

THE JAVA LANGUAGE CHEAT SHEET

Primitive Types:

INTEGER: byte(8bit), short(16bit), int(32bit), long(64bit), **DECIM:** float(32bit), double(64bit), **OTHER:** boolean(1bit), char (Unicode)
HEX: 0x1AF, **BINARY:** 0b00101, **LONG:** 8888888888888L
CHAR EXAMPLES: 'a', '\n', '\t', '\'', '\'', '\'', '\'

Primitive Operators

Assignment Operator: = (ex: int a=5,b=3;)
Binary Operators (two arguments): + - * / %
Unary Operators: + - ++ --
Boolean Not Operator (Unary): !
Boolean Binary: == != > >= < <=
Boolean Binary Only: && ||
Bitwise Operators: ~ & ^ | << >> >>>
Ternary Operator: bool?valtrue:valfalse;

Casting, Conversion

int x = (int)5.5; //works for numeric types
int x = Integer.parseInt("123");
float y = Float.parseFloat("1.5");
int x = Integer.parseInt("7A",16); //fromHex
String hex = Integer.toString(99,16); //toHex
//Previous lines work w/ binary, other bases

java.util.Scanner, input, output

Scanner sc = new Scanner(System.in);
int i = sc.nextInt(); //stops at whitespace
String line = sc.nextLine(); //whole line
System.out.println("bla"); //stdout
System.err.print("bla"); //stderr, no newline

java.lang.Number types

Integer x = 5; double y = x.doubleValue();
double y = (double)x.intValue();
//Many other methods for Long, Double, etc

java.lang.String Methods

//Operator +, e.g. "fat"+"cat" -> "fatcat"
boolean equals(String other);
int length();
char charAt(int i);
String substring(int i, int j); //j not incl
boolean contains(String sub);
boolean startsWith(String pre);
boolean endsWith(String post);
int indexOf(String p); //-1 if not found
int indexOf(String p, int i); //start at i
int compareTo(String t);
// "a".compareTo("b") -> -1
String replaceAll(String str, String find);
String[] split(String delim);

StringBuffer, StringBuilder

StringBuffer is synchronized String Builder
(Use StringBuilder unless multithreaded)
Use the .append(xyz) methods to concat
toString() converts back to String

java.lang.Math

Math.abs(NUM), Math.ceil(NUM), Math.floor(NUM),
Math.log(NUM), Math.max(A,B), Math.min(C,D),
Math.pow(A,B), Math.round(A), Math.random()

IF STATEMENTS:

```
if( boolean_value ) { STATEMENTS }  
else if( bool ) { STATEMENTS }  
else if( ..etc ) { STATEMENTS }  
else { STATEMENTS }  
//curly brackets optional if one line
```

LOOPS:

```
while( bool ) { STATEMENTS }  
for(INIT;BOOL;UPDATE) { STATEMENTS }  
//1INIT 2BOOL 3STATEMENTS 4UPDATE 5->Step2  
do{ STATEMENTS }while( bool );  
//do loops run at least once before checking  
break; //ends enclosing loop (exit loop)  
continue; //jumps to bottom of loop
```

ARRAYS:

```
int[] x = new int[10]; //ten zeros  
int[][] x = new int[5][5]; //5 by 5 matrix  
int[] x = {1,2,3,4};  
x.length; //int expression length of array  
int[][] x = {{1,2},{3,4,5}}; //ragged array  
String[] y = new String[10]; //10 nulls  
//Note that object types are null by default
```

//loop through array:

```
for(int i=0;i<arrayname.length;i++) {  
    //use arrayname[i];  
}
```

//for-each loop through array

```
int[] x = {10,20,30,40};  
for(int v : x) {  
    //v cycles between 10,20,30,40  
}
```

//Loop through ragged arrays:

```
for(int i=0;i<x.length;i++)  
    for(int j=0;j<x[i].length;j++) {  
        //CODE HERE  
    }
```

//Note, multi-dim arrays can have nulls
//in many places, especially object arrays:
Integer[][] x = {{1,2},{3,null},null};

FUNCTIONS / METHODS:

Static Declarations:

```
public static int functionname( ... )  
private static double functionname( ... )  
static void functionname( ... )
```

Instance Declarations:

```
public void functionname( ... )  
private int functionname( ... )
```

Arguments, Return Statement:

```
int myfunc(int arg0, String arg1) {  
    return 5; //type matches int myfunc  
}
```

//Non-void methods must return before ending
//Recursive functions should have an if
//statement base-case that returns at once

CLASS/OBJECT TYPES:

INSTANTIATION:

```
public class Ball { //only 1 public per file  
    //STATIC FIELDS/METHODS  
    private static int numBalls = 0;  
    public static int getNumBalls() {  
        return numBalls;  
    }  
    public static final int BALLRADIUS = 5;
```

//INSTANCE FIELDS

```
private int x, y, vx, vy;  
public boolean randomPos = false;
```

//CONSTRUCTORS

```
public Ball(int x, int y, int vx, int vy) {  
    this.x = x;  
    this.y = y;  
    this.vx = vx;  
    this.vy = vy;  
    numBalls++;  
}  
Ball() {  
    x = Math.random()*100;  
    y = Math.random()*200;  
    randomPos = true;  
}
```

//INSTANCE METHODS

```
public int getX(){ return x; }  
public int getY(){ return y; }  
public int getVX(){ return vx; }  
public int getVY(){ return vy; }  
public void move(){ x+=vx; y+=vy; }  
public boolean touching(Ball other) {  
    float dx = x-other.x;  
    float dy = y-other.y;  
    float rr = BALLRADIUS;  
    return Math.sqrt(dx*dx+dy*dy)<rr;  
}
```

//Example Usage:

```
public static void main(String[] args) {  
    Ball x = new Ball(5,10,2,2);  
    Ball y = new Ball();  
    List<Ball> balls = new ArrayList<Ball>();  
    balls.add(x); balls.add(y);  
    for(Ball b : balls) {  
        for(Ball o : balls) {  
            if(b != o) { //compares references  
                boolean touch = b.touching(o);  
            }  
        }  
    }  
}
```

POLYMORPHISM:

Single Inheritance with "extends"

```
class A{ }
class B extends A{ }
abstract class C { }
class D extends C { }
class E extends D
```

Abstract methods

```
abstract class F {
    abstract int bla();
}
class G extends F {
    int bla() { //required method
        return 5;
    }
}
```

Multiple Inheritance of interfaces with "implements" (fields not inherited)

```
interface H {
    void methodA();
    boolean methodB(int arg);
}
interface I extends H{
    void methodC();
}
interface K {}
class J extends F implements I, K {
    int bla() { return 5; } //required from F
    void methodA(){} //required from H
    boolean methodB(int a) { //req from A
        return 1;
    }
    void methodC(){} //required from I
}
```

Type inference:

```
A x = new B(); //OK
B y = new A(); //Not OK
C z = new C(); //Cannot instantiate abstract
//Method calls care about right hand type
(the instantiated object)
//Compiler checks depend on left hand type
```

GENERICS:

```
class MyClass<T> {
    T value;
    T getValue() { return value; }
}
class ExampleTwo<A,B> {
    A x;
    B y;
}
class ExampleThree<A extends List<B>,B> {
    A list;
    B head;
}
//Note the extends keyword here applies as
well to interfaces, so A can be an interface
that extends List<B>
```

JAVA COLLECTIONS:

List<T>: Similar to arrays

ArrayList<T>: Slow insert into middle
//ArrayList has fast random access
LinkedList<T>: slow random access
//LinkedList fast as queue/stack
Stack: Removes and adds from end

List Usage:

```
boolean add(T e);
void clear(); //empties
boolean contains(Object o);
T get(int index);
T remove(int index);
boolean remove(Object o);
//remove uses comparator
T set(int index, E val);
int size();
```

List Traversal:

```
for(int i=0;i<x.size();i++) {
    //use x.get(i);
}

//Assuming List<T>:
for(T e : x) {
    //use e
}
```

Queue<T>: Remove end, Insert beginning
LinkedList implements Queue

Queue Usage:

```
T element(); // does not remove
boolean offer(T o); //adds
T peek(); //pike element
T poll(); //removes
T remove(); //like poll
Traversal: for(T e : x) {}
```

Set<T>: uses Comparable<T> for uniqueness
TreeSet<T>, items are sorted
HashSet<T>, not sorted, no order
LinkedHashSet<T>, ordered by insert
Usage like list: add, remove, size
Traversal: for(T e : x) {}

Map<K,V>: Pairs where keys are unique
HashMap<K,V>, no order
LinkedHashMap<K,V> ordered by insert
TreeMap<K,V> sorted by keys

```
V get(K key);
Set<K> keySet(); //set of keys
V put(K key, V value);
V remove(K key);
int size();
Collection<V> values(); //all values

Traversal: for-each w/ keyset/values
```

java.util.PriorityQueue<T>

A queue that is always automatically sorted
using the comparable function of an object

```
public static void main(String[] args) {
    Comparator<String> cmp= new LenCmp();
    PriorityQueue<String> queue =
        new PriorityQueue<String>(10, cmp);
    queue.add("short");
    queue.add("very long indeed");
    queue.add("medium");
    while (queue.size() != 0)
        System.out.println(queue.remove());
}

class LenCmp implements Comparator<String> {
    public int compare(String x, String y){
        return x.length() - y.length();
    }
}
```

java.util.Collections algorithms

Sort Example:

```
//Assuming List<T> x
Collections.sort(x); //sorts with comparator
```

Sort Using Comparator:

```
Collections.sort(x, new Comparator<T>{
    public int compareTo(T a, T b) {
        //calculate which is first
        //return -1, 0, or 1 for order:
        return someint;
    }
})
```

Example of two dimensional array sort:

```
public static void main(final String[] a){
    final String[][] data = new String[][] {
        new String[] { "20090725", "A" },
        new String[] { "20090726", "B" },
        new String[] { "20090727", "C" },
        new String[] { "20090728", "D" } };
    Arrays.sort(data,
        new Comparator<String[][]>() {
            public int compare(final String[]
                entry1, final String[] entry2) {
                final String time1 = entry1[0];
                final String time2 = entry2[0];
                return time1.compareTo(time2);
            }
        });

    for (final String[] s : data) {
        System.out.println(s[0]+" "+s[1]);
    }
}
```

More collections static methods:

```
Collections.max( ... ); //returns maximum
Collections.min( ... ); //returns maximum
Collections.copy( A, B); //A list into B
Collections.reverse( A ); //if A is list
```