y; // remove return or add {} |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### wrong type

|  |
| --- |
| type defined in interface Predicate is Integer. |
| class Test{  public static void main(String[] args) {  System.out.println(test((int i) -˃ {return i == 5;}));  }  private static boolean test(**Predicate˂Integer˃ p**) {  return p.test(5);  }  } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

# System Access

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

# Java project

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## class declaration

### main method

class name must be identical with file name.

main() must be defined in the class if the class is expected to be executed by JVM.

main() method must be public, static, void.

|  |
| --- |
| The file name should be "Test.java" |
| public class Test {  public static void main(String[] args){  System.out.println(args);  }  } |

|  |
| --- |
| the approaches below are equal |
| public class MethodMain {  **public static void main(String[] args){**  System.out.println(args);  }  } |
| public class MethodMain {  public static void main(**String args[]**){  System.out.println(args);  }  } |
| public class MethodMain {  public static void main(**String… args**){  System.out.println(args);  }  } |
|  |

|  |
| --- |
| The below define is ok. class name is "\_", argument list is "\_$\_A\_V"  javac \_.java  java \_ |
| **public class \_{**  public static void main(**String[] \_$\_A\_V**){  System.out.println(\_$\_A\_V);  }  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |