Java Cheat Sheet

Print something to the console

```
System.out.println(valueToPrint);
```

```
System.out.println("hello");
```

Define a runnable program

```
public class ProgramName {
    public static void main(String[] args) {
        // Code to run goes here
    }
}
```

```
public class GreeterProgram {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Comments

```
// Single line comment

/* Multi
      line
comment */

/**
* <h1>JavaDoc</h1> comments contain HTML and attributes
*
* @attribute value
*/
```

attribute: author, version, since

Primitive Data Types

Type	**Description**	**Default *Value***	**Example(s)**	**Operators**
int	integers	`0`	`8, -3, 100000`	`+ - * / %`
double	numbers with decimal points	`0.0`	`3.14159`	`+ - * /`
boolean	`true` or `false`	`false`	`true`	`&& || !`
String	sequence of characters	`null`	`"hello", "3.14"`	`+`

Declare a *variable*

```
VariableType variableName;
```

```
int numOranges;
```

Naming rules

1. Cannot be a reserved `keyword`
2. Cannot start with a number
3. No punctuation, delimeters, or special characters

4. No spaces

Define a *variable* (assign a *value*; assumes it's already been declared)

```
variableName = newValue;
```

```
pi = 3.14;
r = pi;
circumference = 2 * pi * r;
```

newValue can be a literal, another variable (already defined), or an expression

Declare and define a *variable* simultaneously

```
VariableType variableName = initialValue;
```

```
double pi = 3.14;
```

Use a specific class from a library

```
import packagename.ClassName;
```

```
import kareltherobot.UrRobot;
import kareltherobot.World;
```

Use any class from a library

```
import packagename.*;
```

```
import turtlefx.*;
```

Create an object

```
new ClassName(required, parameter, values);
```

```
new UrRobot(1, 1, North, infinity);
new Turtle(300, 350, 90);
```

Create an object that you can actually use

```
ClassName objectName = new ClassName(required, parameter, values);
```

```
UrRobot r = new UrRobot(1, 1, North, infinity);
Turtle t = new Turtle(300, 350, 90);
```

Call a *method* on an object

```
objectName.methodName(required, parameter, values);
```

```
r.move();
t.goForward(10);
```

Define an instantiable object type

```
public class TypeName {
    // Convention: define instance variables
    // then constructors then methods
}
```

```
public class Employee {
    // Worthless unless you put something inside
}
```

Define a *constructor* (directly inside a class definition)

```
/**
* Description of this constructor
* @param pName Description of parameter(s)
*/
public TypeName(pType pName, pType pName) {
    // Code to run when creating an object of this type
}
```

```
/**
* Create an emloyee and tell everyone about it
* @param name  Name of the new employee
* @param age   Age of the new employee
*/
public Employee(String name, int age) {
    System.out.println("We've got a new employee: " + name);
}
```

Define a *method* (directly inside a class definition)

```
/**
* Description of this method
* In addition to parameters you can document...
* @return Description of the value that's returned
*/
visibility returnType methodName(pType pName) {
    // Code to run when calling this method
    // If returnType is not void, a value must be returned
    // Refer to the parameter values that were passed in by
    //   using the pName (without the type)
}
```

```
/** (Inside the Employee class)
* Pay the employee for all their hard work
* @param  money Amount of money to be paid
* @return Acknowledgement of receipt of pay
*/
public boolean getPaid(double money) {
    System.out.println("Woohoo! " + money + " bucks!");
    return true;
}
```

`visibility`: `public`, `private`, `protected`

returnType: `void`, primitive data type, object type

use `return` to send back a value to the place where this method was called

Define an object type based on another type

```
public class TypeName extends ParentType {
    /* Constructors and methods:
       use super to refer to the ParentType for
       calling ParentType constructors and methods
    */
}
```

```
public class MyRobot extends UrRobot {
    public MyRobot(int beepers) {
        super(2, 3, East, beepers);
        super.move();
    }
}
```

Override a *method* that already exists in a ParentType

```
@Override
visibility returnType methodName(pType pName) {
    // New code to run when calling this method
}
```

```
public class MyRobot extends UrRobot {
    @Override
    public void move() {
        move();
        move();
    }
}
```

The method name, parameters and return type have to be identical to the original
The `@Override` notation isn't strictly necessary but it's good practice

Send back a value from a method (inside a method)

```
return value;
```

```
return Math.PI * r * r;
```

Define an instance variable

```
// Typically the first thing inside a class definition
visibility VariableType variableName = defaultValue;
```

```
public class Employee {
    private String myName = "Undefined";
    private int myAge = -1;
    public Employee(String name, int age) {
        this.myName = name;
        this.myAge = age;
    }
}
```

`visibility` should almost always be `private`
Use the `this` to disambiguate between parameters and instance variables

Explicitly cast an object to a different type

```
(NewType) objectReference
```

```
// The Employee class inherits from the Person class
Person p = new Employee(); // valid implicit cast
Employee e = (Employee) p; // explicit cast
```

Upcasts are always valid and can be done without the explicit cast
Downcasts can cause runtime errors and as such require the explicit cast

Define an interface

```
// The name of an interface usually describes an ability or a generic type
public interface InterfaceName {
    // Declare public methods but can't define them
    // Classes that implement this interface must define them
}
```

```
public interface Resizeable {
    // public not needed (has to be public)
    void setSize(int size);
    void increaseSize(int increment);
    void decreaseSize(int decrement);
}
```

Implement an interface

```
public class ClassName implements InterfaceName {
    // All methods from the interface must be implemented
}
```

```
public class Square implements Resizeable {
    public void setSize(int size) {
        this.width = size;
    }
    public void increaseSize(int increment) {
        this.width += increment;
    }
    public void decreaseSize(int decrement) {
        this.width -= decrement;
    }
}
```

Define a *non*-instantiable object type

```
public abstract class TypeName {
    // Can contain defined methods
    // Can contain undefined (abstract) methods but
    //   must be impmlented by instantiable sub-types
}
```

```
public abstract class Shape {
}
```

Define an abstract *method* (inside an abstract class)

```
visibility abstract returnType methodName(pType pName);
// abstract methods can't be defined (no method body)
// these methods must be defined in an instantiable sub-type
```

```
/** (Inside the abstract Shape class)
* Different types of shapes calculate their area differently
*   so it can't be defined in the generic shape class
*/
public abstract double calcArea();
```

Create a boolean expression

```
// Any of the following
boolean value
method that returns a boolean value
comparison operation
boolean operation (combining multiple boolean expressions)
```

```
// Mirroring the left
true
frontIsClear()
age > 10
frontIsClear() && anyBeepersInBag()
```

Comparison operators: < <= == != >= >

Boolean operators: && || !

Conditionally execute code

```
if (booleanExpression1) {
    // Code to run when booleanExpression1 is true
} else if (booleanExpression2) {
    // Code to run when booleanExpression1 is false
    //   AND booleanExpression2 is true
} else {
    // Code to run when BOTH conditions are false
}
```

```
if (age >= 20) {
    System.out.println("You're old'");
} else if (age > 13) {
    System.out.println("You're a teenager");
} else {
    System.out.println("You're a little kid");
}
```

The `else` and the `else if` parts are optional but the `if` part is NOT optional

Execute code a specific number of times

```
for (indexVarInit; runCondition; incrementOrDecrement) {
    // Code to run multiple times
}
```

```
// Prints out the numbers from 0 to 99
for (int i = 0; i < 100; i++) {
    System.out.println(i);
}
```

`indexVarInit` tells you where to start
`runCondition` tells you where to stop
`incrementOrDecrement` tells you how to get there

Execute code as long as a condition is true

```
while (loopCondition) {
    // Code to run as long as the loop condition is true
}
```

```
while (karel.frontIsClear()) {
    // Move the robot forward until it encounters a wall
    karel.move();
}
```

Handle an exception

```
try {
    // Code that could throw an exception
} catch (ExceptionType exceptionName) {
    // Code to run when an exception occurs
} finally {
    // Code to run at the end whether
    //   or not an exception occurs
}
```

```
try {
    doDangerousStuff();
} catch (DangerousException de) {
    System.out.println("Error!");
} finally {
    System.out.println("Goodbye!");
}
```

`try` is required, but only need one either `catch` and `finally`

Pass an exception on (to whomever called this method)

```
public void myMethod() throws ExceptionType {
    if (problemSituation == true) {
        throw new ExceptionType();
    }
}
```

```
public void doDangerousStuff() throws DangerousException {
    if (! goodSituation) {
        throw new DangerousException();
    }
}
```

Working with iterators

This is using Java Generics: don't worry about the angle brackets for now

```
Iterator<Type> iteratorName = iteratorMethod();
while (iteratorName.hasNext()) {
    Type varName = iteratorName.next();
    // do something with varName...
}
```

```
Iterator iter = listOfNums.iterator();
while (iter.hasNext()) {
    Integer i = iter.next();
    System.out.println(i);
}
```

Declare a variable to refer to an array

```
ContentType[] arrayName;
```

```
int[] nums;
```

ContentType can be a primitive data type or a *type of Object*

Create an array literal

Must be assigned to a *new* array variable as in the example. You cannot use an array literal to reassign to an existing variable.

```
{value0, value1, value2, etc}
```

```
int[] primes = {1, 3, 5, 7, 11, 13};
// The following line will fail
primes = {1, 3, 5, 7, 11, 13, 17};
```

Get the size of an array

```
arrayVariable.length
```

```
primes.length
```

By itself, this doesn't do anything. It's just a value that you can use in another statement.

Create an "empty" array (with default values)

Usually assigned to a variable as in the example

```
new ContentType[size]
```

```
int[] nums = new int[100];
```

Get the value of an element in an array

By itself, this doesn't do anything. It's just a value that you can use in another statement.

```
arrayName[index]
```

```
nums[0] // First value
nums[nums.length - 1] // Last value
```

Note: indecies are zero-based

Note: if the index used is bigger than the size of the array, you will get an ArrayIndexOutOfBoundsException

Set the value of an element in an array

```
arrayName[index] = newValue;
```

```
nums[3] = 10;
```

Note: if the index used is bigger than the size of the array, you will get an ArrayIndexOutOfBoundsException

Create a 2D array literal

Line breaks and spaces aren't necessary but are useful to better visualize it. The syntax really only allows you to visualize it in row-dominant orientation.

```
{
  {r0c0, r0c1, r0c2, etc},
  {r1c0, r1c1, r1c2, etc},
  {r2c0, r2c1, r2c2, etc}
}
```

```
int[][] multiples = {
  {1, 2, 3,  4,  5,  6},
  {2, 4, 6,  8, 10, 12},
  {3, 6, 9, 12, 15, 18},
};
```

Create a 2-Dimensional array

This assumes row-dominant numbering which is the most common

```
ContentType[][] arrayName = new ContentType[numRows][numColumns];
```

```
int[][] matrix = new int[100][50];
```

Refer to values in a 2D array

This assumes row-dominant numbering which is the most common

```
arrayName[rowIndex][columnIndex]
```

```
matrix[10][5]
```

Can be used to get a value from the array or set a value in an array

Get the dimensions of a 2D array

This assumes row-dominant numbering which is the most common

```
arrayName.length // num rows
arrayName[0].length // num columns
```

```
matrix.length // num rows
matrix[0].length // num columns
```

Using the first row to determine how wide the array is assumes that the array is not jagged, but this is normally the case.

Create collections with different contents using generics

```
Interface<ContentType> collectionName = new CollectionType<ContentType>();
```

```
List<Integer> numbers = new ArrayList<Integer>();
```

You can use the CollectionType on both sides but using the generic `Interface` allows you to easily swap in a different CollectionType that implements that same `Interface`.
Note: ContentType CANNOT be a primitive data type (boolean, int, double) but there are class equivalents for each of them (Boolean, Integer, Double) precisely for this purpose.

for-each loop

Go through the contents of a collection

```
for (ContentType variableName : collectionReference) {
  // do something with each of the items in the collection
  // anywhere you use variableName here it will refer
  //   to the current value in the collection
}
```

```
Integer[] numbers = {1, 2, 3, 4, 5, ...}
for (Integer num : numbers) {
  System.out.print(num * num + ", ");
}
```

The collection can be anything that implements the Iterable interface which includes arrays and all the Java standard collections (among other things)