In this section, we will cover using Classes to help organize our code and to store complex collections of data.

Classes

CSCI 1250 Study Guide

Schneider, Michael Joseph

Class Structure

```
public class ClassName
{
        //Class attributes
        private int dataPoint;

        //Class Constructors
        public ClassName()
        {
                setDataPoint(0);
        }

        //Class methods
        public void setDataPoint(int dataPoint)
        {
                this.dataPoint = dataPoint;
        }

}
```

Attributes/Fields

The attributes, or fields, of a Class define the data that a Class object will keep track of. This data may be used internal for calculations or be accessed externally by calling the attribute's getter method.

Often, we list attributes as private. This guarantees that our class is the only place where the attributes are modified and access is limited to getter/setter methods.

Constructors

A constructor is a unique method, in that it has no listed return type. Instead of returning data, a constructor returns the address of where in memory an instantiated Class object has been stored.

Constructors **MUST** guarantee that all class attributes have been initialized. It is department policy to use setter, or mutator, methods to initialize the values of class attributes.

You can have multiple constructors, but each constructor must have a different set of parameters. If an attribute does not have a corresponding parameter, it must be initialized with a default value.

Methods

By convention, the Class's methods are listed after the Class's attributes and its constructors. The methods are the setters, getters, and any additional calculations the class needs to be able to perform.

Class Example – Student Class

```java
public class Student
{
	private int age;
	private String name;

	//Constructor
	public Student(String name, int age)
	{
		setName(name);
		setAge(age);
	}

	//Setter for name attribute
	public void setName(String name)
	{
		this.name = name;
	}

	//Setter for age attribute
	public void setAge(int age)
	{
		this.age = age;
	}

	//Getter for name attribute
	public String getName()
	{
		return name;
	}

	//Getter for age attribute
	public int getAge()
	{
		return age;
	}
}
```

this.

Since the parameter's name matches the attribute's name, "this." tells the compiler to refer to the class attribute, not the parameter.

-Official Java Tutorial-

https://docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html

Shallow Copy

```java
Student john = new Student(“John”, 17);
Student john2 = john;
```

Shallow Copy

```java
System.out.println(“John’s Age = ” + john.getAge());
System.out.println(“John2’s Age = ” + john2.getAge());
```

Command Window Output

```
John’s Age = 17
John2’s Age = 17
```

```java
john.setAge(32);
```

Modify John’s Data

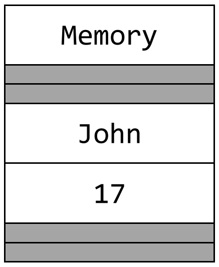
```java
System.out.println(“John’s Age = ” + john.getAge());
System.out.println(“John2’s Age = ” + john2.getAge());
```

Command Window Output

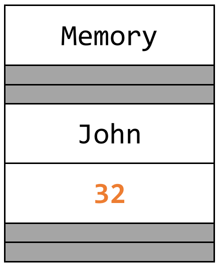
```
John’s Age = 32
John2’s Age = 32
```

In the above example, a shallow copy is used to copy the Student object john. A shallow copy only passes the memory address of an object, not its data. This means that both variables will “point” to the same location in memory. So if either variable is modified, the other variable is also modified!

```java
Student john = new Student(“John”, 17);
Student john2 = john;
```



```java
System.out.println(“John’s Age = ” + john.getAge());
System.out.println(“John2’s Age = ” + john2.getAge());
```



```java
john.setAge(32);
```

```java
System.out.println(“John’s Age = ” + john.getAge());
System.out.println(“John2’s Age = ” + john2.getAge());
```

Deep Copy – Copy Constructor

A deep copy, unlike a shallow copy, will copy the data points of one variable into a NEW location in memory. This means that when the original is modified, the copy remains unchanged. In order to perform a deep copy, a class needs a Copy Constructor. A Copy Constructor will initialize an object's attributes to have the same values as the original.

```java
public Student(Student copy)
{
      setAge(copy.age);
      setName(copy.name);
}
```

Copy Constructor

The copy constructor has only one parameter, a Class object. The passed object's attributes will be copied (even private attributes can be directly accessed in the copy constructor)

Below is the previous shallow copy example, but now modified to use a deep copy.

```java
Student john = new Student("John", 17);
Student john2 = new Student(john);
```

Deep Copy

```java
System.out.println("John's Age = " + john.getAge());
System.out.println("John2's Age = " + john2.getAge());
```

Command Window Output

```
John's Age = 17
John2's Age = 17
```

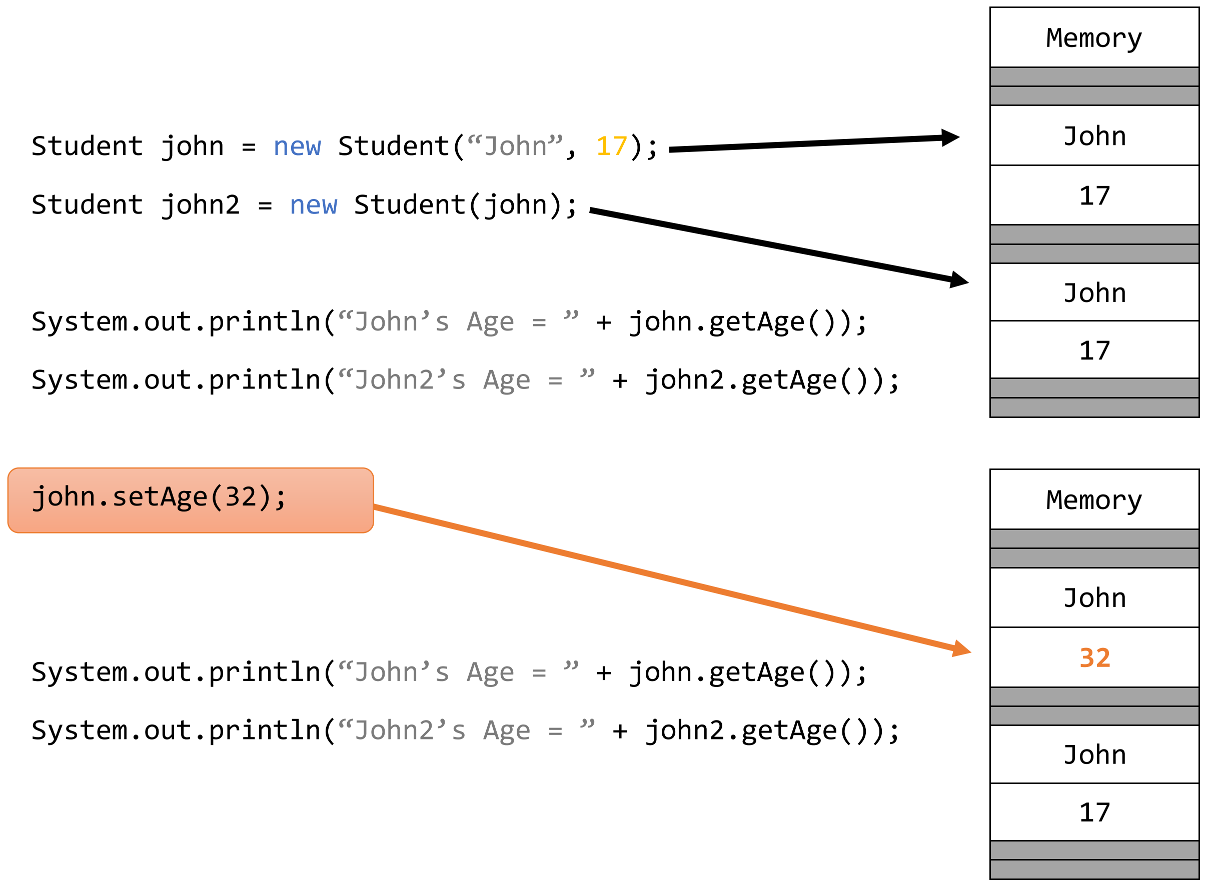
```java
john.setAge(32);
```

Modify John's Data

```java
System.out.println("John's Age = " + john.getAge());
System.out.println("John2's Age = " + john2.getAge());
```

Command Window Output

```
John's Age = 32
John2's Age = 17
```



```
Student john = new Student("John", 17);
Student john2 = new Student(john);

System.out.println("John's Age = " + john.getAge());
System.out.println("John2's Age = " + john2.getAge());
```

```
john.setAge(32);

System.out.println("John's Age = " + john.getAge());
System.out.println("John2's Age = " + john2.getAge());
```

Default toString() Method

The toString() method allows a programmer to represent an object as a String. By default, all Class's have a toString() method provided by the Java language, which will return the name of an object's class and it's memory address.

```
public class Student()
{
        private String name;
        private int age;

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

}

public class Driver()
{
        public static void main(String[] args)
        {
                Student demo = new Student("Bob",5);

                System.out.println(demo);
                System.out.println(demo.toString());
        }
}
```

No toString() method

The Student class does not contain a toString() method. Without a written toString() method, the Java Compiler will use the default Java toString() method.

Accessing toString()

The toString() method can be accessed using dot-notation, or in some cases simply by using the variable's name, like with System.out.println(). In either case, the program will print to the screen:

```
Student@19e0bf
Student@19e0bf
```

ClassName@MemoryAddress

Overridden toString() Method

To provide more information, a Class must override the toString() method. The programmer can then define what information will be displayed when an object's toString() is called. We can modify the previous example, by simply adding a toString() method to the end of the program.

```
public class Student()
{
        private String name;
        private int age;

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

        public String toString()
        {
                String info = "Name: " + name + "\nAge: " + age;
                return info;
        }
}

public class Driver()
{
        public static void main(String[] args)
        {
                Student demo = new Student("Bob",5);

                System.out.println(demo);
                System.out.println(demo.toString());
        }
}
```

Overridden toString() method

The Student class defines its own toString() method, which overrides the default Java toString() method. Now when the toString() method is called, it will return a formatted String representing a Student object's information.

Accessing toString()

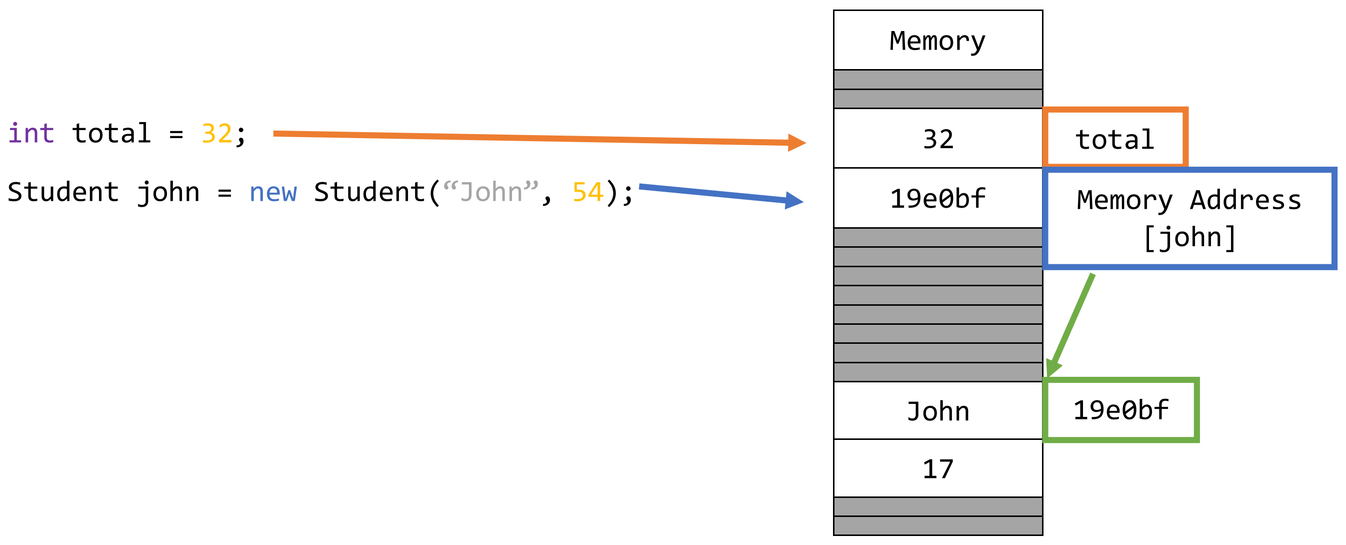
The toString() method can still be accessed in the same manner as the default toString() method was previously accessed. The program now will print the following to the screen.

```
Name: Bob
Age: 5
Name: Bob
Age: 5
```

.equals() Method

In Java, when using “==” the Java compiler compares the actual value of a variable. With data primitives, the variable holds its literal value but with class objects the variable holds the object’s address!

```java
int total = 32;
Student john = new Student(“John”, 54);
```



If we want to compare the actual values of the object’s attributes, we will need to create what is called a .equals(), “dot equals” method. A .equals() method compares the values of two class objects and returns a Boolean (true the objects are equal, false the objects are not equal).

```java
public boolean equals(Student compare)
{
        boolean areEqual = false;
        if(compare.name.equalsIgnoreCase(this.name) && compare.age == this.age)
        {
                areEqual = true;
        }
        return areEqual;
}
```

!!! Strings Are Objects !!!

Remember a String is an object. You cannot use “==” to compare two String objects, you must use one of the String class’s .equals() methods!

`String valA = “first”, valB = “first”;`

`if(valA` ~~`==`~~ `valB)`

Not Allowed!

.equals() Example

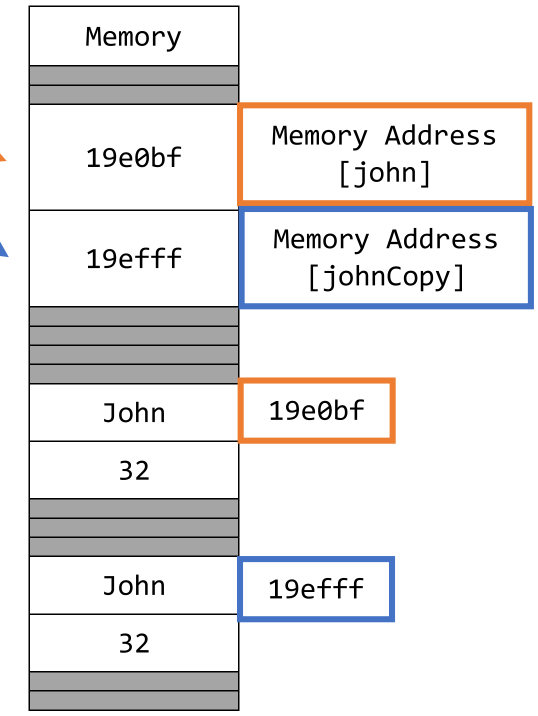
```
Student john = new Student("John", 32);
Student johnCopy = new Student("John", 32);


if(john == johnCopy)    Compares addresses
{
      System.out.println("== returns True!");
}
else
{
      System.out.println("== returns false!");
}


if(john.equals(johnCopy))    Compares attributes
{

      System.out.println(".equals() returns True!");

}
else
{
      System.out.println(".equals() returns false!");
}
```



Command Window Output

```
== returns false

.equals() returns true
```