**Java commands:**

javac Test.java

java Test

Don’t write .class in java command.

javac creates .class from .java file and then java command executes that .class file.

**Variable Naming:**

$ and \_ are valid variables in java

**Switch Statement**

Data types supported by switch statements include the following:

■ int and Integer

■ byte and Byte

■ short and Short

■ char and Character

■ String

■ enum values

Note that boolean, long, float, double and their associated wrapper classes, are not supported by switch statements.

The syntax of default case is “**default**”, not “case default”.

Only constants can be used in switch cases

**Primitive Data Type Rules**:

Characters are initialized to Unicode ‘\u0000’

Characters can be used in array subscripts for index

The \u bits are in HEX, so the Unicode for ‘a’ is '\u0061'.

Order of chars in Unicode table:

048 – 057 : 0 –> 9

065 – 090: A -> Z

097 – 122: a -> z

Prefixes/Suffixes:

Binary: 0b 0b101

Hex: 0x 0xE

Long: L 20L

Float: f 10.5f

byte a = 40;

byte b = 50;

byte result = (byte) a + b; //Compile Error

cast operator has highest precedence

We can use class name as identifier but can’t use reserved word.

String String = “test”; is valid statement

goto is a reserved word but is not used.

char cannot be implicitly converted to byte and short.

char c = ‘a’;

short s =c; //compile error

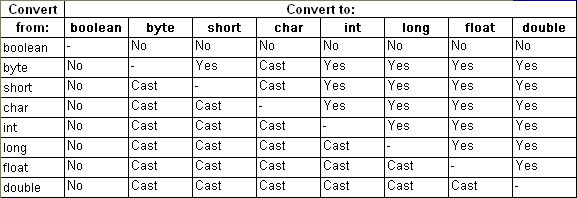
int i = c; //success

Numeric types cannot be converted to char type implicitly.

short s = 10;

char c = s; //compile error

A character can be explicitly casted to any primitive type other than short and byte.



A short CONSTANT can be assigned to a char only if the value fits into a char. short s = 1; byte b = s; => this will also not compile because although value is small enough to be held by a byte but the Right Hand Side i.e. s is a variable and not a constant. final short s = 1; byte b = s; => This is fine because s is a constant and the value fits into a byte. final short s = 200; byte b = s; => This is invalid because although s is a constant but the value does not fit into a byte.

Implicit narrowing occurs only for byte, char, short, and int.

Remember that it does not occur for long, float, or double. So, this will not compile: int i = 129L;

int x = ( x=3 ) \* 4; //valid statement.

i = i++

value of i will never change.

i++ increments the value of i but returns the old value of i. so the old value is assigned again.

**Operators**

|  |  |
| --- | --- |
| a ^ b | Xor operator. False when both are same. Can work on binary and integral types. |
| a | b  a & b | Bitwise operator. Can work on binary and integral types. |
| % | Can be used on float and doubles too |
| !  &&  || | Work with boolean only |
| ~ | Bitwise compliment. Works with integral types only. |
| **a >> b** | Right shift. |
| **a << b** | Left shift |
| **a >>> b** | Zero fill right shift |

**integral types means byte, short, int, long, and char**

Note: There is no <<< operator in java.

The arithmetic operators \*, / and % have the same level of precedence.**Assignment operator has least precedence of all operators.**

**Numeric Promotion**

1. If two values have different data types, Java will automatically promote one of the values to the larger of the two data types.
2. If one of the values is integral and the other is floating-point, Java will automatically

promote the integral value to the floating-point value’s data type.

1. Smaller data types, namely byte, short, and char, are first promoted to int any time

they’re used with a Java binary arithmetic operator, even if neither of the operands is int.

1. After all promotion has occurred and the operands have the same data type, the resulting

value will have the same data type as its promoted operands.

**String**

*Important methods:*

* indexOf(string, fromIndex)
* substring(startIndex, endIndex)
* concat(string)

String concatenation is left to right associative.

These are the six facts on Strings:

1. Literal strings within the same class in the same package represent references to the same String object.
2. Literal strings within different classes in the same package represent references to the same String object.
3. Literal strings within different classes in different packages likewise represent references to the same String object.
4. Strings computed by constant expressions are computed at compile time and then treated as if they were literals.
5. Strings computed at run time are newly created and therefore are distinct.
6. The result of explicitly interning a computed string is the same string as any preexisting literal string with the same contents. (So line 5 prints true.)

Replace returns the same object if there is no change.

The charAt(i) method throws IndexOutOfBoundsException type if passed invalid value.

It mostly throws StringIndexOutOfBoundsException but it is free to choose.

Note: Always choose IndexOutOfBoundsException if it is in options of question

**StringBuilder**

StringBuilder is not immutable

*Important methods:*

* *insert()*
* *delete(from, to)*
* *deleteCharAt()*
* *reverse()*
* ***ensureCapacity(int)***

*increases the capacity to the given number if needed.*

* ***setLength(int)***

*if lesser than current length, will truncate to new length.*

*If greater than current length, will append \u0000 upto the new length.*

* ***trimToSize(int)***

*if capacity is larger than size, will try to reduce capacity.*

*StringBuilder has contructor which take capacity as integer and another which takes string as value*

***StringBuffer is slow because it is thread safe***

Methods not supported by StringBuilder:

* Clear()
* Empty()
* removeAll
* deleteAll()

**Arrays:**

int [] arr = new int[10];

initialized array:

int [] arr = new int[] {10, 20};

or

int [] arr = {10, 20}; //anonymous array as no size and type is specified

int [] arr1, arr2; //both are array type

int arr[], arr2; //only first is array

string [] arr = new [] String {“a”}; //Compile error. [] cannot be before Type.

If you give the elements explicitly you can't give the size.

Arrays.sort(arr); //comparator is optional. Also from and to index can be given

Arrays.binarySearch(arr, number);

if element is not found, the index of the position where element can be placed is negated and 1 is subtracted and then returned.

Collections.binarySearch can work on strings array also

**VarArgs**

Void someMethod(String …args)

Must be the last parameter of a method

Null can be passed as a parameter to var args.

**Multi-dimensional arrays**

int[][] vars1; // 2D array

int vars2 [][]; // 2D array

int[] vars3[]; // 2D array

int[] vars4 [], space [][]; // a 2D AND a 3D array

int[][] differentSize = {{1, 4}, {3}, {9,8,7}};

int [][] args = new int[4][];

for (int[] inner : twoD) {

for (int num : inner)

System.out.print(num + " ");

System.out.println();

}

}

**ArrayList:**

import java.util.ArrayList;

implements RandomAccess

boolean add(E element)

void add(int index, E element)

add always return true. Some other classes in the family use this return type.

add()

remove()

set()

clear()

contains

equals()

**Dead code:**

In case of if else, dead code will be warning and not an error.

But in case of loops it will be unreachable code error given by compiler.

**Wrapper Classes**

int p = Integer.parseInt(“123”);

Integer i = Integer.valueOf(“123”);

2 constructor for each wrapper class:

1. Takes same type primitive
2. Takes String

== operator compares references, not primitive value.

Short k = new Short(9); //Compile error because 9 is considered an int.

short s = 9; //Although 9 is considered an int, but since it will fit into short, it is casted implicitly.

The equals method of all wrapper classes first checks if the two object are of same class or not. If not, they immediately return false.

All the wrapper objects are immutable. When you do i++, what actually happens is something like this:

i = new Integer( i.intValue() + 1); As you can see, a new Integer object is

assigned back to i.

However, to save on memory, Java 'reuses' all the wrapper objects whose values fall in the following ranges:

All Boolean values (true and false)

All Byte values

All Character values from \u0000 to \u007f (i.e. 0 to 127 in decimal)

All Short and Integer values from -128 to 127

Character class doesn’t provide above 2 methods

**Collections**

list.toArray(T [] arr)

If the result fits in arr, it will be copied to arr and returned. Otherwise new array will be created and returned.

Specify 0 sized array to make sure new array is returned

Arrays.asList()

Returns fixed sized arrays and the arraylist created is backed by the given array. Change in one will update the other.

Collections.sort()

Both array and arraylist are ordered

**Java 8 Date Time**

Import java.time.\*;

LocalDate

LocalDate date1 = LocalDate.of(2015, Month.JANUARY, 20);

LocalDate.now()

plusDays()

plusWeeks()

plusMonths

plusYears

*minus versions for all*

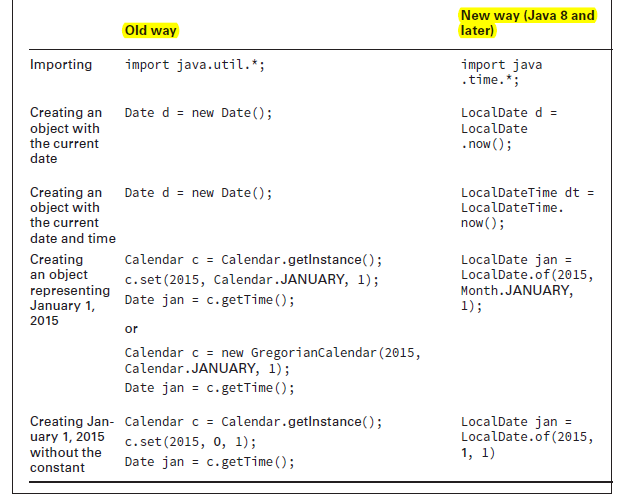
LocalTime

LocalDateTime

LocalDateTime ldt = LocalDateTime.of(ld, lt);

2015-01-20T12:45:18.401

Last value is nanoseconds, not milliseconds

****

Date and time classes are immutable.

**Period**

Period annually = Period.ofYears(1); // every 1 year

Period quarterly = Period.ofMonths(3); // every 3 months

Period everyThreeWeeks = Period.ofWeeks(3); // every 3 weeks

Period everyOtherDay = Period.ofDays(2); // every 2 days

Period everyYearAndAWeek = Period.of(1, 0, 7); // every year and 7 days

date.plus(period)

datetime.plus(period)

**Cannot chain methods when creating periods. It is compile successfully however only the last statement in chain is effective.**

DateTimeFormatter df = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);

DateTimeFormatter df = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);

DateTimeFormatter df = DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);

DateTimeFormatter df = DateTimeFormatter.ofPattern(“MM/dd/yyyy”);

LocalDate.parse(“10/10/2010”)

localDt.format(df)

System.out.println(ld.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT)));

11/5/16

System.out.println(ld.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG)));

November 5, 2016

**Static**

Static methods cannot hide instance methods and vice versa.

//both are legal. Method name and return type should be in order only.

Public final void foo();

Final public void foo();

Top level classes cannot be marked static or private.

Test t = null;

t.count //will not throw exception if count is static. This is because java will call count on the class, not on the object instance.

If static final variables are not initial, there will be compile error.

Static final variables must be set exactly one. It will give compile error, when they are set multiple times.

Instance final members also need to be set exactly once. Same rule as above.

import static class.\*;

static import class.\*; //does not compile

The word import should come before static.

Static imports doesn’t import the class itself.

Static imports are for importing static members of classes. Normal imports are for importing classes.

Static imports can use wildcard (\*)

Parameters names cannot be supplied with static import method name.

**Access Modifiers**

Private

Public

Default (only classes in same package can access) (Package Private)

Protected (classes in same package and for children outside package)

Default access is more restrictive than protected.

Always check for case sensitiveness

Protected rules apply under two scenarios:

1. A member is used without referring to a variable.

In this case, we are taking advantage of inheritance and protected access is allowed.

1. A member is used through a variable.

In this case, the rules for the reference type of the variable are what matter. If it is a subclass, protected access is allowed.

It is legal to import a class in the same file in which class is declared. :O

Remember that you can never access a class that is defined in the default package (i.e. the package with no name) from a class in any other package.

**Overloading Methods**

void test(int []arr);

void test(int …arr) //will not compile

Both are same, so conflict

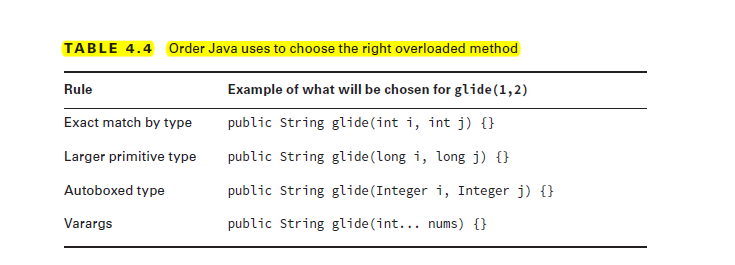
For overloading, the signatures must be same.

void test();

void test(int …arr) //will compile successfully

If argument is primitive, and primitive parameter method is present, it will have high priority. Autoboxing is only done when no other option is available.

Java always picks the most specific overloaded method.



*Java will not convert to large primitive and autobox at the same time.*

Void test(Long l) {}

test(12); //not compile

test(12L) //will compile

if null is passed, most specific method is selected.

If there are more than 1 specific, compile time errors occurs.

**Constructors**

Call to constructor using this() should be the first statement in the constructor.

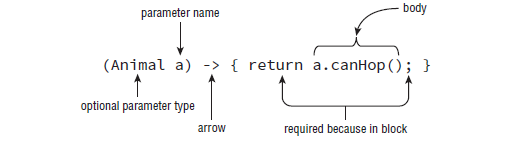
Default constructor is added by compiler if no constructor is provided.

The access type of a default constructor is same as the access type of the class. Thus, if a class is public, the default constructor will be public.

**Lambdas**

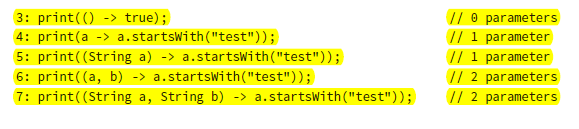
Functional programming uses lambda expressions to write code

Work with interfaces that have only one method.



Parentheses can be omitted if there is a single parameter and its type is not explicitly mentioned.

return keyword can be omitted ONLY if curly braces are NOT used.

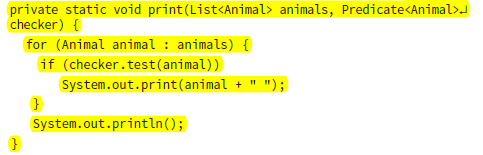


Parameters cannot be skipped (case 3)

(a, b) -> { int a = 0; return 5;} // DOES NOT COMPILE

a cannot be re-declared

Predicate is generic interface present in java.util.function



print(animals, a -> a.canHop());

ArrayList declares a removeIf() method that takes a predicate.

**Inheritance**

super() calls the parent constructor but super is used to access the members

super is a keyword and not an attribute.

super.super is not allowed.

**Overriding**

**Rules for overriding methods:**

1. Method in child class must have same signatures.
2. Method in child class must be at least as accessible or more accessible than the method in parent class.
3. Method in child class should not throw check excerption that is new or broader than the class of any exception thrown in the parent class. Child may eliminate parent’s method exception.
4. If method returns value, it must be same or subclass of the method return type in the parent class.

Private methods cannot be overridden since they are not visible in child class. Child class can define its own method.

A method signature is the method name and the number and type of its parameters. Return types and thrown exceptions are not considered to be a part of the method signature.

If method is overridden, child version will be used always even if it is accessed in the parent class. This is because it is replaced with the child method.

Final methods cannot be overridden.

**Hiding static methods**

First 4 rules from method riding are also applicable

+

If method is static in parent, then it should be static in the child class. If method is instance in parent class, it cannot be static in child class.

Main method can be declared static like other static methods.

**Hiding variables**

Reference determines which variable will be accessed.

Both public and private variables **cannot** be overridden but hidden

**Abstract Classes**

Only an abstract class can have abstract methods.

Abstract class or method cannot be marked as final

Abstract methods cannot be marked **private**

**Interfaces**

Interfaces are abstract by default

Only final variables can be added

All interfaces must be public

Interfaces doesn’t extend java.lang.object

Making interface as private, protected or final will trigger compile error.

All non-default methods in interfaces must be public

Making non-default methods private, protected or final will trigger compile error

Since all methods in interface are public, a class implementing that interface must declare methods as public. Otherwise it will be compile error because class will be trying to make them default access.

If 2 interfaces contains same constants, then class can still implement those interfaces. However using constant without interface name will result in compile time error due to ambiguity.

Unlike a class, an interface can extend from multiple interfaces.

If 2 interfaces contain a method with same signatures but different return type, class implementing both interfaces at a time will not compile due to conflict between 2 methods.

Default methods have implementation in the interface which can be overridden by the implementing class. Default methods must provide implementation in interface.

A default method cannot override a method from java.lang.Object.

If the class implements 2 interfaces that have same method signatures, and at least one of the interface has default implementation, compiler will give error. The class has to provide its own implementation to fix the error or remove the interface.

Functional Interfaces can contain more methods but only one abstract method

**Static methods in interfaces** now also possible. They must be public.

All non-final, non-static and non-private methods are virtual by default.

A class that implements two interfaces containing static methods with the same signature will still compile at runtime, because the static methods are not inherited by the subclass and must be accessed with a reference to the interface name.

Contrast this with the behavior you saw for default interface methods.

Child interface can override parent interface methods just like classes.

A class can implement parent and child interfaces both at the same time. It will not generate any conflict.

Finalize method doesn’t take any parameters.

Casting is only allowed between parent child classes.

String o = (String) new TestClass(); //does not compile

**Exceptions:**

Throwable is subclass of object

Runtime exceptions can be caught but they are not required.

Checked exceptions must be caught or declared.

try statement must have curly brackets even if there is single statement in it.

Runtime exception extends **RuntimeException**.

**Runtime Exceptions:**

NullPointerException

ClassCastException

ArithmeticException

NumberFormatException

SecurityException: Thrown by JVM securitymanager upon security violation

IllegalStateException: Thrown by application e.g when some method is invocated on a thread that is not in appropriate state.

**Checked Exception:**

FileNotFoundException

IOException

**Errors:**

ExceptionInInitializerError: Thrown when there is exception in static initializer

NoClassDefFoundError: Class present at compile time but not at runtime

StackOverflowError:

NumberFormatException and IOException are not thrown by JVM. They are thrown by wrapper/utility classes in java.

Even if you are throwing custom runtime exception, it is not needed to catch or declare it.

For checked exceptions, catch statements must declare only those check exception which are thrown in the code. Compiler doesn’t allow unreachable catch blocks.

If exception is thrown in finally block, that exception will be thrown and any exception thrown from try or catch will be ignored.

Errors are allowed to be handled or declared.

If the exception object thrown is null, jvm will generate NullPointerException.

exception.toString() doesn’t print exception stack. The output format is

<class name>: <message>

e.g java.lang.Exception: some error occurred.

When no main method is found, JVM throws java.lang.NoSuchMethodError

Use of uninitialized local variables in any statement will create compile error.

Remember to count exception object when counting total number of objects created.

Java Exceptions is a mechanism ..

1. that you can use to determine what to do when something unexpected happens.
2. for logging unexpected behavior



**Labelled loops:**

If label is put on statement other than loop label, it cannot be used in break.

public static void main(String[] args) {

int c = 0;

JACK: while (c < 8) {

JILL: {

System.out.println(c);

if (c > 3)

break JACK;

else

c++;

}

}

}

Break JILL will not break the loop but it will compile.

A break statement with no label attempts to transfer control to the innermost enclosing

switch, while, do, or for statement; this statement, which is called the break target,

then immediately completes normally. If no switch, while, do, or for statement

encloses the break statement, a compile-time error occurs.

A break statement with label Identifier attempts to transfer control to the enclosing

labeled statement that has the same Identifier as its label; this statement, which is

called the break target, then immediately completes normally. In this case, the break

target need not be a while, do, for, or switch statement.

**HashCode**

The hashCode() method of objects is used when you insert them into a HashTable, HashMap or HashSet. It is used as key in hashing.

Rules:

1. If object1 and object2 are equal according to their equals() method, they must also have the same hash code.
2. If object1 and object2 have the same hash code, they do NOT have to be equal too.

**Number:**

The abstract class Number is the superclass of classes BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short.

**Important Points:**

For encapsulation, data members must be private. Although setters are allowed but they are not required.

String replace method replaces all instances of given substring :D

Child class cannot access private methods of parent using super()

hashCode can be overridden

**synchronized** can only be applied to methods or blocks.

List of final classes:

* String
* StringBuilder
* StringBuffer
* All wrapper class (Integer, Long …)
* java.lang.System

All non-private method calls are polymorphic.

**int** i = 3;

**int** [][][] arr = **new** **int**[i][i=4][i];

System.***out***.println(arr.length + ", " + arr[0].length + ", " + arr[0][0].length);

**Output:** 3, 4, 4

**int** i = 4;

**int** result = i + (i=3) + i;

System.***out***.println(result);

**Output**: 10

**int** k = 1;

k += (k = 4) \* (k + 2);

System.***out***.println(k);

Output: 25

Object can be instantiated:

Object o = new Object();

System.out.println('a' + 1);

Output: 98

Both operands are promoted to integer.

System.out.println(100/10.0);

Output: 10.0

interface Flyer{ }class Bat { }

if(b instanceof Flyer) System.out.println("f is a Bird");

Note that there is no compilation issue with b instanceof Flyer because Flyer is

an interface and it is possible for b to point to an object of a class that is a sub class of

Bat and also implements Flyer. So the compiler doesn't complain. If you make Bat

class as final, b instanceof Flyer will not compile because the compiler knows

that it is not possible for b to point to an object of a class that implements Flyer.

**Classes:**

A class or interface type T will be initialized at its first active use, which occurs if: T is a class and a method actually declared in T (rather than inherited from a superclass) is invoked. T is a class and a constructor for class T is invoked, or U is an array with element type T, and an array of type U is created. A non-constant field declared in T (rather than inherited from a superclass or superinterface) is used or assigned. A constant field is one that is (explicitly or implicitly) both final and static, and that is initialized with the value of a compile-time constant expression . Java specifies that a reference to a constant field must be resolved at compile time to a copy of the compile-time constant value, so uses of such a field are never active uses. All other uses of a type are passive A reference to a field is an active use of only the class or interface that actually declares it, even though it might be referred to through the name of a subclass, a sub interface, or a class that implements an interface.

An instance member belongs to a single instance, not the class as a whole. An instance member is a member variable or a member method that belongs to a specific object instance. All non-static members are instance members.

transient and volatile modifiers are only valid for member field declarations. abstract and native are only valid for member methods.

native method cannot be abstract.

String s = "Hello";

String s2 = "He" + "llo";

System.***out***.println(s == s2);

Although there is more to it that the following sequence, for the purpose of exam, this is all you need to know:

1. Static blocks of the base class (only once, in the order they appear in the class).

2. Static blocks of the class.

3. Non-static blocks of the base class.

4. Constructor of the base class.

5. Non-static blocks of the class.

6. Constructor of the class.

**ArrayCopy**

**public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)**

**Parameters:**

src - the source array.

srcPos - starting position in the source array.

dest - the destination array.

destPos - starting position in the destination data.

length - the number of array elements to be copied.

Throws:

IndexOutOfBoundsException - if copying would cause access of data outside array

bounds.

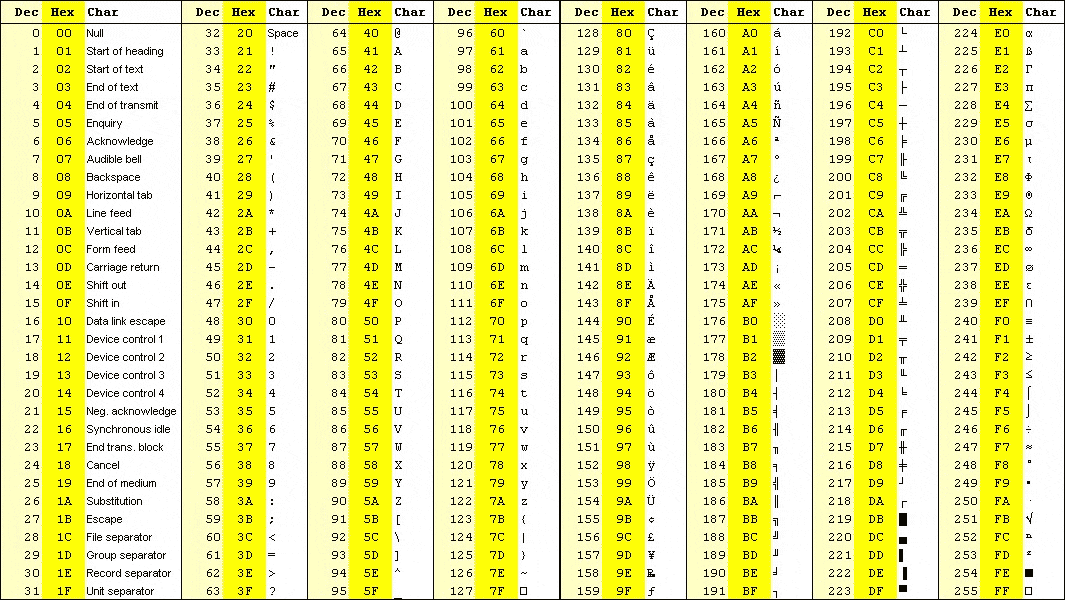
ArrayStoreException - if an element in the src array could not be stored into the dest

array because of a type mismatch.

NullPointerException - if either src or dest is null.

Note that if the src and dest arguments refer to the same array object, then the copying is performed as if the components at positions srcPos through srcPos+length-1 were first copied to a temporary array with length components and then the contents of the temporary array were copied into positions destPos through destPos+length-1 of the destination array.

ASCII table:



**Practice Questions**

