# Data types

Reference: <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

byte: The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large [arrays](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html), where the memory savings actually matters.

short: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with byte, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters.

int: By default, the int data type is a 32-bit signed two's complement integer, which has a minimum value of -231 and a maximum value of 231-1. In Java SE 8 and later, you can use the int data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of 232-1. Use the Integer class to use int data type as an unsigned integer. See the section The Number Classes for more information. Static methods like *compareUnsigned, divideUnsigned* etc have been added to the [Integer](https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html) class to support the arithmetic operations for unsigned integers.

long: The long data type is a 64-bit two's complement integer. The signed long has a minimum value of -263 and a maximum value of 263-1. In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of 264-1. Use this data type when you need a range of values wider than those provided by int. The [Long](https://docs.oracle.com/javase/8/docs/api/java/lang/Long.html) class also contains methods like compareUnsigned, divideUnsigned etc to support arithmetic operations for unsigned long.

float: The float data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the [Floating-Point Types, Formats, and Values](https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.2.3) section of the Java Language Specification. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the [java.math.BigDecimal](https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html) class instead. [Numbers and Strings](https://docs.oracle.com/javase/tutorial/java/data/index.html) covers BigDecimal and other useful classes provided by the Java platform.

double: The double data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the [Floating-Point Types, Formats, and Values](https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.2.3) section of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.

boolean: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.

char: The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

## Default Values:

|  |  |
| --- | --- |
| **Data Type** | **Default Value (for fields)** |
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0f |
| double | 0.0d |
| char | '\u0000' |
| String (or any object) | null |
| boolean | false |

## Floating-Point Literals

A floating-point literal is of type float if it ends with the letter F or f; otherwise its type is double and it can optionally end with the letter D or d.

The floating point types (float and double) can also be expressed using E or e (for scientific notation), F or f (32-bit float literal) and D or d (64-bit double literal; this is the default and by convention is omitted).

double d1 = 123.4;

// same value as d1, but in scientific notation

double d2 = 1.234e2;

float f1 = 123.4f;

## Using Underscore Characters in Numeric Literals

long creditCardNumber = 1234\_5678\_9012\_3456L;

long socialSecurityNumber = 999\_99\_9999L;

float pi = 3.14\_15F;

long hexBytes = 0xFF\_EC\_DE\_5E;

long hexWords = 0xCAFE\_BABE;

long maxLong = 0x7fff\_ffff\_ffff\_ffffL;

byte nybbles = 0b0010\_0101;

long bytes = 0b11010010\_01101001\_10010100\_10010010;

### Notes:

1. char can be assigned to int, long, float, double. Compiler will take the equivalent number and assigned it to corresponding datatype (int, long, float or double). Eg, for float 10.0 and For double 10.0; 1) **char** c = '\u0002'; **int** i4 = c; out:i4 = 2; 2) **char** c = 'c'; **long** i4 = c; out:i4 = 99; 3) **char** c = 'a'; **char** c1 = '2'; **int** i2 = c \* c1; out: i2 = 4850; 4) **char** c = 'a'; **char** c1 = '2'; **char** i2 = c \* c1; Compile time error: Cannot convert from int to char
2. char cannot be converted (assigned) to byte, short. None of the datatypes can be converted to char.
3. byte, short, int and long are can be assigned from small size to big one. Inversely can't be done.
4. float, double are can be assigned from small size to big one. Inversely can't be done.
5. byte, short, int and long can be converted to float and double. Inversely isn't possible.
6. int i = 2147483647; float f2 = i; out:f2 = 2.14748365E9
7. int i = 2147483647; float f2 = i; double d2 =f2; out:d2 = 2.147483648E9
8. int i4 = 1\_10; int i5 = 11\_15; int i6 = i4+i5; out:i6 = 1225
9. char c = '\u0002'; out:c\*1 = 2
10. char c = '\u0002'; out:c = ?
11. **byte** b = 127; char c = '\u0002'; Long l3 = Long.valueOf("10"); out:b\*l3\*c = 2540
12. **long** l11 = 10; **short** sh2 = 10; **short** l12 = l11 \* sh2; Compile time Error: Cannot convert from long to short
13. **char** c = '\u0002'; **int** i4 = c; out:i4 = 2;

# Exceptions

1. Checked exception: These are exceptional conditions that a well-written application should anticipate and recover from. Eg: java.io.FileNotFoundException
2. Error:These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. Errors are not subject to the Catch or Specify Requirement. Errors are those exceptions indicated by *Error* and its subclasses. Eg: java.io.IOError, OutOfMemoryError
3. Runtime exception:These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API. Runtime exceptions are not subject to the Catch or Specify Requirement. Runtime exceptions are those indicated by *RuntimeException* and its subclasses. Eg: NullPointerException

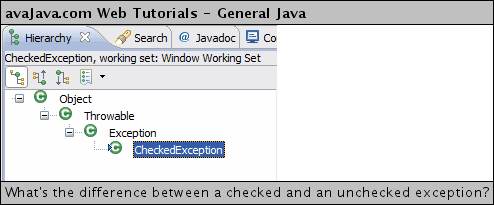
Errors and runtime exceptions are collectively known as *unchecked exceptions*.

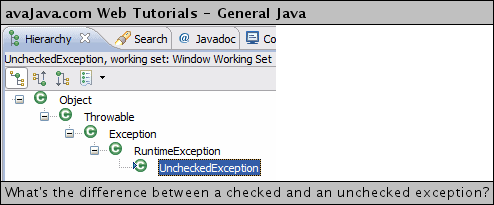
Note: Some programmers consider the Catch or Specify Requirement a serious flaw in the exception mechanism and bypass it by using unchecked exceptions in place of checked exceptions. In general, this is not recommended

In Java, the Exception class is a subclass of Throwable. The RuntimeException class is a subclass of Exception. If an exception is a subclass of Exception but not RuntimeException, it is a checked exception. A checked exception needs to either be caught in a method (in a try/catch block), or the method needs to specify that it can throw the exception on further (via a throws clause of a method declaration). If an exception is a subclass of RuntimeException, it is an unchecked exception. An unchecked exception can be caught, but it does not have to be. Thus, it is 'unchecked'.

An example of a typical checked exception is IOException. An example of a typical unchecked exception is a NullPointerException.

checked exceptions are checked at compile-time while unchecked exceptions are checked at runtime.





There are two types of exceptions: checked exceptions and unchecked exceptions. In this tutorial we will learn both of them with the help of examples. The main **difference between checked and unchecked exception** is that the checked exceptions are checked at compile-time while unchecked exceptions are checked at runtime.

### What are checked exceptions?

Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using [**try-catch block**](http://beginnersbook.com/2013/04/try-catch-in-java/) or it should declare the exception using [**throws keyword**](http://beginnersbook.com/2013/04/difference-between-throw-and-throws-in-java/), otherwise the program will give a compilation error. It is named as ***checked exception*** because these exceptions are ***checked*** at Compile time.

Let’s understand this with this **example**: In this example we are reading the file myfile.txt and displaying its content on the screen. In this program there are three places where an checked exception is thrown as mentioned in the comments below. FileInputStream which is used for specifying the file path and name, throws FileNotFoundException. The read() method which reads the file content throws IOException and the close() method which closes the file input stream also throws IOException.

import java.io.\*;

class Example {

public static void main(String args[])

{

FileInputStream fis = null;

/\*This constructor FileInputStream(File filename)

\* throws FileNotFoundException which is a checked

\* exception\*/

fis = new FileInputStream("B:/myfile.txt");

int k;

/\*Method read() of FileInputStream class also throws

\* a checked exception: IOException\*/

while(( k = fis.read() ) != -1)

{

System.out.print((char)k);

}

/\*The method close() closes the file input stream

\* It throws IOException\*/

fis.close();

}

}

### Output:

Exception in thread "main" java.lang.Error: Unresolved compilation problems:

Unhandled exception type FileNotFoundException

Unhandled exception type IOException

Unhandled exception type IOException

Why this compilation error? As I mentioned in the beginning that checked exceptions gets checked during compile time. Since we didn’t handled/declared the exceptions, our program gave the compilation error.

How to resolve the error? There are two ways to avoid this error. We will see both the ways one by one.

Method 1: Declare the exception using throws keyword.  
As we know that all three occurrences of checked exceptions are inside main() method so one way to avoid the compilation error is: Declare the exception in the method using throws keyword. You may be thinking that our code is throwing FileNotFoundException and IOException both then why we are declaring the IOException alone. Th reason is that IOException is a parent class of FileNotFoundException so it by default covers that. If you want you can declare that too like this public static void main(String args[]) throws IOException, FileNotFoundException.

import java.io.\*;

class Example {

public static void main(String args[]) throws IOException

{

FileInputStream fis = null;

fis = new FileInputStream("B:/myfile.txt");

int k;

while(( k = fis.read() ) != -1)

{

System.out.print((char)k);

}

fis.close();

}

}

Output:  
File content is displayed on the screen.

### **Method 2: Handle them using try-catch blocks.**

The above approach is not good at all. It is not a best [**exception handling**](http://beginnersbook.com/2013/04/java-exception-handling/)practice. You should give meaningful message for each exception type so that it would be easy for someone to understand the error. The code should be like this:

import java.io.\*;

class Example {

public static void main(String args[])

{

FileInputStream fis = null;

try{

fis = new FileInputStream("B:/myfile.txt");

}catch(FileNotFoundException fnfe){

System.out.println("The specified file is not " +

"present at the given path");

}

int k;

try{

while(( k = fis.read() ) != -1)

{

System.out.print((char)k);

}

fis.close();

}catch(IOException ioe){

System.out.println("I/O error occurred: "+ioe);

}

}

}

This code will run fine and will display the file content.

Here are the few other Checked Exceptions –

* SQLException
* IOException
* DataAccessException
* ClassNotFoundException
* InvocationTargetException

### What are Unchecked exceptions?

Unchecked exceptions are not checked at compile time. It means if your program is throwing an unchecked exception and even if you didn’t handle/declare that exception, the program won’t give a compilation error. Most of the times these exception occurs due to the bad data provided by user during the user-program interaction. It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately. All Unchecked exceptions are direct sub classes of **RuntimeException** class.

Let’s understand this with an **example**:

class Example {

public static void main(String args[])

{

int num1=10;

int num2=0;

/\*Since I'm dividing an integer with 0

\* it should throw ArithmeticException\*/

int res=num1/num2;

System.out.println(res);

}

}

If you compile this code, it would compile successfully however when you will run it, it would throw ArithmeticException. That clearly shows that unchecked exceptions are not checked at compile-time, they are being checked at runtime. Lets see another example.

class Example {

public static void main(String args[])

{

int arr[] ={1,2,3,4,5};

/\*My array has only 5 elements but

\* I'm trying to display the value of

\* 8th element. It should throw

\* ArrayIndexOutOfBoundsException\*/

System.out.println(arr[7]);

}

}

This code would also compile successfully sinceArrayIndexOutOfBoundsException is also an unchecked exception.  
**Note**: It **doesn’t mean** that compiler is not checking these exceptions so we shouldn’t handle them. In fact we should handle them more carefully. For e.g. In the above example there should be a exception message to user that they are trying to display a value which doesn’t exist in array so that user would be able to correct the issue.

class Example {

public static void main(String args[])

{

try{

int arr[] ={1,2,3,4,5};

System.out.println(arr[7]);

}catch(ArrayIndexOutOfBoundsException e){

System.out.println("The specified index does not exist " +

"in array. Please correct the error.");

}

}

}

Here are the few most frequently seen unchecked exceptions –

* NullPointerException
* ArrayIndexOutOfBoundsException
* ArithmeticException
* IllegalArgumentException

#### New Syntax

catch (IOException|SQLException ex) {

logger.log(ex);

throw ex;

}

Note**:** If the JVM exits while the try or catch code is being executed, then the finally block may not execute. Likewise, if the thread executing the try or catchcode is interrupted or killed, the finally block may not execute even though the application as a whole continues.

Note: Finally block is executed before the try block returns the value.

Java defines several exception classes inside the standard package **java.lang**.

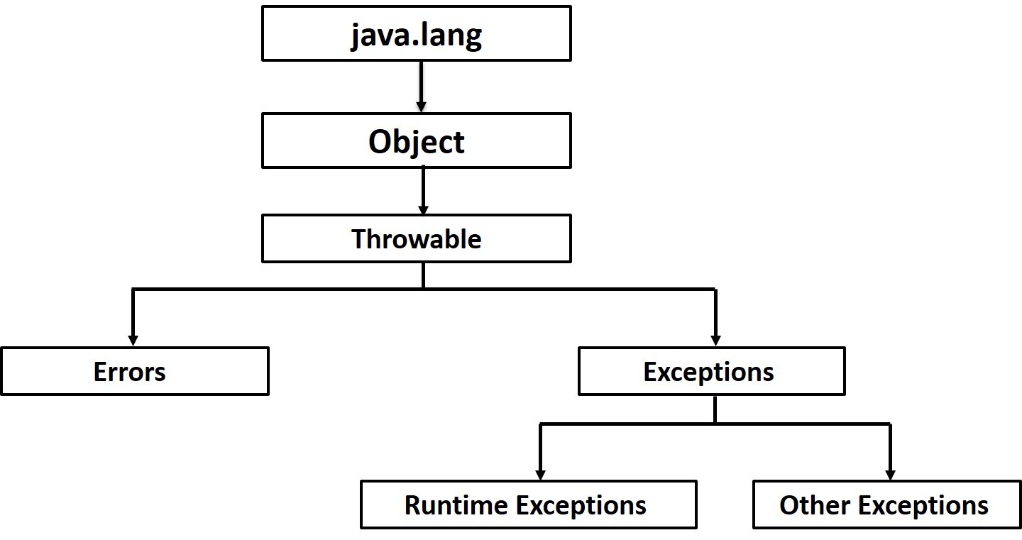
The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.

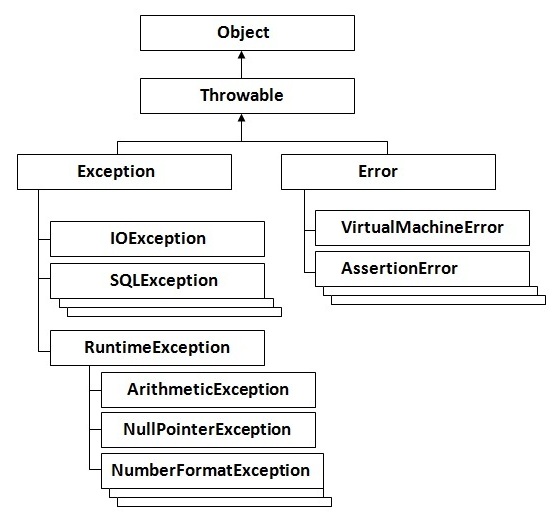
Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked RuntimeException.

|  |  |
| --- | --- |
| Exception | Description |
| ArithmeticException | Arithmetic error, such as divide-by-zero. |
| ArrayIndexOutOfBoundsException | Array index is out-of-bounds. |
| ArrayStoreException | Assignment to an array element of an incompatible type. |
| ClassCastException | Invalid cast. |
| IllegalArgumentException | Illegal argument used to invoke a method. |
| IllegalMonitorStateException | Illegal monitor operation, such as waiting on an unlocked thread. |
| IllegalStateException | Environment or application is in incorrect state. |
| IllegalThreadStateException | Requested operation not compatible with current thread state. |
| IndexOutOfBoundsException | Some type of index is out-of-bounds. |
| NegativeArraySizeException | Array created with a negative size. |
| NullPointerException | Invalid use of a null reference. |
| NumberFormatException | Invalid conversion of a string to a numeric format. |
| SecurityException | Attempt to violate security. |
| StringIndexOutOfBounds | Attempt to index outside the bounds of a string. |
| UnsupportedOperationException | An unsupported operation was encountered. |

Following is the list of Java Checked Exceptions Defined in java.lang.

|  |  |
| --- | --- |
| Exception | Description |
| ClassNotFoundException | Class not found. |
| CloneNotSupportedException | Attempt to clone an object that does not implement the Cloneable interface. |
| IllegalAccessException | Access to a class is denied. |
| InstantiationException | Attempt to create an object of an abstract class or interface. |
| InterruptedException | One thread has been interrupted by another thread. |
| NoSuchFieldException | A requested field does not exist. |
| NoSuchMethodException | A requested method does not exist. |





Here's the bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

## Static Variables / Methods / Blocks

static keyword to create fields and methods that belong to the class, rather than to an instance of the class.

public class Bicycle {

private int cadence;

private int gear;

private int speed;

private int id;

private static int numberOfBicycles = 0;

public Bicycle(int startCadence, int startSpeed, int startGear){

gear = startGear;

cadence = startCadence;

speed = startSpeed;

// **increment number of Bicycles**

// **and assign ID number**

**id = ++numberOfBicycles;**

}

// **new method to return the ID instance variable**

public int getID() {

return id;

}

...

}

A common use for static methods is to access static fields. For example, we could add a static method to the Bicycle class to access the numberOfBicycles static field:

public **static** int getNumberOfBicycles() {

return numberOfBicycles;

}

1. Instance methods can access instance variables and instance methods directly.
2. Instance methods can access class variables and class methods directly.
3. Class methods can access class variables and class methods directly.
4. Class methods ***cannot*** access instance variables or instance methods directly—they must use an object reference. Also, class methods cannot use the this keyword as there is no instance for this to refer to.

#### Constants

The static modifier, in combination with the final modifier, is also used to define constants. The final modifier indicates that the value of this field cannot change.

static final double PI = 3.141592653589793;

Note: If a primitive type or a string is defined as a constant and the value is known at compile time, the compiler replaces the constant name everywhere in the code with its value. This is called a *compile-time constant*. If the value of the constant in the outside world changes (for example, if it is legislated that pi actually should be 3.975), you will need to recompile any classes that use this constant to get the current value.

In java you see "static variables", "static methods", "static classes" and "static blocks". Static variables, static methods and static classes are known to everyone but what is this "static block". Let’s see what, where and how these static blocks are used.

But before going into "static block", lets refresh what other static stuff are. Now "static variables" are class variables i.e., there will be only one copy for each class and not one copy for each object of the class and these variables will be accessed without instantiating the class.

Then what are static methods. Again they are class methods i.e., they can be accessed without creating an instance of the class and like static variables, static methods will be accessed without instantiating the class. Note that static methods cannot access instance variables. They can access only static variables.

Next what are static classes. You cannot declare a top-level class as a static class. Java will throw a compilation error. Only inner classes that are member classes can be declared as static. If we declare member classes as static, we can use it as a top-level class outside the context of top-level class. One catch here is "The static keyword does not do to a class declaration what it does to a variable or a method declaration." - what it means is say for example you have a static variable, then to access that static variable you will use the notation   
*<<Class Name>>.<<Variable Name>>*   
but when you want to use the static inner class, you need to instantiate like   
*<<Top-level class name>>.<<Inner static class name>> newClass = new <<Top-level class name>>.<<Inner static class name>>();*

*Static blocks* are also called *Static initialization blocks* . A static initialization block is a normal block of code enclosed in braces, { }, and preceded by the static keyword. Here is an example:

static {

// whatever code is needed for initialization goes here

}

A class can have any number of static initialization blocks, and they can appear anywhere in the class body. The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code. And dont forget, this code will be executed when JVM loads the class. JVM combines all these blocks into one single static block and then executes. Here are a couple of points I like to mention:

1. If you have executable statements in the static block, JVM will automatically execute these statements when the class is loaded into JVM.
2. If you’re referring some static variables/methods from the static blocks, these statements will be executed after the class is loaded into JVM same as above i.e., now the static variables/methods referred and the static block both will be executed.

Let’s see an example:

public class StaticExample{

static {

System.out.println("This is first static block");

}

public StaticExample(){

System.out.println("This is constructor");

}

public static String staticString = "Static Variable";

static {

System.out.println("This is second static block and "

+ staticString);

}

public static void main(String[] args){

StaticExample statEx = new StaticExample();

StaticExample.staticMethod2();

}

static {

staticMethod();

System.out.println("This is third static block");

}

public static void staticMethod() {

System.out.println("This is static method");

}

public static void staticMethod2() {

System.out.println("This is static method2");

}

}

What will happen when you execute the above code? You will see below output.

This is first static block

This is second static block and Static Variable

This is static method

This is third static block

This is constructor

This is static method2

Now lets the output. First all static blocks are positioned in the code and they are executed when the class is loaded into JVM. Since the static method staticMethod() is called inside third static block, its executed before calling the main method. But the staticMethod2() static method is executed after the class is instantiated because it is being called after the instantiation of the class.

Again if you miss to precede the block with "static" keyword, the block is called "constructor block" and will be executed when the class is instantiated. The constructor block will be copied into each constructor of the class. Say for example you have four parameterized constructors, then four copies of constructor blocks will be placed inside the constructor, one for each. Lets execute the below example and see the output.

public class ConstructorBlockExample{

{

System.out.println("This is first constructor block");

}

public ConstructorBlockExample(){

System.out.println("This is no parameter constructor");

}

public ConstructorBlockExample(String param1){

System.out.println("This is single parameter constructor");

}

public ConstructorBlockExample(String param1, String param2){

System.out.println("This is two parameters constructor");

}

{

System.out.println("This is second constructor block");

}

public static void main(String[] args){

ConstructorBlockExample constrBlockEx =

new ConstructorBlockExample();

ConstructorBlockExample constrBlockEx1 =

new ConstructorBlockExample("param1");

ConstructorBlockExample constrBlockEx2 =

new ConstructorBlockExample("param1", "param2");

}

}

The output is.

This is first constructor block

This is second constructor block

This is no parameter constructor

This is first constructor block

This is second constructor block

This is single parameter constructor

This is first constructor block

This is second constructor block

This is two parameters constructor

The above example is self-explanatory.

Now lets go back to static blocks.

There is an alternative to static blocks —you can write a private static method.

class PrivateStaticMethodExample {

public static varType myVar = initializeClassVariable();

private static varType initializeClassVariable() {

//initialization code goes here

}

}

The advantage of private static methods is that they can be reused later if you need to reinitialize the class variable. So, you kind of get more flexibility with a private static method in comparison to the corresponding static initialization block. This should not mislead that a 'public' static method can't do the same. But, we are talking about a way of initializing a class variable and there is hardly any reason to make such a method 'public'.

#### So what are the advantages of static blocks?

1. If you’re loading drivers and other items into the namespace. For ex, Class class has a static block where it registers the natives.
2. If you need to do computation in order to initialize your static variables, you can declare a static block which gets executed exactly once, when the class is first loaded.
3. Security related issues or logging related tasks

#### Of course there are limitations for static blocks

1. There is a limitation of JVM that a static initializer block should not exceed 64K.
2. You cannot throw Checked Exceptions.
3. You cannot use this keyword since there is no instance.
4. You shouldn’t try to access super since there is no such a thing for static blocks.
5. You should not return anything from this block.
6. Static blocks make testing a nightmare.

Finally how to handle Exceptions in static blocks?   
In methods, an exception can be handled by either passing through the Exception or handling it. But in a static block code, you cannot handle exceptions this way.

Generally a clean way to handle it is using a try-catch block but here since we don’t have this option let’s look at the available three options.

First: After logging the exception throw a RuntimeException which will end the current thread (unless caught by code instantiating / calling a static method on the class for the first time).

Second is calling System.exit(1) but this is not desirable in a managed environment like a servlet. This option is only for java applications and only if the static initializer block performs some critical (without which the program cannot be run successfully) function like loading the database driver.

Third and final option is to set a flag indicating failure. Later the constructors can check the flag and throw exceptions or retry in rare cases.

Finally, if the operation is not important to the functioning of the program then maybe a simple log entry is all that is required.

### Anonymous classes & Lambda expression – Java 8:

##### Example:

public class HelloWorldAnonymousClasses {

interface HelloWorld {

public void greet();

public void greetSomeone(String someone);

}

public void sayHello() {

class EnglishGreeting implements HelloWorld {

String name = "world";

public void greet() {

greetSomeone("world");

}

public void greetSomeone(String someone) {

name = someone;

System.out.println("Hello " + name);

}

}

HelloWorld englishGreeting = new EnglishGreeting();

HelloWorld frenchGreeting = new HelloWorld() {

String name = "tout le monde";

public void greet() {

greetSomeone("tout le monde");

}

public void greetSomeone(String someone) {

name = someone;

System.out.println("Salut " + name);

}

};

HelloWorld spanishGreeting = new HelloWorld() {

String name = "mundo";

public void greet() {

greetSomeone("mundo");

}

public void greetSomeone(String someone) {

name = someone;

System.out.println("Hola, " + name);

}

};

englishGreeting.greet();

frenchGreeting.greetSomeone("Fred");

spanishGreeting.greet();

}

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

HelloWorldAnonymousClasses myApp =

new HelloWorldAnonymousClasses();

myApp.sayHello();

}

}

------------------------------------------------------------------------------------------------------------

btn.setOnAction(**new EventHandler<ActionEvent>() {**

**@Override**

**public void handle(ActionEvent event) {**

**System.out.println("Hello World!");**

**}**

**}**);

Because the EventHandler<ActionEvent> interface contains only one method, you can use a lambda expression instead of an anonymous class expression.

Anonymous classes are ideal for implementing an interface that contains two or more methods.

final TextField sum = **new TextField() {**

**@Override**

**public void replaceText(int start, int end, String text) {**

**if (!text.matches("[a-z, A-Z]")) {**

**super.replaceText(start, end, text);**

**}**

**label.setText("Enter a numeric value");**

**}**

**@Override**

**public void replaceSelection(String text) {**

**if (!text.matches("[a-z, A-Z]")) {**

**super.replaceSelection(text);**

**}**

**}**

**};**

The highlighted code creates a text field that only accepts numeric values. It redefines the default implementation of the TextField class with an anonymous class by overriding the replaceText and replaceSelection methods inherited from the TextInputControl class.

The previous section, [Anonymous Classes](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html), shows you how to implement a base class without giving it a name. Although this is often more concise than a named class, for classes with only one method, even an anonymous class seems a bit excessive and cumbersome. Lambda expressions let you express instances of single-method classes more compactly.

##### Example:

public static void processPersons(

List<Person> roster,

Predicate<Person> tester,

**Consumer<Person> block**) {

for (Person p : roster) {

if (tester.test(p)) {

**block.accept(p);**

}

}

}

processPersons(

roster,

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25,

**p -> p.printPerson()**

);

##### Example:

public static void processPersonsWithFunction(

List<Person> roster,

Predicate<Person> tester,

Function<Person, String> mapper,

Consumer<String> block) {

for (Person p : roster) {

if (tester.test(p)) {

String data = mapper.apply(p);

block.accept(data);

}

}

}

processPersonsWithFunction(

roster,

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25,

p -> p.getEmailAddress(),

email -> System.out.println(email)

);

##### Example:

public static <X, Y> void processElements(

Iterable<X> source,

Predicate<X> tester,

Function <X, Y> mapper,

Consumer<Y> block) {

for (X p : source) {

if (tester.test(p)) {

Y data = mapper.apply(p);

block.accept(data);

}

}

}

processElements(

roster,

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25,

p -> p.getEmailAddress(),

email -> System.out.println(email)

);

##### Example:

roster

.stream()

.filter(

p -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25)

.map(p -> p.getEmailAddress())

.forEach(email -> System.out.println(email));