* A case constant must evaluate to the same type that the switch expression can use, with one additional

final int a = 1;

final int b;

b = 2;

int x = 0;

switch (x) {

case a: // ok

case b: // compiler error

* int temp = 90;

switch(temp) {

case 80 : System.out.println("80");

case 80 : System.out.println("80"); // won't compile!

case 90 : System.out.println("90");

default : System.out.println("default");

}

* String s = "Monday";

switch(s) {

case "Monday": // matches!

But the following would NOT match:

String s = "MONDAY";

switch(s) {

case "Monday": // Strings are case-sensitive, DOES NOT match

* class SwitchString {

public static void main(String [] args) {

String s = "green";

switch(s) {

case "red": System.out.print("red ");

case "green": System.out.print("green ");

case "blue": System.out.print("blue ");

default: System.out.println("done");

}

}

}

In this example case "green": matched, so the JVM executed that code block and all subsequent code blocks to produce the output:

green blue done

* int x = 1;

while (x) { } // Won't compile; x is not a boolean

while (x = 5) { } // Won't compile; resolves to 5

// (as the result of assignment)

while (x == 5) { } // Legal, equality test

while (true) { } // Legal

* int [][] twoDee = {{1,2,3}, {4,5,6}, {7,8,9}};

for(int[] n : twoDee) ; // loop thru the array of arrays

for(int n2 : twoDee[2]) ; // loop thru the 3rd sub-array

* boolean isTrue = true;

outer:

for(int i=0; i<5; i++) {

while (isTrue) {

System.out.println("Hello");

break outer;

} // end of inner while loop

System.out.println("Outer loop."); // Won't print

} // end of outer for loop

System.out.println("Good-Bye");

OP:

Hello

Good-Bye

The case constant must be a literal or final variable, or a constant expression, including an enum or a String. You cannot have a case that includes a non-final variable or a range of values.

If the condition in a switch statement matches a case constant, execution will run through all code in the switch following the matching case statement until a break statement or the end of the switch statement is encountered. In other words, the matching case is just the entry point into the case block, but unless there's a break statement, the matching case is not the only case code that runs.

The default block can be located anywhere in the switch block, so if no preceding case matches, the default block will be entered, and if the default does not contain a break, then code will continue to execute (fall-through) to the end of the switch or until the break statement is encountered.

Checked exceptions are subject to the handle or declare rule; any method that might throw a checked exception (including methods that invoke methods that can throw a checked exception) must either declare the exception using throws, or handle the exception with an appropriate try/catch.

Subtypes of Error or RuntimeException are unchecked, so the compiler doesn't enforce the handle or declare rule. You're free to handle them or to declare them, but the compiler doesn't care one way or the other.

A case constant must evaluate to the same type that the switch expression can use, with one additional

* final int a = 1;

final int b;

b = 2;

int x = 0;

switch (x) {

case a: // ok

case b: // compiler error

* int temp = 90;

switch(temp) {

case 80 : System.out.println("80");

case 80 : System.out.println("80"); // won't compile!

case 90 : System.out.println("90");

default : System.out.println("default");

}

* String s = "Monday";

switch(s) {

case "Monday": // matches!

But the following would NOT match:

String s = "MONDAY";

switch(s) {

case "Monday": // Strings are case-sensitive, DOES NOT match

* class SwitchString {

public static void main(String [] args) {

String s = "green";

switch(s) {

case "red": System.out.print("red ");

case "green": System.out.print("green ");

case "blue": System.out.print("blue ");

default: System.out.println("done");

}

}

}

In this example case "green": matched, so the JVM executed that code block and

all subsequent code blocks to produce the output:

green blue done

* int x = 1;

while (x) { } // Won't compile; x is not a boolean

while (x = 5) { } // Won't compile; resolves to 5

// (as the result of assignment)

while (x == 5) { } // Legal, equality test

while (true) { } // Legal

* An object is eligible for GC when there is no reference to an object in currently live thread.
* Try{

}

Finally{

}

// compile time error if any statement here when finally block **throw exception or return anything**.

* When calling method check method throws any Exception. If it throws any Exception we must handle otherwise compile time error.
* If while loop contains true or false directly then if any statement after that then it won’t reachable.

While(false)//compilation error here

{

}

* While(true) //compilation error here

{

}

* For(;false;){}//compilation error here
* For(;true;){}// compilation error here
* Check super or this is in static method or not.
* new Object[3] {null, new Object(), new Object()};

// not legal; size must not be specified

* int[][] books = new int[3][];

int[] numbers = new int[6];

int aNumber = 7;

books[0] = aNumber; // NO, expecting an int array not an int

books[0] = numbers; // OK, numbers is an int array

* Car[] cars;

Honda[] cuteCars = new Honda[5];

cars = cuteCars; // OK because Honda is a type of Car

Beer[] beers = new Beer [99];

cars = beers; // NOT OK, Beer is not a type of Car

* int[] weightList = new int[5];

byte b = 4;

char c = 'c';

short s = 7;

weightList[0] = b; // OK, byte is smaller than int

weightList[1] = c; // OK, char is smaller than int

weightList[2] = s; // OK, short is smaller than int

* int[] splats;

int[] dats = new int[4];

char[] letters = new char[5];

splats = dats; // OK, dats refers to an int array

splats = letters; // NOT OK, letters refers to a char array

* You'll get a NullPointerException if you try to use an array element in an object array, if that element does not refer to a real object.
* We can’t declare any other modifiers when declaring a variable inside an interface, but we can skip any of above three modifiers or all of them as they will be added implicitly.
* We can’t create any method with any modifiers except public and abstract, when declaring a method inside an interface, but we can skip any of above two modifiers or all of them as they will be added implicitly.
* If classes are in two different package and it has has-a relation check whether the class is public or not. Only public classes can be accessed outside of the package with has –a relationship.
* You can’t assign a negative value to a primitive char data type without casting.
* However making the main method final using the keyword “final” doesn’t cause any problem.
* Instance variable and static variable should not be same.

Method parameter and local variable and block(if, loop) variable should not be same.

**AUTOMATIC TYPE PROMOTION:**

When java evaluating an expression using variables of one type or more types, there is a possibility that intermediate values or final result exceeds the range of the type, we are used in the expression. For example;

byte b = 10;

byte c = 18;

byte d = c\*b; // the output will be 180, which will exceed the range of byte.

To avoid these types of problems, java provides automatic type promotions. The output of the expressions which evaluate byte, short, int and char, will be promoted to type int. so above give example the output of the c\*b result an int.

If expression contains long then the output will be promoted to long.

Ex: long x = 10;

byte b = 5;

long result = b\*x; // you can also use long, float or double as int type can automatically convert to these types.

* New StringBuilder(“muni”).capacity();//16+4=22
* We should not declare local variable and block variable same.
* While defining ArrayList with generic, generic type should be match both sides.