Topic 2.1: Objects

Learning Goals (Week 2):

- Write simple to moderately complex classes.
- Constructors, instance variables and instance methods, the this keyword
- **Class variables and class methods**
- Compile & run a Java code with multiple files in the same directory.
- Use instances of user-defined classes
- **User-defined classes in other user-defined class and in main methods**
- **Explain how and why the concept of encapsulation is useful**
- **How Encapsulation is achieved via accessors / mutators.**
- Understand object references and use them appropriately in code,
- Deep versus shallow object copies.

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

] this is our focus, we may talk about the others later

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

     public void haveBirthday() {
          age++;
     }
}
```

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
    - notice this isn't the same as **same file.**

HOPEFULLY, you put one class per file so we can "pretend" it is the same but it is not.

- data variables like name, and age **should** be private

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

	public void haveBirthday() {
		age++;
	}
}
```

Access Modifiers: General 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. We will see other cases later

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

	public void haveBirthday() {
		age++;
	}
}
```

Principle of Encapsulation

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

- **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

- 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:

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

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}
}
```

ifunny.co

Why not everything public?

- Suppose you use: `public String name;`
  - You have Person objects throughout the U of M 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 everything public?

- Suppose you use: `public String name;`
  - You have Person objects throughout the U of M 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…
  - If it was private, you could just modify getName() and setName() and that's it (hiding the implementation behind the method names)
    - **maintainability is key**

Under the hood terminology

- Words we've used but haven't fully defined...until now
  - **Declaration**
    - Telling our Java program we intend to use a variable.
  - **Creation/Instantiation**
    - Giving a declared variable a value, can be done in the same line as the declaration
  - **Runtime Memory/Runtime Stack**
    - A special segment of computer memory our program uses to actually execute our program
  - **Heap Memory**
    - A special segment of computer memory our program has access to when it needs extra memory for certain things, such as Objects (Arrays, user-defined, etc).
  - **Allocate/Reserve Memory**
    - Grabbing a chunk of free memory from heap memory to store our object value(s)
  - **Free/Release Memory**
    - When we are done using reserved memory we 'free' it.
    - Letting it go so that the memory can be used by another part of the program if it needs it
    - Think of it like clearing the contents of a column or row in excel

Mutability

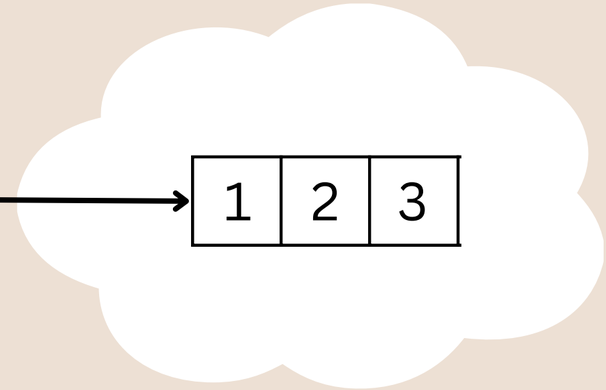
- An object whose contents can be changed after the object is constructed is a mutable object.
- That is, it has a setter (or mutator) method, or any other method that changes the value of an instance variable.
- Arrays are also mutable objects because their contents can be modified after we create them.
- This means we can change some values without using the ‘new’ keyword again, or having to copy over all the information to a new object with an updated value
- Objects that are immutable are ‘remade’ each time their value changes.
    - As in the memory the object took up is released and then reallocated/reserved

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.
  - The change happened in **heap memory**
  - since our whole program has access to this memory, the change in there is seen everywhere

```
main() {
    // create an int[] arr 1, 2, 3
    changeIt(arr);
}

changeIt(int[] a) { // mutable objects are passed by reference
   a[0] = -1;
}
```

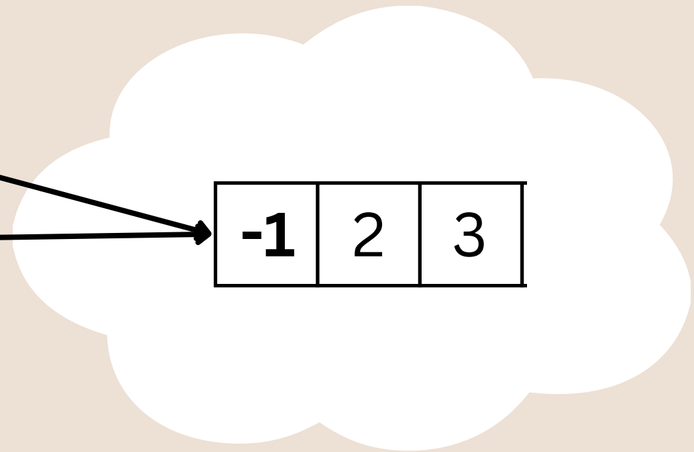


Mutable Objects

- 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.
  - The change happened in **heap memory**
  - since our whole program has access to this memory, the change in there is seen everywhere

```
main() {
    // create an int[] arr 1, 2, 3
    changeIt(arr);
}

changeIt(int[] a) {
   a[0] = -1;
}
```



- 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.
- **Primitives are naturally immutable (i.e. int, float, boolean, char, double, etc.)**

Immutable Objects

```
main() {
    int x = 50;
    changeIt(x);
    // x is still 50 here
}

changeIt(int y) {
   y = 100;
}
```

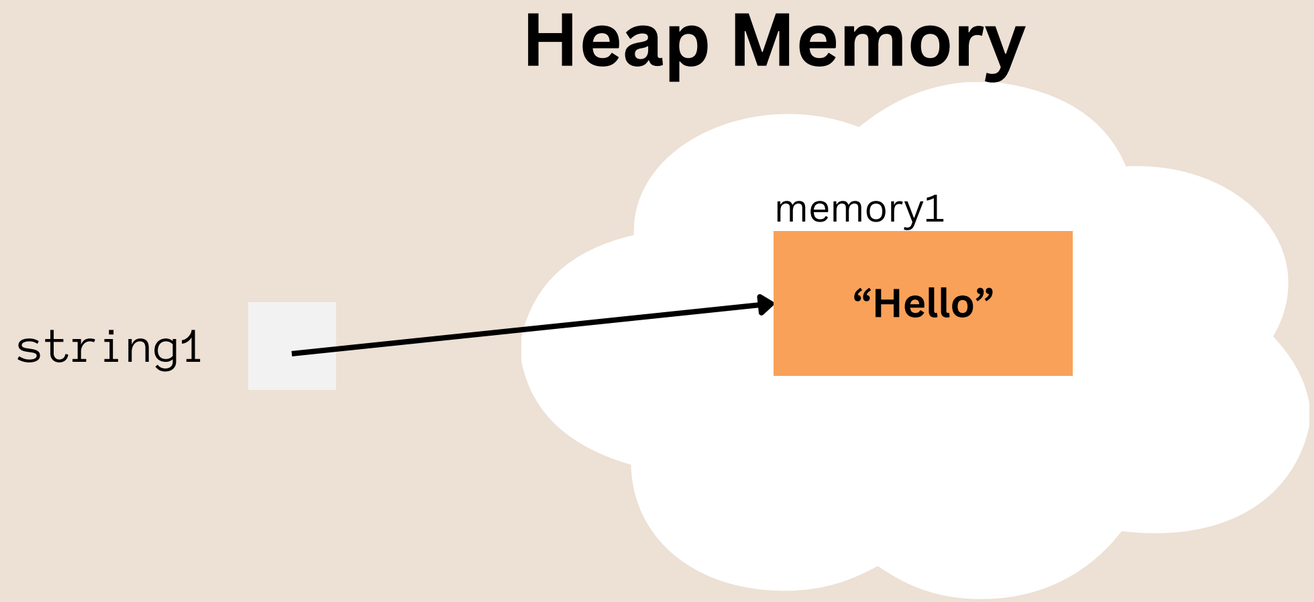
- Primitives and Immutables are passed **by-value.** This means when the method ends, the changes disappear along with it.
- This comes down to where and how these values are stored versus the Mutable Objects.
- Without going too deep into the complexities, primitives are stored in **runtime memory** and are passed by value, when the method ends, the method-made changes are cleaned up and disappear

Mutability Exception: String

- 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

```
String string1 = “Hello”;
```

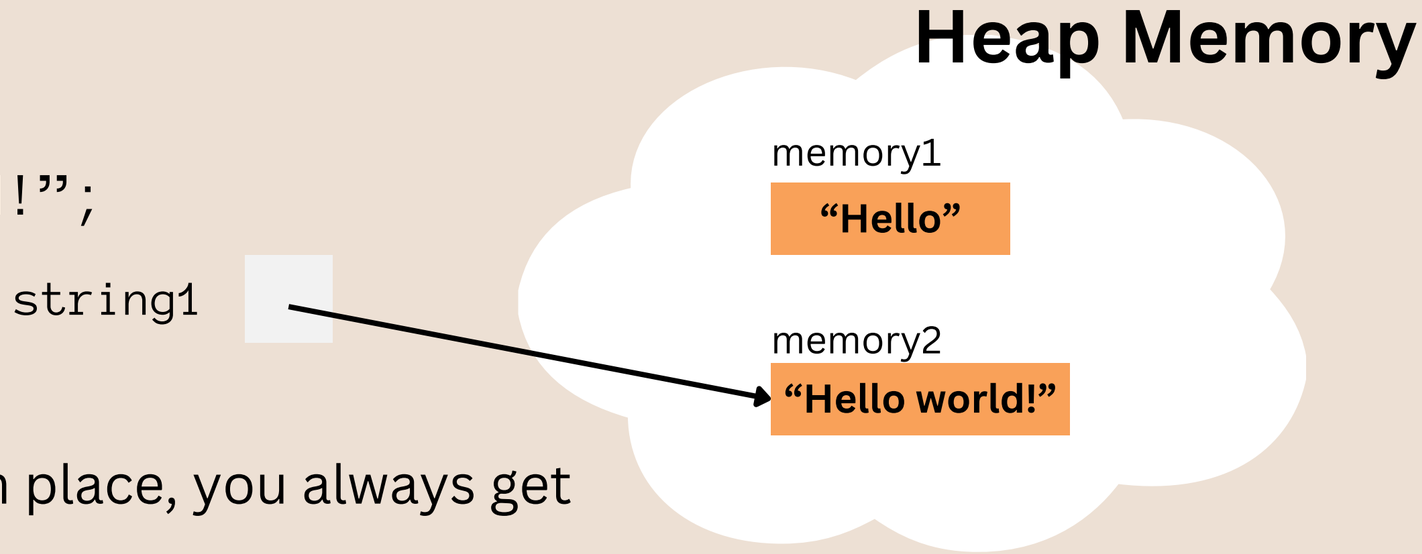


Mutability Exception: String

- 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

```
String string1 = “Hello”;
string1 = string1 + “ world!”;
```



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

Class Variables/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 (all created Person objects can access it)

- 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!)

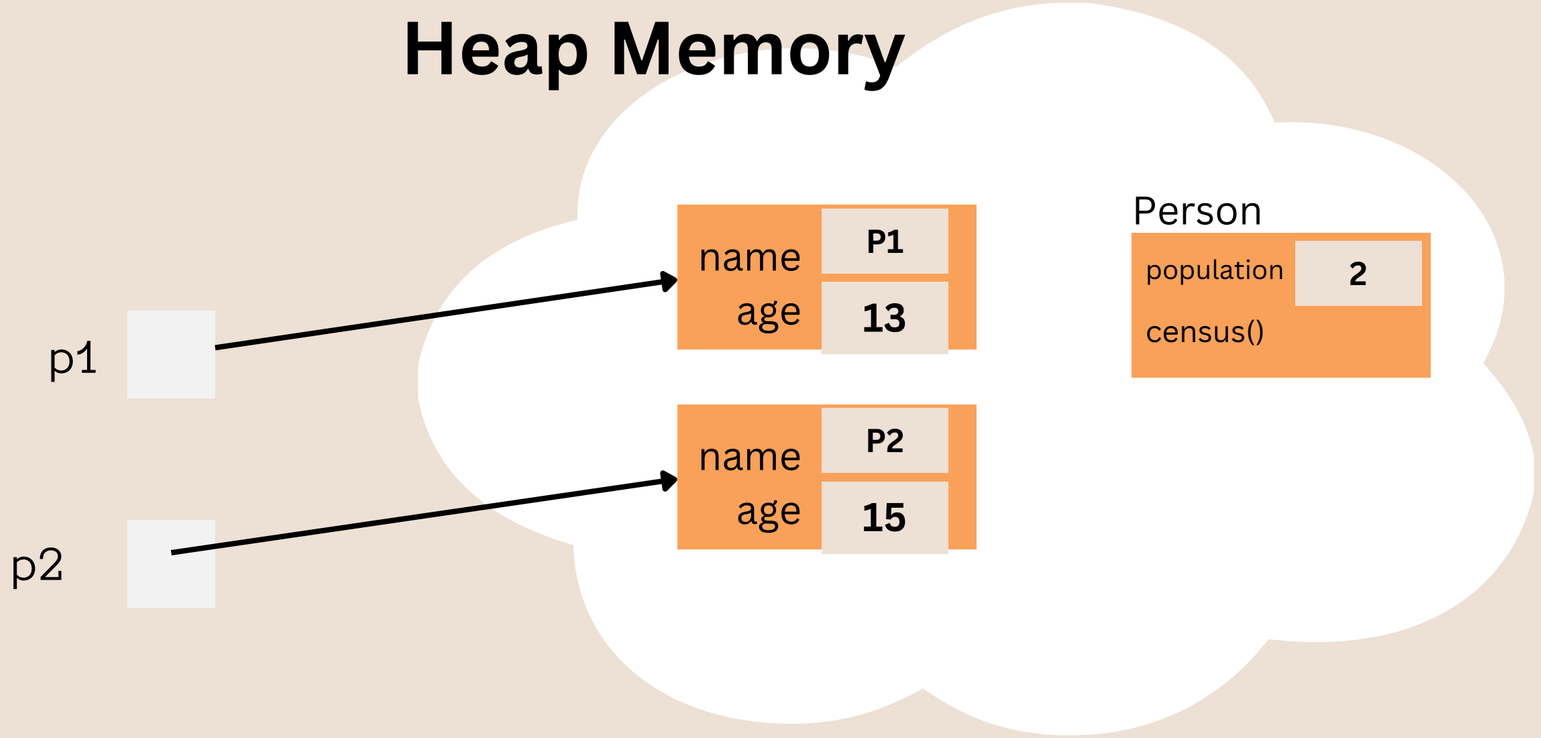
Class Variables

```
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; // set outside the constructor

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

     public Person() {
         name = “newborn”;
         age = 0;
         population++;  //remember to add population++ to all constructors!
    }
}
```

Class Variables: Code in Memory



Class Variables

- 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;
	public static final int AGE_OF_MAJORITY = 18;
}
```

Calling Methods: A summary

- 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/non-static methods
    - //Same as <ThisClass>.method for static methods

Calling Methods: An example

```
Person hermione = new Person(“Hermione”, 19);

hermione.haveBirthday(); // age up the hermione object, an instance of the Person class

int totalNbPeople = Person.census(); // just the 1 for now, called on the class itself

Person harry = new Person(“Harry”, 19);

totalNbPeople = Person.census(); // now it’s 2
```

Class Level Information you have seen before

- 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.

- 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)

```
Person p1 = new Person("Melika", 21);
Person p2 = new Person("Melika", 21);
System.out.println(p1 == p2); // false because the references are different
Person p3 = p1;
System.out.println(p1 == p3); // true because the reference is the same
```

- Not particularly helpful

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 these methods check for case-sensitive equals or a comparison of “alphabetical order”

- Similar to the toString() instance method, you should write these methods (equals & compareTo) for your own objects. Other methods can use them.
  - There are default ones, but they’re not useful.
  - When it comes to broad/complex use, these methods do have some differences to toString() but for our simple purposes, our own equals()/compareTo() methods will be sufficient

Using Objects: When? Where? Who?

- 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.

- 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)

Pause & Practice

- Consider a library system:
  - What type of data would we store in a Book object?
  - What about a LibraryMember class?
    - Consider the types of data you might want to store

- Consider a vehicle service system:
  - What might you put in a Vehicle class?
  - When someone goes to a service center they book an appointment for their vehicle, what kind of data would you store in a ServiceAppointment object?

- Start looking for objects everywhere! Consider what data would be useful to clump together and how it might be used!