COMP 1020 - Basic objects

UNIT 2

OOP

- OOP stands for Object Oriented Programming

OOP

- OOP stands for Object Oriented Programming
- It is a programming model, or paradigm, based on the construction and use of objects:
  - a program built under this model can be seen as a set of objects interacting together

OOP

- OOP stands for Object Oriented Programming
- It is a programming model, or paradigm, based on the construction and use of objects:
  - a program built under this model can be seen as a set of objects interacting together
- Many programming languages support OOP

OOP

- In this course, we're going to see the basic concepts of objects and how they work in Java

OOP

- In this course, we're going to see the basic concepts of objects and how they work in Java

- There is a second-year course focusing entirely on the concepts of object oriented programming:
  COMP 2150 - Object orientation

  - COMP 2150 goes a lot deeper into the concepts of OOP and shows you how they work in three different languages: Java, C++ and Ruby

OOP

- In this course, we're going to see the basic concepts of objects and how they work in Java

- There is a second-year course focusing entirely on the concepts of object oriented programming:
  COMP 2150 - Object orientation

  - COMP 2150 goes a lot deeper into the concepts of OOP and shows you how they work in three different languages: Java, C++ and Ruby
  - Very important that you understand the basic concepts to get ready for 2150

The purpose of class

- Breaking news: a class is not just a container in which we put a main method

```
public class MyProgram {

    public static void main (String[] args) {

        System.out.println("Hello!");
    }
}
```

Class means type

- When you create a class, you're actually defining a new type of data (i.e. an object)

Class means type

- When you create a class, you're actually defining a new type of data (i.e. an object)
- Like any other type, a class has:
  - A name (the class name, which should begin with an uppercase letter by convention)

Class means type

- When you create a class, you're actually defining a new type of data (i.e. an object)
- Like any other type, a class has:
  - A name (the class name, which should begin with an uppercase letter by convention)
  - Some kind of data that is stored: a collection of variables called instance variables (& class variables)

Class means type

- When you create a class, you're actually defining a new type of data (i.e. an object)
- Like any other type, a class has:
  - A name (the class name, which should begin with an uppercase letter by convention)
  - Some kind of data that is stored: a collection of variables called instance variables (& class variables)
  - Some kind of actions that can be performed on the data: a collection of methods called instance methods (& class methods)

Use of objects

- Objects in programs are often used to represent real life objects…
  - person, student, car, bank account, store, etc.
- … or more abstract concepts, such as data structures or GUI elements
  - list, tree, node, panel, button, etc.

Class definition

- A class is defined in a .java file, with the same name as the class

Class definition

- A class is defined in a .java file, with the same name as the class

- Example:

```
public class Person{
    public String name;
    public int age;
}
```

Class definition

- A class is defined in a .java file, with the same name as the class

- Example:

```
public class Person{
    public String name;
    public int age;
}
```

This is just the class definition, i.e a specification / model / template for your new type → it does not create anything in memory.

When you actually build an instance of this class, you get what we call an object.

Class definition

- A class is defined in a .java file, with the same name as the class

- Example:

```
public class Person{
    public String name;
    public int age;
}
```

These are the instance variables, i.e. the object's data → each instance will have its own specific set of values for the instance variables.

We'll talk about the "public" later.

Class definition

- A class is defined in a .java file, with the same name as the class

- Example:

```
public class Person{
    public String name;
    public int age;
}
```

We normally assign values to the instance variables in the **constructor** → we'll see that in a bit

Class definition

- A class is defined in a .java file, with the same name as the class

- Example:

```
public class Person{
    public String name;
    public int age;
}
```

When you compile this file, you get a Person.class file (as expected), and then you're ready to use your new type called Person

Creating objects

- Once your class has been compiled, you use this new type, in a main for example:

```
Person john;
Person jane;
```

Creating objects

- Once your class has been compiled, you use this new type, in a main for example:

```
Person john;
Person jane;
```

- What you see above is just a declaration, no object has been created yet

Creating objects

- Once your class has been compiled, you use this new type, in a main for example:

```
Person john;
Person jane;
```

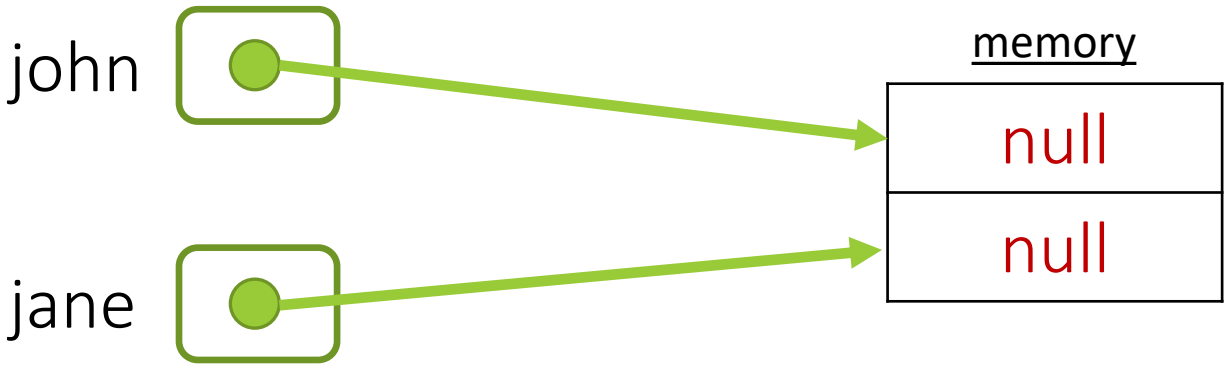
- What you see above is just a declaration, no object has been created yet

- All variables of an object type contain a reference, not the object itself: just like we have seen for arrays (and Strings too), because they are objects as well

Creating objects

```
Person john;
Person jane;
```

- When you have a reference that points nowhere, it points to the special reference null



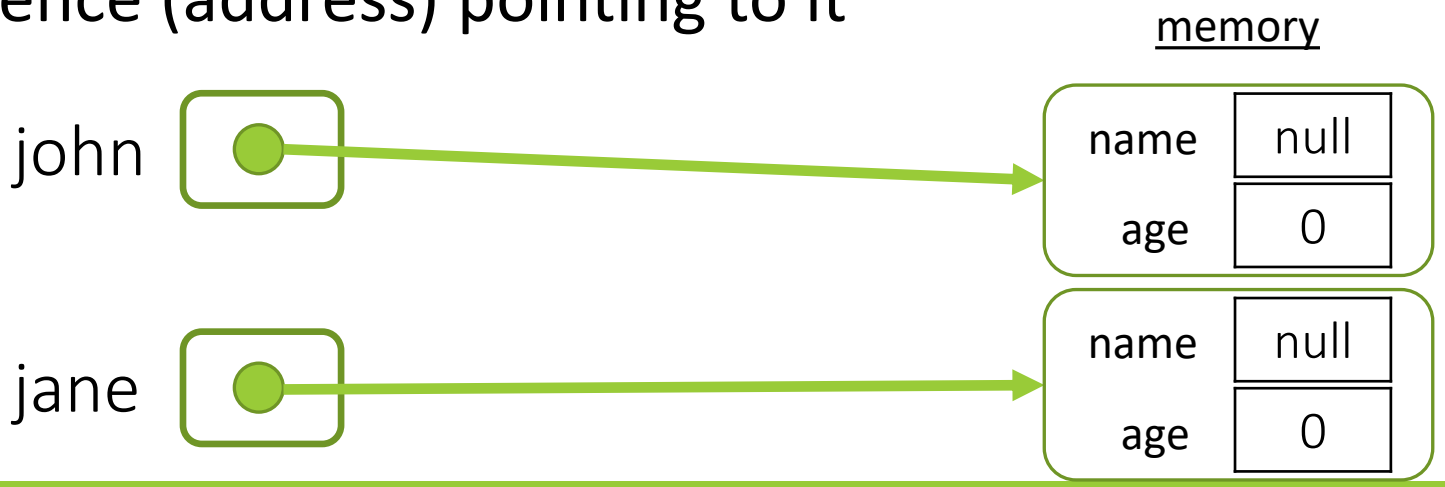
Creating objects

- To create an instance (an actual object), the basic syntax is:

```
Person john = new Person();
Person jane = new Person();
```

this calls the constructor!

- This creates the object in memory, and returns a reference (address) pointing to it



Using the instance variables

- Given a reference to an object (e.g. john or jane below), you can access the instance variables using the syntax below, assuming that they are "public" (but normally they shouldn't be… more on that soon)

```
john.name = "John";
jane.name = "Jane";
john.age = 55;
jane.age = 42;
if (john.age > jane.age)
    System.out.println(john.name + " is older!");
```

Instance methods

- Instance methods are methods that can only be used on instances (instantiated objects)
- These methods have access to the instance variables of that specific instance

Instance methods

- Instance methods are methods that can only be used on instances (instantiated objects)
- These methods have access to the instance variables of that specific instance
- They can be defined similarly to the methods we introduced last week, but for instance methods you must
  - omit the static keyword
  - add the public keyword (in most cases… more on that soon)

Instance methods

- Example:

```
public class Person{
    public String name;
    public int age;

    //Instance methods below:
    public void haveBirthday() { age++; }

    public int getNbLettersInName() {
        return name.length();
    }
}
```

See how the instance methods refer directly to the instance variables of some specific object (a specific instance)

Using an instance method

- You use an instance method by applying it as an operation to one particular object instance:

```
jane.haveBirthday();
john.haveBirthday();
int numLetters = jane.getNbLettersInName();
```

objectInstance.instanceMethod();

Using an instance method

- You use an instance method by applying it as an operation to one particular object instance:

```
jane.haveBirthday();
john.haveBirthday();
int numLetters = jane.getNbLettersInName();
```

objectInstance.instanceMethod();

- The instance methods are called “on” a specific instance, and they will be able to access/change the instance variables (e.g. name/age) of this specific instance

Message terminology

- This is traditionally known as “sending a message to an object”
- E.g. send the haveBirthday() message to the jane object:

```
jane.haveBirthday();
```

Access modifiers

- Each variable or method in a class can have 4 different access modifiers, which affect their visibility/accessibility:
  - public
  - private
  - protected
  - package-private

Access modifiers

- Each variable or method in a class can have 4 different access modifiers, which affect their visibility/accessibility:
  - public
  - private

  We'll just focus on these 2 for now
  - protected
  - package-private

  We'll come back to these ones in a few weeks; ignore these for now!

Access modifiers

- public: means any code, anywhere, can access or use it
- private: means only methods in this same class can access or use it

Access modifiers

Rule of thumb:

- Use private for instance variables
  - Objects should deal with their own data and provide public methods for others to access/modify it

- Use public for most instance methods (unless you have a method that should only be used internally, then you can use private)
  - methods are (normally) supposed to be used by others

Principle of encapsulation

- Encapsulation is one of the main features of object-oriented programming
- It's the idea that you can restrict access to some of the object's fields, you can hide some information from other classes that use the object

Principle of encapsulation

- Goal: protecting the internals, preventing other classes from misusing the object

Principle of encapsulation

- Goal: protecting the internals, preventing other classes from misusing the object

- As a result of using encapsulation:

  - all code that can affect the object's members is local (to that specific class)

  - code is more reliable, easier to debug, easier to update and maintain

Principle of encapsulation

- If the instance variables are private, then we provide "accessor" and "mutator" methods (get/set methods) if needed:

```
private String name;
…
public String getName() { return name; }  //accessor
public void setName(String newName) {  //mutator
    name = newName;
}
```

Why not just public?

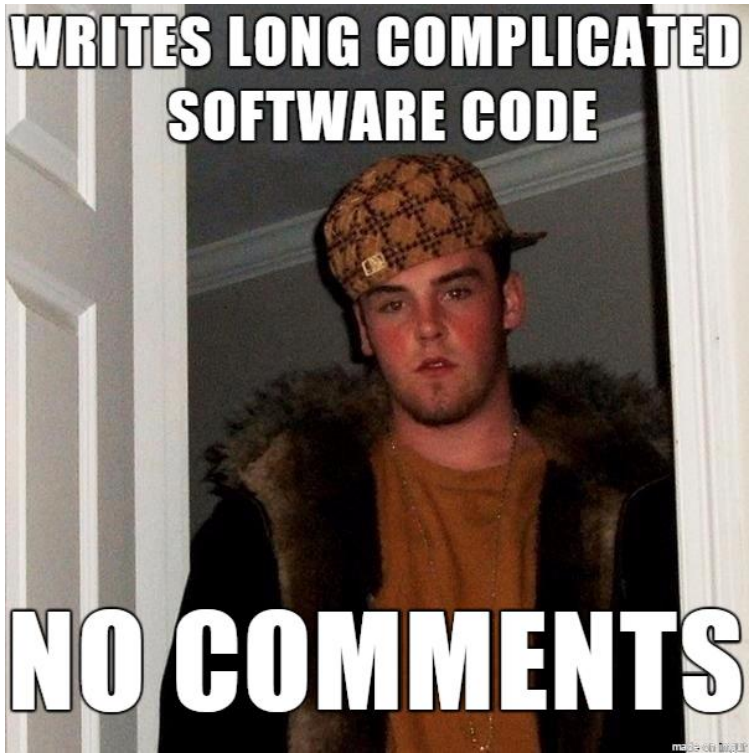
- Suppose you use: public String name;
- You have Person objects throughout the U of M and ICM student records system. ".name" is used in 6,328 different places in the code.
- Now, for some good reason, you must change to
  public char[ ] name; //name is now a char array
- You now need to update the code to make it work…

Why not just public?

- That means: you need to find and change 6,328 other places in the system, in 219 other classes…

Why not just public?

- That means: you need to find and change 6,328 other places in the system, in 219 other classes…
- … and Steve wrote some of them and he was fired in 2012 and nobody understands his code because he didn't use any comments



Why not just public?

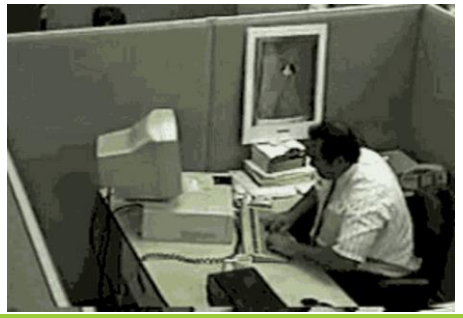
- That means: you need to find and change 6,328 other places in the system, in 219 other classes…
- … and Steve wrote some of them and he was fired in 2012 and nobody understands his code because he didn't use any comments, and nothing is working and…

Why not just public?

- That means: you need to find and change 6,328 other places in the system, in 219 other classes…
- … and Steve wrote some of them and he was fired in 2012 and nobody understands his code because he didn't use any comments, and nothing is working and…

Why not just public?

- That means: you need to find and change 6,328 other places in the system, in 219 other classes…
- … and Steve wrote some of them and he was fired in 2012 and nobody understands his code because he didn't use any comments, and nothing is working and…



Why not just public?

- If it was private, you could just modify getName() and setName() and that's it
- Maintainability is by far the most important thing!

Scope and the this keyword

- In an instance method, if you have a local variable (e.g. a parameter) with the same name as an instance variable → the local one will be used

Scope and the this keyword

- In an instance method, if you have a local variable (e.g. a parameter) with the same name as an instance variable → the local one will be used

- You can use the keyword this to represent the current instance of the object (the instance on which the method was called)

Scope and the this keyword

- In an instance method, if you have a local variable (e.g. a parameter) with the same name as an instance variable → the local one will be used

- You can use the keyword this to represent the current instance of the object (the instance on which the method was called)

- this can be used to specify that you want to access the instance variable for the object that the message was sent to

Scope and the this keyword

- Example:

```
public void setAge (int age)
{
    this.age = age;  //this. is necessary to disambiguate
}
```

Scope and the this keyword

- Example:

```
public void setAge (int age)
{
    this.age = age;  //this. is necessary to disambiguate
}
```

Just age refers to the parameter in this context

age of the object the instance method was called on

Scope and the this keyword

- Example 2:

```
public void setSameAgeAs (Person other)
{
    this.age = other.age;  //this. is optional here
}
```

Scope and the this keyword

- Example 2:

```
public void setSameAgeAs (Person other)
{
    this.age = other.age;  //this. is optional here
}
```

There is no “conflict” here, nothing to disambiguate. You can use this if you want, but it’s not necessary in this case.

The toString() method

- toString() is a very useful instance method, which you should always try to supply using this signature:

```
public String toString()
```

The toString() method

- toString() is a very useful instance method, which you should always try to supply using this signature:

```
public String toString()
```

- This method is automatically called by Java to get a String representation of your object (called when printing or concatenation is needed with your object)

The toString() method

- When you define your toString() method for your object, you control how your object will be displayed as a String

```
public String toString() {
    return name + "(" + age + ")";  //e.g. Jane (42)
}
```

The toString() method

- Once toString() is defined, you can now get readable results from, for example:

```
System.out.println(jane + " and " + john);
```

The toString() method

- Note that there is always a toString() method → if you don't supply one, Java uses a default one
  - However in most cases it will not give you a useful String → try it!

Constructors

- A constructor is a special method that is used to instantiate (create an instance of) an object
- We normally use it to initialize the instance variables
- We can also use to do any kind of processing that needs to be done when the object is created (input, output, calculations, creation of other objects, initialization of GUI panels, etc.)

Constructors

- A constructor is a special case of an instance method:
  - no return type at all (not even void)
  - it must have the exact same name as the class

- It's run automatically when an object instance is created (by new <ClassName>(…))

Defining a constructor

- Syntax:

```
public Person(String name, int age)
{
    //instructions to do during object instantiation
}
```

Defining a constructor

- Syntax:

public makes it accessible outside of this class

```
public Person(String name, int age)
{
    //instructions to do during object instantiation
}
```

Defining a constructor

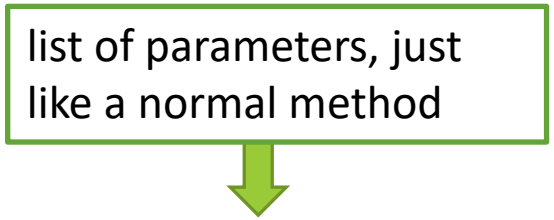
- Syntax:

name of the constructor:
same as class name

```
public Person(String name, int age)
{
    //instructions to do during object instantiation
}
```

Defining a constructor

- Syntax:




```
public Person(String name, int age)
{
    //instructions to do during object instantiation
}
```

Defining a constructor

- Syntax:

```
public Person(String name, int age)
{
    //instructions to do during object instantiation
}
```

- Although it's possible to have a private constructor, we normally make them public

Defining a constructor

- You can define multiple constructors, as long as they have different signatures (lists of parameters)

```
public class Person{
    public String name;
    public int age;

    public Person(){
        name = "Newborn";
        age = 0;
    }

    public Person(String name, int age){
        this.name = name;
        this.age = age;
    }
}
```

2 different constructors with different parameters

Defining a constructor

- You can define multiple constructors, as long as they have different signatures (lists of parameters)

```
public class Person{
    public String name;
    public int age;

    public Person(){
        name = "Newborn";
        age = 0;
    }

    public Person(String n, int a){
        name = n;
        age = a;
    }
}
```

Note: You could also use different parameter names to just avoid the conflicts, so that 'this.' isn't required

Constructing new objects

- If any constructors are supplied, the correct parameters for one of them must be used when creating an instance:

```
Person john = new Person("John", 29); //2nd one
Person newborn = new Person( ); //1st one
Person you = new Person(0); //error → this
                            //constructor does not exist!
```

Constructing new objects

- When a constructor does not initialize the instance variables, they just keep their default value

- Default value depends on type, as seen before (either 0, 0.0, false, '\0000', or null)

Default constructors

- A default constructor is provided by Java in case no constructor is defined in a class

```
public NameOfTheClass() { }
```

- This default constructor does not do anything except instantiating the object

Default constructors

- A default constructor is provided by Java in case no constructor is defined in a class

```
public NameOfTheClass() { }
```

- This default constructor does not do anything except instantiating the object

- Note that this default constructor disappears as soon as one constructor is defined in the class (no matter the list of parameters it uses)

Default constructors

- Example:

```
//in Person.java file:
public class Person {
    public String name;
    public int age;
}
```

```
//in Test.java file
public class Test{
    public static void main (String[] args) {
        Person p = new Person();
    }
}
```

Default constructors

- Example:

```
//in Person.java file:
public class Person {
    public String name;
    public int age;
}
```

```
//in Test.java file
public class Test{
    public static void main (String[] args) {
        Person p = new Person();
    }
}
```

- This compiles perfectly, no errors → default constructor is there to instantiate the Person object

Default constructors

- Example 2:

```
//in Person.java file:
public class Person {
    public String name;
    public int age;

    public Person(String n, int i){
        //statements here
    }
}
```

```
//in Test.java file
public class Test{
    public static void main (String[] args) {
        Person p = new Person();
    }
}
```

Default constructors

- Example 2:

```
//in Person.java file:
public class Person {
    public String name;
    public int age;

    public Person(String n, int i){
        //statements here
    }
}
```

```
//in Test.java file
public class Test{
    public static void main (String[] args) {
        Person p = new Person();
    }
}
```

- This does not compile anymore → no default constructor!

  cannot find symbol constructor Person()

Mutable objects

- An object whose contents can be changed after the object is constructed is a mutable object.
- That is, it has a setter (or mutator) method, or any other method that changes the value of an instance variable.
- Arrays are also mutable objects because their contents can be modified after we create them.

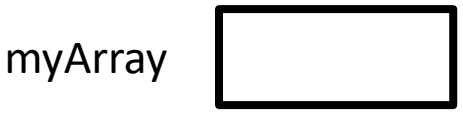
Mutable objects

- When we pass a mutable object to a method, the method can change the contents of the object.
- This change is “permanent”: that is, the object is changed after the method ends.
- In Java, this only happens with (mutable) objects, not with primitives.
- Here’s an example using arrays.

Mutable objects

- Example:

int[] myArray; // myArray is a reference, no object yet…



Mutable objects

- Example:

```
int[] myArray;              // myArray is a reference
myArray = new int[3]; // the array is the object
```

Mutable objects

- Example:

```
int[] myArray;              // myArray is a reference
myArray = new int[3]; // the array is the object
```


```
public static void f(int i, int[] arr) {
  i = -1;
  arr[0] = -1;
}
```

Mutable objects

- Example:

```
int myInt = 0;                // myInt is not a reference
int[] myArray;                // myArray is a reference
myArray = new int[3]; // the array is the object
// what happens when we call: f(myInt, myArray); ?
```

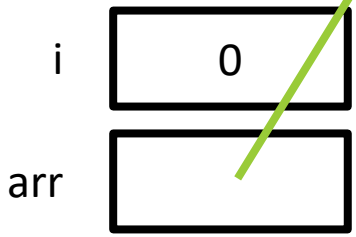

```
public static void f(int i, int[] arr) {
  i = -1;
  arr[0] = -1;
}
```

Mutable objects

- Example:

```
int myInt = 0;                // myInt is not a reference
int[] myArray;                // myArray is a reference
myArray = new int[3]; // the array is the object
f(myInt, myArray);
```


```
public static void f(int i, int[] arr) {
  i = -1;
  arr[0] = -1;
}
```



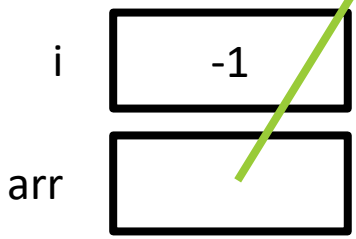
At the start of the call to the method f()

Mutable objects

- Example:

```
int myInt = 0;                // myInt is not a reference
int[] myArray;                // myArray is a reference
myArray = new int[3]; // the array is the object
f(myInt, myArray);
```


```
public static void f(int i, int[] arr) {
  i = -1;
  arr[0] = -1;
}
```



At the end of the call to the method f()

Mutable objects

- Example:

```
int myInt = 0;                // myInt is not a reference
int[] myArray;                // myArray is a reference
myArray = new int[3]; // the array is the object
f(myInt, myArray);
```

- Even after the method ends, the change made to the contents of the mutable object still remains.

Immutable objects

- An object whose contents **cannot** be changed after the object is constructed is an immutable object.
- That is, it has **no** setter (or mutator) or equivalent methods, and all instance variables are private.
- Immutable objects are preferred where possible, because their contents are always predictable.
- Don't write setter methods just because you can. Only write them if you need them.

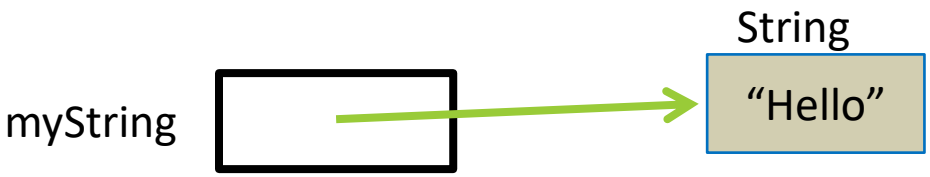
Strings are immutable

- Every String is immutable: once it's created, you cannot change its value
- That means, every time you "modify" the value of a String variable, what actually happens, behind the scenes:
  - A new String object is created, and the new reference to it is returned

Strings are immutable

- Example:

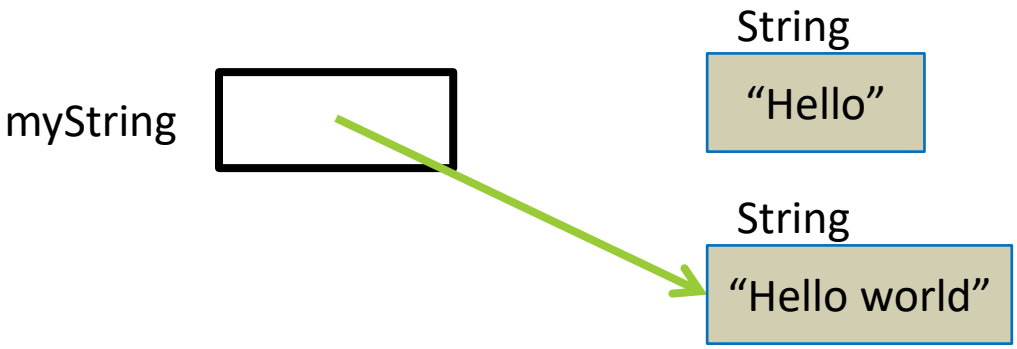
String myString = “Hello”;



Strings are immutable

- Example:

String myString = “Hello”;
myString = myString + “ world”;



- You are never modifying a String in place, you always get a new one → String is immutable

Class variables and methods

- You can create variables and methods which do not refer to any one specific instance (e.g. one actual Person object), but belong to the class as a whole

- We call those:
  - class variables
  - class methods

- To create them, we need to use the static keyword (yes, the one we saw in the first week of classes → finally we learn what it means!)

Class variables and methods

- Example:

```
public class Person{
    //instance variables - one per object created
    private String name;
    private int age;

    //class variable - only one for the whole class
    private static int population = 0;

    //class method - cannot be applied to an object
    public static int census() { return population; }

    //remember to add population++ to all constructors!
}
```

Class variables and methods

- Example:

```
public class Person{
    //instance variables - one per object created
    private String name;
    private int age;

    //class variable - only one for the whole class
    private static int population = 0;

    //class method - cannot be applied to an object
    public static int census() { return population; }

    //remember to add population++ to all constructors!
}
```

class variables are initialized at the same time they are declared

Class variables and methods

- Example:

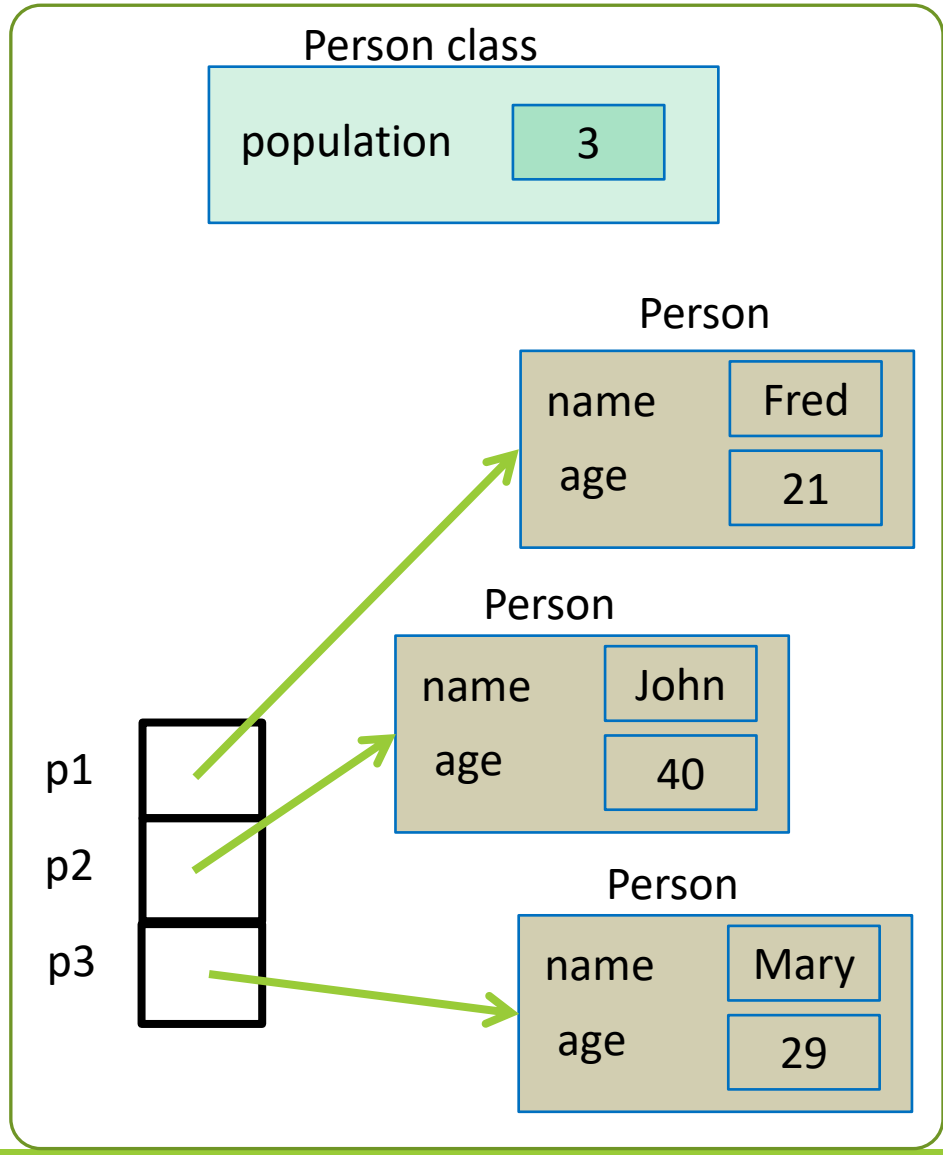
```
//Constructor could look like this:
public Person(String n, int a)
{
    name = n;
    age = a;
    population++;
}
```

Class variables and methods

- Example:

```
//Constructor could look like this:
public Person(String n, int a)
{
    name = n;
    age = a;
    population++;
}
```

In a main, somewhere else, you create 3 objects of type Person, and this is what you get in the memory:



Class variables and methods

- You can also have class constants → just add final:

```
public class Person{
    //instance variables - one per object created
    private String name;
    private int age;

    //class constant
    public static final boolean NEEDS_TO_EAT = true;
}
```

Calling methods

- Instance (non-static) methods:
  someObject.method(…)

- Class (static) methods:
  ClassName.method(…)

- Any method from inside the same class:
  method(…) //the same class is assumed
  //It can either be static or not.
  //Same as this.method(…) for instance methods
  //Same as <ThisClass>.method for static methods

Calling methods

- Example, trying to call instance and class methods of Person from a main in another class:

```
Person john = new Person("John", 55);

john.haveBirthday();

int totalNbPeople = Person.census();
```

Methods you have used

- System.out.println(...)
  - System is a class (google "Java System class")
  - out is a public static variable in that class
    - Its type is PrintStream
  - println is a public instance method in the PrintStream class.
  - You're sending a println message to the object referred to by the static out variable in the System class.

Methods you have used

- Math.sqrt(…) is a public static method in Math
- Math.PI is a public static constant in Math
- main is a public static void method in your class

Comparing objects

- Comparing Object variables using == or != is not usually what you want to do
  - It only compares the references
  - It does not look inside the objects, to check if the instance variables have the same values (which is normally what you're trying to do)

Comparing objects

- Standard methods for comparing the actual data inside Objects:
  - object1.equals(object2) //gives a boolean
  - object1.compareTo(object2) //gives an int
    - Gives a negative value if object1 is “smaller”
    - Gives a positive value if object1 is “larger”
    - Gives a zero if they are “equal”
    - For Strings, this checks “alphabetical order”

Comparing objects

- Similarly to the toString() instance method, you should write these methods for your own objects. Other methods can use them.

- There are default ones, but they're not useful.

Places to use objects

- You can use an object type anywhere you can use any other type
  - as a parameter to a method
  - as the return type of a method
  - as the elements of an array
  - as an instance variable in another object
  - etc. etc.

Places to use objects

- Just remember that it's always a reference to an object that is passed/returned/stored/etc.
- This was done many times in COMP 1010 with Strings and arrays (which are objects)