**Java Cheat Sheet**

**Object-Oriented Programming**

**Access Modifiers**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Classes in same package can access | Classes in other packages can access | Subclasses in same package can access | Subclasses in other packages can access |
| **public** | Yes | Yes | Yes | Yes |
| **protected** | Yes | No | Yes | Yes |
| **no modifier** | Yes | No | Yes | No |
| **private** | No | No | No | No |

**Static**

Anything declared as static within a class will be shared by all objects instantiated from that class. This includes any objects instantiated from subclasses that inherit from a base class with a static member.

**Static variables:** Shared by all objects instantiated from that class and can be accessed using Class.staticVariable.

**Static methods:** Shared as well and can be called without creating an object of that class. Simply put Class.staticMethod.

**Static Class:** Can only have a static class nested within another class.

**Final**

**Variables:** Variables declared as final are constant and can't be redefined.

**Methods:** Methods declared as final can't be overridden.

**Classes:** Classes defined as final can't be extended.

**const** - same as final keyword in other languages but in java it is an unimplemented keyword. (Use final instead)

**Polymorphism - (many forms)**

The ability for an object to take on many forms.

Traditional polymorphism happens when a subclass overrides or overloads a method that is in a base class and that method is being called by an object whose type is of the base class but who instance is of a subclass.

**Example Below:**

class Animal {

public void animalSound() {

System.out.println("The animal makes a sound");

}

}

class Dog extends Animal {

public void animalSound() {

System.out.println("The dog says: bow wow");

}

}

class Main {

public static void main(String[] args) {

Animal myAnimal = new Animal(); // Create a Animal object

Animal myPig = new Pig(); // Create a Pig object

myAnimal.animalSound();

myPig.animalSound();

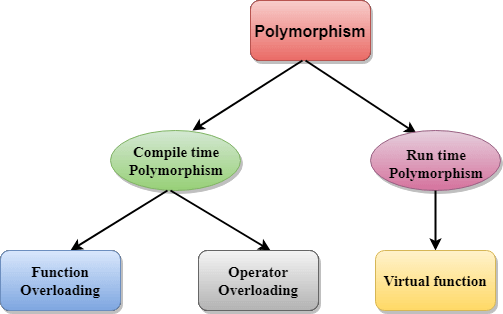
}

}

Polymorphism is useful because you can have polymorphic arrays. Polymorphic arrays that can hold more than one type of object that all inherit from a base class.

**More Info:**

1. Static Binding (or Compile time) Polymorphism, e.g., Method Overloading
2. Dynamic Binding (or Runtime) Polymorphism, e.g., Method overriding
3. Some even say that operator overloading is a form of Polymorphism



In **Java** all methods by default are considered **virtual functions** unless they are marked with the keyword final, then they are **non-virtual**.

**Instance vs Object**

Vehicle car = new Vehicle();

The new keyword creates an instance of the Vehicle class and allocates memory for the Vehicle object then returns a reference to that object. So "car" is technically a reference (sometimes called an instance) pointing to that object in memory.

From what I understand, "Instance" is the action of creating an object. And that object is physically stored in memory.

**Abstract Classes**

An abstract class shares both implementation and signature. It's kind of of a combination of classes and interfaces.

A class declared with the **abstract** keyword.

1) Cannot create an instance or object from an abstract class.

2) Must have at least on abstract method and can have multiple concrete methods.

**Interfaces**

Interfaces share behavior but not implementation. We don't need to share the code but just the function signature.

**Immutable**

Strings are immutable meaning their references do not change. If they were not immutable when you declare string literals of the same string, when one changes the others would change too.

**Example:**

String a = "hi";

String b = "hi";

String c = "hi";

a = "bye"

// Since strings are immutable, only variable a changes.

// If strings where not immutable, since b and c point to the same string literal in memory, they would also change.