Java Basics Cheatsheet

(Tested with Java 11)

The Java Basics Cheatsheet for beginners is designed in an easy to understand flow with examples to learn the basics of Java. It can also be considered as a guide to Getting Started with Java. It covers the basic usage of primitives, for loop, while loop, if/else statement with examples. It also provides an overview of classes, constructors, methods, interfaces, and exception handling in Java. It covers the basic usage of enums, strings, and math operations in Java. At the end, it provides the link to download the PDF having all the sections of the Java Basics Cheat Sheet.

0 - Installers

Java 11 On macOS, Java 11 On Windows, Java 11 On Ubuntu, Java 8 On macOS, Java 8 On Windows, Java 8 On Ubuntu, IntelliJ IDEA on Windows, IntelliJ IDEA on Ubuntu, Visual Studio Code on Windows, Visual Studio Code on Ubuntu, Eclipse on Windows, Eclipse on Ubuntu, Eclipse on Mac, and NetBeans on Windows

1 - Hello World

```
# The simplest source file in Java i.e. Hello.java to print hello world is
shown below.

public class Hello {

        public static void main( String[] args ) {

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

# We can compile the java file using javac command as shown below.
# We can compile multiple source files using the same command.

javac [options] [source files]

# Example
javac Hello.java

# On compilation of source file, the compiler generates class file.

# We can use the java command to invoke the Java Virtual Machine (JVM).

java [options] class [args]

# Example
java Hello
```

```
# Output
Hello World !!
```

2 - Primitive Data Types

```
# A primitive literal is source code representation of the primitive data
types.
# Integer numbers in Java can be represented in four ways: decimal(base 10),
octal(base 8), hexadecimal(base 16), and binary(base 2).
# Numeric literals can be declared using underscore characters(_).
# Underscore cannot be used at the start or end of the literal.
# Underscore can be used for short, int, long, double, and float.
# Underscore cannot be used directly next to the decimal point for double and
float.
# Underscore cannot be used next to X or B in hex or binary numbers.
# Binary literals starts with either 0B or 0b.
# Octal literals starts with zero(0). We can have up to 21 digits in octal
number excluding the leading zero.
# Hexadecimal literals starts with 0X or 0x. We can have up to 16 digits in
hexadecimal number excluding 0X or 0x.
# All the four integer literals decimal, binary, octal, and hexadecimal can
also be specified as long by adding L or l at the last.
# A literal integer is always an int.
# The compiler implicitly casts the int value to short within the range of
short. The compilation fails if int literal is out of range.
# The compiler implicitly casts the int value to byte within the range of
byte. The compilation fails if int literal is out of range.
# A literal floating point number is always an double.
# The literal assigned to float must end with F or f.
# It's optional to attach D or d at the last of double.
# A boolean literal can be only true or false.
# The char literal can use unicode notation by prefixing the value with \u.

Primitives: char, boolean, byte, short, int, long, double, float

# Literal Examples
char: 'a', '\u004E' (letter N), '\"' (double quote), '\n' (newline), '\t'
(tab)
boolean: true or false
byte: 41, short: 198
int: 12321, 1_000_000 (1 million), 0B1010 (decimal 10)
long: 1221321, 1_221_321
float: 12.342f, double: 12121.87
```

3 - Primitives Range

Type	Bits	Bytes	Min	Max
byte	8	1	-2^{7}	$2^{7} - 1$
char	16	2	0	65,535
short	16	2	-2^{15}	$2^{15} - 1$
int	32	4	-2^{31}	$2^{31} - 1$
long	64	8	-2^{63}	$2^{63} - 1$
float	32	4	-	-
double	64	8	-	-

4 - Non-Primitive Data Types

```
# The non-primitive data types in Java includes String, Array, Class, and
Interface.
# Strings are objects used to store characters in a consecutive manner.
# Strings are immutable i.e. once created on heap, it cannot be modified.
# Arrays are used to store data in a consecutive manner.
# All the arrays in Java are objects.
# Classes are used to create objects.
# Interfaces are abstract in nature i.e. they do not provide implementation
of the methods declared in them.
# Since Java 8, interfaces can provide default or static methods
implementation.
```

5 - Keywords & Identifiers

```
# Keywords are reserved words in Java

# Keywords: char, byte, short, int, long, float, double, boolean
# Keywords: class, interface, extends, implements, this, super, new, return,
instanceof, package, import
# Keywords: public, private, protected
# Keywords: default, abstract, final, native, strictfp, synchronized,
transient, volatile, static, void
# Keywords: if, else, for, while, do, switch, case, goto, break, continue
# Keywords: try, catch, finally, throw, throws
# Keywords: const, assert, enum

# Identifier can be used as name of variables, classes, or methods

# Identifier Rules
# Must start with a letter, currency characters (e.g. $), or connecting
```

```
characters (e.g. _)
# Cannot start with a digit
# After first character, identifiers can have numbers, letters, currency
characters, or connecting characters
# Cannot use special characters including !, @, #, %
# Keywords cannot be used as Identifier
# Can be of any length
# Identifiers are case sensitive

# Identifier Examples
# Correct: $alpha, _bravo, alpha_1_2_$, alpha_bravo, alpha_123,
this_is_a_long_name
# Correct: alpha, Alpha, aLpha, and alphA : all are different
# Wrong: 12alpha, @alpha, al!pha, else, while
```

6 - Naming Conventions

```
# Classes and Interfaces

# First letter should be capital
# First letter of inner words should be uppercase (CamelCase convention)
# The Class name should be noun
# The Interface name should be adjective

# Methods

# First letter should be small
# First letter of inner words should be uppercase (CamelCase convention)
# The method name should be verb-noun pair

# Variables

# First letter should be small
# First letter of inner words should be uppercase (CamelCase convention)

# Constants

# The variables marked as static and final
# The constants should be named using uppercase letters
# The words should be separated by _ (underscore)
```

7 - Assignment Operators

# We can assign a primitive variable using a literal.
# We can assign a primitive variable using result of an expression.

Assignment Operator

= : Equal : int b = 5, int a = b, short s = 5, byte b1 = 2
int c = a + b,
byte d = (byte) (a + b)
short e = (short) (a + b)
String str = "Hello"

Compound Assignment Operators

+= : Increment : int a += b ~ int a = a + b
-= : Decrement : int a -= b ~ int a = a - b
***= : Multiplication** : int a *= b ~ int a = a * b
/= : Division : float a /= b ~ float a = a / b
%= : Modulus : int a %= b ~ int a = a % b

Notes:
# The compiler do implicit casting of int to short and byte.
# The compiler won't do implicit casting of result of an expression.

```
short s = 5;

s += 2; // Ok
s += 2 + 5; // Ok
s = s + 2 + 5; // Compilation fails
```

8 - Relational Operators

# Relational operators always result in a boolean(true or false) value.
# Equality operators include Equal and Not equal.
# Equality operators can be used for two numbers, characters, boolean(true or false), or object reference variables.
# Equality operators cannot be used to compare incompatible types.
# The relational operators 'Greater than', 'Less than', 'Greater than or equal to', and 'Less than or equal to' can be used to compare any combination of integers, floating-point numbers, or characters.

== : Equal : a == b
!= : Not equal : a != b
> : Greater than : a > b
< : Less than : a < b
>= : Greater than or equal to : a >= b

```
 <= : Less than or equal to : a <= b

# Comparison Examples

int a = 12;
float b = 15.25f;
boolean c = a > b;
```

9 - Arithmetic Operators

```
 + : Addition : a + b, a + b + c
 - : Subtraction : a - b, a + b - c
 * : Multiplication : a * b, a * (b + c)
 / : Division : a / b, ( a + b ) / c
 % : Modulus : a % b
 ++ : Increment : ++b (prefix), a++ (postfix)
 -- : Decrement : --a (prefix), a-- (postfix)

Notes:
# Increment and Decrement operators cannot be applied to final variables.
# The prefix operator runs before the value is used in the expression.
# The postfix operator runs after the value is used in the expression.
# We can also use the addition operator to concatenate strings.
# Since strings are immutable, new string object will be created on the heap
if it does not exist in the pool.
# If either operand is a String, the addition operator becomes string
concatenation operator.

String a = "Hello"; // Creates first string object on heap
String b = "World"; // Creates second string object on heap
String c = "HelloWorld"; // Creates third string object on heap
String d = a + b; // Refers to third string object(searches in the pool) on
heap
String e = "Hello" + " Java"; // Creates fourth string object on heap
String f = "Hello" + 12; // Results in Hello12
```

10 - Logical Operators

```
# The Logical Operators are used to perform logical AND(& or &&), OR(| or
||), XOR(^) and NOT(!) operations.
# The logical operators & and && returns true if both the operands or
conditions return true.
# The logical operators & and && returns false if either of the condition
return false.
```

```
# The logical operator & evaluates both the operands or condition.
# The logical operator && returns false if first condition return false and
it won't check the second condition.
# The logical operators | or || returns true if either of the operands or
conditions return true.
# The logical operators | or || returns false if both the conditions return
false.
# The logical operator | evaluates both the operands or condition.
# The logical operator || returns true if first condition return true and it
won't check the second condition.
# The logical operator ^ returns true if only one operand returns true.
# The logical operator ! is a unary operator.
# The logical operator ! returns true if condition under consideration is
false.
# The logical operator ! returns false if condition under consideration is
true.

 &  : (x == 25) & (y == 35)
 |  : (x == 25) | (y == 35)
 ^  : (x == 25) ^ (y == 35)
 !  : !(x == 25)
 &&  : (x == 25) && (y == 35)
 ||  : (x == 25) || (y == 35)
```

11 - Bitwise Operators

```
 & : AND : Copies 1 if both bits are 1, else use 0
 | : OR : Copies 1 if either of the bits is 1, else use 0
 ^ : XOR : Copies 1 if only one of the bits is 1, else use 0
 ~ : NOT (Compliment) : Simply flip the bits
 << : left shift : Shift left and fill by zero on right
 >> : right shift : Shift right and fill by zero on left
 >>> : unsigned right shift : Shift right and fill by zero on left, set
leftmost to 0
```

12 - Conditional Operator

```
# Conditional Operator is the only operator which takes three operands.
# It's also called as ternary operator.
# It's used to evaluate boolean expressions.

 variable = Expression1 ? Expression2 : Expression3

# It operates similarly to that of the if-else statement.
# It can also be nested to evaluate more complex logic.
# If Expression1 evaluates to true, Expression2 will be executed.
# If Expression1 evaluates to false, Expression3 will be executed.

int a = 10;
int b = 5;
int sum = ( a > b ) ? ( a + b ) : ( a - b ); // Results in 15
int sub = ( b < a ) ? ( a - b ) : ( a + b ); // Results in 5
```

13 - instanceof Operator

```
# The instanceof operator is used to perform IS-A relationship test.
# It can be used to test whether the given object is of the given type.
# It can only be used to test null or objects against class types in the same class hierarchy.
# An object passes instanceof operator test for interfaces if the class or it's superclasses implements the interface.

int[] arr = { 1, 5, 10, 15 };
boolean a = null instanceof Object; // It returns false
boolean b = arr instanceof Object; // It returns true
```

14 - Primitives Type Conversions

```
# Implicit Type Casting - Implicit conversions or widening happens when we put a smaller number to a larger number.
# Explicit Typecasting - Explicit conversion or narrowing is required when we put a larger number to a small number. Some information might be lost in this case.

# Integers - Implicit
byte a = 5; // 5
int b = 121;   // 121
long c = b; // 121
b = a; // 5
short s = 1234;

# Integers - Explicit
int a = (int) 21.21; // 21
short b = (short) ( a + 25 ); // 25
int c = (int) 55.45f; // 55
byte d = (byte) 42L; // 42
short s = (short) 1234223;
```

```
# Floating Points
double a = 100L; // Implicit
float b = 20.25f; // Implicit
float c = (float) 20.25; // Explicit
float f = (float) (a + 20.25); // Explicit
```

15 - Arrays

```
# All the arrays are objects.
# All the array elements are given default value on instantiation.

int[] arr1 = { 1, 5, 10, 15 };

int[] arr2 = new int[ 5 ];

arr1[ 0 ] = 2;
arr2[ 3 ] = 25;
```

16 - Conditional Statements

```
# Statement Blocks - if, else, else if, and switch.
# Statement block executed only if given condition is true.
# The expression in an if statement must resolve to a boolean expression.
# The switch expression must evaluate to a char, byte, short, int, enum, or
String.
# The switch can only check for equality.
# The case constant must evaluate to a char, byte, short, int, enum, or
String.
# The case constant must be a literal or a compile-time constant.

int a = 20;
int b = 15;

# if

if( a > b ) {

        System.out.println( "A is greater than B" );
}

# else

if( a > b ) {
        System.out.println( "A is greater than B" );
}
else {

        System.out.println( "A is smaller than or equal to B" );
}
```

# **else if**

```
if( a > b ) {

        System.out.println( "A is greater than B" );
}
else if( a < b ) {

        System.out.println( "A is smaller than B" );
}
else {

        System.out.println( "A is equal to B" );
}
```

# **switch**

```
switch( a ) {

        case 10: {

                System.out.println( "A is equal to 10." );

                break;
        }
        case 20: {

                System.out.println( "A is equal to 20." );

                break;
        }
        default: {

                System.out.println( "No matching result found." );
        }
}
```

17 - Iterative Loop Statement

```
# Iterative Loop - for

# Iterate Array
String[] fruits = { "Apple", "Banana", "Orange", "Grapes" };
String fruit = "Banana";

# Basic For Loop

for( int i = 0; i < fruits.length; i++ ) {

        if( fruit.equals( fruits[ i ] ) ) {

                System.out.println( "Found fruit: " + fruit );

                break;
        }
}

# Enhanced For Loop

for( String element : fruits ) {

        if( fruit.equals( element ) ) {

                System.out.println( "Found fruit: " + fruit );

                break;
        }
}
```

18 - Conditional Loop Statement

```
# Conditional Loops - while and do
# Statement block executes till the condition is true.
# In while loop, statement block executes after checking the condition.
# In do loop, statement block executes before checking the condition.
# In do loop, the statement block executes at least once.

# while with condition

int limit = 5;
int start = 1;
int factor = 2;

while( start <= limit ) {
        int result = start * factor;

        System.out.println( "Multiplied " + start + " by " + factor + ": " +
result );

        start = start + 1;
}
```

# **while with condition and break**

```
String[] fruits = { "Apple", "Banana", "Orange", "Grapes" };

int count = fruits.length;
int index = 0;

while( index < count ) {

        if( "Orange".equals( fruits[ index ] ) ) {

                System.out.print( "Found Orange" );

                break;
        }
        else {

                index = index + 1;

                continue;
        }
}
```

# **do with while**

```
int limit = 5;
int start = 1;
int factor = 2;

do {

        int result = start * factor;

        System.out.println( "Multiplied " + start + " by " + factor + ": " +
result );

        start = start + 1;
}
```

```
while( start <= limit );
```

19 - Modifiers

# **Access Modifiers**

# There are three access modifiers in Java including public, private, and protected.
# Apart from these three modifiers, a class or member can have default visibility without specifying any access modifier.
# **public** Visible within the same or other packages.
# **protected** Visible within the same package or child classes in same or other packages.
# **private** Visible within the same class.
# **none** Visible within the same package.

# **Non-Access Modifiers**

# Apart from the three access modifiers, there are non-access modifiers including abstract, final, strictfp, default, static, native, synchronized, transient, and volatile.
# **abstract** Applicable on classes, interfaces, and methods.
# **final** Applicable on classes, methods, instance variables, local variables, class variables, and interface variables.
# **strictfp** Applicable on classes, and methods. The floating points in classes or methods marked as strictfp must adhere to IEEE 754 standard.
# **default** Applicable on interface methods with body since Java 8.
# **static** Applicable on nested classes, methods, class variables, and interface variables. Also, applicable on interface methods since Java 8.
# **native** Applicable on methods to call platform dependent method.
# **synchronized** Applicable on methods and method blocks. The synchronized methods and blocks can be accessed by one thread at a time.
# **transient** The instance variables marked as transient won't be serialized.
# **volatile** The instance variables marked as volatile will force threads to occupy their own copy of the variable with the master copy in memory.

Access Modifiers Scope

Scope	Private	Default	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package	No	Yes	Yes	Yes
Different Package Sub Class	No	No	Yes	Yes

Different Package Non-Sub Class	No	No	No	Yes

20 - Classes

```
# A Class can be declared using the keyword class.
# A class can be considered as blueprint or prototype to create the instances
of the class.
# Instances of a class are also known as objects.
# We can create multiple objects of the same class using the keyword new.
# Every Object has it's own state and behavior in the form of instances
fields and methods respectively.
# The classes marked as public are visible within other packages.
# The classes without any access modifier are visible within the same
package.
# The class declaration cannot use the private or protected access modifier.
# The nested class declaration can use the public, private or protected
access modifier.
# The nested class declaration can use the static modifier.
# The file name of the file having public class must be the same as that of
the class.
# The source code file can have any name in absence of public class. The name
must not match any class name of the classes declared within the file.
# A file can have only one public class.
# A file can have multiple non-public classes.
# If the class is part of a package, the package statement must be the first
line.
# If there are import statements, they must go between the package and first
class in the same source code file.
# In absence of package statement, import statement must be the first line.
# In absence of package and import statements, class declaration must be the
first line.
# It's preferred to follow Camel Case naming convention while naming the
method. The first letter can be capital and capital letter can be used for
the inner words.
```

# Class Examples

```
// Package Level
class Vehicle { }

public class Vehicle { }

public class Car { }

public class Truck { }
```

```
// Create Objects
Car car = new Car();
Truck truck = new Truck();
```

21 - Constructors

```
# The Java Compiler creates a default no-args constructor in case no
constructor is defined for a class.
# The default no-args constructor has the same access modifier as that of the
class.
# The default no-args constructor includes call to the superclass no-args
constructor using super().
# We can explicitly declare and define the no-args constructor.
# The constructor name must be the same as that of the class.
# The constructor must not be associated with return type.
# A class can have multiple overloaded constructors.
# The class objects can be created using the appropriate constructor.
# Every constructor, as it's first statement, implicitly invokes no-args
constructor of the superclass by calling super().
# We can explicitly invokes no-args constructor or overloaded constructor
with arguments of the same class by calling this() or superclass by calling
super() as the first statement of the constructor.
# In absence of no-args constructor and presence of constructor with
arguments, the compiler won't create the default no-args constructor.
# In absence of no-args constructor and presence of constructor with
arguments in superclass, the constructor must explicitly invokes no-args or
overloaded constructor of same class by calling this() or overloaded
constructor of superclass by calling super().
# In absence of no-args constructor and presence of constructor with
arguments, we cannot create an object without passing appropriate constructor
arguments.
# It's preferred to define a no-args constructor in presence of constructors
with arguments. Though, in several scenarios we might not be required to
define the no-args constructor.
# Constructors can use any access modifiers including public, protected,
private, or none.
# A constructor cannot be invoked like a method. A constructor can be invoked
within another constructor by calling this() or super() as the first
statement.
```

# Constructor Examples

```
public class Car {

        private String name;
        // Constructor without any argument and without return type
        public Car() {

                this.name = "Subaru";
        }

        // Constructor with argument and without return type
        public Car( String name ) {

                this.name = name;
        }
}

// Create Objects
Car subaru = new Car();
Car ferrari = new Car( "Ferrari" );
```

22 - Methods

```
# A Method is a block of code defining the behavior of the class.
# A method must always specify the return type.
# A class can have multiple methods with different names.
# A class can have multiple overloaded methods having same name, but
different parameters.
# Though it's not preferred, the method name can be the same as that of the
class with return type.
# A method can be either instance method or class method.
# The class methods must use the keyword static.
# The static methods can be called without creating the object of the class.
# The static methods can't be overridden.
# The static methods can't call non-static method or use instance variables.
# The static methods can call static methods or use static variables.
# The methods without any access modifiers remains visible within the same
package.
# The methods with public access modifier remains visible within the same and
other packages.
# The methods with protected access modifier remains visible within the same
package and child classes in the same package or other packages.
# It's preferred to follow Camel Case naming convention while naming the
method. The first letter can be small and capital letter can be used for the
inner words.
```

# Method Examples

```
public class Car {
        private String name;

        public Car() {

                this.name = "Subaru";
        }

        public Car( String name ) {

                this.name = name;
        }

        public String getName() {

                return name;
        }

        public void setName( String name ) {

                this.name = name;
        }
}

// Create Objects
Car myCar = new Car();

// Print Name
System.out.println( myCar.getName() );
```

23 - Interfaces

```
# An Interface can be declared using the keyword interface.
# An interface is implicitly abstract.
# An interface can be either package level with default visibility or marked
as public.
# An interface can be implemented by any class.
# The class implementing the interface must be marked as abstract if it does
not implement all the abstract methods of the interface.
# An interface can extend multiple interfaces.
# An interface cannot implement another interface.
# An interface cannot extend class.
# All the interface methods are implicitly public and abstract unless
declared as static or default.
# All the variables defined in the interface are implicitly public, static
and final.
# Interface methods cannot be marked as final, strictfp, or native.
# An interface cannot have constructor.

# Interface Examples

// Package Level
interface Bounceable { }

public abstract interface Bounceable { }

public interface Bounceable { }

public interface Bounceable {

        public void bounce();
}
```

24 - Inheritance

```
# A class can be sub-classed using the keyword extends.
# A class cannot extend more than one class.
# A class cam implement multiple interfaces.
# A class marked as final cannot be sub-classed.
# We can test the object type using the keyword instanceof.
# We can perform Is-A relationship test to identify the type of object.
# A constructor cannot be inherited.
# The child class inherits all the public and protected members of parent
classes.
# The child class can override parent class methods.
# The child class can overload parent class methods.
# Inheritance makes it possible to re-use the code by extending existing
classes without re-implementing the same logic.

# Inheritance Examples

public interface Bounceable {

        public void bounce();
}

public class Vehicle implements Bounceable {

        public void bounce() {

                System.out.println( "Bounce on Jump" );
        }
}
```

```
public class Car extends Vehicle { }

public class Truck extends Vehicle { }

// Create Objects
Car car = new Car();
Truck truck = new Truck();
Vehicle myCar = new Car();

# IS-A relationship tests

// car Is-A Bounceable
// car Is-A Car
// car Is-A Vehicle
// truck Is-A Vehicle
// myCar Is-A Car
// myCar Is-A Vehicle
```

25 - Variable Initialization

# **Static Variables**

# Static variables are declared in a class and outside of constructors and methods.
# Static variables are declared using the static keyword.
# Static variables are also called as class variables.
# Static variables can be public, private, or protected.
# Static variables declared using static and final keywords are called as constants.
# Static variables are shared among all the objects of a class.

# **Instance Variables**

# Instance variables are variables defined at class level.
# The instance variables are also called as member variables.
# Instance variables can be public, private, or protected.
# Each instance variable get's a default value each time a new object is created, in case explicit value is not assigned.
# If array is initialized or constructed, all array elements will get default value.

# **Instance Variables - Default Values**

# **Object Reference** - null
# **byte, short, int, long** - 0

# **float, double** - 0.0
# **boolean** - false
# **char** - '\u0000'
# **Array** - null

# **Local Variables**

# Local variables are declared and initialized within a method body, constructor, or block.
# Local Variables are also called as Stack or Automatic Variables.
# Local variables cannot use the access modifiers including public, private, or protected.
# Local variables can use final non-access modifier.
# Local Variable must be assigned a value before accessing it.
# Local Variable won't be assigned with default value as compared to Instance Variable.
# Local Variables are defined within a method including method parameters.
# It's not required to explicitly initialize the array elements. The array elements will get default values on constructing it.

# **Notes**

# Local variables or method variables live on the stack.
# Objects and their instance variables live on the heap.

26 - Exception Handling

# Occurrence of exceptional condition alters the normal program flow.
# An exception is said to be thrown, when an exceptional event occurs.
# The exception handler code catches the thrown exception.
# Exception handling transfers the program execution to exception handler.
# Exceptions can be handled using **try** and **catch** keywords.
# The keyword **try** is used to define a block of code where exception might happen.
# The catch clauses followed by the try block catches the exceptions and handle them.
# The **finally** block will always execute irrespective of the exceptional condition except when the try or catch block calls System.exit().
# The finally clause is optional in presence of catch clauses.
# The finally clause is required in absence of catch clauses.
# The try block must be followed by catch clauses to handle **checked exceptions**, unless the method throws the exceptions.
# The method declaration can use **throws** keyword and list the exceptions it throws.
# The method can exclusively throw an exception using the **throw** keyword. It must declare it using the **throws** keyword if not handled within the method

body i.e. it propagates the exception to the calling method.
# All **non-RuntimeExceptions** are considered as **checked exceptions** since compiler checks such exceptions.
# **RuntimeException** and their sub-classes are considered as **unchecked exceptions**.
# Similar to **RuntimeException** and their sub-classes, **Error** and their sub-classes do not need to be handled or declared.
# The compiler checks the checked exceptions to make sure that they are handled or declared.

# **Exception Handling**

```
String myfileName = "myfile.txt";

try {


        FileReader fileReader = new FileReader( new File( myfileName ) );
        BufferedReader reader = new BufferedReader( fileReader );

        String firstLine = reader.readLine();

        System.out.println( "Line 1: " + firstLine );

        reader.close();
}
catch( FileNotFoundException ex ) {

        System.out.println( "File not found." );
}
catch( IOException e ) {

        System.out.println( "Failed reading the file." );
}
finally {

        System.out.println( "Done with file handling." );
}
```

27 - Enums

# Enums allows a variable of type enum to have only one value from an enumerated list.
# Enums can have constructors, variables, and methods.
# Enums can have overloaded constructors.
# Enums constructor cannot be invoked directly to create objects like we do with classes.

# Enums outside the class can be declared with public or default access modifiers.
# Enums within a class can be declared with public, private, protected or default access modifiers.
# Enums cannot be declared within a method.
# Enum constants supports anonymous inner class known as **constant specific class body**.
# Enums can implement interfaces.

# **Enum Example**

```
enum TeaBag {

        SMALL(5), MEDIUM(10), LARGE(15);

        private int size;

        TeaBag( int size ) {

                this.size = size;
        }

        public int getSize() {

                return this.size;
        }
}
```

# **Enum Usage**

```
TeaBag a = TeaBag.SMALL;
TeaBag b = TeaBag.LARGE;

System.out.println( TeaBag.MEDIUM ); // Prints MEDIUM
System.out.println( a ); // Prints SMALL
System.out.println( b ); // Prints LARGE
System.out.println( a.getSize() ); // Prints 5
System.out.println( b.getSize() ); // Prints 10
System.out.println( a == TeaBag.SMALL ); // Prints true

// We can also iterate enum by calling the static method values()
for( TeaBag teaBag : TeaBag.values() ) {

        System.out.println( teaBag );
}
```

28 - String and StringBuilder

# String objects are immutable, i.e. once created on heap, they cannot be changed.
# Each character in a string is a 16-bit Unicode character.
# StringBuilder and StringBuffer objects are mutable.
# StringBuilder is more efficient than StringBuffer, since StringBuffer methods are synchronized.

# **String Methods**

charAt(int n): Returns the nth character of the string (n starts from zero)
compareTo(String str): Returns negative if string is lesser than the given string, positive if the string is greater than the given string, else returns zero if both are equal.
concat(): Concatenates two strings and returns the resulting string object reference
contains(String str): Returns true if string contains the given string
equals(String str): Returns true if strings are equal
equalsIgnoreCase(String str): Returns true if strings are equal, ignoring the case of characters
indexOf(char c): Returns the position of first occurrence of character c in the string.
indexOf(char c, int startIndex): Returns the position of first occurrence of character c after startIndex in the string.
length(): Returns the length of the string
replace(char charA, char charB): Replaces all occurrence of charA with charB and returns a string object reference
substring(int startIndex): Returns substring starting after character startIndex (startIndex starts from zero)
substring(int startIndex, int endIndex): Returns substring starting after character startIndex to endIndex (startIndex starts from zero, endIndex starts from 1)
toLowerCase(): Converts the string to lower case and returns a string object reference
toUpperCase(): Converts the string to upper case and returns a string object reference
toString(): Returns the value of the string
trim(): Removes the whitespaces at the beginning and at the end, and returns a string object reference
valueOf(Object obj): Converts the given parameter into string

# **String Examples**

```
// Creates an empty string object and assigns to reference variable
String a = new String();
```

```
// Creates a string object and assigns to reference variable
// Also, adds another object i.e. "Animal" to the pool
String b = new String( "Animal" );

// Creates a string object "Cat" in the pool and assigns to reference
variable
String c = "Cat";

// Creates a string object "Dog" in the pool and assigns to reference
variable
a = "Dog";

System.out.println( a ); // Prints Dog
System.out.println( b ); // Prints Animal
System.out.println( c ); // Prints Cat
```

# **StringBuilder Examples**

```
StringBuilder a = new StringBuilder();
StringBuilder b = new StringBuilder( "Animal" );

a.append( "Cat" );

System.out.println( a ); // Prints Cat
System.out.println( b ); // Prints Animal
```

29 - Console

# We can use the Scanner and BufferedReader classes to get input from the console.

# **Scanner Examples**

```
// Create scanner object and assign to scanner reference variable
Scanner scanner = new Scanner( System.in );

System.out.println( "Enter your name: " );

// Read String
String name = scanner.next();

System.out.println( "Enter your age: " );

// Read Integer
int age = scanner.nextInt();
```

```
System.out.println( "Name: " + name + " Age: " + age );
```

# BufferedReader Examples

```
// Create BufferedReader object and assign to reader reference variable
BufferedReader reader = new BufferedReader( new InputStreamReader( System.in
) );

try {

        System.out.println( "Enter your name: " );

        // Read String
        String name = reader.readLine();

        System.out.println( "Enter your age: " );

        // Read Integer
        int age = Integer.parseInt( reader.readLine() );

        System.out.println( "Name: " + name + " Age: " + age );
}
catch( NumberFormatException e ) {

        System.out.println( "Entered wrong age." );
}
catch( IOException e ) {

        System.out.println( "Failed to read." );
}
```

30 - Maths

# **Math.ceil(double num)** - Smallest double(equivalent mathematical integer) greater than or equal to num
# **Math.floor(double num)** - Largest double(equivalent mathematical integer) less than or equal to num
# **Math.round(double num)** - Closest long to num
# **Math.round(float num)** - Closest int to num
# **Math.pow(double num, double power)** - Returns double - num raised to the power
# **Math.random()** - Returns double value with a positive sign, greater than or equal to 0.0 and less than 1.0
# **Math.sqrt(double num)** - Returns double - Square root of num
# **Math.asin(double num)** - Returns double - Arc sine of num radians

```
# Math.acos( double num ) - Returns double - Arc cosine of num radians
# Math.atan( double num ) - Returns double - Arc tangent of num radians
# Math.sin( double num ) - Returns double - Sine of num radians
# Math.cos( double num ) - Returns double - Cosine of num radians
# Math.tan( double num ) - Returns double - Tangent of num radians
# Math.toDegrees( double num ) - Returns double - Angle num from radians to
degrees
# Math.toRadians( double num ) - Returns double - Degree num from degrees to
radians
# Math.exp( double x ) - Returns double - e raised to the power x, e =
2.718281
# Math.expm1( double x ) - Returns double - e raised to the power x minus 1,
e = 2.718281
# Math.log( double x ) - Returns double - natural logarithm of x (to base e)
# Math.log10( double x ) - Returns double - natural logarithm of x (to base
10)

# Examples

double a = Math.ceil( 12.75 ); // return 13
double b = Math.floor( 12.75 ); // return 12
double c = Math.round( 0.25 ); // returns 0
double d = Math.round( 0.55 ); // returns 1
```

31 - Files

```
# File class in Java is not used to read/write data. It's an abstract
representation of file and directory pathnames.
# FileReader and FileWriter classes are used to read and write character
files.
# BufferedReader and BufferedWriter classes are used to read and write large
chunks of data minimizing the operations.
# FileInputStream and FileOutputStream classes are used to read or write
bytes from files. Both the classes can be used for binary as well as text.

# Examples

File file = new File( "myfile.txt" );

# FileReader

try {

        FileReader fr = new FileReader( file );

        int content;
        while( ( content = fr.read() ) != -1 ) {

                System.out.print((char) content);
        }

        fr.close();
}
catch( FileNotFoundException e ) {

        e.printStackTrace();
}
catch( IOException e ) {

        e.printStackTrace();
}

# BufferedReader

try {

        BufferedReader br = new BufferedReader( new FileReader( file ) );

        String line;

        while( ( line = br.readLine() ) != null ) {

                System.out.println( line );
        }
}
catch( FileNotFoundException e ) {

        e.printStackTrace();
}
catch( IOException e ) {

        e.printStackTrace();
}
```

The Java Basics Cheatsheet by https://www.tutorials24x7.com. Download PDF.

Notes

This cheatsheet does not cover all the programming concepts of Java.
It can be considered as a reference to quickly revise or learn the basic concepts of Java.
Submit the Contact Form in case you face any issues in using it or finds any discrepancy.

We at Tutorials24x7 are happy to share our experience and problems faced by us in our day to day activities with their resolutions.

We are also happy to share our experience in the form of Cheatsheets for quick references of popular programming languages and frameworks.

Tutorials 24x7

Connect With Us

Explore

About
Contact
Terms
Feedback
Privacy
Blog
Testimonial