JavaScript: Basic, Advanced, & More Cheat Sheet

by AC Winter (acwinter) via cheatography.com/21349/cs/4084/

About This Document

the purpose of this cheat sheet is to briefly describe the core elements of the JavaScript language for those of studying it who have taken in much more than we can hold onto well. nothing here is explained *in full* but rather meant to get you on the right track. also, this document purposely does not cover browser-specific methods / syntax / objects and the like.

this cheat sheet is a work in progress and may be updated -- check back on occasion!

Types

Type	typeOf evaluation	Primitive?
Null	object	yes
Undefined	undefined	yes
Boolean	boolean	yes
String	string	yes
Number	number	yes
Object	object	no --*an object*
Function	function	no --*an object*
Array	object	no --*an object*
Symbol	symbol	no --*an object*
[]	object	no --*an object*
{}	object	no --*an object*

Number & Math Methods

» `someNum.toFixed(num)`

• shortens someNum to have only num decimal places

» `num.toExponential()`

• converts num to exponential notation (i.e. 5.569e+0)

» `num.toString()`

• converts num to a string

» `num.toPrecision(#)`

• converts num to a num with # places starting with whole numbers

» `String(someValue)`

• converts or coerces someValue to a string - someValue can be any type, ie "Boolean(1)" returns true

» `parseInt(string, radix)`

» `parseFloat(string, radix)`

• converts a string into an integer. the optional radix argument defines the base -- i.e., base 10 (decimal) or base 16 (hexadecimal).

» `Math.round(num)`

• rounds num to nearest integer

Number & Math Methods (cont)

» `Math.ceil(num)`

• rounds num *up* to nearest integer

» `Math.floor(num)`

• rounds num *down* to nearest integer

» `Math.max(num1, num2)`

• returns larger num

» `Math.min(num1, num2)`

» `Math.pow(num1, num2)`

• returns num1 to the power num2

» `Math.sqrt(num)`

» `Math.random()`

• returns decimal between 0 (inclusive) and 1(exclusive)

» `Math.abs(num)`

• returns absolute value of num•

Array "Extra Methods"

Note: these "extra methods," which are "higher-order" functions, ignore holes in the array (i.e.: [“apples”, , , , “oranges”]). they also have more arguments than shown here -- best to look them up for more info!

Note: array-*like* objects, for example `arguments` and `NodeLists`, can also make use of these methods.

» `arr.some(callback)`

» `arr.every(callback)`

• returns a boolean value. returns true if*some* or *every* element in the array meets the evaluation. example:

```
var a = [1,2,3];
var b = a.every(function(item){
  return item > 1;
}); // false
```

» `arr.reduce(function(prev, next){..}, startVal)`

» `arr.reduceRight(function(prev, next){..}, startVal)`

• returns a value. *reduce* employs a callback to run through the elements of the array, returning "prev" to itself with each iteration and taking the next "next" value in the array. for it's first "prev" value it will take an optional "startVal" if supplied. an interesting example:

```
var arr = ["apple", "pear", "apple", "lemon"];
var c = arr.reduce(function(prev, next) {
 prev[next] = (prev[next] += 1) || 1;
 return prev;
```

Array "Extra Methods" (cont)

```
 }, {});
 // objCount = { apple: 2, pear: 1, lemon: 1 }
```

» `arr.filter(function(){..})`

• returns an array. *filter* returns an array of elements that satisfy a given callback. example:

```
 var arr2 = ["jim", "nancy", "ned"];
 var letter3 = arr2.filter(function(item) {
  return (item.length === 3);
 });
 console.log(letter3); // ['jim', 'ned']
```

» `arr.sort(function(){..})`

• returns *the original* array, mutated. *sort* returns the elements sorted with a given criteria. for example:

```
 var stock = [{key: “r”, num: 12}, {key: “a”, num:
2}, {key: “c”, num: 5}];
 var c = stock.sort(function(a,b) {
  return a.num - b.num;
 } ); // [ { key: ‘a', num: 2 }, { key: ‘c', num: 5
}, { key: ‘r', num: 12 } ]
```

» `arr.map()`

• returns an array. *map* goes over every element in the array, calls a callback on the element, and sets an element in the new array to be equal to the return value the callback. for example:

```
 var stock = [{key: "red", num: 12}, {key: "blue",
num: 2}, {key: "black", num: 2}];
 var b = stock.map(function (item){
  return item.key;
 }) // ["red","blue","black"]
```

» `arr.forEach()`

• no return value. *forEach* performs an operation on all elements of the array. for example:

```
 var arr = [“jim”, “mary”];
 a.forEach (function (item) {
   console.log("I simply love “ +item);
 }); // “I simply love jim”, “I simply love mary"
```

Note: you can combine array methods in a*chain* where the result of the leftmost operation is passed to the right as such:

`array.sort().reverse()...`

Functions & Etc.

Callbacks: placing ( ) after a function call *executes it immediately*. leaving these off allows for a callback.

Function Declaration

» `function aFunctionName (args) {...`

• functions created in this manner are evaluated when the code is parsed and are 'hoisted' to the top and are available to the code *even before* they're formally declared. Note: Due to JS's odd construction, using function declarations within a flow control statement can be wonky and is best avoided.

Function Expression / Anonymous Functions

» `var bar = function (args) {...`

• (also referred to as 'Function Operators') anonymous functions are evaluated at 'runtime' and are therefore less memory intensive. they must be provided a variable name but need not have a function name (therefore: anonymous). [these are]

Named Function Expression

» `var bar = function foo (args) {...`

• confusingly, this is still an 'anonymous function.' assigning a name is useful for debugging purposes and also allows for self-referential / recursive calls

Function Constructor

» `var anotherFunction = new Function (args, function () {... }) {...}`

• equivalent to a functional expression

Self-Invoking Anonymous Functions

» `( function (args) { doSomething; } ) ( );`

• (also known as IIFEs / 'Immediately Invoked Function Expressions') and invokes immediately

Loops / Control Flow Statements

if .. else if .. else

```
 if (considtion1) {
   doSomething;
 } else if {
   doSomethingElse;
 } else {
   doSomethingMore;
 }
```

for loop

```
 for (var i = 0; i < someNumber; i++) {
  doSomething;
 }
```

switch loop

Loops / Control Flow Statements (cont)

```
switch (someEvaluation) {
  case "evaluatesAsThis" :
    doSomething;
  case "evaluatesAsThat" :
    doSomethingElse;
}
```

while loop

```
while (someEvaluation === true) {
  doSomething;
}
```

do .. while

```
do {
  doSomething;
}
while (someEvaluation === true);
```

for .. in (objects)

```
for (anItem in anObject) {
   doSomething With anItem;
    // will be the key
   doSomethingWith Object[anItem];
    // will be the value of that key
}
```

"this"

coming soon

String Methods, Properties & Etc

a string *can be coerced into an array* so many array methods are applicable as well

» `str.charAt(num)`
• returns the character in str at index num
» `str.charCodeAt(num)`
• returns the unicode value of the char
String.fromCharCode(num)`
• returns the character with unicode's num
» `str.indexOf(char)`
• returns -1 if char not found in str
» `str.lastIndexOf(subString)`

String Methods, Properties & Etc (cont)

• returns the index of the last occurrence of subString
» `str.length`
• returns length of str starting at 1
» `str.match(pattern)`
• returns null if not found. returns an*array* of all matches
» `str.match(/pattern/g)`
• provides global search of string
» `str.replace(old, new)`
» `str.search(pattern)`
• returns index of first match or -1 if not found
» `str.substring(index1, index2)`
• char at index1 is returned, index2 is not
» `str.split(char)`
• returns an array of str split on char
» `str.substr(index1, num)`
• returns substring starting at index1 and running num letters
» `str.toLowerCase()`
» `str.toUpperCase()`
» `str.toLocaleLowerCase()`
• takes local language settings into account
» `str.toLocaleUpperCase()`
• ibid
» `Number(var/string/object)`
• converts to number. "true" converts to 1, etc
» `one.concat(two)`
• concatenates string/array one with two
» `JSON.stringify( )`
• converts a javascript value/object into a string
» `JSON.parse ( )`
• converts a JSON string into a javascript object

Date Methods

Note: Unix epoch is January 1, 1970
» `var today = new Date();`
• creates date object for now
» `var someDate = new Date("june 30, 2035");`
• creates date object for arbitrary date
» `var today = Date.now();`

Date Methods (cont)

• returns number of milliseconds since epoch

» `parse()`

• returns milliseconds between date and Unix epoch.

» `toDateString()`

» `toTimeString()`

» `toLocalTimeString()`

Get / Set Date Methods

• getDate()	• getHours()
• getDay()	• getMilliseconds()
• getFullYear()	• getMinutes()
• getMonth()	• getSeconds()
• getTime()	• getTimezoneOffset()

Note: there are also 'set' methods such as *setMonth()*.

Note: getDay and getMonth return numeric representations starting with 0.

Miscellaneous Instructions

» `break;`

• breaks out of the current loop

» `continue;`

• stops current loop iteration and increments to next

» `isNaN(someVar)`

• returns true if not a number

» `isFinite(someVar)`

» `var aVar = anObject[anAttribute] || "nonesuch";`

• assigns a default value if none exists

» `var aVar = anEvaluation ? trueVal : falseVal;`

• ternary operator. assigns trueVal to aVar if anEvaluation is true, falseVal if not

» `delete anObject[anAttribute]`

» `(aProperty in anObject)`

• returns true or false if aProperty is a property of anObject

» `eval(someString)`

• evaluates a someString as if it was JavaScript. i.e. eval("var x = 2+3") returns 5

Array Methods (basic)

Note: index numbers for arrays start at 0

» `arr.length()`

» `arr. push(val)`

• adds val to end of arr

» `arr. pop()`

• deletes last item in arr

» `arr. shift()`

• deletes first item in arr

» `arr.unshift(val)`

• adds val to front of arr

» `arr.reverse ()`

» `arr1.concat(arr2)`

• concatenates arr1 with arr2

» `arr.join(char)`

• returns string of elements of arr joined by char

» `arr.slice(index1, index2)`

• returns a new array from arr from index1 (inclusive) to index2 (exclusive)

» `arr.splice(index, num, itemA, itemB,..)`

• alters arr. starting at index and through index+num, overwrites/adds itemsA..

Definitions & Lingo

Higher Order Functions

functions that accept *other functions* as an argument

Scope

the set of variables, objects, and functions available within a certain block of code

Callback

(also *event handler*) a reference to executable code, or a piece of executable code, that is passed as an argument to other code.

the % operator

% returns the remainder of a division such that "3 % 2 = 1" as 2 goes into 3 once leaving 1. called the "remainder" or "modulo" operator.

Composition

the ability to assemble complex behaviour by aggregating simpler behavior. *chaining* methods via dot syntax is one example.

Definitions & Lingo (cont)

Chaining

also known as *cascading*, refers to repeatedly calling one method after another on an object, in one continuous line of code.

Naming Collisions

where two or more identifiers in a given namespace or a given scope cannot be unambiguously resolved

DRY

Don't Repeat Yourself

ECMAScript

(also *ECMA-262*) the specification from which the JavaScript implementation is derived. version 5.1 is the current release.

Arity

refers to the number of arguments an operator takes. ex: a binary function takes two arguments

Currying

refers to the process of transforming a function with multiple arity into the same function with less *arity*

Recursion

an approach in which a function calls itself

Predicate

a calculation or other operation that would evaluate either to "true" or “false.”

Asynchronous

program flow that allows the code following an*asynchronous* statement to be executed immediately without waiting for it to complete first.

Callback Hell

code thickly nested with callbacks within callback within callbacks.

Closure

a function with access to the global scope, it's parent scope (if there is one), and it's own scope. a closure may retain those scopes even after it's parent function has *returned.*

IIFE

Immediately Invoked Function Expressions. *pronounced "iffy."* a function that is invoked immediately upon creation. employs a unique syntax.

Definitions & Lingo (cont)

Method

an object property has a function for its value.

Reserved Words

abstract	arguments	boolean	break
byte	case	catch	char
class	const	continue	debugger
default	delete	do	double
else	enum	eval	export
extends	false	final	finally
float	for	function	goto
if	implements	import	in
instanceof	int	interface	let
long	native	new	null
package	private	protected	public
return	short	static	super
switch	synchronized	this	throw
throws	transient	true	try
typeof	var	void	volatile
while	with	yield	

Prototype-based Inheritance

coming soon

Core Java Cheat Sheet

by evanescesn09 via cheatography.com/88543/cs/20288/

OOP

Encapsulation	The process of binding related classes, objects and operations together is called Encapsulation	Using access modifiers, packages
Abstraction	The process of specifying what to do without specifying how to do it	Using abstract classes and interfaces
Inheritance	When one class inherits the properties of another class	Using Aggregation, Composition
Polymorphism	Same thing is done in different ways	Using compile-time and run-time polymorphism

Encapsulation

default	accessible to classes only in the same package
public	accessible to all classes in any package
private	accessible to only a specific method, or class
protected	accessible to classes in the same package, and sub-classes of this class

Abstraction

Abstract Class	When a class has one or more unimplemented methods, it should be declared abstract. The sub-classes should provide implementation for the unimplemented methods, else they should also be declared abstract	Used: When default implementation is needed for some methods, and specific implementations for other methods based on the class implementing them

Abstraction (cont)

Interface	Blueprint of a class. It contains only static, final variables and only unimplemented methods. Classes implement the interface should implement all methods of the interface, or be declared abstract	Used: to support multiple inheritance

Abstract classes don't support multiple inheritance, whereas interfaces do

Inheritance

Aggregation	When one class contains a reference of another class	Loosely coupled classes
Association	When one class is made up of another class	Tightly coupled classes

Java does't support multiple inheritance directly, it supports it only via Interfaces

Polymorphism

Compile-time	Also called overloading. When methods have same name but different signature (return-type, number of parameters, type of parameters etc)
Run-time	Also called overriding. When child-classes over-write method implementations of parent-class.

static keyword

static field	Shared by all members of the class. It can be accessed before objects are created
static method	Can be accessed without creating an instance of the class. They can only access static variables and static methods. Cannot access this or super

static keyword (cont)

static block	Used when some computation is to be done to initialize the static variables. This block is executed once when the class is initially loaded into memory
static class	We cannot declare top-level classes as static. Only inner classes can be static. A static class cannot access non-static members of the Outer class. It can access only static members of Outer class

final

fields	treated as constants
methods	cannot be overridden by child classes
classes	cannot be inherited

finalize( )

finalize() method is a protected and non-static method of java.lang.Object class. This method will be available in all objects you create in java. This method is used to perform some final operations or clean up operations on an object before it is removed from the memory

String Creation

Literal : String s = " "	Creates Strings in String pool, in JVM. Multiple strings can have same value. Only one copy of the word exists in the String pool, and the references of it are updated.
Object: String s = new String( );	Creates a string object in heap. The heap in-turn checks the JVM String Pool to see if there exists a string with same value.

```
String s1 = "abc";
String s2 = "abc";
s1 == s2 returns true;
======================
String s1 = new String("abc");
String s2 = new String("abc");
s1 == s2 returns false;
But s1.equals(s2) returns true;
```

String Immutability

Strings in java are immutable because changing the value of a String literal changes the value of other Strings that reference the literal, which leads to inconsistency in the program. To prevent this, strings in java are immutable.

Storing passwords in Strings

It is best to store passwords as char[ ] because if passwords are stored as Strings, the string tends to be in the JVM pool even after all references to it no longer exist. This causes a vulnerability in the system. In case of Char[ ], once all the references to char[ ] are gone, the Java Garbage Collector deletes the char[ ] to preserve memory. So, it's safer.

StringBuilder, StringBuffer

StringBuilder	To create mutable strings in Java
StringBuffer	To create thread-safe mutable strings in Java

String methods

```
s.charAt(int index)
s.compareTo(s2), s.compareToIgnoreCase(s2)
s.concat(s2)
s.contains(sequence of characters)
s.equals(s2), s.equalsIgnoreCase(s2)
s.length()
s.replace(character, replacement) )
s.replaceAll(character, replacement)
s.subString(int startIndex)
s.subString(int startIndex, int endIndex)
s.toUpperCase( ), s.toLowerCase( )
s.toCharArray()
s.trim( )
String s = String.valueOf(int, or long or double)
String[] s1 = s.split( String regex)
String[] s1 = s.split(String regex, int limit )
```

Core Java Cheat Sheet

by evanescesn09 via cheatography.com/88543/cs/20288/

StringBuffer, StribgBuilder methods
s.append(s2)
s.deleteCharAt(int index)
s.indexOf(string), s.indexOf(string, fromIndex)
s.insert(int index, objectValue)
s.replace(int startIndex, int endIndex, String)
s.reverse( )
s.toString( )
s.trimToSize( )
s.setCharAt(int index, charSequence)

Sorting Cheat Sheet

by evanescesn09 via cheatography.com/88543/cs/20303/

Sorting

Sorting is rearranging the given data in a specific format- ascending or descending. The purpose of sorting elements is to greatly improve the efficiency of searching while doing computations on the data.
In Java, we can sort primitive data (using sorting algorithms) , or user-defined data (using Comparable or Comparator interfaces)

We have 2 main kinds of sorting:
1. Internal Sorting - done in main memory
2. External Sorting - algorithms that use external memory like tape, or a disk for sorting.
External sorting is used when data is too large to fit into main memory.

Bubble Sort

Technique	Brute Force
How?	Each element is compared with every other element, and when they are not in correct order, they are swapped
Time Complexity	n^2
Space Complexity	O(1) - because it is done in place

Merge Sort

Technique	Divide and Conquer
Best Used	when merging 2 or more already sorted input lists
How?	Dividing the input data into half recursively till each input is of size=1. Then merging the sorted data into one

Merge Sort (cont)

Time Complexity	O(nlogn) - logn to sort half the data in each recursion, n to merge all the input data to give the final sorted output
Space Complexity	O(n) extra space required when merging back the sorted data

Merge Sort does not preserve ordering or elements of the same value

Insertion Sort

Technique	
Best used	for small data
Advantages	Preserves insertion order of elements of same values
How?	Removes an element from the input list and insert into the correct position of the already sorted list.
Time Complexity	O(n^2)
Space Complexity	O(1) - because it is done in place

Quick Sort

Technique	Divide and Conquer
How?	Select a pivot element. Split the array into 2 parts - elements less than pivot and elements > pivot. Recursively repeat this process till all the elements are sorted.
Time Complexity	O(n logn)
Space Complexity	O(1) - it is done in place

Selection Sort

Technique	
Best Used	only for small set of data, it doesn't scale well for larger data sets
How?	Find min value in the list, swap it with element at current index. Repeat the process till the list is sorted.
Time Complexity	O(n^2)
Space Complexity	O(1) - because it is done in place

Heap Sort

Technique	Divide and Conquer
Best Used	Priority Queues
How?	Insert all elements into a heap (minHeap or MaxHeap). Then remove elements from the heap till the heap is empty.
Heap	The main property of a heap is that it always contains the min/max element at the root. As elements are inserted and deleted, the heap makes use of the "heapify" property to ensure that the min/max element is always at the root. So, always the min/max element is returned when the heap is dequeued
Time Complexity	O(nlogn)
Space	O(n)

Design Patterns: Observer, Interpreter, Memento Cheat Sheet

by ppesq via cheatography.com/20490/cs/3257/

OBSERVER

one-to-many dependency between subject and observers, so that when subject changes, the observers are notified and updated.

a way of notifying change to a number of classes

Questions

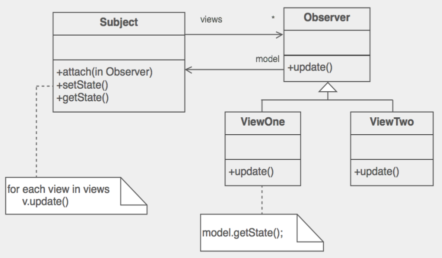
What is it?

> many objects need to be notified of changes in the state of one object

Where have I seen it before?

> RSS feeds or any pub/sub system you might have used/coded

Ok, this is cool. What do I need to implement it?



1. A Subject Abstract Class and an Observer Abstract Class
2. Concrete subject and observer class that implement above pattern.
The concrete subject registers its observers

Example

A **referee** (*concrete subject*) notifies all the **players** and **commentators** (*concrete observers*) about changes in the state of a Soccer match. Each **player** must be notifiable.

MEMENTO

provides the ability to restore an object to its previous state

a memento is like a magic cookie that encapsulates a checkpoint capability

Questions

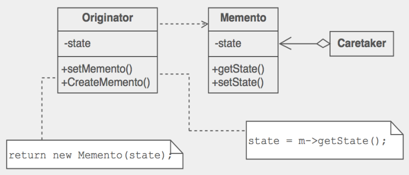
What problem does it solve?

> you want to save the state of an object so that you can restore it later.

Where have I seen it before?

> Git or any version control system for that matter. A memento makes rollbacks possible.

Ok, this is cool. What do I need to implement it?



1. An originator class (class that has a state that needs to be remembered)
2. A caretaker class (class that wants to modify the state of the originator)
3. A memento class that holds originator information that can't be modified by any other class. It is merely a container.

Example

A **programmer** (caretaker) asks for a copy (memento) of the **code** (originator) he/she is modifying. Later he/she decides he doesn't like the new state of the **code** so he restores it with the copy it still has.

INTERPRETER

Represent the grammar of a language with a hierarchical object-oriented design.

The language is usually a domain specific language.

Questions

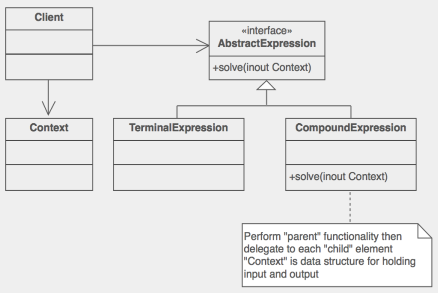
What problem does it solve?

> A language must be parsed and interpreted.

Where have I seen it before?

> Parsers

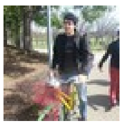
Ok, this is cool. What do I need to implement it?


Client
«interface»
AbstractExpression
+solve(inout Context)
Context
TerminalExpression
CompoundExpression
+solve(inout Context)
Perform "parent" functionality then delegate to each "child" element
"Context" is data structure for holding input and output

1. A Context class that contains the input.
2. An AbstractExpression class, a composite object of terminals and non-terminals.
3. The client passes the context to the abstract expression, which calls the interpret() function on its children.

Example

A **roman numeral** (context) is converted into decimal notation by the **parser** (Abstract Expression).
Derivation: LIV => 50 + IV => 50 + (-1 + 5) => 50 + 4 => 54



Advantages

Observer: minimal coupling; easy addition and removal of observers

Memento: it is an encapsulated copy so it avoids exposing its info; the storage burden is on the caretaker, not on originator

Interpreter: easy to change/extend/implement/evaluate a language

Disadvantages

Observer: Possible memory leak; Objects might need to work hard to deduce what changed in the subject.

Memento: Copy operation to a memento can be costly for the originator; Caretaker might have large storage costs.

Interpreter: Complex grammars are hard to maintain and debug.

Java Fundamentals Cheat Sheet

by sschaub via cheatography.com/1000/cs/464/

Java Data Types

byte / short / int / long	-123, 10
float / double	235.13
char	'U'
boolean	true, false
String	"Greetings from earth"

Java Statements

If Statement

```
if ( expression ) {
  statements
} else if ( expression ) {
  statements
} else {
  statements
}
```

While Loop

```
while ( expression ) {
  statements
}
```

Do-While Loop

```
do {
  statements
} while ( expression );
```

For Loop

```
for ( int i = 0; i < max; ++i) {
  statements
}
```

For Each Loop

```
for ( var : collection ) {
  statements
}
```

Switch Statement

Java Statements (cont)

```
switch ( expression ) {
  case value:
    statements
    break;
  case value2:
    statements
    break;
  default:
    statements
}
```

Exception Handling

```
try {
  statements;
} catch (ExceptionType e1) {
  statements;
} catch (Exception e2) {
  catch-all statements;
} finally {
  statements;
}
```

Java Data Conversions

String to Number

int i = Integer.parseInt(*str*);

double d = Double.parseDouble(*str*);

Any Type to String

String s = String.valueOf(*value*);

Numeric Conversions

int i = (int) *numeric expression*;

Java String Methods

s.length()	length of *s*
s.charAt(*i*)	extract *i*th character
s.substring(*start*, *end*)	substring from *start* to *end*-1
s.toUpperCase()	returns copy of *s* in ALL CAPS
s.toLowerCase()	returns copy of *s* in lowercase
s.indexOf(*x*)	index of first occurence of *x*
s.replace(*old*, *new*)	search and replace
s.split(*regex*)	splits string into tokens
s.trim()	trims surrounding whitespace
s.equals(*s2*)	true if s equals s2
s.compareTo(*s2*)	0 if equal/+ if s > s2/- if s < s2

See http://docs.oracle.com/javase/6/docs/api/java/lang String.html for more.

java.util.ArrayList Methods

l.add(*itm*)	Add *itm* to list
l.get(*i*)	Return *i*th item
l.size()	Return number of items
l.remove(*i*)	Remove *i*th item
l.set(*i*, *val*)	Put *val* at position *i*

```
ArrayList<String> names =
  new ArrayList<String>();
```

See http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html for more.

Java Fundamentals Cheat Sheet

by sschaub via cheatography.com/1000/cs/464/

java.util.HashMap Methods

m.put(*key*,*value*)	Inserts *value* with *key*
m.get(*key*)	Retrieves value with *key*
m.containsKey(*key*)	true if contains *key*

```
HashMap<StÂrinÂg,String> names =
  new HashMap<StÂrinÂg, String>();
```

See http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html for more.

Java Hello World

```
import java.util.Date;
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello, world!");
    Date now = new Date();
    System.out.println("Time: " + now);
  }
}
```

* Save in Hello.java
* Compile: **javac Hello.java**
* Run: **java Hello**

Java Arithmetic Operators

x + y	add	x - y	subtract
x * y	multiply	x / y	divide
x % y	modulus	++x / x++	increment
		--x / x--	decrement

Assignment shortcuts: x *op*= y
Example: x += 1 increments x

Java Comparison Operators

x < y	Less	x <= y	Less or eq
x > y	Greater	x >= y	Greater or eq
x == y	Equal	x != y	Not equal

Java Boolean Operators

! x (not)	x && y (and)	x || y (or)

Java Text Formatting

printf style formatting

```
System.out.printf("Count is %d\n", count);
s = String.format("Count is %d", count);
```

MessageFormat style formatting

```
s = MessageFormat.format(
  "At {1,time}, {0} eggs hatched.",
  25, new Date());
```

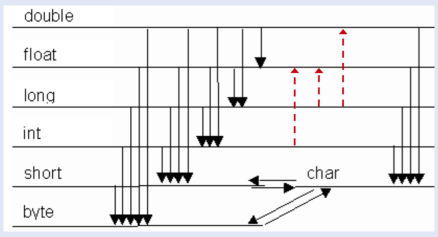
Individual Numbers and Dates

```
s = NumberFormat.getCurrencyInstance()
  .format(x);
s = new SimpleDateFormat(""h:mm a"")
  .format(new Date());
s = new DecimalFormat("#,##0.00")
  .format(125.32);
```

See http://docs.oracle.com/javase/6/docs/api/java/text/package-frame.html for MessageFormat and related classes

OO_Java Cheat Sheet
by tarinya via cheatography.com/25055/cs/6436/

Typ Konversion



schwarze Pfeile: expliziet
rote Pfeile: impliziet, evt Genauigkeitsverlust
alles andere impliziet

Switch-Statement

```
switch (Ausdruck) {
case Wert1:
   Anweisungen;
   break;
case Wert2:
   Anweisungen;
   break;
default:
   Anweisungen;
}
```

Wenn Ausdruck Wert1 entspricht, wird Anweisung 1 ausgeführt...

Typ Polymorphismus

```
car extends vehicle
@Override
drive(){}
Vehicle v = new Car();
//calls drive of Car not Vehicle
v.drive();
```

Klasse hat eigenen Typ plus alle Typen der Superklassen.
verschiedene "Brillen"
Dynamic Dispatch = dynamischer Typ zur Laufzeit entscheidet über aufgerufene Methoden.

Rekursion

```
public boolean groupSum(int start, int[] nums, int target) {
  if (start >= nums.length) return (target == 0);
  if (groupSum(start + 1, nums, target - nums[start])) return true;
  if (groupSum(start + 1, nums, target)) return true;
  return false;}
Collection<String> validParentheses(int nofPairs){
  Collection<String> list = new ArrayList<>();
  if (nofPairs == 0) {list.add("");} else {
    for (int k = 0; k < nofPairs; k++) {
      Collection<String> infixes = validParentheses(k);
      Collection<String> suffixes = validParentheses(nofPairs - k - 1);
      for (String infix : infixes) {
        for (String suffix : suffixes) {
          list.add("(" + infix + ")" + suffix);}}}}
  return list;}
```

Generics

allg. Form

class Pair<T,U>

TypeBound (Bedingungen für Type)

class Node<T extends Comparable<T>>
Node<Person>//ok Node<Student>//Error

Multiple TypeBounds (mit & anhängen)

class Node<T extends Person & Comparable<Person>>

generische Methode

Generics (cont)

public <T extends Person> T set (T element) {...}
<T>: Bedingungen für Input
T : Rückgabetyp

Wildcard-Typ (Nutzen: Typ**argument** nicht relevant)

Node<?> undef --> kein Lesen & schreiben ausser: Object o = undef.getValue();

Lambdas

```
(p1, p2) -> p1.getAge() - p2.getAge()
```

```
p -> p.getAge() >= 18
```

```
people.sort((p1, p2) -> p1.getName().compareTo(p2.getName()));
```

```
people.sort(Comparator.comparing(Person::getLastName).thenComparing(Person::getFirstName));
```

```
Utils.removeAll(people, person -> person.getAge() < 18);
```

```
people.sort(Comparator.comparing(p1 -> p1.getLastName().length()));
```

Syntax: (Parameter) -> {Body}

StreamAPI

```
people
.stream()
.filter(p -> p.getAge() >= 18)
.map(p -> p.getLastName())
.sorted()
.forEach(System.out::println);
people.stream().mapToInt(p -> p.getAge())
.average().ifPresent(System.out::println);
Map<String, Integer> totalAgeByCity =
```

OO_Java Cheat Sheet

by tarinya via cheatography.com/25055/cs/6436/

StreamAPI (cont)

```
  people.stream()
  .collect(
      Collectors.groupingBy(Person:
:getCity,
      Collectors.summingInt(Person:
:getAge)));
```

endliche Quelle: `IntStream.range(1, 100)`

unendl. Quelle: `Random random = new Random();`

`Stream.generate(random::nextInt).limit(100)`

FileReader/Writer

```
private static void reverteText()
throws FileNotFoundException,
IOException{
  try (FileReader reader = new
FileReader("input.txt");
      FileWriter writer = new
FileWriter("outpur.txt")){
       int value = reader.read();
       String text = "";
       while (value >=0){
          text = (char)value + text;
          value = reader.read();}
       writer.write(text);}}
```

Sichtbarkeit

public	alle Klassen
protected	Package und Sub-Klassen
private	nur innerhalb Klasse
(keines)	innerhalb Package

Datentypen

byte	8 bit (2^7 bis 2^7-1)
short	16 bit
int	32 bit
long	64 bit (1L)
float	32 bit (0.1f)
double	64 bit

Operator-Prio

+,-,++,--,! (unär)

*, /, %

+, - (binär)

<, <=, >, >=

==, !=

&&

||

Rundungsfehler

0.1+0.1+0.1 != 0.3

Problem: double/float	`x == y`
Lösung	`Math.abs(x - y) < 1e-6`

Integer Literal

binär	0b10 = 2
oktal	010 = 8
hex	0x10 = 16
opt.	1000 = 1_000

Arithmetik

```
1 / 0 --> ArithmeticException: /
by zero
1 / 0.0 --> Infinity
-1.0 / 0 --> -Infinity
0 / 0.0 --> NaN
```

Arithmetik Overloading

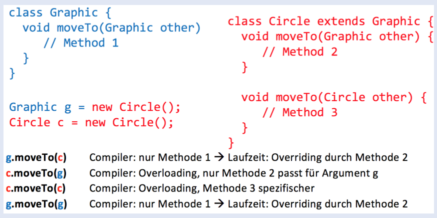
```
int operate(int x, int y) { ... }
double operate(int x, double y) {
... }
double operate(double x, double y)
{ ... }
```

operate(1, 2); --> meth 1

operate(1.0, 2); --> meth 3

operate(1, 2.0); --> meth 2

Overloading



```
class Graphic {
  void moveTo(Graphic other)
     // Method 1
  }
}

Graphic g = new Circle();
Circle c = new Circle();
```

```
class Circle extends Graphic {
  void moveTo(Graphic other) {
     // Method 2
  }

  void moveTo(Circle other) {
     // Method 3
  }
}
```

g.moveTo(c)	Compiler: nur Methode 1 → Laufzeit: Overriding durch Methode 2
c.moveTo(g)	Compiler: Overloading, nur Methode 2 passt für Argument g
c.moveTo(c)	Compiler: Overloading, Methode 3 spezifischer
g.moveTo(g)	Compiler: nur Methode 1 → Laufzeit: Overriding durch Methode 2

Bedingungsoperator

`max = left > right ? left : right;`

wenn left>right wird max = left, sonst max=right

Package prio

own class (inc. nested class)

single type imports -> import p2.A;

class in same package

import on demand -> import p2.*

Enum

```
public enum Color {
   BLUE(1), RED(2);
   private final int code;
   private Color(int code) {
      this.code = code;}
   public int getColorValue() {
      return code;}}
```

Methodenreferenz

```
people.sort(this::compareByAge)
int compareByAge(Person p1, Person
p2){return p1.age - p2.age;}
```

```
people.sort(Sorter::compareByAge)
static int compareByAge(Person p1,
Person p2){return p1.age - p2.age;}
```

```
Sorter sorter = new Sorter();
people.sort(sorter::compareByAge);
```

Serialisieren

```
OuputStream fos = new
FileOutputStream("serial.bin");
try (ObjectOutputStream stream =
new ObjectOutputStream(fos)){
   stream.writeObject(person);
}
```

needs Serializable interface
serialisiert alle direkt & inderekt referenz. Obj

Input/Output

```
try (BufferedInputStream
inputBuffer = new bis(new
FileInputStream(pathSource));
BufferedOutputStream outputB = new
bos(new
FileOutputStream(pathTarget))) {
   byte[] byteBuffer = new
byte[4096];
   int value =
inputBuffer.read(byteBuffer);
   while (value >= 0) {
      outputB.write(byteBuffer, 0,
value);
      value =
inputBuffer.read(byteBuffer);}}
catch (FileNotFoundException e) {
   System.out.println("File not
found: "+e);
} catch (IOException e) {
   System.out.println("reading
Error:"+e);}
```

String Pooling

```
String a = "OO", b = "Prog", c =
"OOProg";
//true
a == "OO"; c == "OO" + "Prog";
(a+b).equals(c);
//false
a + b == c;
```

String Literals gepoolt. True Fälle --> es werden keine neuen Objekte erstellt
Integer-Pooling -128 bis 127

Collections

Array	`int[] intArray = new int[3];`
List	`List<T> al = new ArrayList<>();`

Collections (cont)

Set	`Set<T> hs = new HashSet<>();`
Map	`Map<U,T> hm = new HashMap<>();`

```
Iterator<T> it = a.iterator();
while(it.hasNext()) {..}
```

`Collections.sort(Object o)`	needs Comparable on o

```
for (Map.Entry<Character, Double>
entry: map.entrySet())
```

LIST/SET: add(), remove()
LIST: get(index)
MAP: put(key, value), remove(key),getKey()

Interface vs. Abstract Class

Methoden	implizit public, abstract	evt abstract, nie private
Variablen	KONSTANTEN (impl. public static final)	normal, Constructor mit super()
Vererbung	`implements a,b`	`extends a`

Abstract Method (in abs. cl.): `public abstract void report();` kein Rumpf{}
in interface: `void report();`
I. Methode mit Rumpf: `default void report(){...}`
`interface I1 extends I2,I3`
`class C2 extends C1 implements I1, I2`
wenn I1&I2 nun gleiche Methoden haben (signatur) --> C2 muss ine der beiden überschreiben. Zugriff möglich mit
`I1.super.methode();`
`I2.super.methode();`

Unchecked vs Checked Exceptions

Unchecked	Checked
Error	Try/catch oder Methodenkopf
RunTime-Exceptions	mögl. catch-Block: `e.printStackTrace()`

-> RunTime, NullPointer, IllegalArgument, IndexOutOfBounds

eigene exception

```
class myE extends Exception { myE()
{}
MyE(String message) {
super(message);} }
```

Junit

```
@Before
public void setUp() {...}
@Test (timeout = 500, expected =
Exception.class)
public void testGoodName(){
  assertEquals(expected, actual);
  assertTrue(condition);}
```

@After -> tearDown()
EdgeCases testen (Grenzwerte, null,...)

equals

```
@Override
public boolean equals(Object obj) {
   if(this == obj) {return true;}
   if (obj == null) {return false;}
   if (getClass() !=
obj.getClass()) {
      return false;}
   Student other = (Student)obj;
   return regNumber ==
other.regNumber;}}
```

HashCode überschreiben!
` x.equals(y) → x.hashCode() == y.hashCode()
Grund: inkonsistenz bei Hashing. obj wird nicht gefunden, obwohl in Hash-Tabelle,
x.contains(y) --> nutzt hashCode() & equals()

HashCode()

```
@Override
public int hashCode() {
  return 31* firstName.hashCode()
   + 31 * lastName.hashCode();
}
```

Funktionsschnittstelle

```
@FunctionalInterface
interface Comparator<T> {
  int compare(T first, T second);
}
```

genau eine Methode --> java.util.function

RegEx

Pattern pattern = Pattern.compile("*reg*")

vor ? optionaler Teil	([0]?[0-9]|2[0-3])
(){}*+?|\ als norm text	* \(\\ ...
Gruppennamen	(?<NAME>)

Matcher matcher = pattern.matcher(string);
if (matcher.matches()){String one = matcher.group("NAME")}