Prog 2 Good to Know

Variables:

* Basic: type + variable name (noun, descriptive)
* Complex: Visibility modifier + static + final + type + name

Visibility modifiers:

* Public: Visible for everyone
* Private: Visible only in the given class
* Default: Accessible within the same class and package within which its class is defined.
* Protected: Accessible within the package in which it is defined and, in its subclass(es). Even if they are in another package.

Static:

The static keyword indicates that the member belongs to a type itself rather than to an instance of that type. (Can be used for variables, methods, nested classes, code blocks)

Final:

In Java the final keyword is a non-access specifier that is used to restrict a class, variable or a method.

* Variable: Restricts modification.
* Method: Cannot be overridden.
* Class: Cannot be extended.

Type:

In java there are:

* Primitives:
  1. Byte
  2. short
  3. Int
  4. Long
  5. Char
  6. Double
  7. Float
  8. boolean
* Objects:
  1. String
  2. Built-in types
  3. Custom types

Name:

The name of the variable is always a noun, written in camelCase and always describes the reason of the variable.

Method declaration:

Visibility modifier + static + final + return type + name + (parameter list) + { method body }

*See visibility modifiers, static and final descriptions above.*

* Return type: The type of the value the method will return with the return statement. Our method might not want to return anything, in that case use the void keyword as a return type.
* Name: The name of the method is always a verb written in camelCase. It describes what the method does.
* Parameter list: The number of parameters can vary from 0 to infinite basically. Although there is no upper limit to the number of parameters, we do to have more than 3 (except in case of constructors).
* Method body: It defines what the method does. You write the logic here that will be executed. It always must return something (except in case of void methods).

Objects and classes:

Primitive types simply hold a value, while objects have other functionalities:

* An object holds a state that consents from the properties(variables) it holds.
* An object has a unique behavior that is defined by the methods in it.
* An object is identified uniquely by its name.

Classes are used to provide a blueprint on how to create an object. A class defines what properties will be held in the object and defines its behavior by determining its methods.

Class definition:

Every java file must have one top level class and its name must be the same as the files name.

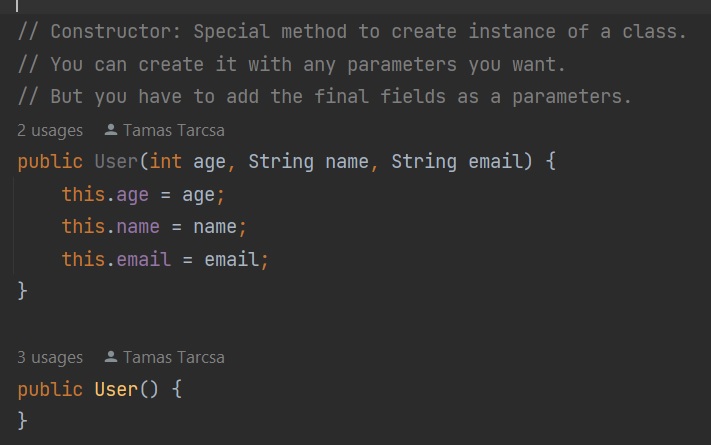
This class must be public. We use nouns as the name of the classes. They start with an upper-case letter and after that written in camelCase.

In case of inner classes: The naming conventions are the same, but we can use other visibility modifiers not just public. Inner classes are classes within another class.

You can create an object from a class by using the new keyword and its constructor.

What is the constructor?

The constructor is a special method used to create instances. Every class has an empty constructor by default. In the constructor, you can define or set the properties of the object or call initializer methods.



Wrapper Objects:

In java there is a wrapper object for every primitive type. Why?

* Primitives cannot be used everywhere (for example: Collections)
* The wrapper objects provide useful methods (for example valueOf())

The wrapper object is just simply holding the value of the primitive in an object, and they provide some useful behaviors for them.

In java there is an automatic process that converts a primitive into its wrapper object. This process is called Autoboxing. The opposite of this process is Unboxing.

String:

In java String is an object backed by a character array. It provides us a lot of valuable methods to handle and manipulate texts. Strings are immutable, which means: Once they have a value, it cannot be changed. When you change the value of the String, you just simply create a new String in the memory and set the pointer of your variable to it. This is where the String heap comes handy.

We never create a String like this: new String(“value”);

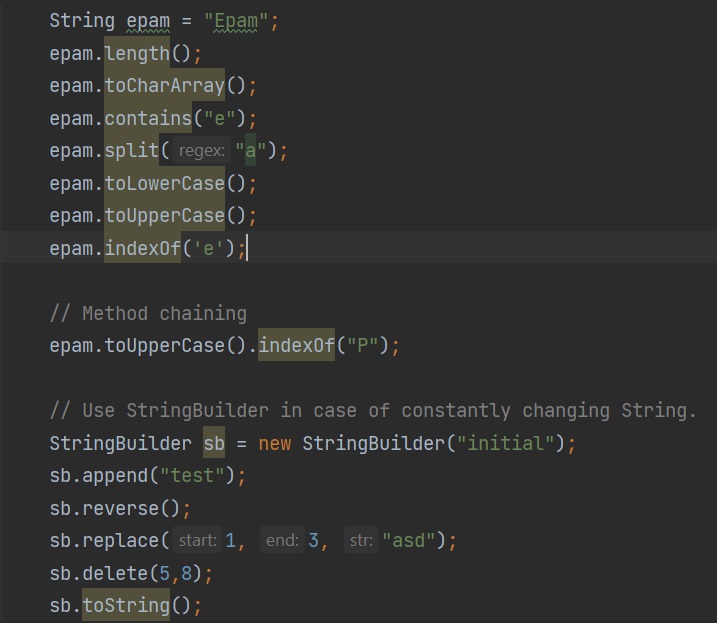
Its because in this case new parts of the memory will always be allocated for your variable. In this case you are wasting memory. We use the String literal to create variables:

String epam = “Epam”;

In this case, our String will be saved in the String heap and if you will create a new String in the future and assign the same “Epam” value to it, then it will be used from the heap and no new memory will be allocated.

If you would like to modify a String’s value often, then use StringBuilder. StringBuilder does not saves its value to the memory unless you tell it to do it. With this you can create a “mutable String”.

Useful String methods:



Abstract classes

An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

Abstract class definition:

Visibility modifier + abstract + class + name

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

Interfaces

An interface is a completely "abstract class" that is used to group related methods with empty bodies.

To access the interface methods, the interface must be "implemented” by another class with the implements keyword (instead of extends). The body of the interface method is provided by the "implement" class.

A class can only have one parent class, but it can implement multiple interfaces (just separate them with a comma)

Iterator

An Iterator is an object that can be used to loop through collections, like ArrayList and HashSet. It is called an "iterator" because "iterating" is the technical term for looping.

Iterable

Implementing this interface allows an object to be the target of the "for-each loop" statement.

Equals and hashcode

hashCode() returns an integer representing the current instance of the class. We should calculate this value consistent with the definition of equality for the class. Thus, if we override the equals() method, we also have to override hashCode(). Contracts:

* internal consistency: the value of hashCode() may only change if a property that is in equals() changes
* equals consistency: objects that are equal to each other must return the same hashCode
* collisions: unequal objects may have the same hashCode