* Only constants and final variable allowed in switch case. Boolean, long, float, double are not allowed in switch case.
* Method parameters public void m1(int a, int b) and method arguments m1(2,3).
* An ArrayList preserves the order of insertion, Iterator and ListIterator and foreach loop will return the elements in the same order.
* We can’t use top lever class or interface with static keyword

Ex: public static class Person{}

public static interface Person{}

* The classes and interfaces can be defined in any order of occurrence in a Java source code file.
* Per Java naming conventions, package names should all be in lowercase.
* For packaged classes and interfaces, the package statement is the first statement in a Java source file (a .java file). The exception is that comments can appear before or after a package statement.
* The nonpublic class Book cannot be accessed outside of the package.
* Local variables and method parameters can’t be defined using access modifiers except final.
* A derived class can inherit and access protected members of its base class, regardless of the package in which it’s defined. A derived class in a separate package can’t access protected members of its base class using reference variables.
* Non Access modifiers
* abstract
* static
* final
* synchronized
* native
* strictfp
* transient
* volatile
* You can’t mark classes, interfaces, or variables with synchronized modifier.
* A final variable can’t be reassigned a value. It can be assigned a value only once.
* If a reference variable is defined as a final variable, you can’t reassign another object to it, but you can call methods on this variable:
  1. class Person {
  2. final StringBuilder name = new StringBuilder("Sh");
  3. Person() {
  4. name.append("reya");🡪Can call methods on a final variable
  5. name = new StringBuilder();🡪Won’t compile. You can’t reassign another object to a final variable.

}

}

* You can’t use the import statement to access multiple classes or interfaces with the same names from different packages.

Ex: util.Date & sql.Date (use only one import at a time)

* You can’t import classes from a subpackage by using the wildcard character, an asterisk (\*), in the import statement.
* Classes and interfaces defined using the public access modifier are accessible to related and unrelated classes outside the package in which they redefined.
* The members of a class defined using the protected access modifier are accessible to classes and interfaces defined in the same package and to all derived classes, even if they’re defined in separate packages.
* The members of a class defined without using an explicit access modifier are defined with package accessibility (also called default accessibility).The members with package access are accessible only to classes and interfaces defined in the same package.
* A class defined using default access can’t be accessed outside its package.
* The members of a class defined using a private access modifier are accessible only to the class in which they are defined. It doesn’t matter whether the class or interface in question is from another package or has extended the class. Private members are not accessible outside the class in which they’re defined.
* The private access modifier is the most restrictive access modifier.
* An interface is an abstract entity by default. The Java compiler automatically adds the keyword abstract to the definition of an interface (which means that adding the keyword abstract to the definition of an interface is redundant).
* A variable can’t be defined as an abstract variable.
* A method can’t be defined both as abstract and static.
* A static method or variable can’t access non-static variables or methods of a class. But the reverse is true: non-static variables and methods can access static variables and methods.
* The comments can appear anywhere in a class. They can appear before and after package and import statements. They can appear before or after a class, method, or variable declaration.
* You can’t start or end a literal value with an underscore.

int intLiteral = \_100;

int intLiteral2 = 100\_999\_;

long longLiteral = 100\_L;

* You can’t place an underscore right after the prefixes 0b, 0B, 0x, and 0X, which are used to define binary and hexadecimal literal values.
* You can place an underscore right after the prefix 0, which is used to define an octal literal value.
* You can’t place an underscore prior to an L suffix (the L suffix is used to mark a literal value as long).
* You can’t use an underscore in positions where a string of digits is expected.

The following line of code will compile successfully but will fail at runtime:

int i = Integer.parseInt("45\_98");

* You can’t place an underscore adjacent to a decimal point.

float floatLiteral = 100.\_48F;

double doubleLiteral = 100\_.87;

* char values are unsigned integer values, so if you try to assign a negative number, the code will not compile.

char c3 = -122;

* But you can forcefully assign a negative number to a char by casting it to char, as follows:

char c3 = (char)-122;

* You can’t define a variable or method with the same name as Java keywords or reserved words. As these names suggest, they’re reserved for specific purposes.
* Remember that you can’t apply these operators to incomparable types. In the following code snippet, the code that compares an int variable to a boolean variable will fail to compile:

int a = 10;

boolean b1 = false;

System.out.println(a == b1);

Here’s the compilation error:

incomparable types: int and boolean

System.out.println(a == b1);

EXAM TIP

The result of the relational operation is always a boolean value. You can’t assign the result of a relational operation to a variable of type char, int, byte, short, long, float, or double.

**Valid identifiers:**

* A valid identifier starts with a letter (a–z, upper- or lowercase), a currency sign, or an underscore. There is no limit to its length.
* A valid identifier can contain digits, but not in the starting place.
* A valid identifier can use the underscore and currency sign at any position of the identifier.
* A valid identifier can’t have the same spelling as a Java keyword, such as switch.
* A valid identifier can’t use any special characters, including !, @, #, %, ^, &, \*, (,), ', :, ;, [, /, \, or }
* You can’t define a static variable and an instance variable with the same name in a class.

class MyPhone {

static boolean **softKeyboard** = true;

boolean **softKeyboard** = true;

* You can’t define a static method and an instance method with the same name in a class.
* Local variables and loop variables and method parameters can’t be defined with the same name.

void myMethod(int **weight**) {

int **weight** = 10;

}

**class** Phone {

String phoneNumber = "123456789";

**void** setNumber() {

String phoneNumber;

phoneNumber = "987654321";

}

}s

**class** TestPhone {

**public** **static** **void** main(String[] args) {

Phone p1 = **new** Phone();

p1.setNumber();

System.***out***.println(p1.phoneNumber);

}

}

Output: 123456789

* An object’s life cycle starts when it’s created and lasts until it goes out of its scope or is no longer referenced by a variable.

EXAM TIP

An object is marked as eligible to be garbage collected when it can no longer be accessed, which can happen when the object goes out of scope. It can also happen when an object’s reference variable is assigned an explicit null value or is reinitialized.

public void myMethod()

{

int result = 88;

if (result > 78) {

Exam myExam1 = new Exam();

myExam1.setName("Android");

}

}

myExam1 scope starts from the line where it is declared until the end of the if block. Then its’ eligible for GC.

* double newWeight = p.setWeight(20.0);

Code won’t compile because the method setWeight doesn’t return a value. p.setWeight(20.0); is fine.

RULES TO REMEMBER WHEN DEFINING A RETURN STATEMENT

* For a method that returns a value, the return statement must be followed immediately by a value.
* For a method that doesn’t return a value (return type is void), the return statement must *not* be followed by a return value.
* If the compiler determines that a return statement isn’t the last statement to *execute* in a method, the method will fail to compile.
* The returned value from a method may or may not be assigned to a variable. If the value is assigned to a variable, the variable type must be compatible with the type of the return value.
* A return statement must be the last statement in a method. Statements placed after the return statements aren’t reachable and fail to compile.

RULES TO REMEMBER FOR DEFINING OVERLOADED METHODS

Here are a few rules for defining overloaded methods:

* Overloaded methods must have different method parameters from one another.
* Overloaded methods may or may not define a different return type.

Methods can’t be defined as overloaded methods if they differ only in their return types: (compile time error)

double calcAverage(int marks1, int marks2) {

return (marks1 + marks2)/2.0;

}

int calcAverage(int marks1, int marks2) {

return (marks1 + marks2)/2.0;

}

* Overloaded methods may or may not define different access modifiers.

Methods can’t be defined as overloaded methods if they only differ in their access

modifiers:

public double calcAverage(int marks1, int marks2) {

return (marks1 + marks2)/2.0;

}

private double calcAverage(int marks1, int marks2) {

return (marks1 + marks2)/2.0;

}

* Overloaded methods can’t be defined by only changing their return type or access modifiers.
* If the class is public then only I can import and I can use the class in different package

EXAM TIP

You can define a constructor using all four access modifiers: public, protected, default, private.

* What happens if you define a return type for a constructor? Java will treat it as another method, not a constructor, which also implies that it won’t be called implicitly when you create an object of its class:

class Employee {

**void** Employee() {

System.out.println("Constructor");

}

}

INITIALIZER BLOCKS VERSUS CONSTRUCTORS

An *initializer block* is defined within a class, not as a part of a method. It executes for every object that’s created for a class.

class Employee {

**{**

**System.out.println("Employee:initializer");**

**}**

}

* If you define both an initializer and a constructor for a class, both of these will execute. The initializer block will execute prior to the constructor:

class Employee {

Employee() {

System.out.println("Employee:constructor");

}

{

System.out.println("Employee:initializer");

}

}

The purpose of initializer block is to initialize anonymous classes. An *anonymous class* is a type of inner class. In the absence of a name, anonymous classes can’t define a constructor and rely on an initializer block to initialize their variables.

* When you invoke an overloaded constructor using the keyword this, it must be the first statement in your constructor.

EXAM TIP

The terms *encapsulation* and *information hiding* are used interchangeably. By exposing object functionality only through methods, you can prevent your private variables from being assigned any values that don’t fit your requirements. One of the best ways to create a well-encapsulated class is to define its instance variables as private variables and allow access to these variables using public methods.

EXAM TIP

When you pass a primitive variable to a method, its value remains the same after the execution of the method. The value doesn’t change, regardless of whether the method reassigns the primitive to another variable or modifies it.

When you pass an object to a method, the method can modify the object’s state by executing its methods. In this case, the modified state of the object is reflected in the calling method.

* Read 155-157 of pass by value and pass by reference.

***Review notes***

* Variables can have multiple scopes: class, instance, local, and method parameters.
* The scope of local variables is less than the scope of a method if they’re declared in a sub-block (within braces, {}) in a method. This sub-block can be an if statement, a switch construct, a loop, or a try-catch block.
* Local variables can’t be accessed outside the method in which they’re defined.
* Instance variables are defined and accessible within an object. They’re accessible to all the instance methods of a class.
* Class variables are shared by all of the objects of a class—they can be accessed even if there are no objects of the class.
* Local and instance variables can be defined using the same name. In a method, if a local variable exists with the same name as an instance variable, the local variable takes precedence.
* An initializer block can create local variables. It can access and assign values to instance and static variables. It can call methods and define loops, conditional statements, and try-catch-finally blocks.
* class ContString {

public static void main(String... args) {

String summer = new String("Summer");🡪1

String summer2 = "Summer";🡪2

System.out.println("Summer");🡪3

System.out.println("autumn");🡪4

System.out.println("autumn" == "summer");🡪5

String autumn = new String("Summer");🡪6

}

}

I’ll walk through the code with you step by step to calculate the total number of String objects created:

* The code at 1 creates a new String object with the value "Summer". This object is not placed in the String constant pool.
* The code at 2 creates a new String object with the value "Summer" and places it in the String constant pool.
* The code at 3 doesn’t need to create any new String object. It reuses the String object with the value "Summer" that already existed in the String constant pool.
* The code at 4 creates a new String object with the value "autumn" and places it in the String constant pool.
* The code at 5 reuses the String value "autumn" from the String constant pool. It creates a String object with the value "summer" in the String constant pool.
* The code at 6 creates a new String object with the value "Summer".
* The previous code creates a total of five String objects.

EXAM TIP

Strings are immutable. Once initialized, a String value can’t be modified. All the String methods that return a modified String value return a new String object with the modified value. The original String value always remains the same.

* String day = "SunDday";

day.replace('D', 'Z').substring(3);

System.out.println(day);

**String is immutable no change in the value variable day. Prints SunDday.**

String day = "SunDday";

day = day.replace('D', 'Z').substring(3);

System.out.println(day);

**Calls methods replace and substring on day, and assigns the result back to variable day.**

* When you use += to concatenate String values, ensure that the variable you’re using has been initialized (and doesn’t contain null). Look at the following code:

String lang = "Java";

lang += " is everywhere!";

System.out.println(lang);🡪**“Java is everywhere”**

String initializedToNull = null;

initializedToNull += "Java";

System.out.println(initializedToNull);🡪 **nullJava**

* In string’s indexOf() and replace() methods we can use char’s

indexOf(‘c’) | replace(‘old char’,’new char’)

***Creating StringBuilder objects***

* You must use class StringBuilder when you’re dealing with larger strings or modifying the contents of a string often.

StringBuilder sb1 = new StringBuilder();

When you create a StringBuilder object using its default constructor, the following code executes behind the scenes to initialize the array value defined in the class StringBuilder itself:

StringBuilder() {

value = new char[16]; **Creates an array of length 16**

}

StringBuilder sb2 = new StringBuilder(sb1);

**Constructor that accepts a StringBuilder object**

StringBuilder sb3 = new StringBuilder(50);

**int value specifying initial capacity.**

StringBuilder sb4 = new StringBuilder("Shreya Gupta"); **Constructor that accepts a String**

When you create a StringBuilder object by passing it a String, the following code executes behind the scenes to initialize the array value:

public StringBuilder(String str) {

value = new char[str.length() + 16];

append(str);

}

**APPEND()**

The append method adds the specified value at the end of the existing value of a StringBuilder object. Because you may want to add data from multiple data types to a StringBuilder object

sb1.append(true);

sb1.append(10);

sb1.append('a');

sb1.append(20.99);

sb1.append("Hi");

System.out.println(sb1); **Prints true10a20.99Hi**

You can append a complete char array, StringBuffer, or String or its subset as follows:

StringBuilder sb1 = new StringBuilder();

char[] name = {'J', 'a', 'v', 'a', '7'};

sb1.append(name, 1, 3);

System.out.println(sb1);

Because the method append also accepts a method parameter of type Object, you can pass it any object from the Java API or your own user-defined object:

sb1.append(new Person("Oracle"));

System.out.println(sb1); **Prints JavaPerson@126b249(**class name @unsigned hexadecimal representation of the hash code of the object**)**

EXAM TIP

For classes that haven’t overridden the toString method, the append method appends the output from the default implementation of method toString defined in class Object.

**INSERT()**

The main difference between the append and insert methods is that the insert method enables you to insert the requested data at a particular position, but the append method only allows you to add the requested data at the end of the String-Builder object.

1. StringBuilder sb1 = new StringBuilder("Bon");

sb1.insert(2, 'r');🡪insert(Object obj) same as append().

System.out.println(sb1); **Prints Born**

1. StringBuilder sb1 = new StringBuilder("123");

char[] name = {'J', 'a', 'v', 'a'};

sb1.insert(1, name, 1, 3);

System.out.println(sb1); **Prints 1ava23**

**DELETE() AND DELETECHARAT()**

The method delete removes the characters in a substring of the specified String-

Builder. The method deleteCharAt removes the char at the specified position.

StringBuilder sb1 = new StringBuilder("0123456");

sb1.delete(2, 4);

System.out.println(sb1); **Prints 01456**

StringBuilder sb1 = new StringBuilder("0123456");

sb1.deleteCharAt(2);

System.out.println(sb1); **Prints 013456**

TRIM()

Unlike the class String, the class StringBuilder doesn’t define the method trim. An

attempt to use it with this class will prevent your code from compiling. The only reason

I’m describing a nonexistent method here is to ward off any confusion.

REVERSE()

As the name suggests, the reverse method reverses the sequence of characters.

* The methods of the class StringBuffer are synchronized where necessary, whereas the methods of the class StringBuilder aren’t.

ARRAYS:

* The members of an array are defined in contiguous (continuous) memory locations and hence offer improved access speed.

EXAM TIP

Once allocated, all the array elements store their default values. Elements in an array that store objects default to null. Elements of an array that store primitive data types store 0 for integer types (byte, short, int, long), 0.0 for decimal types (float and double), false for boolean, or /u0000 for char data.

EXAM TIP

Code to access an array index will throw a runtime exception if you pass it an invalid array index value. Code to access an array index will fail to compile if you don’t use a char, byte, short, or int.

EXAM TIP

When you combine an array declaration, allocation, and initialization in a single step, you can’t specify the size of the array.

ARRAY LIST:

When you add an element to the end of the list, the ArrayList first checks whether its instance variable elementData has an empty slot at the end. If there’s an empty slot at its end, it stores the element at the first available empty slot. If no empty slots exist, the method ensureCapacity creates another array with a higher capacity and copies the existing values to this newly created array. It then copies the newly added value at the first available empty slot in the array. When you add an element at a particular position, an ArrayList creates a new

array and inserts all its elements at positions other than the position you specified. If there are any subsequent elements to the right of the position that you specified, it shifts them by one position. Then it adds the new element at the requested position.

* The return type of an overriding method in the derived class can be the same, or a subclass of the return type of the overridden method in the base class. When the overriding method returns a subclass of the return type of the overridden method, it is known as a *covariant return type*. This is applicable for objects only.
* Access modifiers for an overriding method can be the same or less restrictive than the method being overridden, but they can’t be more restrictive.

EXAM TIP

To implement polymorphism with classes, you can define abstract or non-abstract methods in the base class and override them in the derived classes.

* An interface can inherit zero or more interfaces. An interface cannot inherit a class.
* The method signatures of a method defined in an interface and in the class that implements the interface must match; otherwise, the class won’t compile.
* Inheritance enables you to reuse existing code.
* Inheritance saves you from having to modify common code in multiple classes.

EXAM TIP

The method equals defines a method parameter of type Object, and its return type is boolean. Don’t change the name of the method, its return type, or the type of method parameter when you define (*override*) this method in your class to compare two objects.

Ex: public boolean equals(**BankAccount anObject**) {

}-🡪 **Wrong**

***Contract of the equals method***

* The method indexOf can be used to search a String for the occurrence of a char or a String, starting from the first position or a specified position.
* The method delete removes the characters in a substring of the specified StringBuilder. The method deleteCharAt removes the char at the specified position.
* Unlike the class String, the class StringBuilder doesn’t define the method trim.
* remove(Object o) removes the first occurrence of the specified element from the list, if it’s present.
* remove(int) removes the element at the specified position in the list.
* When you override the equals method in your class, make sure that you use the correct method signature for the equals method.
* According to the contract of the method equals, if a null value is passed to it, the method equals should return false.
* If the equals method modifies the value of any of the instance variables of the method parameter passed to it, or of the object on which it is called, it will violate the contract.

class EJavaGuruArray {

public static void main(String args[]) {

int[] arr = new int[5];

byte b = 4; char c = 'c'; long longVar = 10;

arr[0] = b;

arr[1] = c;

arr[3] = longVar;

System.out.println(arr[0] + arr[1] + arr[2] + arr[3]);

}

}

* Java does support implicit widening conversions for variables, the previous code fails to compile.
* arr[1] = ‘a’--- correct
* int[] arr1;

char[] arr3 = {'a', 'b'};

arr1 = arr3; --- wrong

* The switch statement doesn’t accept arguments of type boolean, long, float, or double, or any object besides String.
* The following code won’t compile because the type of history is double, which is a type that isn’t accepted by the switch statement:

double history = 20;

switch (history) {

// ..code

}

EXAM TIP Watch out for questions in the exam that try to pass a primitive decimal type such as float or double to a switch statement. Code that tries to do so will not compile.



CASE VALUES SHOULD BE COMPILE-TIME CONSTANTS

The value of a case label must be a compile-time constant value; that is, the value should be known at the time of code compilation:

int a=10, b=20, c=30;

switch (a) {

case **b+c**: System.out.println(b+c); break;---1

case **10\*7**: System.out.println(10\*7512+10); break;---2

}

Note that b+c in the previous code defined at 1 can’t be determined at the time of compilation and isn’t allowed. But 10\*7 defined at 2 is a valid case label value.You can use variables in an expression if they’re marked final because the value of final variables can’t change once they’re initialized:

final int a = 10;---1

final int b = 20;

final int c = 30;

switch (a) {

case b+c: System.out.println(b+c); break;

}

Because the variables b and c are final variables here, at 1 the value of b+c can be known at compile time. This makes it a compile-time constant value, which can be used in a case label.

You may be surprised to learn that if you don’t assign a value to a final variable with its declaration, it isn’t considered a compile-time constant:

final int a = 10;

final int b = 20;

final int c;---1

c = 30;---2

switch (a) {

case b+c: System.out.println(b+c); break;---3

}

This code defines a final variable c at line 1 but doesn’t initialize it. The final variable c is initialized at line 2. Because the final variable c isn’t initialized with its declaration, at 3 the expression b+c isn’t considered a compile-time constant, so it can’t be used as a case label.

byte myByte = 10;

switch (myByte) {

case **1.2**: System.out.println(1); break; --- wrong don’t use float in switch.

}

String name = "Paul";

switch (name) {

case "Paul": System.out.println(1);

break;

case **null**: System.out.println("null");--- wrong don’t use null in switch.

}

* for (**int j=10, long longVar = 10**; j <= l; ++j) { } --- **Can’t define variables of different types in an initialization block**
* int tableOf = 25;

for (int ctr = 1; ctr <= 5; ++ctr) {

System.out.println(tableOf \* ctr);

}

ctr = 20; **Variable ctr isn’t accessible outside for loop**

EXAM TIP

Use the for-each loop to iterate arrays and collections. Don’t use it to initialize, modify, or filter them.

NOTE

Don’t forget to use a semicolon (;) to end the do-while loop after specifying its condition. Even some experienced programmers overlook this step!

WITH OUT LABEL

String[] programmers = { "Outer", "Inner" };

**for** (String outer : programmers)

{

**for** (String inner : programmers)

{

**if** (inner.equals("Inner"))

**break** ;

System.***out***.print(inner + ":");

}

}

OP: Outer:Outer:

WITH LABEL

String[] programmers = { "Outer", "Inner" };

outer: **for** (String outer : programmers)

{

**for** (String inner : programmers) {

**if** (inner.equals("Inner"))

**break** outer;

System.***out***.print(inner + ":");

}

}

OP: Outer:

* int i = 10;

do

while (i < 15)

i = i + 20;

while (i < 2);

System.out.println(i);

OP: 30

* int i = 10;

do

while (i++ < 15)

i = i + 20;

while (i < 2);

System.out.println(i);

OP: 32 (Loop execute 2 times because of ++ operator)

* int a = 10;

if (a++ > 10) {

System.out.println("true");

}

{

System.out.println("false");

}

OP: **false**

**ABC**

Hence, the value false will print no matter what, regardless of whether the condition in the if construct evaluates to true or false. Because the opening and closing braces for this code snippet are placed right after the if construct.

***INHERITANCE:***

* *Default—*Members with default access can be accessed in a derived class only if base and derived classes reside in the same package.
* protected*—*Members with protected access are accessible to all the derived classes, regardless of the packages in which the base and derived classes are defined.
* public*—*Members with public access are visible to all the other classes.

A derived class doesn’t inherit the following members:

* private members of the base class.
* Base class members with default access, if the base class and derived classes exist in separate packages.
* Constructors of the base class. A derived class can call a base class’s constructors, but it doesn’t inherit them.

EXAM TIP

The method signatures of a method defined in an interface and the classes that implement the interface must match, or the classes won’t compile.

EXAM TIP

You may see multiple questions in the exam that try to assign an object of a base class to a reference variable of a derived class. Note that a derived class can be referred to using a reference variable of its base class. The reverse is not allowed and won’t compile.

Ex: Child c = new Parent(); --- compile time error

* Base b = new Derived();

b is referring to the Derived object.

b

b can access only Base class variable and methods, not Derived class.

If you want access you can type cast it forcefully.

Ex: ((Child)b).parentvariablename/method

* Don’t use this and super keywords in static block.

EXAM TIP

Watch out for code that returns a value from the catch block and modifies it in the finally block. If a catch block returns a primitive data type, the finally block can’t modify the value being returned by it. If a catch block returns an object, the finally block can modify the value being returned by it.

Refer MultipleReturn1 and MultipleReturn2.

* A try block may be followed by multiple catch blocks, and the catch blocks may be followed by a single finally block.
* A try block may be followed by either a catch or a finally block or both. But a finally block alone wouldn’t suffice if code in the try block throws a checked exception. In this case, you need to catch the checked exception or declare it to be thrown by your method. Otherwise your code won’t compile.

EXAM TIP

In the OCA Java SE 7 Programmer I exam, you may have to select which type of reference variable to use to store the object of the thrown checked exception in a handler. To answer such questions correctly, remember that a checked exception is a subclass of the java.lang.Exception class, but not a subclass of java.lang.RuntimeException.

* A runtime exception may not be a part of the method signature, even if a method may throw it.

NOTE

When you try to access an invalid array position, ArrayIndexOutOf-BoundsException is thrown. When you try to access an invalid ArrayList position, IndexOutOfBoundsException is thrown.

* **ArrayList<String> list;list.add();-- NullPointerException.**

EXAM TIP

In the exam, watch out for code that tries to use an uninitialized local variable. Because such variables aren’t initialized with even a null value.

* oldLaptops[3][6] in sop won’t call toString().

System.out.println(Integer.parseInt("-123"));

System.out.println(Integer.parseInt("+123"));

System.out.println(Integer.parseInt("123**\_**45"));

System.out.println(Integer.parseInt("12**ABCD**"));

Above all are **NumberFormatException.**

* Starting in Java 7, you can use underscores (\_) in numeric literal values. But you can’t use them in String values passed to the method parseInt.
* You can convert the hexadecimal literal value "12ABCD" to the decimal number system by specifying the base of the number system as 16.

Ex: System.out.println(Integer.parseInt("123**ABCD**", **16**)); **Prints 1223629**

***ExceptionInInitializerError***

The ExceptionInInitializerError error is typically thrown by the JVM when a static initializer in your code throws any type of RuntimeException.

A static initializer block is defined using the keyword static, followed by curly braces, in a class. This block is defined within a class, but not within a method. It’s usually used to execute code when a class loads for the first time. Runtime exceptions arising from any of the following will throw this error:

* Execution of an anonymous static block
* Initialization of a static variable
* Execution of a static method (called from either of the previous two items)

The static initializer block of the class defined in the following example will throw a

NumberFormatException, and when the JVM tries to load this class, it’ll throw an

ExceptionInInitializerError:

public class DemoExceptionInInitializerError {

static {

int num = Integer.parseInt("sd", 16);

}

}

Following is the error message when JVM tries to load the class DemoExceptionIn-

InitializerError:

java.lang.ExceptionInInitializerError

Caused by: java.lang.NumberFormatException: For input string: "sd"

at

java.lang.NumberFormatException.forInputString(NumberFormatException.jav

a:48)

at java.lang.Integer.parseInt(Integer.java:447)

at

DemoExceptionInInitializerError.<clinit>(DemoExceptionInInitializerError

.java:3)

EXAM TIP

Beware of code that seems to be simple in the OCA Java SE 7 Programmer I exam. The class DemoExceptionInInitializerError (mentioned previously) seems deceptively simple, but it’s a good candidate for an exam question. As you know, this class throws the error ExceptionInInitializer-Error when the JVM tries to load it.

In the following example, initialization of a static variable results in a NullPointer- Exception being thrown. When this class is loaded by the JVM, it throws an Exception-InInitializerError:

public class DemoExceptionInInitializerError1 {

static String name = null;

static int nameLength = name.length();

}

The error message when the JVM tries to load the DemoExceptionInInitializer-

Error1 class is as follows:

java.lang.ExceptionInInitializerError

Caused by: java.lang.NullPointerException

EXAM TIP

ExceptionInInitializerError can be caused by an object of RuntimeException only. It can’t occur as the result of an error or checked exception thrown by the static initialization block.

***NoClassDefFoundError***

What would happen if you failed to set your classpath and, as a result, the JVM was unable to load the class that you wanted to access or execute? Or what happens if you try to run your application before compiling it? In both these conditions, the JVM would throw NoClassDefFoundError.

***OutOfMemoryError***

What happens if you create and use a *lot* of objects in your application—for example, if you load a large chunk of persistent data to be processed by your application. In such a case, the JVM may run out of memory *on the heap*, and the garbage collector may not be able to free more memory for the JVM. In this case, the JVM is unable to create any more objects on the heap. An OutOfMemoryError will be thrown.

You’ll always work with a finite heap size, no matter what platform you work on, so you can’t create and use an unlimited number of objects in your application. To get around this error, you need to either limit the number of resources or objects that your application creates or increase the heap size on the platform you’re working with.

* If a catch block returns a primitive data type, a finally block can’t modify the value being returned by it.
* If a catch block returns an object, a finally block can modify the value being returned by it.
* String s = null;

Sop(s); // null

* class Ink{}

interface Printable {}

class ColorInk extends Ink implements Printable {}

class BlackInk extends Ink{}

class TwistInTaleCasting {

public static void main(String args[]) {

Printable printable = null;

BlackInk blackInk = new BlackInk();

printable = (Printable)blackInk;

}

}

printable = (Printable)blackInk; ClassCastException

printable = blackInk; compiletime error