Java Cheat Sheet for AP

Identifiers

for variable, parameter, constant, user-defined method, or userdefined class

a...zA...Z_ followed by a...zA...Z_0...9

1. language keywords forbidden
2. lower/UPPER case-sensitive
3. begin with a lowercase letter, each subsequent word with a capital letter

a toto xy y_max BigOne bigOne getName preTaxTotal

~~by~~ ~~and~~ ~~for~~

Floating-point Numbers

for float, double

double-precision number

1-bit *52-bits* *11-bits*

sign * mantissa * 2^{exponent}

0.1 * 26 == 0.1+0.1+...+0.1 (26 terms) ---> false

Built-in Type

AP Java build-in type: int, boolean, double

	Primitive Type	Size(bit)	Value Range	Wrapper Type	example
	void	------	------	Void	
	boolean	**1**	**true or false**	**Boolean**	**true, false**
	byte	8	-128 ~ 127	Byte	
not in AP	char	16	Unicode 0 ~ 2^{16}-1	Character	
	short	16	-2^{15} ~ 2^{15}-1	Short	
	int	**32**	**-2^{31} ~ 2^{31}-1**	**Integer**	**2, -26,3000**
	long	64	-2^{63} ~ 2^{63}-1	Long	
not in AP	float	32		Float	
	double	**64**		**Double**	**2.718, -23.21, 1.6e4**

```
int x;    // declare a variable   can not use a variable before initialize
int x = 0;  // declare and initialize a variable
final int CLASS_SIZE = 35;    // Const Variable,
a quantity whose value will not change
```

Chaining of assignment statements

```
int next, prev, sum;
next = prev = sum = 0;
```

Conversions

Implicit Type Casting (Widening)
Automatic type conversion

*also used in small type to a lager type (**Extends Class**)*

byte → short → int → long →float → double

widening

Explicit Type Casting (Narrowing)

double → float → long → int →short → byte

narrowing

*no relationship between the java will throw **ClassCastException***

JAVA DATA TYPE CONVERSION CHART http://interviewquestionjava.blogspot.com/	byte 8-bit (1 byte) signed two's complement integer Range: -128 to 127	short 16-bit (2 bytes) signed two's complement integer Range: -32,768 to 32,767	int 32-bit (4 bytes) signed two's complement integer Range: -2147483648 to 2147483647	long 64-bit (8 bytes) two's complement integer Range: -9223372036854775808L to 9223372036854775807L	float single-precision 32-bit (4 bytes) IEEE 754 floating point Range: 1.4E-45F to 3.4028235E+38F	double double-precision 64-bit (8 bytes) IEEE 754 floating point Range: 2.2250738585072014E-308D to 1.7976931348623157E+308D	boolean represents one bit of information as flag (possible values true/false) Range: NA	char single 16-bit Unicode character Range: '\u0000' (or 0) to '\uffff' (or 65,535)	String Immutable non-primitive data type (it's an Object) , supports character string within double quotes Range: NA
byte [default value : 0]	NA	byte b = 65; short s =(short)b;	byte b = 65; int i = b;	byte b = 65; long l = b;	byte b = 65; float f = b;	byte b = 65; double d = b;	NA	byte b = 65; char c = (char) b;	byte b = 65; String str = new String(new byte[]{b});
short [default value : 0]	short s = 65; byte b =(byte)s;	NA	short s = 65; int i = s;	short s = 65; long l = s;	short s = 65; float f = s;	short s = 65; double d = s;	NA	short s = 65; char c =(char)s;	short s = 65; String str = Short.*toString*(s); String str1 = String.*valueOf*(s);
int [default value : 0]	int i = 65; byte b =(byte)i;	int i = 65; short s = (short) i;	NA	int i = 65; long l = i;	int i = 65; float f = i;	int i = 65; double d = i;	NA	int i = 65; char c =(char)i;	int i = 65; String str = Integer.*toString*(i); String str1 = String.*valueOf*(i);
long [default value : 0L]	long l = 65l; byte b =(byte)l;	long l = 65l; short s = (short) l;	long l = 65l; int i = (int)l;	NA	long l = 65l; float f = l;	long l = 65l; double d = l;	NA	long l = 65l; char c =(char)l;	long l = 65l; String str = Long.*toString*(l); String str1 = String.*valueOf*(l);
float [default value : 0.0f]	float f = 65f; byte b=(byte)f;	float f = 65f; short s = (short)f;	float f = 65f; int i = (int)f;	float f = 65f; long l = (long)f;	NA	float f = 65f; double d = f;	NA	float f = 65f; char c =(char)f;	float f = 65f; String str = Float.*toString*(f); String str1 = tring.*valueOf*(f);
double [default value : 0.0d]	double d = 65d; byte b =(byte)d;	double d = 65d; short s = (short)d;	double d = 65d; int i = (int)d;	double d = 65d; long l = (long)d;	double d = 65d; float f = (float)d;	NA	NA	double d = 65d; char c =(char)d;	double d = 65d; String str = Double.*toString*(d); String str1= String.*valueOf*(d);
boolean [default value : false]	NA	NA	NA	NA	NA	NA	NA	NA	boolean bln = true; String str = Boolean.*toString*(bln); String str1 = String.*valueOf*(bln);
char [default value : '\u0000']	char c = 'A'; byte b=(byte)c;	char c = 'A'; short s = (short)c;	char c = 'A'; int i = (int)c;	char c = 'A'; long l=(long)c;	char c = 'A'; float f = (float)c;	char c = 'A'; double d = (double)c;	NA	NA	char c = 'A'; String str = Character.*toString*(c); String str1 = String.*valueOf*(c);

String
[default value : null]
	String str = "65";
	byte b = Byte.*parseByte*(str);
	byte b1 = Byte.*valueOf*(str);
		String str = "65";
		short s = Short.*parseShort*(str);
		short s1 = Short.*valueOf*(str);
			String str = "65";
			int i = Integer.*parseInt*(str);
			int i1 = Integer.*valueOf*(str);
				String str = "65";
				long l = Long.*parseLong*(str);
				long l1 = Long.*valueOf*(str);
					String str = "65";
					float f = Float.*parseFloat*(str);
					float f1 = Float.*valueOf*(str);
						String str = "65";
						double d = Double.*parseDouble*(str);
						double d1 = Double.*valueOf*(d);
							String str = "true\";
							boolean bln = Boolean.*parseBoolean*(str);
							boolean bln1 = Boolean.valueOf(bln);
								String str = "A";
								char c = str.charAt(0);
									NA

Operators

precedence

Operator	Symbol and Precedence	example
Post-increment/post-decrement	i++, i--	return value then increment
Pre-increment/pre-decrement	++i, --i	first increment then return value
Unary operatora	+, -, ~, !	+5, -3
Multiplication/division/modulus	*, /, %	
Addition/subtraction	+, -	
Shift operators	<<, >>, >>>	
Relational operators	<, >, <=, >=, instance of	return boolean expressions
Equal to/not equal to	==, !=	return boolean expressions
Bitwise And, exclusive OR, inclusive OR	&, ^, |	
Logical AND, OR	&&, ||	Short-circuit evaluation
Ternary operator	?:	
Assignment Operators	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=	assignment

postfix and prefix increment

```
i = 0
i++;  // i = 1; i++ = 0
++i;  // i = 1; ++i = 1
```

equals (==)

Do not routing use == to test for equality of floating-point numbers

Logical AND, OR(&& ||)

Short-circuit evaluation:
A && B : A is false , B is not executed
A || B : A is true, B is not executed

Relational Operators

1. evaluate to boolean, true or false
2. support implicit type casting
3. used only in the comparison of primitive types
4. be careful when comparing floating-point values.

Precedence

1. same line have equal precedence
2. equal precedence is from left to right
3. () has the highest precedence

Input and Output

```
Scanner keyboard = new Scanner(System.in);
keyboard.next();
keyboard.nextInt();
keyboard.nextDouble();

System.out.print("...");
System.out.println("...");
System.out.println("Value x is " + 3);
```

Escape Sequence

a backslash followed by a single character

Escape Characters	Description
\t	tab
\'	**single quoto**
\"	**double quoto**
\r	carriage return
****	**backslash chatacter**
\n	**new line**
\f	form feed
\b	backspace

packages

```
import packagename.*;
import packagename.ClassName;
import packagename.subpackagename.ClassName;

import java.uril.ArrayList;
```

Conditional Statement

statement block executed only if a condition is true

```
if (boolean expression) {
   statements block
}
```

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

```
if (boolean expression) {
   statements block
} else if (boolean expression) {
   statements block
}
...
else {
   statements block
}
```

block without {} can only contain one line

```
if (boolean expression)
   if (boolean expression)
      statement;
```

Can go with several else if, and only one final else. Only the block of first true condition is executed

```
if (boolean expr1 && boolean expr2)
   statement;
```

```
String state = "";
if (bmi<16) {
   state = "Severe Thinness";
} else if (bmi >= 16 && bmi < 17) {
   state = "Moderate Thinness";
} else if (bmi >= 17 && bmi <= 18.5) {
   state = "Mild Thinness";
} else if (bmi >= 18.5 && bmi <= 25) {
   state = "Normal";
} else {
   state = "Overweight";
}
```

While Loop

infinite loops: a loop that repeats forever

```
while (boolean test) {
   statements    // loop body
}
```

initializations before the loop

```
int i = 1, mult3 = 3;
while (mult3 < 20){
   System.out.print(mult3 + " ");
   i++;
   mult *= i;
}
```

1. Used in infinite loops.
2. Don't forget to change the loop variable in the body of the loop

For loop

iterator loop: the number of loops we know

go over sequence's index

For Loop

```
for (initialization; termination condition;
update statement) {
   statements      // body of loop
}
```

initializations before the loop

```
int sum = 0;
for (int i = 0; i<100; i++){
   if (i % 2 == 0){   loop variable i
      sum += i;
   }
}
System.out.println(sum);
```

go over sequence's value

For-Each Loop

```
for (SomeType element: collection) {
   statements      // body of loop
}
```

```
for (int element: arr){
   System.out.println(element);
}
```

Nested Loop

```
for (int i = 1; i <= 6; i++) {
   for (int j = 1; j <= i; j++)
      System.out.print("+");
   for (int j = 1; j <= 6 - i; j++)
      System.out.print("*");
   System.out.println();
}
```

1. The loop variable should not have its value changed inside the loop body
2. The initializing and update statements can use any valid constants, variables, or expressions.
3. The scope of the loop variable.

Exception

Exception	example
AirthmeticException	int a = 5 / 0
NullPointerException	String a = null; a.length()
ArrayIndexOutOfBoundsException	arr[100]
IndexOutOfBoundsException	arrayList[100]
IllegalArgumentException	g.setColor(5,10,10)
ConcurrentModificationException	for iterator with list.remove()

```
try{
   // statements that might cause exception
} catch {
   // statements that handle an exception
   //examples, closing a connection, closing
   //file, exiting the process after writing
   //details to a log file
}
throw new ArithmeticException();
```

Class

class is a software blue print for implementing objects of a given type.

object is a single instance of the class

public: usable by all program outside the class

default: used only by classes in its own package

private: accessed only by methods of that class

static: variable contains a value that is shared by all instances of the class.

final : the value can not be changed

Data encapsulation: Combining an object's data and methods into a single unit

There is and only one public class in a java file, and any classes in this file.

```
public class BankAccount {
   private String password;
   private double balance;
   public static final double OVERDRAWN_PENALTY = 20.00;
   private static double intRate;

   public BankAccount() {  default constructor
      this.password = "";
      this.balance = 0.0;
   }

   public BankAccount(String acctPassword, double acctBalance) {
      this.password = acctPassword;
      this.balance = acctBalance;
   }

   public double getBalance() {
      /* implementation code */
      return this.balance;
   }

   public void deposit(String acctPassword, double amount) {
      /* implementation code */
      if (!acctPassword.equals(password)){
         /* throw an exception */
      }else {
         this.balance += amount;
      }
   }

   public void withdraw(String acctPassword, double amount) {
      /* implementation code */
      if (!acctPassword.equals(password)){
         /* throw an exception */
      }else {
         this.balance -= amount;
         if (this.balance <= 0){
            this.balance -= OVERDRAWN_PENALTY;
         }
      }
   }

   public static double getInterestRate(){
      System.out.println("Enter interest rate for bank account");
      System.out.println("Enter in decimal form: "); // read User input
      intRate = 10;
      return intRate;
   }

   public static void main(String[] args) {
      BankAccount b1 = new BankAccount("mattW", 500.00);
      BankAccount b2 = new BankAccount("DannyB", 650.00);
      if (b1.getBalance() > b2.getBalance()){
         System.out.println("b1 is rich");
      }
      double interestRate = BankAccount.getInterestRate();
   }
}
```

constructors:

a constructor creates an object of the calss

1. *the same name as the class*
2. *no return type*

constructor with parameters

Accessors:

accessors method is a public method that accesses a class object without altering the object.

it allows other objects to get the value of a private instance variable.

Mutators:

A mutator method changes the state of an object by modifying at least one of its instance variables.

```
public class DoOperations {
   public int product(int n) {
return n * n; }
   public double
product(double x) { return x * x;
}
   public double product(int x,
int y) { return x * y; }
   ...
}
```

Method Overloading:

1. the method's name
2. a list of the parameter types

1. primitive date type
2. reference data type

instance method: accessors, mutators , all operate on individual objects of a calss

static method: performs an operation for the entire class.

a static method can only use a static variable in its code

a static method is invoked by using the class name with the dot operator

Inheritance

Inheritance defines a relationship between objects that share characteristics.

Subclass
superclass

extends

```
public class UnderGrad extends Student
```

super

```
super();
super.computeGrade();
```

Declaring

```
Student s = new Student();
Student g = new GradStudent();
Student u = new UnderGrad();
GradStudent g = new Student();
UnderGrad u = new Student();
```

Polymorphism is the mechanism of selecting the appropriate method for a particular object in a class hierarchy.

Dynamic binding(Late binding): Making a run-time decision about which instance method to call

Static binding(Early binding): The compiler selects the correct overloaded method at compile time by comparing the methods' signatures.

String

```
String a1 = "hello";
String a2 = "hello";
System.out.println(a1 == a2); //true
String b1 = new String("hello");
String b2 = new String("hello");
System.out.println(b1 == b2); //false
System.out.println(b1.equals(b2)); //true
```

String is immutable

String pool

```
s1.equals(obj); --boolean  // true argument is not null and is a
s1.equalsIgnoreCase(str); --boolean //ignoring case considerations
s1.length(); --int   //length of String, num of unicode code units
s1.charAt(index);  --char  // char value at the specified index
s1.substring(beginIndex); --String  // beginIndex to the end
s1.substring(beginIndex, endIndex);  --String  // beginIndex to endIndex - 1
s1.indexOf(ch); --int // first occurrence of int ch. ch is a char
s1.lastIndexOf(ch);  --int //  last occurrence of int ch. ch is a char
s1.compareTo(str); --int // ==0: equals; >0 lexicographically follows;
<0: lexicographically precedes
s1.compareToIgnoreCase(s2);
s1.toUpperCase(); --String // convert to upper case
s1.toLowerCase(); --String // convert to lower case
s1.trim() --String  // all leading and trailing space removed
```

```
"unhappy".substring(2)      //returns "happy"
"cold".substring(4)         //returns ""
"cold".substring(5)         //StringIndexOutOfBoundsException
"strawberry".substring(5,7) //returns "be"
"crayfish".substring(4,8)   //returns "fish"
"crayfish".substring(4,9)   //StringIndexOutOfBoundsException
"crayfish".substring(5,4)   //StringIndexOutOfBoundsException
```

Object

```
Object Methods:
toString()
equals()
Do not use == to test objects for equality. Use the equals method.
compareTo()
< >  are not used for objects. use compareTo method
```

Wrapper Class

```
Integer:
Integer(int value) // Constructs an Integer object from an int. (Boxing.)
int intValue() // Returns the value of this Integer as an int. (Unboxing.)
Integer.MIN_VALUE  //A constant equal to the minimum value represented by an int or Integer.
Integer.MAX_VALUE //A constant equal to the maximum value represented by an int or Integer.
Double:
Double(double value)  //Constructs a Double object from a double. (Boxing.)
double doubleValue()  //Returns the value of this Double as a double. (Unboxing.)
```

```
Integer intOb1 = 4;
Integer intOb2 = 4; // (intOb1 == intOb2)
Integer intOb3 = 4;
Integer intOb4 = new Integer(4); // (intOb3 != intOb4)
Integer intOb5 = 4;
int intOb6 = 4; //(intOb5 == intOb6)
Integer intOb7 = 4;
Integer intOb8 = new Integer(4); //(intOb7.intValue() == intOb8.intValue())
Integer intOb9 = 4;
Integer intOb10 = 8; // (intOb9 < intOb10)
Integer intOb11 = 4;
int n = 8;  // (intOb11 < n)
```

Math

```
Math:
static int abs(int x) //Returns the absolute value of integer x.
static double abs(double x)  //Returns the absolute value of real number x.
static double pow(double base, double exp)
static double sqrt(double x)
static double random() // Returns a random number r, where 0.0 ≤ r < 1.0.
```

Random

```
Random:
double x1 = 6 * Math.random(); // range 0.0 <= x < 6.0
double x2 = Math.random() + 2;  //range 2.0 <= x < 3
double x3 = 2 * Math.random() + 4; // range 4.0 <= x < 6.0
int num1 = (int) (Math.random() * 100); // 0 to 99
int num2 = (int) (Math.random() * 100) + 1; // 1 to 100
int num3 = (int) (Math.random() * 20) + 5; // 5 to 24
```

Array

Array

```
declare:
int[] dataInt = new int[25];
double dataDouble[] = new double[25];
String[] dataStr;
dataStr = new String[25];
int[] intList1, intList2;
int[] arr1 = neZ int[15], arr2 = neZ int[30];

initialization:
int[] coins = neZ int[4];
coins[0] = 1;
coins[1] = 5;
coins[2] = 10;
coins[3] = 25;
int[] coins = {1, 5, 10, 25};

array.length    // property  length of array
```

ArrayList

ArrayList

```
declare:
private ArrayList<Clown> clowns;
clowns = new ArrayList<Clown>();

ArrayList() // Constructor empty list
int size()    // number of elements
boolean add(E obj) // Appends obj (end)
void add(int index, E element) // insert obj (index)
E get(int index)  // return obj (index)
E set(int index, E element)  //Replaces item (index)
E remove(int index) // Removes and returns (index)
```

Two-dimensional

two-dimensional

```
declare:
int[][] table;
double[][] matrix = new double[3][4];
String[][] strs = new String[2][5];

int[][] mat = { {3, 4, 5},      //row 0
                {6, 7, 8}};     //row 1


for (row = 0; row < mat.length; row++)
   for (col = 0; col < mat[0].length; col++)
      processElements();
```