Not Isaiah Carrington
Not Isaiah.carrington@mycavehill.uwi.edu

Java Cheat Sheet

No long talk for this one. I just hope this helps. As per the other documents, the idea is the same. If you have any queries or if I have any mistakes, just send me an email.

1… 2… 3… GO

Table of Contents

Not Isaiah Carrington
Not Isaiah.carrington@mycavehill.uwi.edu

Glossary

- Java – A statically typed, interpreted, Object Oriented based programming language. Much fun.
- Statically Typed – All variables and objects must have an explicitly defined typed before being used. This helps prevent typing errors.
- Case Sensitive – The case matters. Ex:
    - cat, Cat and CAT are all seen as different names to Java.
- Naming Convention – A standard method of naming items.
- camelCase – The first letter is lowercase, and all subsequent first letters of following words is capitalized: Ex:
    - validName
    - aValidVariableName
- TitleCase – Similar to camelCase, difference being each first letter of all words are capitalized. Ex:
    - ValidName
    - AValidClassName
- Variable – A label for a block of memory where data can be stored
- Data Type – The type of data a value is. Ex:
    - Integer
    - String
    - Character
- Primitive Data Type – A builtin data type.
- Class – The smallest unit of definition. Used as a template/blueprint for creating objects. Ex:
    - A blueprint of a house, or computer
- User Defined Class – A class the user created.
- Object – An instance of a class. I.E, an object created from a blueprint. Ex:
    - A house made from a blueprint.
- Property – A value of a class used to describe it. Ex:
    - Color, Height, Width
- Method – A function of a class used to define a behaviour. Ex:
    - Walk, getArea, shrink, grow
- Array – A homogeneous contiguous data structure. (Stores data of the same type sequentially in memory)
- OOP – Object Oriented Programming. A style of programming built around objects and classes.
- Encapsulation – When data and methods that operate on that data is enclosed in a single class.
- Data Hiding – This works in tandem with Encapsulation, and is when data is hidden from the outside world.
- Access Modifiers – These modify the access a value has from within a class.
- Private Access – These values can only be used from within the class they are defined.
- Protected Access - These values can “only” (exceptions exist.) be used from within the class they are defined, or from child classes of the class from where they were defined.
- Public Access – These Values can be accessed from outside the class from where they were defined, including from other packages.
- Package – A package is a set of related classes in a specified namespace.

Not Isaiah Carrington
Not Isaiah.carrington@mycavehill.uwi.edu

- Namespace – A space for names (variable, classes, objects, etc) , where each name must be unique.
- Default Access – These values can be accessed only from other classes within the same package
- Package – A collection of related classes all accessible under the same namespace
- Inheritance – This is when a class is made using another class as a template, and as such has the same properties and methods. Ex:
    - A child getting traits from their fore parents.
    - A human inheriting certain traits from being a Mammal.
- Composition – This is when we use an object from a User Defined Class, as a property for another class.
- Polymorphism – This is the ability of a class to take on multiple forms, and is often used in tandem with Inheritance.
- Constructors – These are methods of classes which are often used to instantize the properties of an object when created.
- Message Passing – This is the method by which classes communicate with each other via method calls, without having to expose the underlying workings.
- Abstract Classes – This is a class that represents a concept, and not a concrete object. As such, it does not have any behaviors on its own, and objects cannot be made directly from it Ex:
    - An item is a concept, and a book is an implementation of that concept.
- Interface – This is a class that defines certain behaviors that any class which implements it must adhere to.
- Exceptions – A situation that occurs within the program due to “undefined” behaviour, which if it is not handled, will result in the program crashing.
- Error – A problem that occurs within the program which causes it to crash.
- Assertion – A statement used to confirm that a situation is true. If the situation is false, halts the program, by raising an error.
- Preconditions – A condition which is checked before any further instructions are executed.
- Design By Contract – An agreement between two classes to ensure defined behaviour by use of preconditions and postconditions.
- Postconditions – A condition which is checked after all previous executions were executed.
- Operand – Something that is being operated on.
- Overload - This refers to creating methods of the same name but different parameters
- Override – This refers to the process of a child class creating a method with the same name as one in the parent class.

Fundamentals

Arithmetic Operators:

+ (Adds the left operand to the right operand)

Ex: 3 + 5

- (Subtracts the right operand from the left operand)

Ex: 10 – 7

Not Isaiah Carrington
Not Isaiah.carrington@mycavehill.uwi.edu

* (Multiplies the left operand by the right operand)

/ (Divides the left operand by the right operand)

Naming Convention

This refers to the style of which names for objects can and should be written.

- Variables & Objects & Function names: camelCase
- Classes: TitleCase
- Constants: UPPERCASE, each word separated by _ (underscore)
- Can not contain symbols other than _ (underscore)
- Can not start with a number
- Cannot contain spaces

Examples:

VALID	INVALID
VALIDVARIABLENAME	!nvalidVariableName
VALIDCLASSNAME	123NotValid
VALID_CONSTANT	package

Creating Variables:

Creating variables of primitive data types can be done as follows:

```
data_type variableName = value
```

Example:

Creating an integer called num to store a value of 5.

```
int num = 5
```

Creating Functions / Methods

Functions, more correctly referred to as Methods given our OOP environment, are created as follows:

```
access_modifier return_type functionName (parameters) {
         // Function Code
}
```

Example, let’s create a method that displays “Hello World”

```
public void sayHello(){
         System.out.println(“Hello World”);
}
```

Now let’s create one, that displays a message that is provided to it via the parameter.

```
public void sayMessage(String message){
        System.out.println(message);
}
```

Lastly, let's create a function called sum, which takes 2 integers and returns their total.

```
public int sumNumbers(int num1, int num2){
        return num1 + num2;
}
```

New Keywords:

- void : This indicates that no value is to be returned.
- Parameter: These are variable names that are defined in the function. When a function receives arguments, they are sent to these variable names in order.
- Arguments: These are values that are passed to the function from the outside world.

Calling a Function:

Let's first call our function that takes no arguments.

```
sayHello();
```

Now, let's call the sayMessage function which takes a string as an argument.

```
sayMessage("Hello outside");
```

Lastly, let's call our sum function, and store the returned value in a variable.

```
int sum = sumNumbers(10, 20)
```

Overloading Functions:

Functions can be overloaded by creating a function with the same name, but with different number or types of parameters. Let's take our sum function, and create one to add 2 doubles, and one to concatenate 2 strings.

```
public double add(double x, double y){
        return x + y;
}

public String add(String x, String y){
        return x + y;
}
```

Now whether we want to add 2 numbers, or concatenate 2 strings, is dependent on the arguments we pass. For Example:

```
add(10.34, 4) // This will call the first one to add two numbers and return the value
add("String 1 ", "String 2") // This will concatenate the 2 strings and return the value
```

Logical Operators

&& (Checks if both the left and right operands are true. Returns true if so, returns false otherwise)

Not Isaiah Carrington
Not Isaiah.carrington@mycavehill.uwi.edu

Ex: true && true

|| (Checks if either the left or right operands are true. Returns false if neither are true, returns true otherwise)

! (Negates whatever follows. I.E, true becomes false and false becomes true)

IF Statements

Used to check a given condition, and if true, will run the code inside. Otherwise, will skip the code inside and carry on execution. If it contains an else branch, this branch will be executed instead. If it contains an else if, its condition will be checked and executed if true.

Format:

```
          if (condition) {
                    // Code to run if true
          } else if (condition) {
                    // Code to run if true
          } else {
                    // Code to run if everything else is false
          }
```

For loop

A for loop is used to repeat a block of code a specified number of times

Format:

```
for (counter = start_value; condition; change condition){
          //  (Code to be repeated)
}
```

Example, counting to 10:

```
for (int I = 0; I < 10; I++){
          System.out.println(I);
}
```

While Loop

A while loop can be used to repeat a block of code until a condition is false.

Format:

```
while (condition) {
          // Code to be repeated
}
```

Example, counting to 10.

```
int I = 0;
while (I < 10){
          System.out.println(I++);
}
```

Not Isaiah Carrington
Not Isaiah.carrington@mycavehill.uwi.edu

Creating and Using Arrays

Arrays are created using the following format.

data_type arrayName[] = new data_type[array_size];

Where data_type is the type of data that will be stored. The square brackets [] are necessary.

Example: let's create an array of 10 integers.

```
int numArray[] = new int[10];
```

Initializing an array using a for loop:

An array can be initialized using a for loop. For example, let's take our numArray, and set each of the values to 3 times the index.

```
for (int I = 0; I < 10; I++){
		numArray[I] = I * 3;
}
```

OOP

Creating Classes:

Classes are created using the following syntax:

```
access_modifier class ClassName {

	// Class Properties


	// Class Methods

}
```

The access_modifier is optional, and will use the default modifier mentioned in the glossary.

For example, creating a class called Cat, whose properties consist of size and color, and use behaviours consist of talking and walking.

```
public class Cat{
		private int size;
		private String color;

		public void talk(){
				// Code to talk
		}

		public void walk(){
				// Code to walk
		}
}
```

Creating Classes with No Argument Constructor

A constructor is defined as a method with the same name as the class, and can be used to initialize the properties of a class. One is included by default even if you don't explicitly create it, and takes no arguments. We can create our own constructor to control the behaviour we want.

Let's create a class Cat, which has a constructor to set the size to 10, and color to blue.

P.S (***this*** is a keyword used to refer to the current class.)

```
public class Cat{
          private int size;
          private String color;

          Cat(){
                    this.size = 10;
                    this.color = "Blue";
}

          public void talk(){
                    // Code to talk
          }

          public void walk(){
                    // Code to walk
          }
}
```

Creating Classes with Constructors that take Arguments

Sometimes we want the behaviour to be more controllable, and as such, may want to take information from outside the class to initialize the properties to.

Let's take our Cat class, and add a constructor that accepts a *size* and *color* from outside the class, and set our class properties to those values. We can use the ***this*** keyword so we can use the same name as shown below.

```
public class Cat{
          // Properties

          Cat() {
                    this.size = 10;
                    this.color = "Blue";
          }
          Cat (size, color){
                    this.size = size;
                    this.color = color;
          }

          // Methods
}
```

By using the ***this*** keyword, we can give our parameters the same name as our properties, which makes the code much easier to understand.

Not Isaiah Carrington
Not Isaiah.carrington@mycavehill.uwi.edu

Creating Objects

Objects are created using the ***new*** keyword.

Creating variables of custom data types (User Defined Classes)

ClassName nameOfObjectVariable = new ClassName()

Let’s create an object for our cat.

```
Cat myCat = new Cat()
```

By calling Cat() with no arguments, we are saying that we want the default constructor. Therefore, our cat will have a size of 10, and a color of “Blue”

Let’s create another cat, but this time using the second constructor by passing in the values to the Class call.

```
Cat alsoMyCat = new Cat(20, “Red”)
```

Now our second cat has a size of 20 and a color of “Red”

Composition

Composition is when we use a class as a property of another class.

Let’s create a class to represent a Player, and for its weapon, let’s create another class of sword.

```
class Sword{
        int damage;
        int size;

        Sword(){
                this.damage = 15;
                this.size = 4;
        }
}

class Player{
        int health;
        Sword sword;

        Player(){
                this.health = 100;
                this.sword =  new Sword();
        }
}
```

Inheritance

Let’s create a class of an Enemy, and then create a Zombie class to inherit the properties of Health and Damage from the Enemy class, along with the Attack behaviour.

```
class Enemy{
        protected int health;
        protected int damage;

        Enemy(int health, int damage){
                this.health = health;
                this.damage = damage;
        }

        public void attack(){
                // Code to attack
        }
}
class Zombie extends Enemy{
        Zombie(){
                super(100, 10);
        }
}
```

Zombie has the same properties of health and damage, and the same attack method.

Keywords:

extends – This is used to indicate that a class will inherit from another class. Format:

class base_class extends parent_class

super – Super refers to the parent class. By calling super(arguments), the arguments will be passed to the constructor of the Parent Class, and the properties will be initialized to these values.

Method Overriding

We can use method overriding to provide unique behaviour for our zombie's attack.

Overriding is as simple as creating a method in the subclass with the same name and parameters as a method in the parent class.

```
class Enemy{
        protected int health;
        protected int damage;

        Enemy(int health, int damage){
                this.health = health;
                this.damage = damage;
        }

        public void attack(){
                // Code to attack
        }
}
class Zombie extends Enemy{
        Zombie(){
                super(100, 10);
        }

        public void attack(){
                // Code for the zombie to attack.
        }
}
```

Not Isaiah Carrington
Not Isaiah.carrington@mycavehill.uwi.edu

Polymorphism

Polymorphism allows an object to have multiple forms, or behaving in different ways and is normally used with inheritance.

Let’s use the player class from before, and instead of having the sword field set as sword, we’ll set it as weapon instead. For example:

```
class Player{
          int health;
          Weapon weapon;

          Player(){
                    this.health = 100;
          }

          public void setWeapon(Weapon weapon){
                    this.weapon = weapon;
          }

          public void attack(){
                    this.weapon.attack();
          }

class Weapon{
          int damage;

          Weapon(int damage){
                    this.damage = damage;
          }
}

class Sword extends Weapon{
          Sword(){
                    super(20);
          }

          public void attack(){
                    // code for sword attack
          }
}

class Spear extends Weapon{
          Spear(){
                    super(30);
          }

          public void attack(){
                    // Code for spear attack
          }
}
```

With the Player’s weapon class done like this, we can then assign it any of the subclasses for Weapon, and it will work the way it’s supposed to. For example:

```
Player p1 = new Player();
p1.setWeapon(new Sword());
p1.attack() // Calls the code for the Sword’s attack
p1.setWeapon(new Spear());
p1.attack() // Now calls the code for the Spear’s Attack.
```

Not Isaiah Carrington
Not Isaiah.carrington@mycavehill.uwi.edu

Same weapon variable, but different objects and by extension, different behaviours. Such cool.

Abstract Classes

Refer to the glossary for what this is.

The format for an abstract class is:

```
access_modifier abstract class ClassName{
         // Properties

         // Abstract methods
         access_modifier abstract return_type methodName(parameters);
}
```

For an example, let's use the weapon class from before, and make it abstract.

```
public abstract class Weapon{
         protected int dmg;
         protected int size;

         Weapon(int dmg, int size){
                  this.dmg = dmg;
                  this.size = size;
         }

         public abstract void swing();
         public abstract void increaseDmg(int amount);
}
```

For abstract methods, we declare them, and then we define them in the subclass.

For example, let's create our sword.

```
class Sword extends Weapon{
         Sword(){
                  super(10, 12);
         }

         public void swing(){
                  // Code for swinging
         }

         public void increaseDmg(int amount){
                  this.dmg += amount;
         }
}
```

All abstract methods should be implemented.

Interfaces

Again, refer to the glossary. Format:

```
access_modifier interface InterfaceName{
         // Methods to be implemented.
}
```

Not Isaiah Carrington
Not Isaiah.carrington@mycavehill.uwi.edu

For example, let's create an interface to represent a vehicle. This will have methods which everything that implements this interface, must define.

```
public interface Vehicle{
          public void drive();
          public int getNumWheels();
}

public class Car implements Vehicle{
          public void drive(){
                    // Code to drive
          }

          public int getNumWheels(){
                    // Code to return number of wheels
          }
}

public class Truck implements Vehicle{
          public void drive(){
                    // Code to drive
          }

          public int getNumWheels(){
                    // Code to return number of wheels
          }
}
```

For an example as to how this can work:

```
Vehicle v1 = new Car();
Vehicle v2 = new Truck();
v1.drive();
v2.drive()
```

Because they have the same interface, they can be generalized as the Vehicle, and can therefore be used in that way. You should see this as a form of polymorphism.

Exceptions:

Handling:

Exceptions should be handled using the Try / Catch / Finally statement. The format for handling an exception is as follows:

```
try{
          // Code to execute
}
catch ( Exception) {
          // Code to run if the above exception is thrown
} finally {
          // Code to run whether the exception is raised or not
}
```

Catching:

Several exceptions can be caught by using several catches.

Not Isaiah Carrington
Not Isaiah.carrington@mycavehill.uwi.edu

```
try{
          // Code to execute
} catch ( Exception1) {
          // Code if Exception1 occurs
} catch (Exception2) {
          // Code if Exception2 occurs
} catch (Exception3){
          // Code if Exception3 occurs
} finally {
          // Code to execute regardless
}
```

Creating:

When wanting to create a custom Exception class, one must inherit from Exception. For example:

Let's create an Exception class, that we want to have thrown, whenever we receive invalid input.

```
class InvalidInputException extends Exception{

          InvalidInputException(){
                    super();
          }

          InvalidInputException(String message){
                    super(message);
          }
}
```

Throwing:

Exceptions are thrown using the throw keyword. For example:

```
throw InvalidInputException();
```

Note, you need to have an exception handler when you want to throw an exception, otherwise the program will not compile.

Throws:

To throw an exception from within a function, where the error handler is present in an outer function, the function you're throwing from MUST indicate that it throws the exception. For example:

```
class ExceptionExample{
          public void outerFunction(){
                    try {
                              ExceptionFunction(10);
                    } catch (InvalidInputException e){
                              System.out.println(e.getMessage());
                    }
          }

          public void ExceptionFunction(int x) throws InvalidInputException{
                    if (x != 5) throw InvalidInputException("The value was supposed to be 5.");
          }
}
```

Not Isaiah Carrington
Not Isaiah.carrington@mycavehill.uwi.edu

Conclusion:

I hope this helps or something. I don’t know. But… The information’s all there. Have fun.