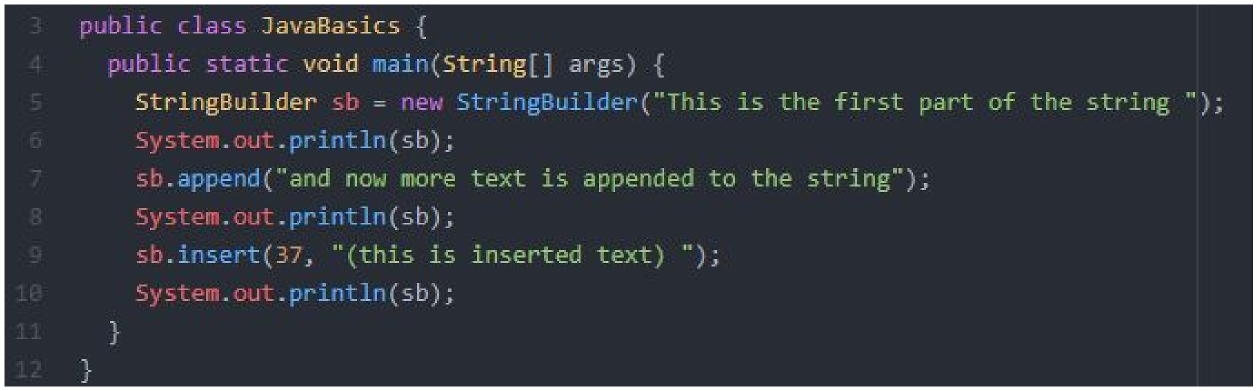
Per Scholas

Java Programming - 02

More on Strings

- StringBuilder

  - Similar to String class, but StringBuilder can append and insert new character sequences



```java
public class JavaBasics {
  public static void main(String[] args) {
    StringBuilder sb = new StringBuilder("This is the first part of the string ");
    System.out.println(sb);
    sb.append("and now more text is appended to the string");
    System.out.println(sb);
    sb.insert(37, "(this is inserted text) ");
    System.out.println(sb);
  }
}
```

Output:

```
This is the first part of the string
This is the first part of the string and now more text is appended to the string
This is the first part of the string (this is inserted text) and now more text is
appended to the string
```

  - StringBuilder objects can be accessed from multiple threads at any given time

  - Read these two web pages on the StringBuilder class:

    - https://docs.oracle.com/javase/tutorial/java/data/buffers.html

    - https://www.geeksforgeeks.org/g-fact-27-string-vs-stringbuilder-vs-stringbuffer/

- StringBuffer

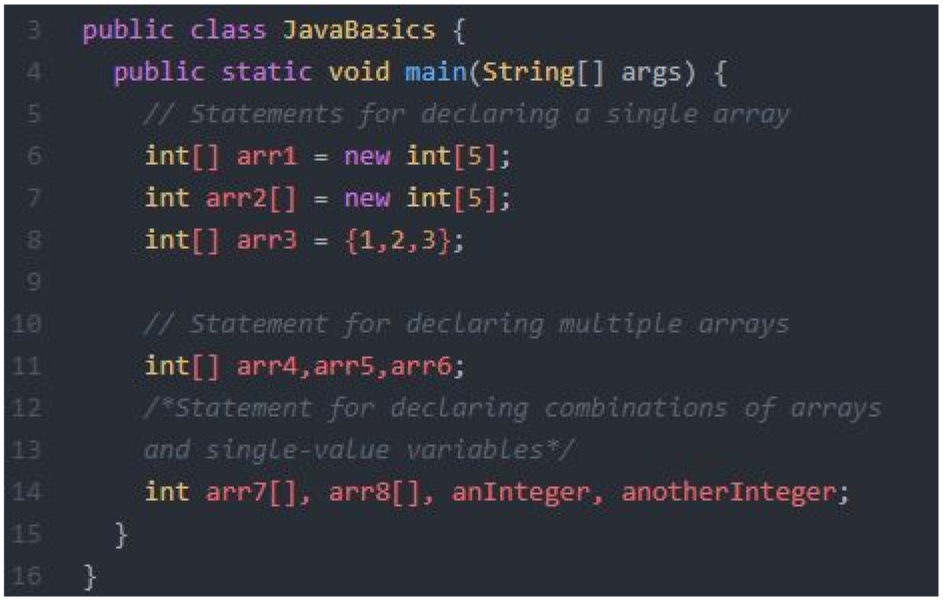
  - Similar to StringBuilder, but is synchronized so it can only be accessed by a single thread at a time

- Read this two web page on the StringBuffer class:
    - https://www.geeksforgeeks.org/stringbuffer-class-in-java/
- Use these web pages as references for the StringBuffer class:
    - https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuffer.html
    - https://www.tutorialspoint.com/java/java_string_buffer.htm

---

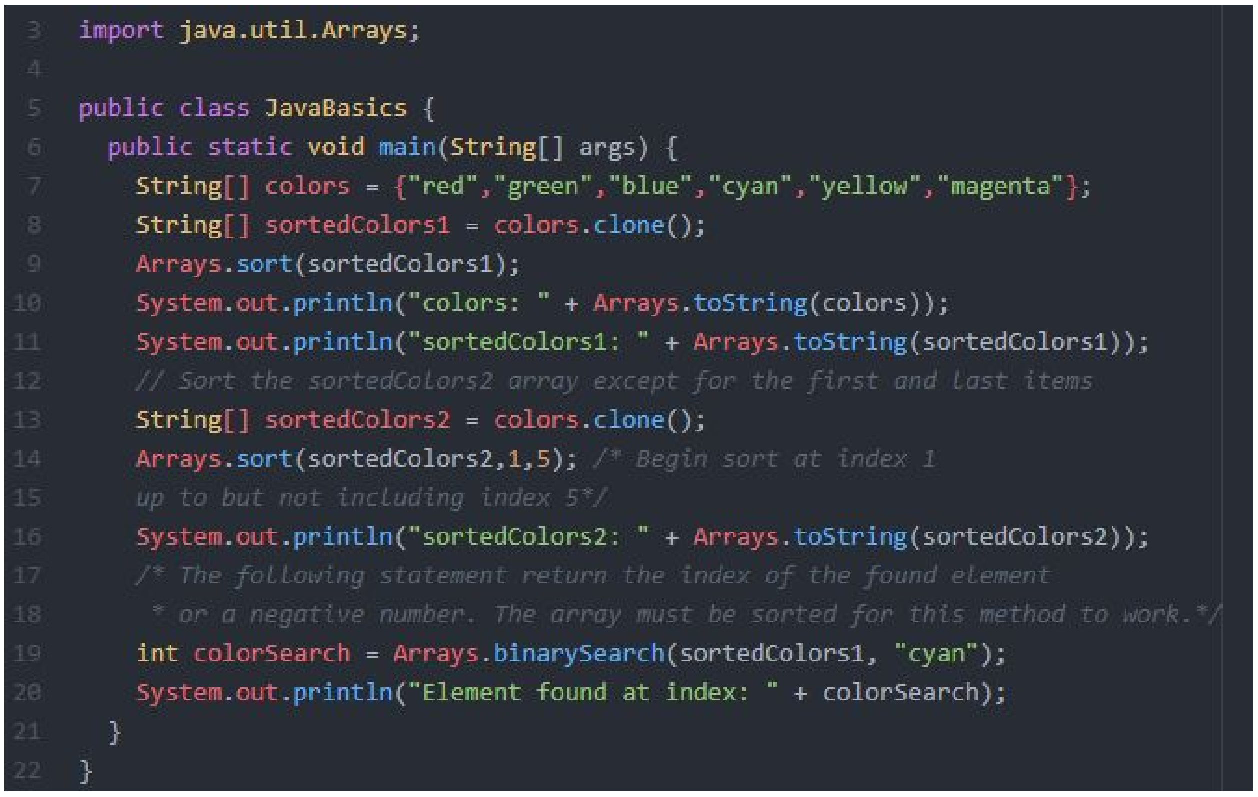
Arrays

- Data structures which store a fixed-size sequential collection of elements
    - Index based - first element of the array is stored at index 0
    - Collection of elements of the same data type
    - Arrays in Java are fixed size - size cannot be changed once the array is created
    - Declaring an array: int[ ] myArray = new int[10];
    - new array of int data type with length of 10 elements.
    - Other ways to declare arrays:



```java
public class JavaBasics {
  public static void main(String[] args) {
    // Statements for declaring a single array
    int[] arr1 = new int[5];
    int arr2[] = new int[5];
    int[] arr3 = {1,2,3};

    // Statement for declaring multiple arrays
    int[] arr4,arr5,arr6;
    /*Statement for declaring combinations of arrays
    and single-value variables*/
    int arr7[], arr8[], anInteger, anotherInteger;
  }
}
```

- Access an array element by it’s index: myArray[0], myArray[1], etc.
- Assign value by referring to an index: myArray[5] = 125; sixth element of array will hold the value 125.
- Read this Array tutorial: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html
- The Arrays class contains several static methods to perform useful functions on arrays such as sort and binary search

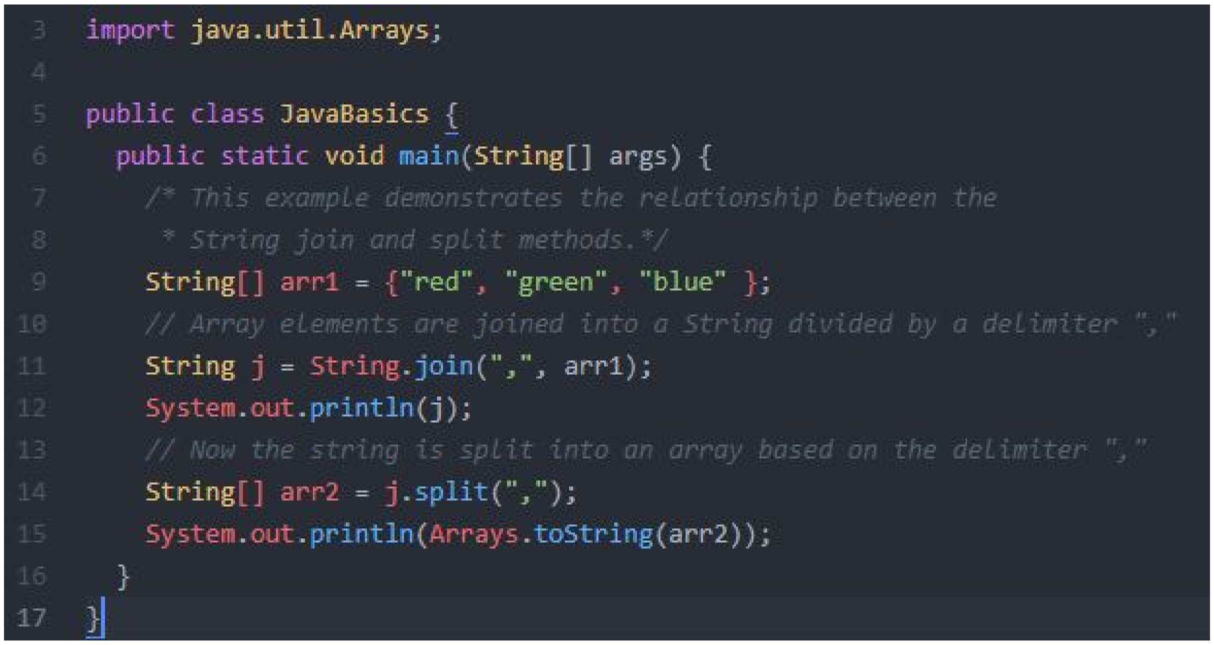


```java
import java.util.Arrays;

public class JavaBasics {
  public static void main(String[] args) {
    String[] colors = {"red","green","blue","cyan","yellow","magenta"};
    String[] sortedColors1 = colors.clone();
    Arrays.sort(sortedColors1);
    System.out.println("colors: " + Arrays.toString(colors));
    System.out.println("sortedColors1: " + Arrays.toString(sortedColors1));
    // Sort the sortedColors2 array except for the first and last items
    String[] sortedColors2 = colors.clone();
    Arrays.sort(sortedColors2,1,5); /* Begin sort at index 1
    up to but not including index 5*/
    System.out.println("sortedColors2: " + Arrays.toString(sortedColors2));
    /* The following statement return the index of the found element
     * or a negative number. The array must be sorted for this method to work.*/
    int colorSearch = Arrays.binarySearch(sortedColors1, "cyan");
    System.out.println("Element found at index: " + colorSearch);
  }
}
```

- Review and be familar with these references on the Arrays class: https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html
- https://www.geeksforgeeks.org/array-class-in-java/

- StringTokenizer & String split( ) Method

- StringTokenizer is a legacy class that is retained for compatibility purposes. However, current use is discouraged. The recommended replacements are the String split( ) method or java.regex.util package
  - https://docs.oracle.com/javase/8/docs/api/index.html?java/util/StringTokenizer.html
  - https://www.geeksforgeeks.org/stringtokenizer-class-java-example-set-1-constructors/
- String split( ) method
  - Splits a string based on a specified delimiter (e.g., “-”) and loads the split elements into an array



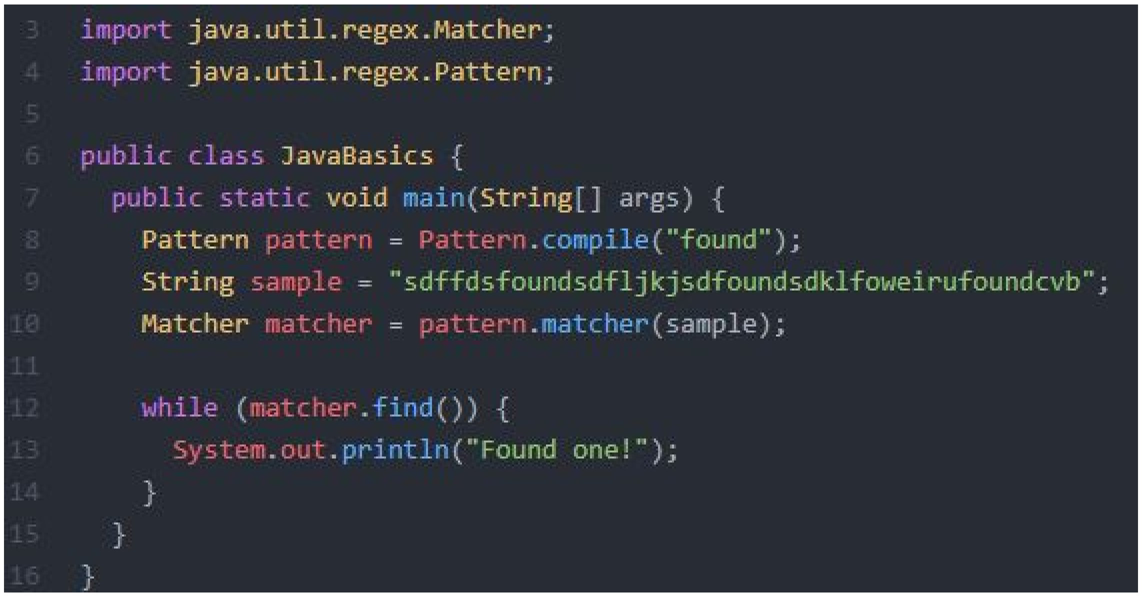
```java
import java.util.Arrays;

public class JavaBasics {
  public static void main(String[] args) {
    /* This example demonstrates the relationship between the
     * String join and split methods.*/
    String[] arr1 = {"red", "green", "blue" };
    // Array elements are joined into a String divided by a delimiter ","
    String j = String.join(",", arr1);
    System.out.println(j);
    // Now the string is split into an array based on the delimiter ","
    String[] arr2 = j.split(",");
    System.out.println(Arrays.toString(arr2));
  }
}
```

Output:
```
red,green,blue
[red, green, blue]
```

  - https://docs.oracle.com/javase/tutorial/essential/regex/pattern.html - (discussion of split( ) method in the lower half of the page)
  - https://www.geeksforgeeks.org/split-string-java-examples/

- Regular Expressions
  - Pattern Class
    - a compiled representation of a regular expression
    - this is the pattern that is being matched against (i.e., the string that is being searched for)
  - Matcher Class
    - the engine that interprets the pattern
    - this is the text that is being searched for the pattern
    - https://docs.oracle.com/javase/tutorial/essential/regex/matcher.html

    An example of the use of the Pattern and Matcher classes:

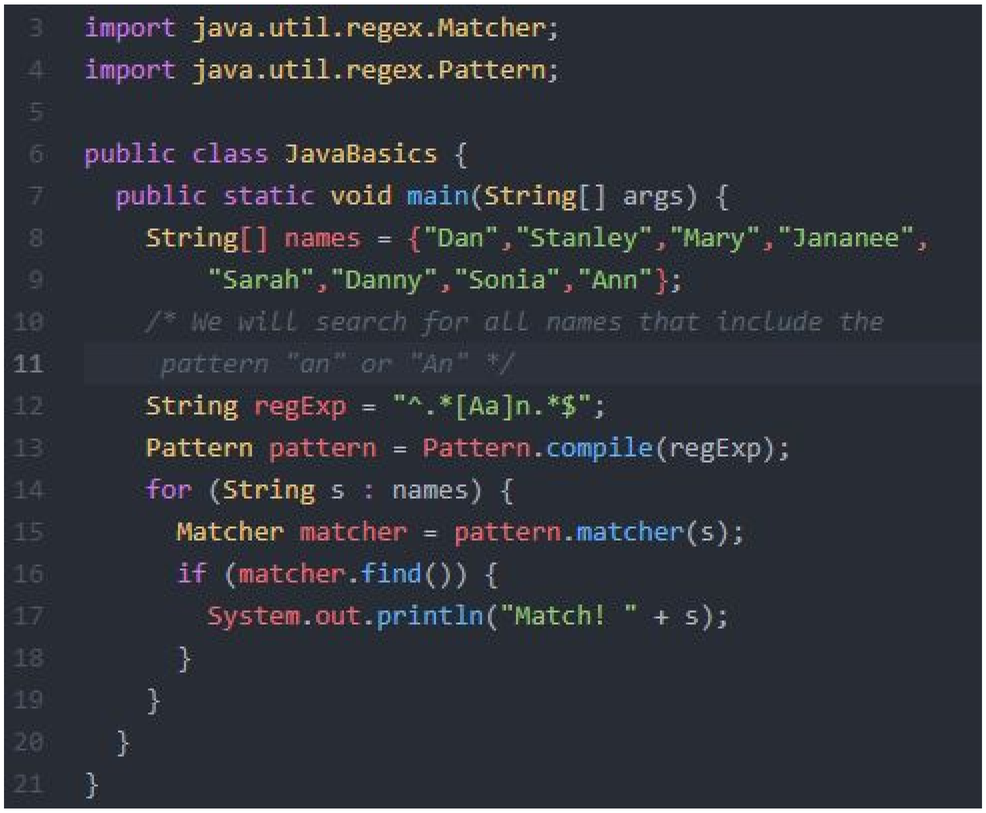


```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class JavaBasics {
  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("found");
    String sample = "sdffdsfoundsdfljkjsdfoundsdklfoweirufoundcvb";
    Matcher matcher = pattern.matcher(sample);

    while (matcher.find()) {
      System.out.println("Found one!");
    }
  }
}
```

Output:

```
Found one!
Found one!
Found one!
```

Another example using a for-loop:



```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class JavaBasics {
  public static void main(String[] args) {
    String[] names = {"Dan","Stanley","Mary","Jananee",
        "Sarah","Danny","Sonia","Ann"};
    /* We will search for all names that include the
     pattern "an" or "An" */
    String regExp = "^.*[Aa]n.*$";
    Pattern pattern = Pattern.compile(regExp);
    for (String s : names) {
      Matcher matcher = pattern.matcher(s);
      if (matcher.find()) {
        System.out.println("Match! " + s);
      }
    }
  }
}
```

```
Output:
Match! Dan
Match! Stanley
Match! Jananee
Match! Danny
Match! Ann
```

- https://regexone.com/
- https://www.tutorialspoint.com/java/java_regular_expressions.htm
- Advanced Challenge - complete the following tutorial: https://docs.oracle.com/javase/tutorial/essential/regex/index.html

Memory Allocation & De-Allocation (New Operator & Garbage Collection)

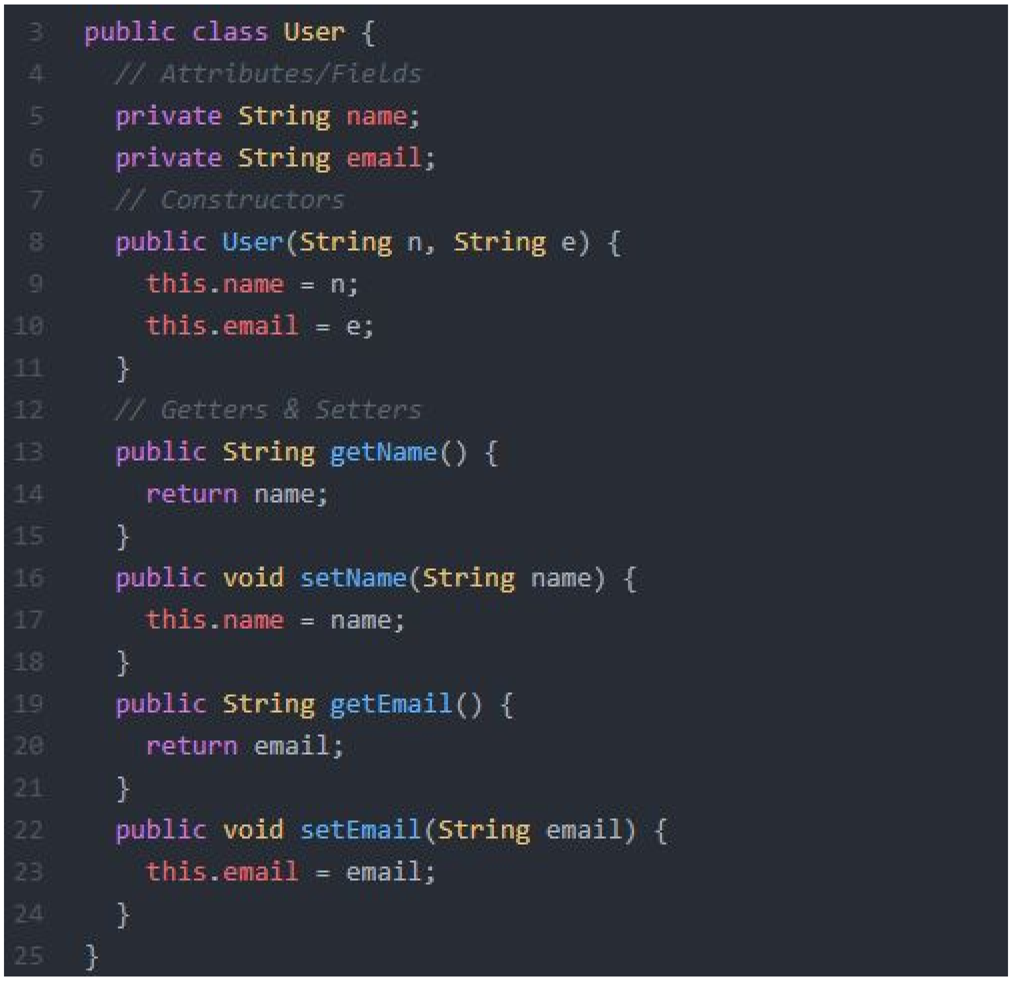
- The “new” operator instantiates a class by allocating memory for a new object and returning a reference to that memory address
- The new operator is responsible for the creation of a new object or instance of a class

- It dynamically allocates new memory in the heap which is created when the JVM starts up and may increase or decrease in size when the application runs. When the heap becomes full, unused variables and objects are automatically removed through a process called garbage collection, thus making room for new objects. There is no explicit need to destroy an object in Java because garbage collection handles de-allocation automatically.
- Memory remains allocated to an object until there are no longer any references to it
- Programs that do not de-allocate memory can eventually crash when there is no memory left in the heap. These programs are said to have memory leaks.
- The finalize( ) method is a protected and non-static method of the java.lang.Object class and thus available to all objects. This method can be used to perform final operations or clean up operations on an object before it is removed from memory.
- Read this section on objects from the Oracle Java tutorial along with the subsequent two pages on creating and using objects: https://docs.oracle.com/javase/tutorial/java/javaOO/objects.html
- https://www.geeksforgeeks.org/new-operator-java/
- https://www.geeksforgeeks.org/garbage-collection-java/

Object-Oriented Programming (OOP) - Concepts

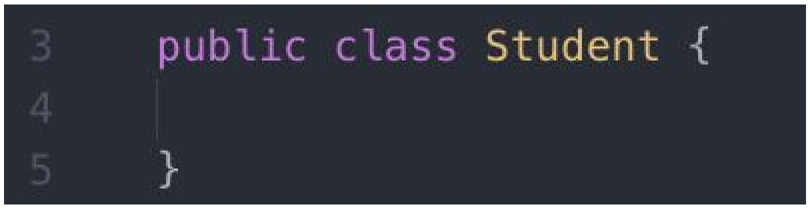
- Object-Oriented Programming is an approach to software development which focuses on objects that contain data/attributes and procedures/methods.
- A Class is a blueprint/prototype from which objects (i.e., instances) are created.

  Class Example:



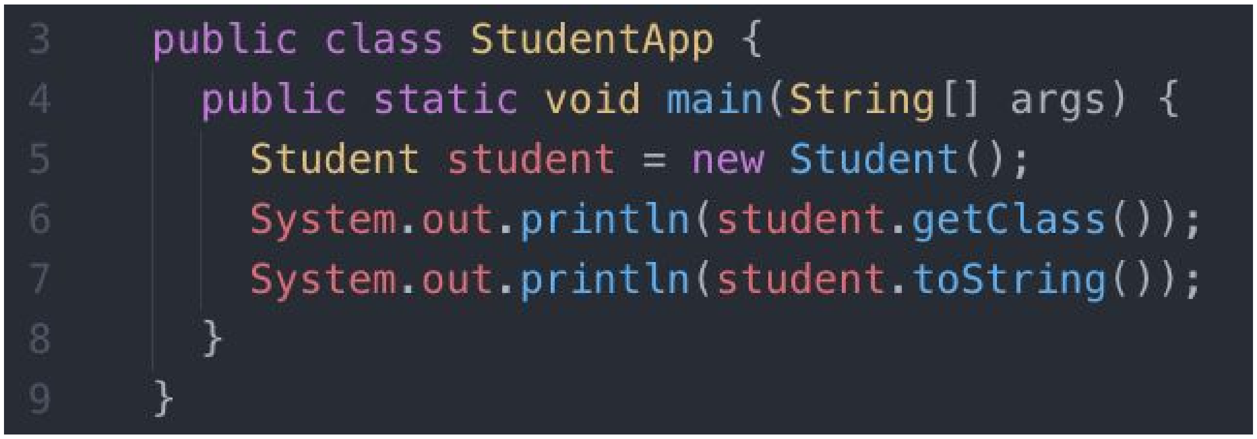
```java
public class User {
  // Attributes/Fields
  private String name;
  private String email;
  // Constructors
  public User(String n, String e) {
    this.name = n;
    this.email = e;
  }
  // Getters & Setters
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getEmail() {
    return email;
  }
  public void setEmail(String email) {
    this.email = email;
  }
}
```

- Objects
    - Instances of a class
    - https://docs.oracle.com/javase/tutorial/java/javaOO/objects.html
- Here a basic class called Student:



```java
public class Student {

}
```

- It doesn’t have any added features yet such as fields or methods, but we can still instantiate it and call methods that are inherited from its default parent class, Object:



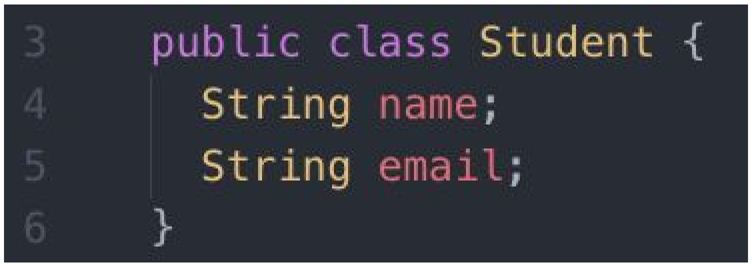
```java
public class StudentApp {
  public static void main(String[] args) {
    Student student = new Student();
    System.out.println(student.getClass());
    System.out.println(student.toString());
  }
}
```

Output:

```
class com.perscholas.student.Student
com.perscholas.student.Student@33909752
```

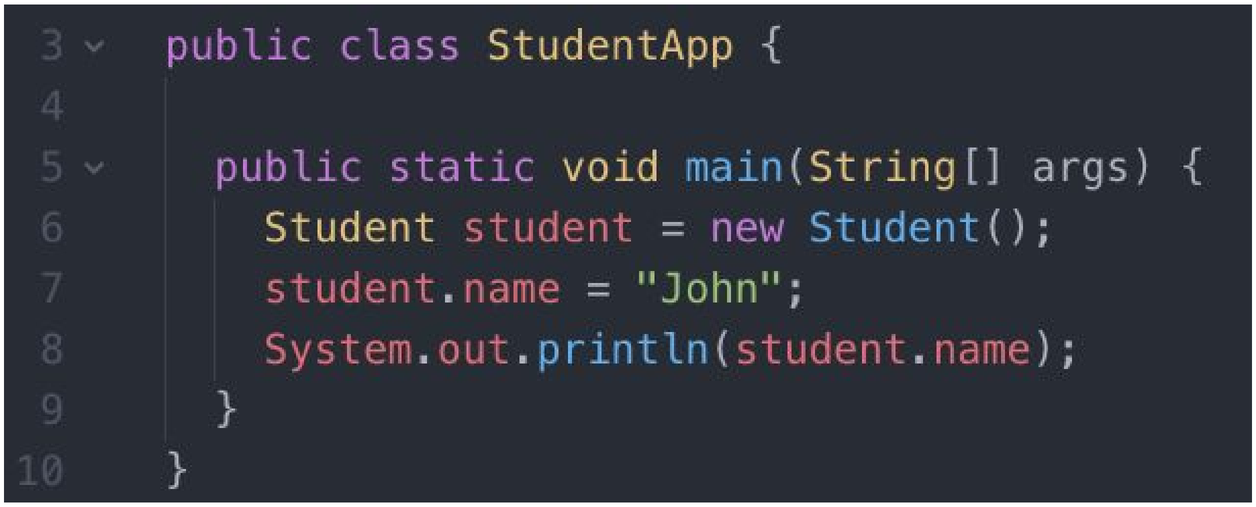
- Here we printed the results of the Object methods getClass and toString. The getClass method returns the path and class name and the toString method returns the class path and name, along with the hashcode. Don’t worry if you don’t understand what those are right now. Just know that we have access to class attributes/fields and methods from the parent class. This is inheritance which will be discussed further in another section.

- Next, we will add attributes/fields to the Student class:



```java
public class Student {
  String name;
  String email;
}
```

These are variables which will come with each instance/object created (i.e., instantiated) such as when we created the instance and named it “student”. We can now save and access values held in these variables for each instance we create.

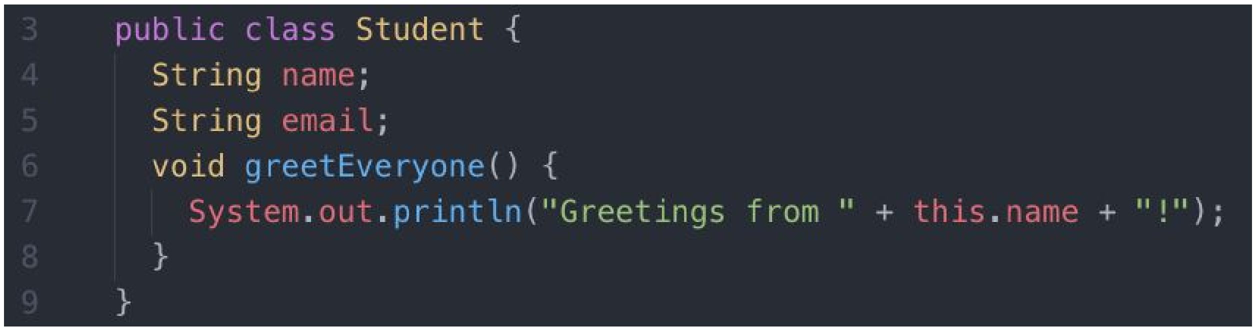


```java
public class StudentApp {

  public static void main(String[] args) {
    Student student = new Student();
    student.name = "John";
    System.out.println(student.name);
  }
}
```

Output: John

- We saved the string “John” to the variable student.name and then passed that value to the System.out.println method to print it out to the console.

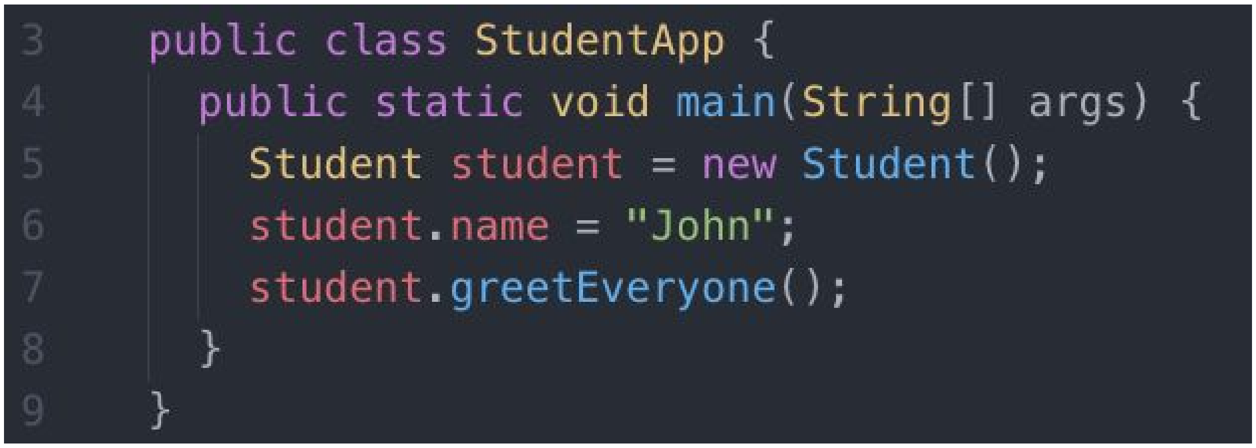
- Now we’ll add a method to this class:



```java
public class Student {
  String name;
  String email;
  void greetEveryone() {
    System.out.println("Greetings from " + this.name + "!");
  }
}
```

- The “this” keyword is used to refer to the instantiated (i.e., created) object so in our example this.name will be referencing student.name

Now we can use our instance “student” to call this method:

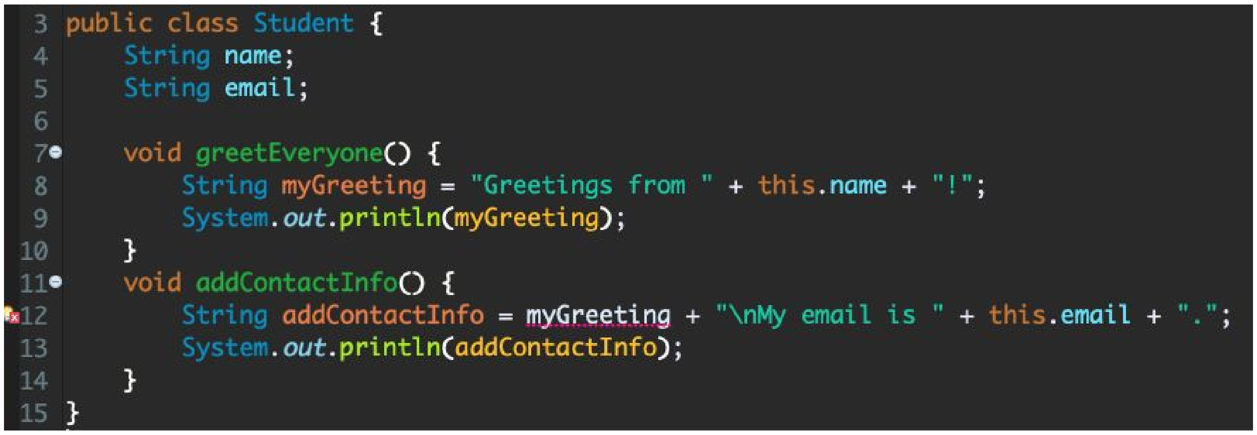


```
3   public class StudentApp {
4     public static void main(String[] args) {
5       Student student = new Student();
6       student.name = "John";
7       student.greetEveryone();
8     }
9   }
```

We instantiated the class in line 5 and assigned it to “student”, then we assigned the string “John” to the instance variable student.name in line 6, and then ran the method student.greetEveryone( ) in line 7 which resulted in the console output: Greetings from John!

- This is a basic Java class. Next we will discuss more class features and gradually add more features to our classes
- Variables
  - Local Variables
    - Local variables are defined inside methods, constructors or blocks
    - The local variable will be declared and initialized within the method and will be destroyed when the block has been completed
    - Scope is limited to the block in which the variable was created and any blocks nested within

      Example:

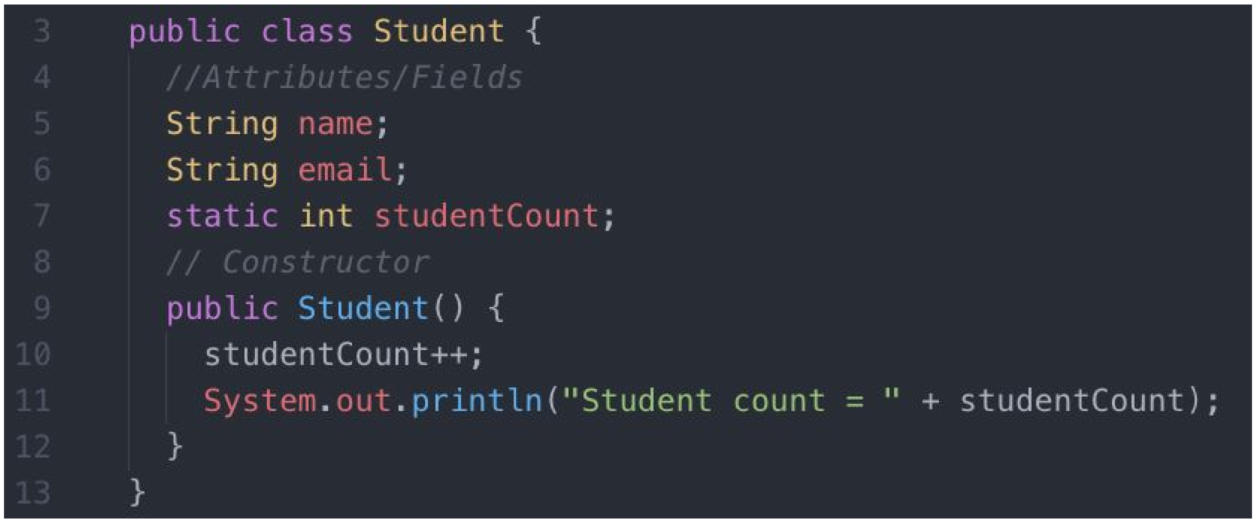


```java
public class Student {
    String name;
    String email;

    void greetEveryone() {
        String myGreeting = "Greetings from " + this.name + "!";
        System.out.println(myGreeting);
    }
    void addContactInfo() {
        String addContactInfo = myGreeting + "\nMy email is " + this.email + ".";
        System.out.println(addContactInfo);
    }
}
```

The String variable myGreeting is declared in the method greetEveryone and so is local to that method, but not available to the addContactInfo method. Refer back to the section on Scope for more on this topic.

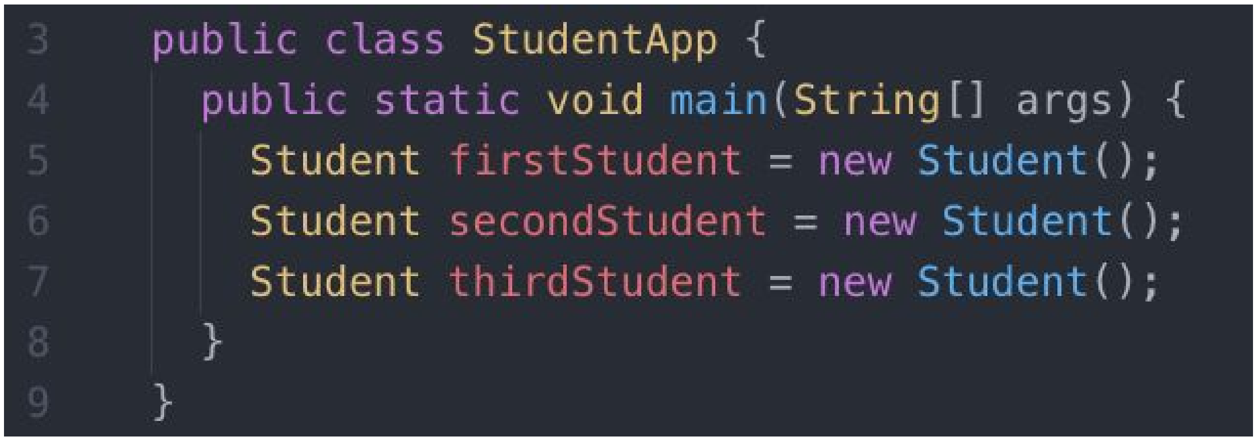
- Instance Variables (i.e., attributes/fields)
  - We've already seen examples of these in the preceding Student class example
  - Instance variables are variables within a class but outside any method
  - These variables are initialized when the class is instantiated
  - Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- Static (Class) Variables
  - Declared inside a class using the "static" keyword
  - A single copy of the variable is created and accessed at the class level (i.e., by referencing the class rather than an instance of the class)
  - A static variable gets allocated only once per class within the lifecycle of the application.
  - Read this page about static variables: https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html

    Example:



```java
public class Student {
  //Attributes/Fields
  String name;
  String email;
  static int studentCount;
  // Constructor
  public Student() {
    studentCount++;
    System.out.println("Student count = " + studentCount);
  }
}
```

We've added a static variable studentCount to the Student class along with a constructor which we will discuss in the next section. The constructor runs each time an instance is created. In the case of the Student class, the static variable studentCount will be incremented by 1 and the count printed to the console each time an instance of Student is created as in the following example:



```java
public class StudentApp {
  public static void main(String[] args) {
    Student firstStudent = new Student();
    Student secondStudent = new Student();
    Student thirdStudent = new Student();
  }
}
```

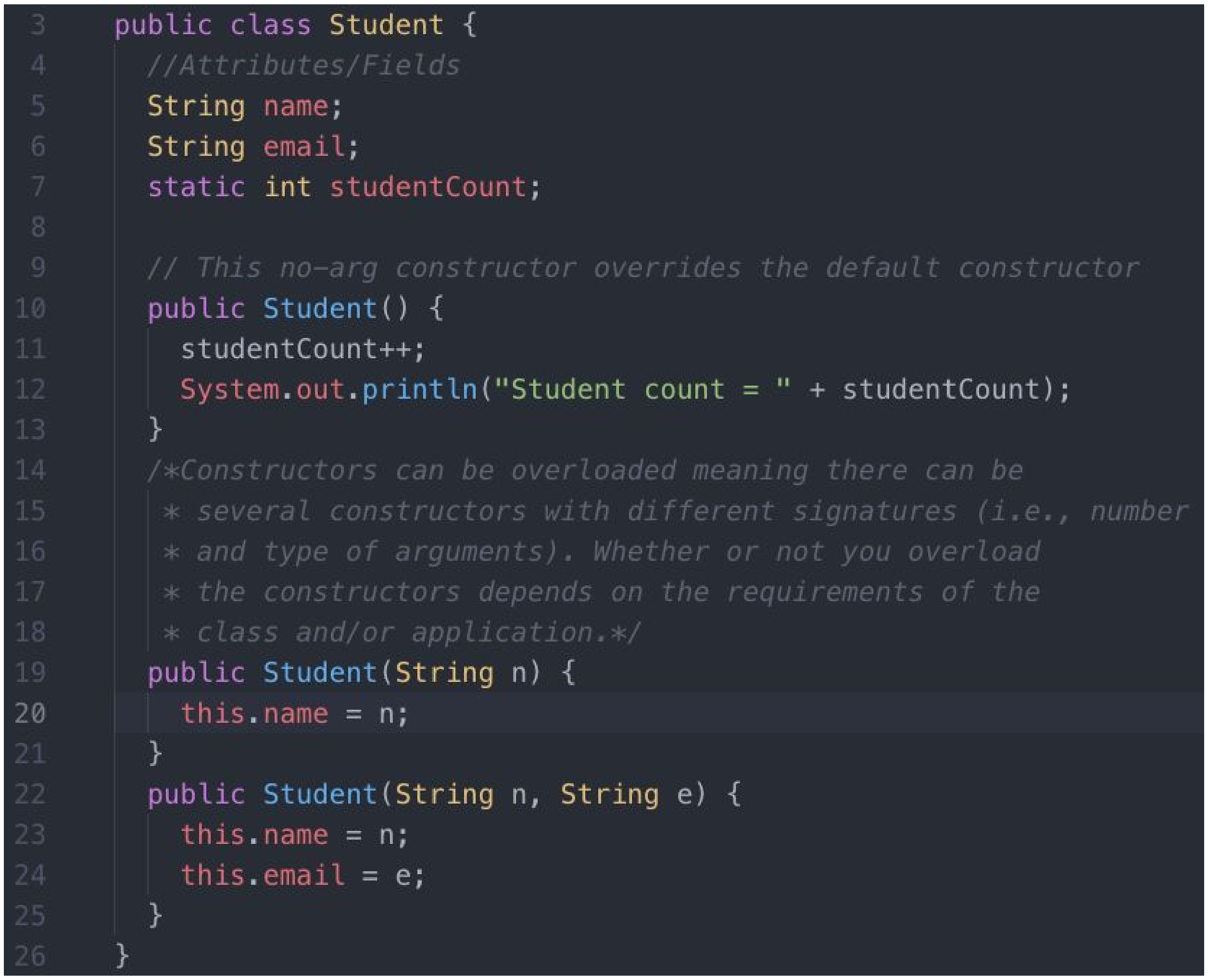
Output:

```
Student count = 1
Student count = 2
Student count = 3
```

- Constructor - method that is called when a new instance of the class is created

- By default, Java includes a no-arg (i.e., no arguments) constructor when no constructor is written in the class, however the default constructor can be overridden and overloaded
  - If no constructor is written in the class then the default constructor is included but won't show up in the code
  - If a custom constructor is written in the code, then the default constructor is no longer available - you must write a no-arg constructor in this case if your class requires this
- Constructors do not have a return type
- The name of the constructor must be the same as the class
- Primarily used to initialize class attributes/fields

Example:



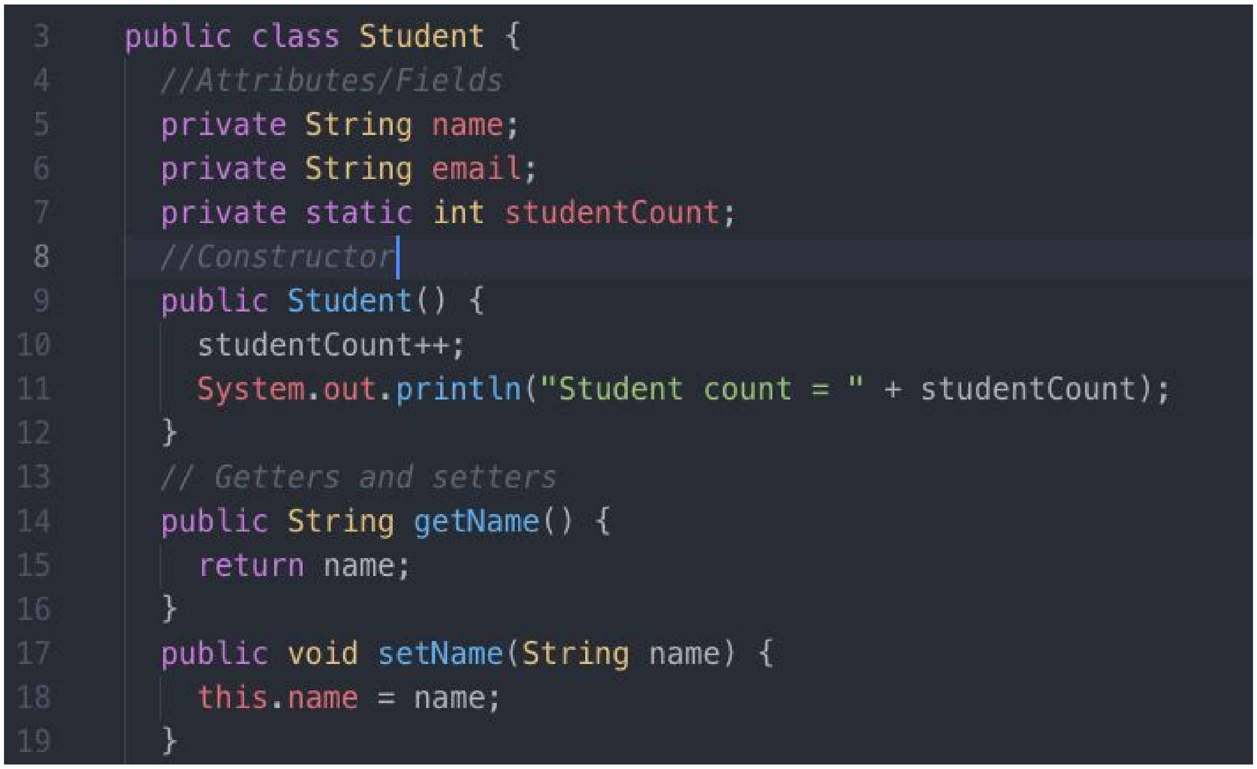
```java
public class Student {
  //Attributes/Fields
  String name;
  String email;
  static int studentCount;

  // This no-arg constructor overrides the default constructor
  public Student() {
    studentCount++;
    System.out.println("Student count = " + studentCount);
  }
  /*Constructors can be overloaded meaning there can be
   * several constructors with different signatures (i.e., number
   * and type of arguments). Whether or not you overload
   * the constructors depends on the requirements of the
   * class and/or application.*/
  public Student(String n) {
    this.name = n;
  }
  public Student(String n, String e) {
    this.name = n;
    this.email = e;
  }
}
```

- Read these two pages on constructors:
  - https://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html
  - https://www.geeksforgeeks.org/constructors-in-java/

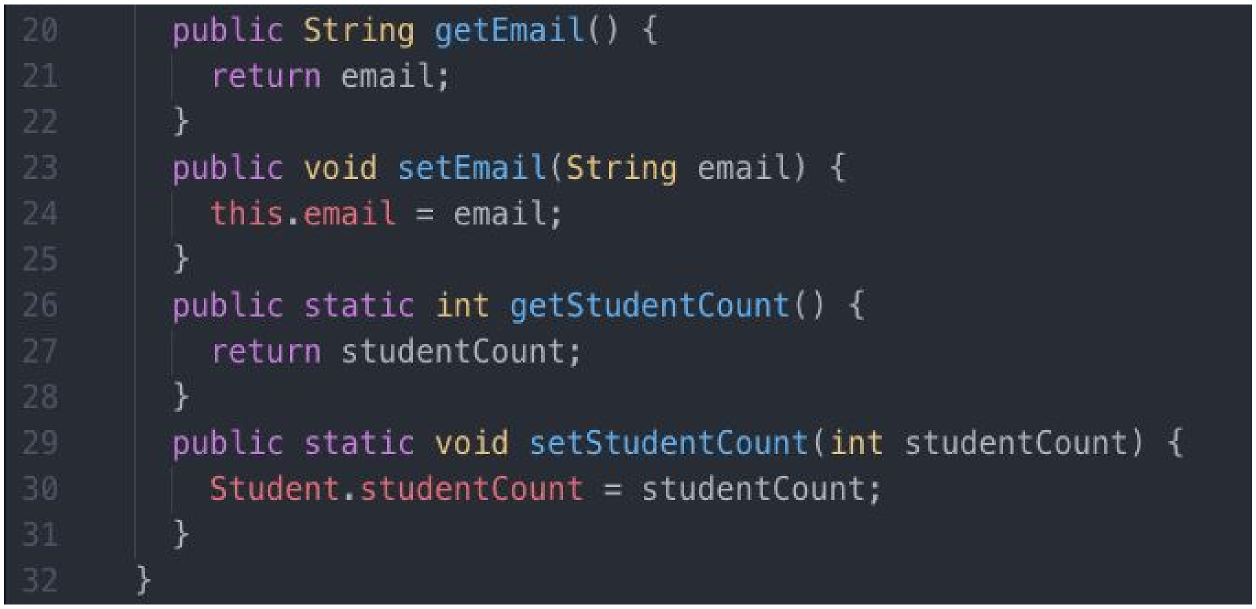
- Accessors/Mutators (aka, getters and setters)
  - Used to access private or protected attributes/fields of a class
  - Access modifiers: public, protected, private and default (package)
    - Access modifiers will be discussed further in the section on Encapsulation
    - Read more about this here: https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

- In most cases, fields should be private with public getters and setters to access those fields

We will continue refining the Student class we created earlier by making the attributes private and including getters and setters - we will also remove the overloaded constructors because we no longer need them



```java
public class Student {
  //Attributes/Fields
  private String name;
  private String email;
  private static int studentCount;
  //Constructor
  public Student() {
    studentCount++;
    System.out.println("Student count = " + studentCount);
  }
  // Getters and setters
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
```



```java
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public static int getStudentCount() {
        return studentCount;
    }
    public static void setStudentCount(int studentCount) {
        Student.studentCount = studentCount;
    }
}
```

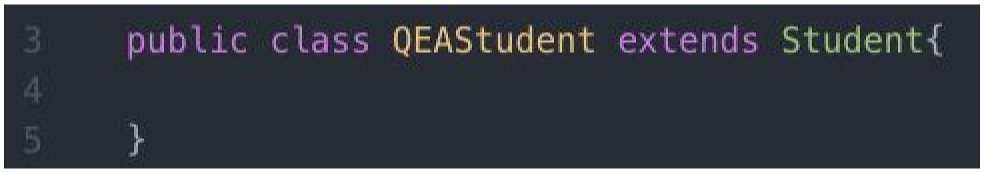
You can write/code the getters and setters manually, but Eclipse has a menu item that can generate them for you: Right click anywhere inside the class (but preferably where you want the getters and setters placed) and select Source -> Generate Getters and Setters… In the “Generate Getters and Setters” window, select “Select All” in the upper right of the window and then click “OK”. You can then layout and/or align the getter and setter methods to your preference.

- Methods
  - Classes can have additional methods to handle logic related to the class
    - In a previous section we created two methods: greetEveryone and addContactInfo
    - Getters and setters are also methods in a class
  - Read these two pages on methods and arguments
    - https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html
    - https://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html
  - Static keyword as applied to methods:

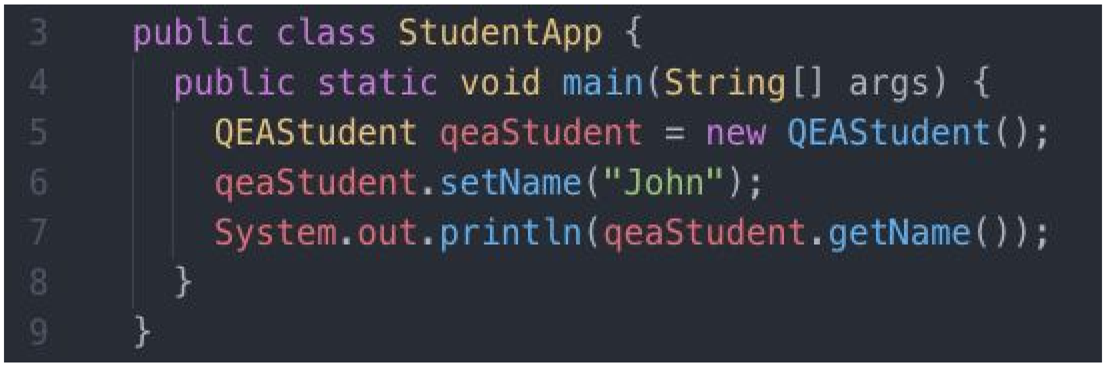
    - As previously mentioned, the static keyword assigns attributes/fields to the class, but the static keyword can also be applied to methods
    - https://www.geeksforgeeks.org/static-methods-vs-instance-methods-java/

- Final Keyword
  - Applied to variables to make them constants
  - Applied to methods to prevent overriding
  - Applied to classes to prevent inheritance
  - Read this page on the use of the "final" keyword: https://www.geeksforgeeks.org/final-keyword-java/
- Basic Concepts of OOP: Inheritance, Polymorphism, Abstraction and Encapsulation
  - Inheritance: when a class (subclass) inherits members (fields and methods) from another class (super class)
    - Read these pages on creating classes and subclasses and the concept of inheritance
      - https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html
      - https://docs.oracle.com/javase/tutorial/java/javaOO/classes.html
      - https://www.geeksforgeeks.org/inheritance-in-java/
    - The super keyword
      - Accessing superclass members
      - Calling the super( ) constructor
      - Read these two pages on uses of the "super" keyword
        - https://docs.oracle.com/javase/tutorial/java/IandI/super.html
        - https://www.geeksforgeeks.org/super-keyword/

In a previous section, we created a class Student which had no explicit features but inherited features from its parent class Object. Now, we'll create a new class "QEAStudent" which will inherit from "Student" and thus inherit all of its attributes and methods as demonstrated in the following example:



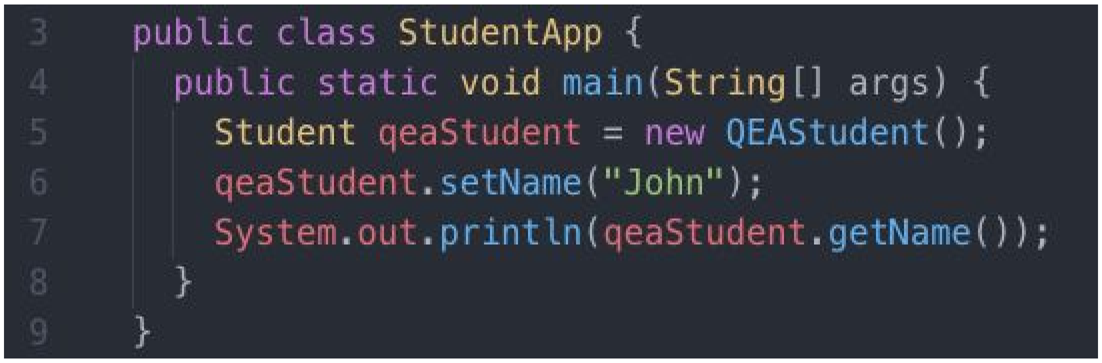
```java
public class QEAStudent extends Student{

}
```



```java
public class StudentApp {
  public static void main(String[] args) {
    QEAStudent qeaStudent = new QEAStudent();
    qeaStudent.setName("John");
    System.out.println(qeaStudent.getName());
  }
}
```

Output:
Student count = 1
John

- Polymorphism: An object can take on more than one form. It can reference objects of its subclasses. "Is-a" relationship (e.g., Car "is-a" Vehicle.)
  - An object can be referenced by any class or abstract class it extends (inherits) and/or any interfaces it implements
  - Read this page on polymorphism: https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html



```java
public class StudentApp {
  public static void main(String[] args) {
    Student qeaStudent = new QEAStudent();
    qeaStudent.setName("John");
    System.out.println(qeaStudent.getName());
  }
}
```

Notice how the object of QEAStudent named “qeaStudent” is referenced as a Student type. This is an example of polymorphism. This is possible because the the QEAStudent class extends the Student class and passes the “is-a” test (i.e., QEAStudent is a Student).

- Compile Time Polymorphism (Dynamic Binding) versus Runtime Polymorphism (Static Binding)
- We’ve already discussed the concepts of overriding and overloading, but here are more explanations on how these concepts relate to polymorphism. Be familiar with this section and use it as a reference.
  - Compile time polymorphism - overloading
    - Operator Overloading
      - The ‘+’ operator can be used to add two numbers and also can be used to concatenate two strings
      - The ‘+’ operator is the only operator in java which is used for operator overloading
    - Constructor Overloading
      - We can include multiple constructors in a class
      - This provides multiple ways to create a class by passing different sets of arguments/parameters
    - Method Overloading
      - We can have different forms of the same method in the same class, similar to constructor overloading
      - Overloaded methods are differentiated according to their signature (i.e., number and/or types of parameters)
  - Runtime Polymorphism - overriding
    - Five rules to apply when overriding a method

        - The name of the method must be the same as that of the super class method
        - The return type of the overriding method (i.e., subclass method) must be compatible with the return type of the method being overridden (i.e., super class method)
        - You must not reduce the visibility of a method when overriding it or change the parameter list of the method
        - Overriding methods can throw unchecked exceptions regardless of whether the super class throws these exceptions or not (see the web link below for information on checked and unchecked exceptions)
            - However, the overriding method should not throw checked exceptions (i.e., declared using the "throws" statement) which are new or broader than those in the overridden method from the super class.
            - Exceptions will be discussed in a later section. For now, here is a reference on checked and unchecked exceptions: https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/
        - Reference on method overriding: https://www.geeksforgeeks.org/overriding-in-java/
    - You can read more about static and dynamic binding here: https://www.geeksforgeeks.org/static-vs-dynamic-binding-in-java/
- Abstraction: displaying functionality and hiding implementation details. User knows the "what" and not necessarily the "how".
    - A classic example of abstraction is a person driving a car who knows that turning the steering wheel causes the vehicle to turn, pressing on the accelerator makes it move and pressing on the brakes stops it. The driver doesn't necessarily know how this happens, but just knows how to "call" these functions of the vehicle (the vehicle api so to speak) to make it work. Classes and applications work in a similar fashion. We can call methods of other classes and have them perform certain functions and/or return certain values without knowing exactly how this was done.

- Abstraction will be discussed further in the sections on abstract classes and interfaces
- Read this page on abstraction: https://www.geeksforgeeks.org/abstraction-in-java-2/

- Encapsulation: wrapping the data and the code acting on that data into a single entity (object)
  - Access modifiers allow a programmer to control access to fields and methods by instances of other classes
    - Access Modifiers (aka, access specifiers)
      - public - access to the class's attributes/fields or methods from other classes is unrestricted
      - protected - access to the class's attributes/fields or methods is available within the same package and to its subclasses
      - private - access to the class's attributes/fields or methods is available only within the class
      - default - access to the class's attributes/fields or methods is available only within the same package
      - Read this tutorial on access modifiers: https://www.geeksforgeeks.org/access-modifiers-java/
    - In most situations, attributes/fields will be declared as private with public getters and setters
  - data hiding: hiding data from code that is outside the object by making data/attributes/fields private
  - The purpose of data hiding is to have more control on how data gets manipulated and/or accessed
    - Example: If an email field should only allow valid email addresses, a setter method could screen for this prior to setting the email field with an erroneous

(i.e., non-email) string passed in by a user. Another example would be if we wanted a class's field(s) to be read only we could remove all setter methods.

- Watch this video and read the web page on encapsulation:
    - https://www.youtube.com/watch?v=4VBdNbWeTZw
    - https://www.tutorialspoint.com/java/java_encapsulation.htm
- Cohesion & Coupling
    - Cohesion
        - Classes should model a single entity and keep to a single well-defined purpose
        - Aim for high cohesion
            - Classes have a well focused/defined purpose
            - Classes are easy to read
            - Classes are reusable
        - Read this page on cohesion in java: https://www.geeksforgeeks.org/cohesion-in-java/
    - Coupling
        - The degree that one class depends on another
        - Aim for loose coupling
        - Read this page on coupling in Java: https://www.geeksforgeeks.org/coupling-in-java/

- Advanced Challenge: Read through this tutorial from Oracle - https://docs.oracle.com/javase/tutorial/java/javaOO/index.html

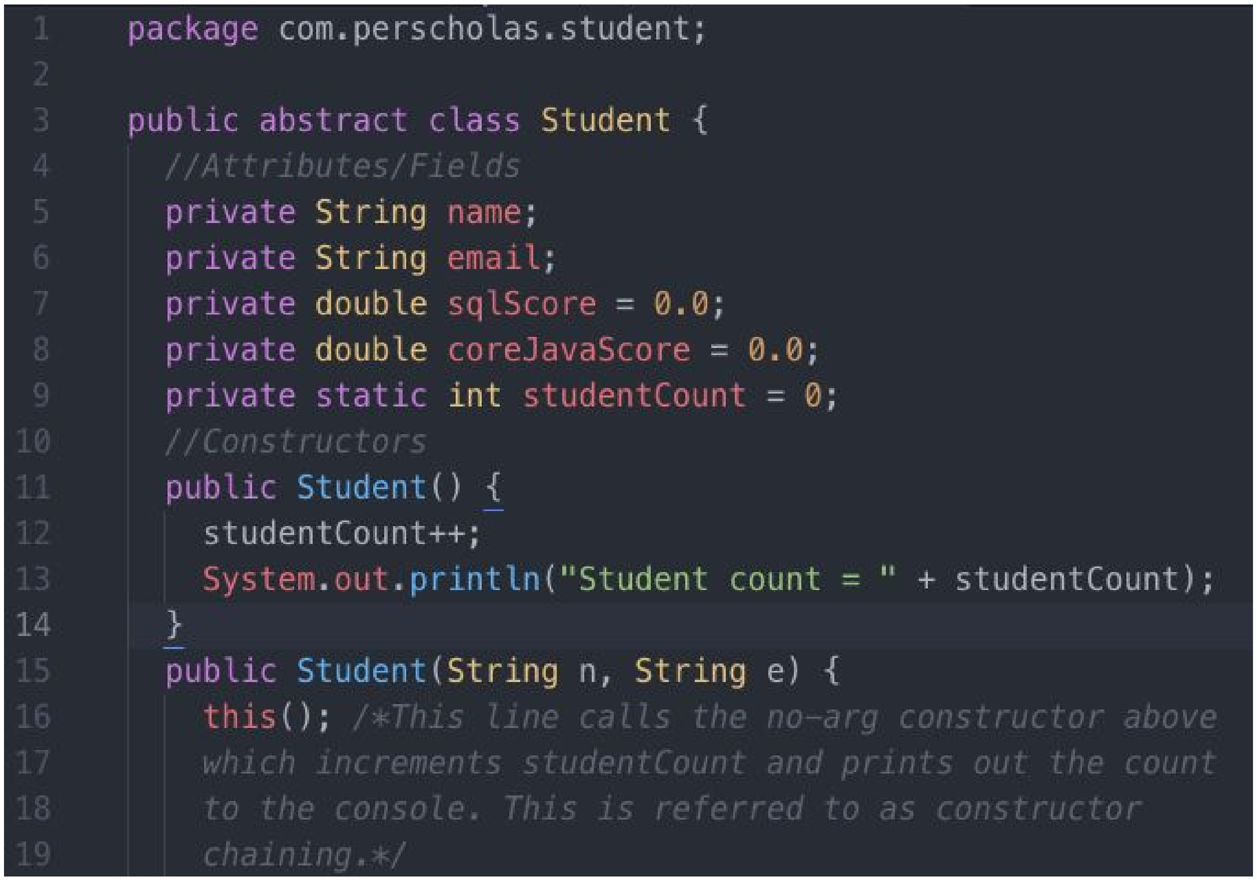
Abstract Classes

- Useful for sharing code among several closely related classes

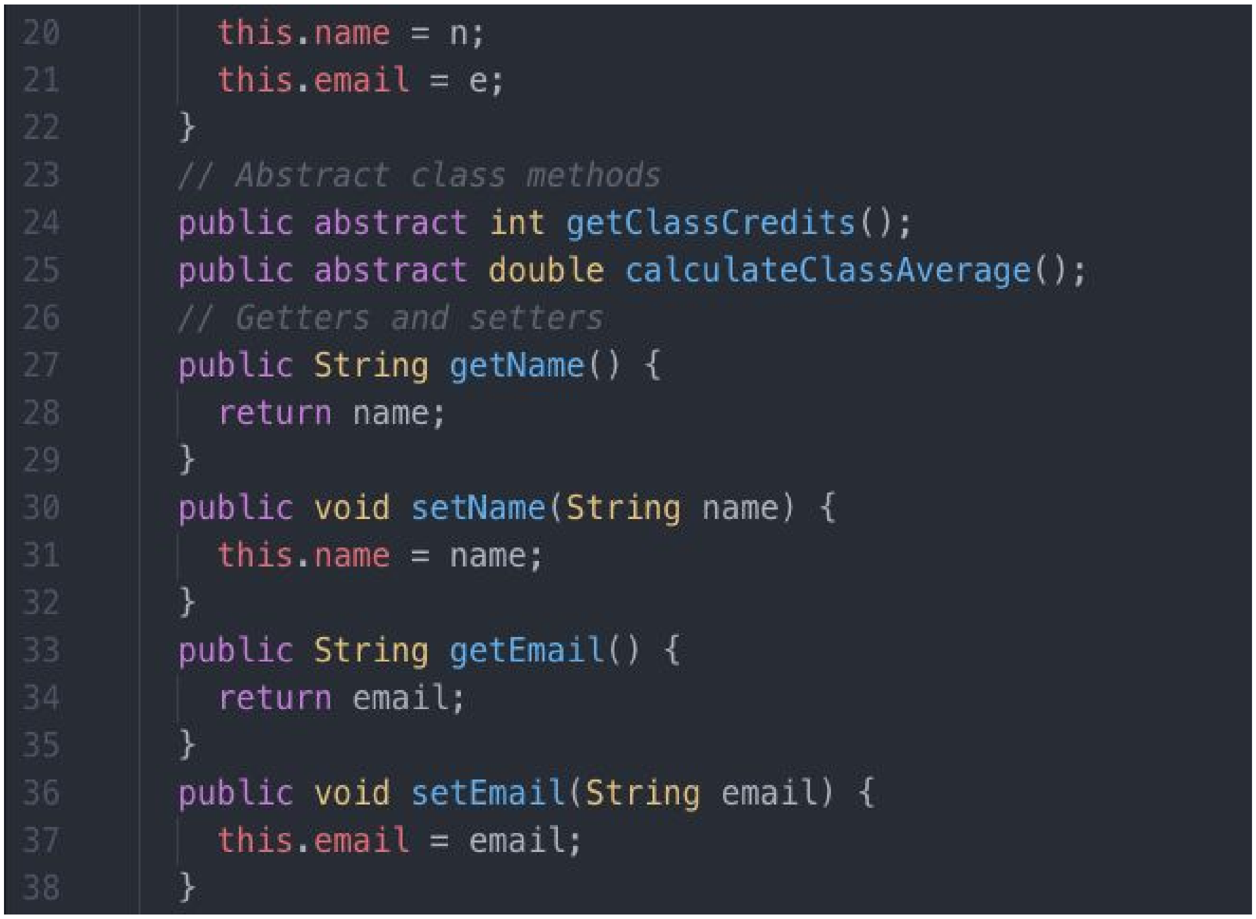
- Is not instantiated, but is utilized as a superclass
- Can contain non-static fields and methods with access modifiers (public, protected or private)
- Abstract methods
  - Include a header but no body
  - Must be overridden in the subclass
- https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html
- https://www.youtube.com/watch?v=1PPDoAKbaNA
- https://www.youtube.com/watch?v=RcIsb9iFKH8
-

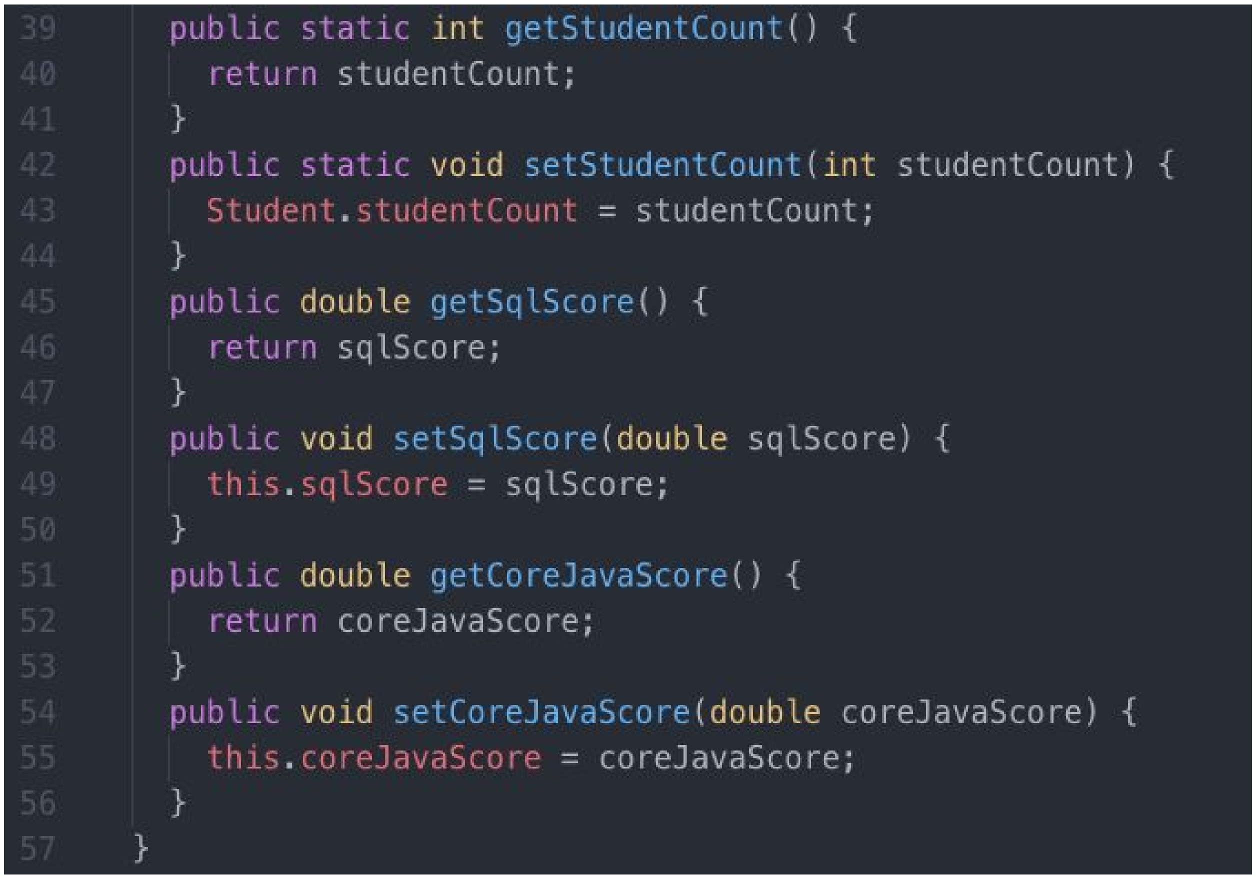
- Abstract class example:

  In this example, we make the Student class abstract by adding the “abstract” keyword to its declaration. We can no longer instantiate this class, but will instantiate its subclasses. We have also added two abstract methods that must be implemented in the subclasses.



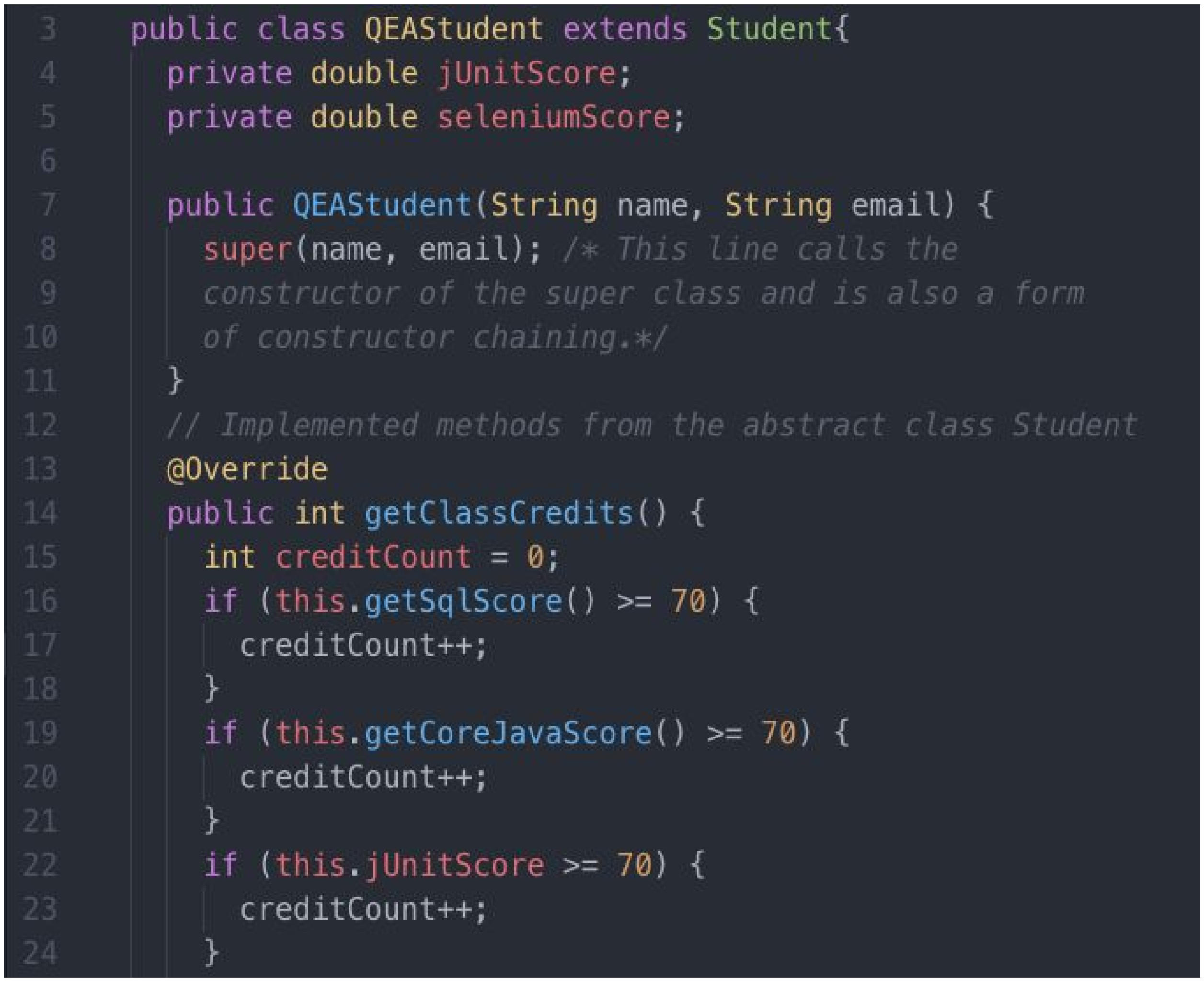
```java
package com.perscholas.student;

public abstract class Student {
  //Attributes/Fields
  private String name;
  private String email;
  private double sqlScore = 0.0;
  private double coreJavaScore = 0.0;
  private static int studentCount = 0;
  //Constructors
  public Student() {
    studentCount++;
    System.out.println("Student count = " + studentCount);
  }
  public Student(String n, String e) {
    this(); /*This line calls the no-arg constructor above
    which increments studentCount and prints out the count
    to the console. This is referred to as constructor
    chaining.*/
```


```java
        this.name = n;
        this.email = e;
    }
    // Abstract class methods
    public abstract int getClassCredits();
    public abstract double calculateClassAverage();
    // Getters and setters
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
```

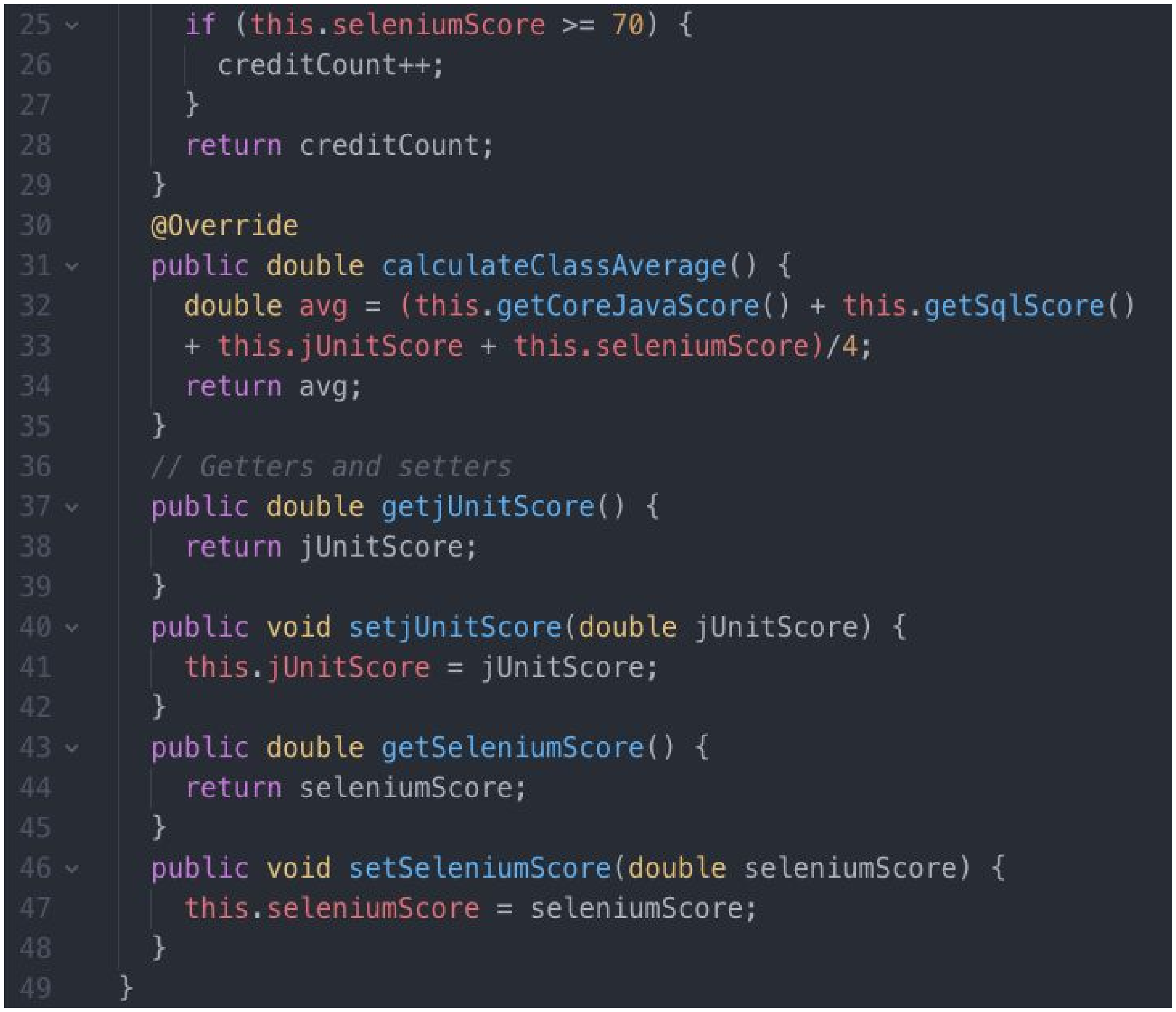


```
    public static int getStudentCount() {
      return studentCount;
    }
    public static void setStudentCount(int studentCount) {
      Student.studentCount = studentCount;
    }
    public double getSqlScore() {
      return sqlScore;
    }
    public void setSqlScore(double sqlScore) {
      this.sqlScore = sqlScore;
    }
    public double getCoreJavaScore() {
      return coreJavaScore;
    }
    public void setCoreJavaScore(double coreJavaScore) {
      this.coreJavaScore = coreJavaScore;
    }
  }
```

Here is an implementation of the Student abstract class named QEAStudent:

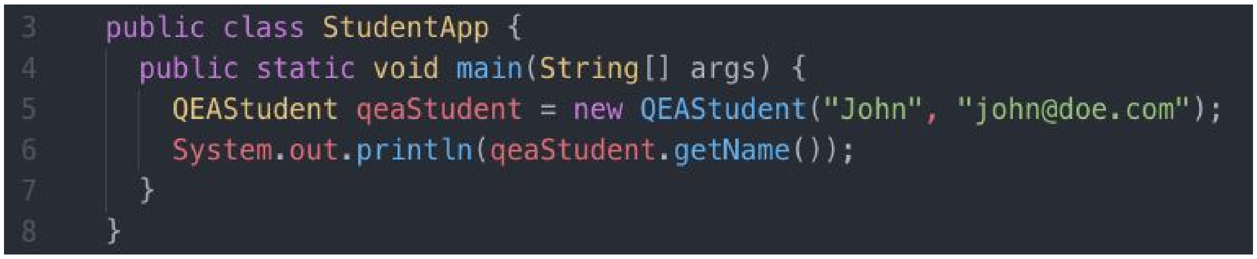


```
public class QEAStudent extends Student{
  private double jUnitScore;
  private double seleniumScore;

  public QEAStudent(String name, String email) {
    super(name, email); /* This line calls the
    constructor of the super class and is also a form
    of constructor chaining.*/
  }
  // Implemented methods from the abstract class Student
  @Override
  public int getClassCredits() {
    int creditCount = 0;
    if (this.getSqlScore() >= 70) {
      creditCount++;
    }
    if (this.getCoreJavaScore() >= 70) {
      creditCount++;
    }
    if (this.jUnitScore >= 70) {
      creditCount++;
    }
```



```java
    if (this.seleniumScore >= 70) {
      creditCount++;
    }
    return creditCount;
  }
  @Override
  public double calculateClassAverage() {
    double avg = (this.getCoreJavaScore() + this.getSqlScore()
    + this.jUnitScore + this.seleniumScore)/4;
    return avg;
  }
  // Getters and setters
  public double getjUnitScore() {
    return jUnitScore;
  }
  public void setjUnitScore(double jUnitScore) {
    this.jUnitScore = jUnitScore;
  }
  public double getSeleniumScore() {
    return seleniumScore;
  }
  public void setSeleniumScore(double seleniumScore) {
    this.seleniumScore = seleniumScore;
  }
}
```

Here is instantiation of the QEAStudent class



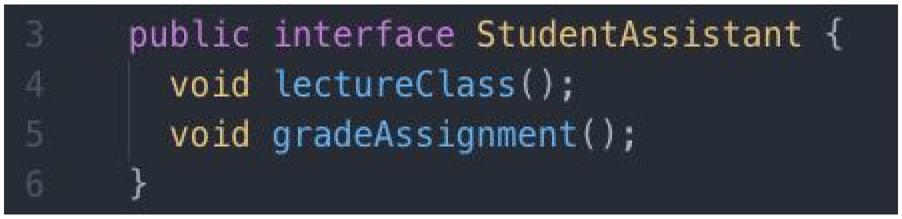
```java
public class StudentApp {
  public static void main(String[] args) {
    QEAStudent qeaStudent = new QEAStudent("John", "john@doe.com");
    System.out.println(qeaStudent.getName());
  }
}
```

Output:

```
Student count = 1
John
```

Interfaces

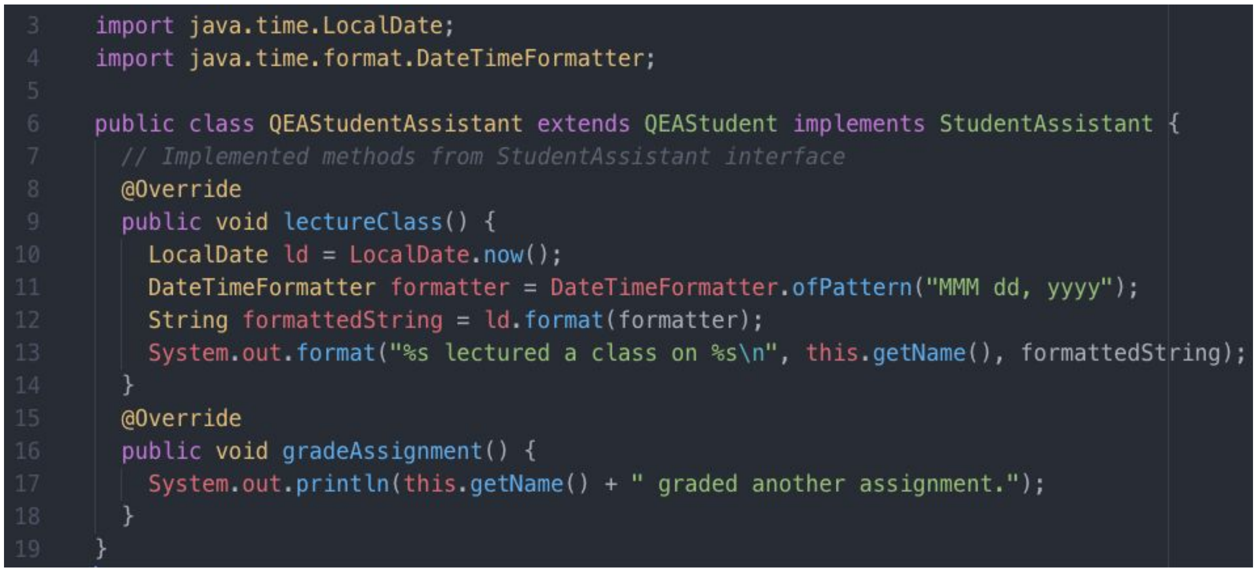
- Similar to abstract classes
- Specify behavior for other classes
- Classes can implement multiple interfaces
- Classes which implement an interface must provide all methods listed in the interface
- Can contain fields, but all are static and final
- All methods specified by an interface are public
- Here is an interface called StudentAssistant:



```java
public interface StudentAssistant {
  void lectureClass();
  void gradeAssignment();
}
```

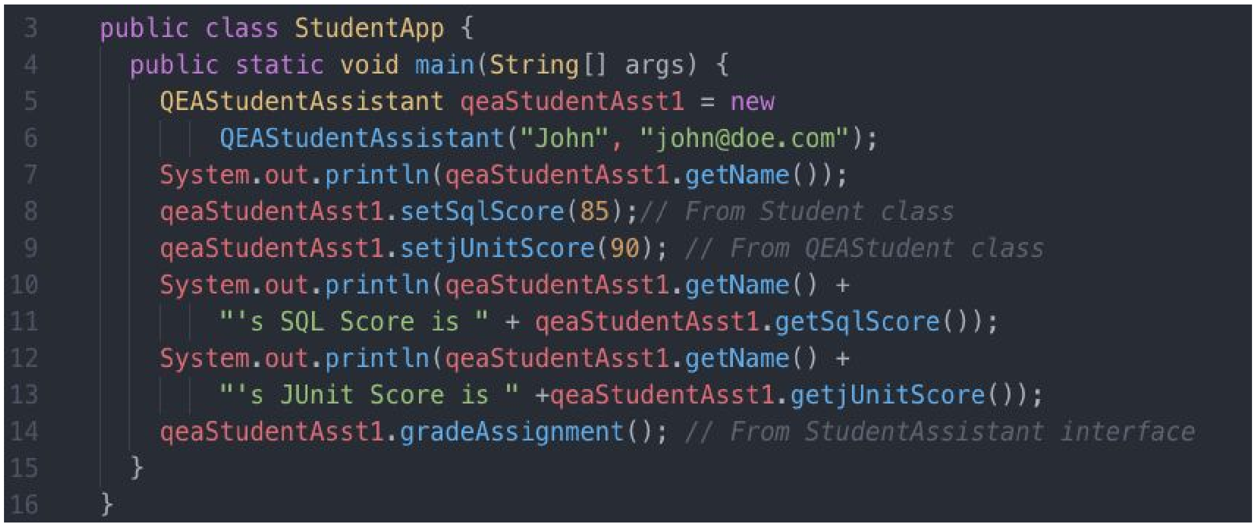
-

- Now we will implement the interface in a new class called QEAStudentAssistant which also extends the class QEAStudent which extends the abstract class Student. We've added very simple implementations of the interface methods for this example.



```java
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

public class QEAStudentAssistant extends QEAStudent implements StudentAssistant {
  // Implemented methods from StudentAssistant interface
  @Override
  public void lectureClass() {
    LocalDate ld = LocalDate.now();
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("MMM dd, yyyy");
    String formattedString = ld.format(formatter);
    System.out.format("%s lectured a class on %s\n", this.getName(), formattedString);
  }
  @Override
  public void gradeAssignment() {
    System.out.println(this.getName() + " graded another assignment.");
  }
}
```

- Here is an instantiation of QEAStudentAssistant with some methods inherited from each of the super classes and interface



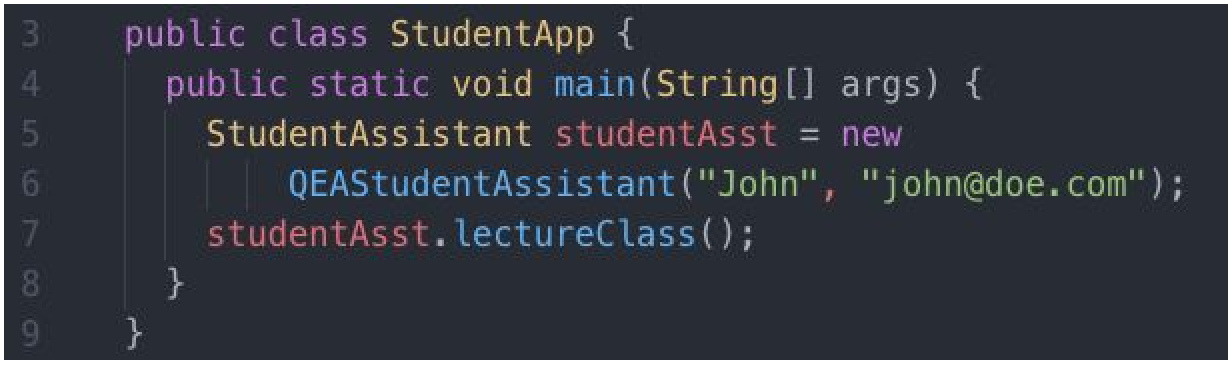
```java
public class StudentApp {
  public static void main(String[] args) {
    QEAStudentAssistant qeaStudentAsst1 = new
        QEAStudentAssistant("John", "john@doe.com");
    System.out.println(qeaStudentAsst1.getName());
    qeaStudentAsst1.setSqlScore(85);// From Student class
    qeaStudentAsst1.setjUnitScore(90); // From QEAStudent class
    System.out.println(qeaStudentAsst1.getName() +
        "'s SQL Score is " + qeaStudentAsst1.getSqlScore());
    System.out.println(qeaStudentAsst1.getName() +
        "'s JUnit Score is " +qeaStudentAsst1.getjUnitScore());
    qeaStudentAsst1.gradeAssignment(); // From StudentAssistant interface
  }
}
```

Output:

```
Student count = 1
John
John's SQL Score is 85.0
John's JUnit Score is 90.0
```

John graded another assignment.

- An interface reference variable can reference any object that implements that interface (interface inheritance). Only attributes and methods inherited from the interface will be available to the instance. However, since we are instantiating a QEAStudentAssistant, the chained constructors all run and studentCount from the Student class is incremented and printed.



```java
public class StudentApp {
  public static void main(String[] args) {
    StudentAssistant studentAsst = new
        QEAStudentAssistant("John", "john@doe.com");
    studentAsst.lectureClass();
  }
}
```

Output:
Student count = 1
John lectured a class on Nov 06, 2018

- Read the following pages on interfaces and watch the video:

  - A brief introduction to the concept of interfaces:
    https://docs.oracle.com/javase/tutorial/java/concepts/interface.html

  - Read this web page and the suqsequent pages (Defining an Interface and Implementing an Interface):
    https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html

  - This is another useful page on interfaces:
    https://www.geeksforgeeks.org/interfaces-in-java/

  - https://www.youtube.com/watch?v=Ggjxn8Q9VuE

Inner Classes

- Classes nested inside other classes

- Visible only to code inside the outer class
- Only code in the outer class can create an instance of the inner class - must use obj of outer class to create an instance of inner class
- If the inner class is static, then use the outer class to create an instance of inner class
- Read the following two pages on inner classes and watch the video (inner classes can be difficult to understand so you may have to read through these articles and watch the video a few times to understand this concept):
  - https://www.geeksforgeeks.org/inner-class-java/
  - https://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html
  - https://www.youtube.com/watch?v=iqp7NQCN2ck

Anonymous Classes

- Can override a method from the same class
- Declared during creation of a new instance of a class
- Can extend exactly one class or implement exactly one interface
- Read these two pages on anonymous classes and watch the video (anonymous classes can be difficult to understand so you may have to read through these articles and watch the video a few times to understand this concept):
  - https://www.geeksforgeeks.org/anonymous-inner-class-java/
  - https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html
  - https://www.youtube.com/watch?v=V7yVbG9_xkM

Wrapper Classes

- A wrapper class encloses around a primitive data type and makes it an object

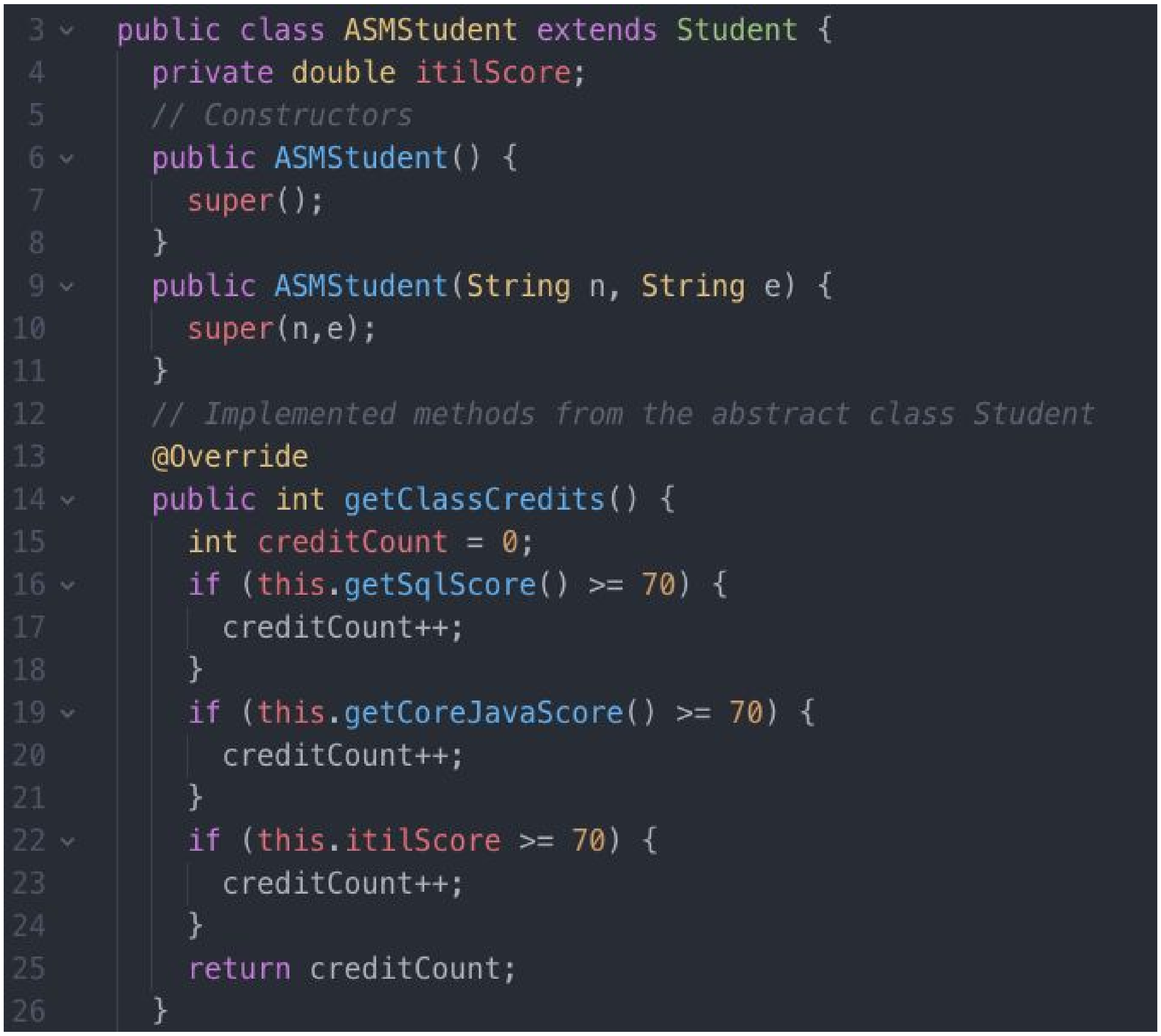
- Contains methods to perform operations related to the data type
- Wherever the data type is required to be an object, wrapper class instances can be used
- Can convert strings into wrapper data types (referred to a parsing)
- Autoboxing - conversion of primitive data type to a non-primitive wrapper class
- Unboxing - conversion of non-primitive wrapper class to a primitive data type
- Read these two pages on wrapper classes
  - https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html
  - https://www.geeksforgeeks.org/wrapper-classes-java/

Generics

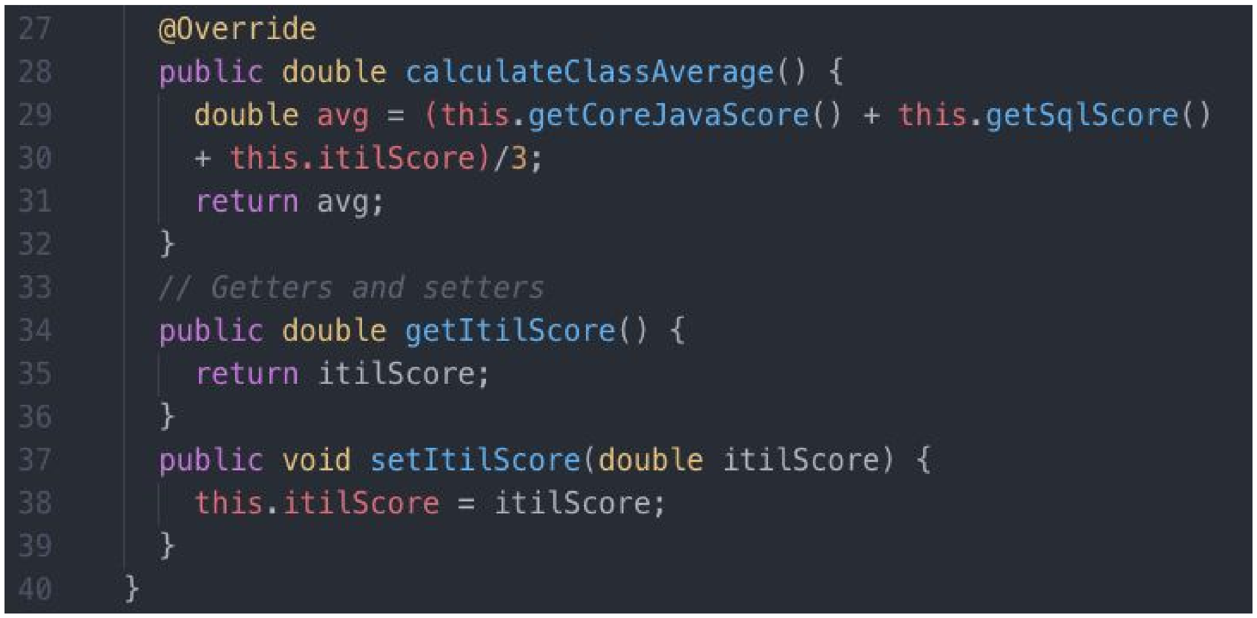
- Refers to interfaces or classes that take type parameters (e.g., List<Integer>, HashMap<String, String>, etc.)
- Allows programmers to specify the type of data allowed in a given Collection
- We will see more examples of this in the Collections Framework section - look for data types enclosed in angle brackets (e.g, ArrayList<Student>)
- Read this page and watch the video on generics:
  - https://www.geeksforgeeks.org/generics-in-java/
  - https://www.youtube.com/watch?v=HhrQqmp3hXI
- Advanced Challenge: Read through these tutorials on generics
  - https://docs.oracle.com/javase/tutorial/extra/generics/index.html
  - https://docs.oracle.com/javase/tutorial/java/generics/index.html

Collections Framework

- https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html
- List
  - Interface (subinterface) which extends the Collection interface (superinterface)
    - An ordered collection
    - Elements are accessed by an integer index
    - Implementing classes include: ArrayList, LinkedList and Stack
  - https://docs.oracle.com/javase/tutorial/collections/implementations/list.html
  - https://docs.oracle.com/javase/8/docs/api/java/util/List.html
  - https://www.geeksforgeeks.org/list-interface-java-examples/
- ArrayList
  - Implementation of the List interface
  - Size (length) changes depending on elements added or removed
  - Here is a program which demonstrates implementation of the List interface through the ArrayList class and also demonstrates the use of polymorphism and the enhanced for-loop. First we'll create a new class called ASMStudent as follow:

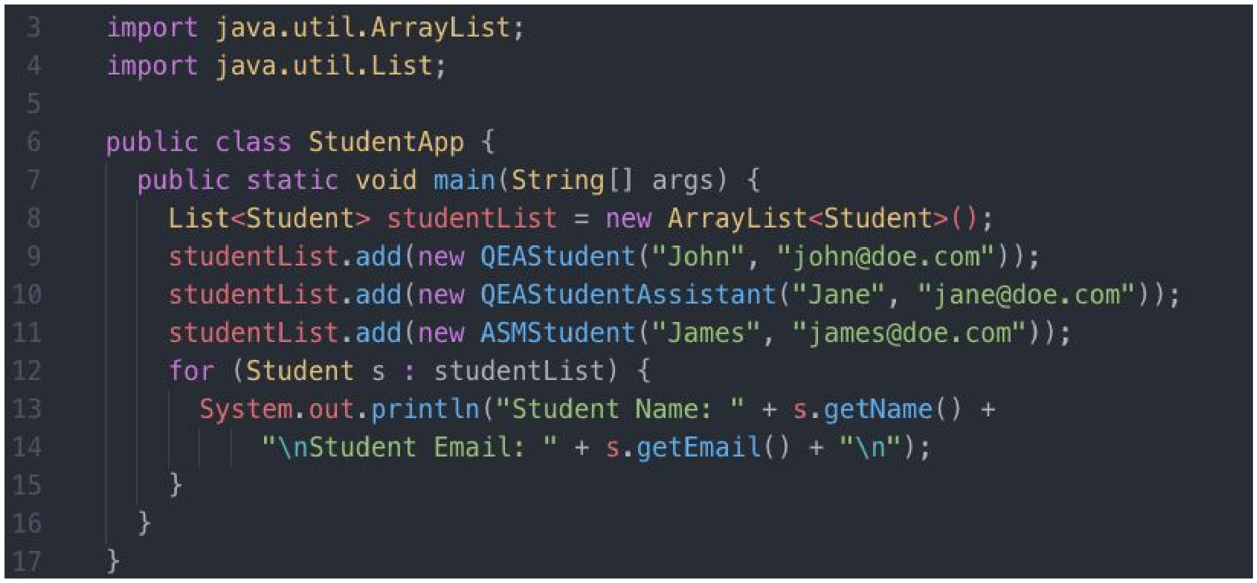


```java
public class ASMStudent extends Student {
  private double itilScore;
  // Constructors
  public ASMStudent() {
    super();
  }
  public ASMStudent(String n, String e) {
    super(n,e);
  }
  // Implemented methods from the abstract class Student
  @Override
  public int getClassCredits() {
    int creditCount = 0;
    if (this.getSqlScore() >= 70) {
      creditCount++;
    }
    if (this.getCoreJavaScore() >= 70) {
      creditCount++;
    }
    if (this.itilScore >= 70) {
      creditCount++;
    }
    return creditCount;
  }
```



```java
    @Override
    public double calculateClassAverage() {
        double avg = (this.getCoreJavaScore() + this.getSqlScore()
        + this.itilScore)/3;
        return avg;
    }
    // Getters and setters
    public double getItilScore() {
        return itilScore;
    }
    public void setItilScore(double itilScore) {
        this.itilScore = itilScore;
    }
}
```

- Next we'll create a List of Student type and add instantiations of QEAStudent, QEAStudentAssistant, and ASMStudent and then loop through and print out the name and email for each student:



```java
import java.util.ArrayList;
import java.util.List;

public class StudentApp {
    public static void main(String[] args) {
        List<Student> studentList = new ArrayList<Student>();
        studentList.add(new QEAStudent("John", "john@doe.com"));
        studentList.add(new QEAStudentAssistant("Jane", "jane@doe.com"));
        studentList.add(new ASMStudent("James", "james@doe.com"));
        for (Student s : studentList) {
            System.out.println("Student Name: " + s.getName() +
                    "\nStudent Email: " + s.getEmail() + "\n");
        }
    }
}
```

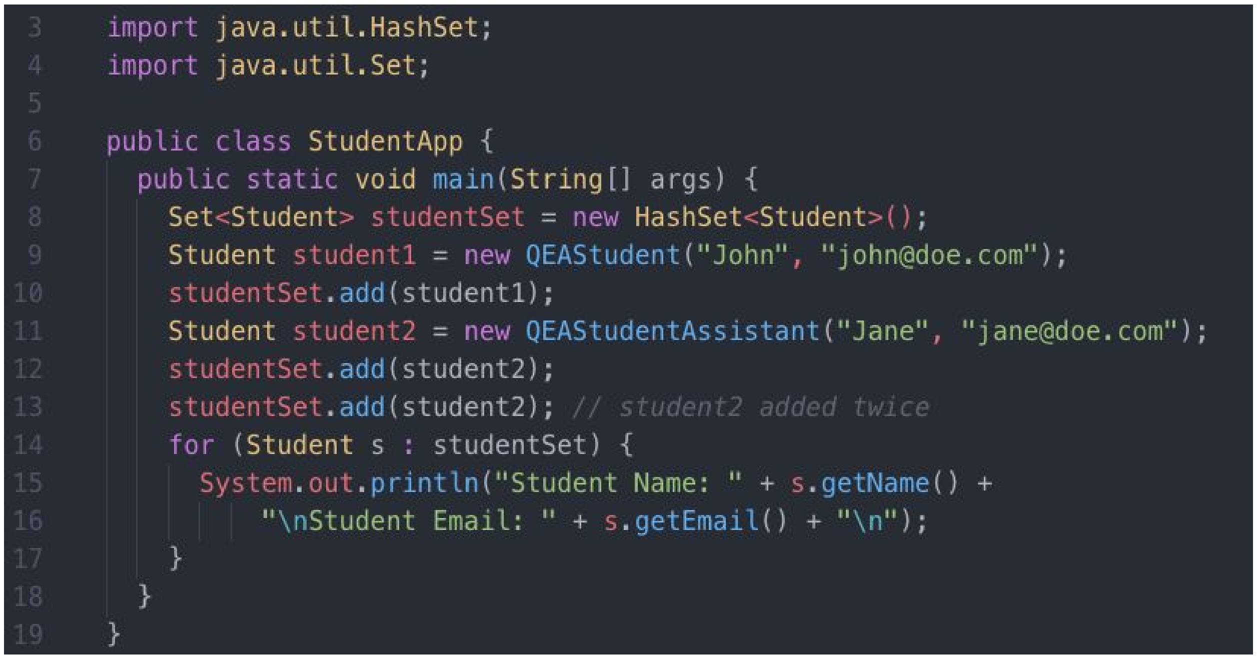
Output:

```
Student count = 1
Student count = 2
Student count = 3
Student Name: John
Student Email: john@doe.com
Student Name: Jane
Student Email: jane@doe.com

Student Name: James
Student Email: james@doe.com
```

- https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
- https://www.geeksforgeeks.org/arraylist-in-java/
- https://www.youtube.com/watch?v=IEqvmsqjpT0

- Set
  - An interface which also extends the Collection interface
    - Similar to a List, but a set cannot have duplicate elements
    - Models the mathematical set abstraction
  - https://docs.oracle.com/javase/tutorial/collections/implementations/set.html
  - https://docs.oracle.com/javase/8/docs/api/java/util/Set.html
  - https://www.geeksforgeeks.org/set-in-java/
- HashSet
  - Implementation of Set interface
  -

- Example:



```java
import java.util.HashSet;
import java.util.Set;

public class StudentApp {
  public static void main(String[] args) {
    Set<Student> studentSet = new HashSet<Student>();
    Student student1 = new QEAStudent("John", "john@doe.com");
    studentSet.add(student1);
    Student student2 = new QEAStudentAssistant("Jane", "jane@doe.com");
    studentSet.add(student2);
    studentSet.add(student2); // student2 added twice
    for (Student s : studentSet) {
      System.out.println("Student Name: " + s.getName() +
          "\nStudent Email: " + s.getEmail() + "\n");
    }
  }
}
```

Even though student2 is added twice in the code, it is only added once to the HashSet because no duplicates are allowed. Keep in mind that duplicates are only considered with regards to the variable name or literal value (e.g., 5, “aString”). If we had created another instance called student3 with the same content as student2 (“Jane”, “jane@doe.com”) and added it to the HashSet, it would have been allowed.

Output:

```
Student count = 1
Student count = 2
Student Name: John
Student Email: john@doe.com

Student Name: Jane
Student Email: jane@doe.com
```

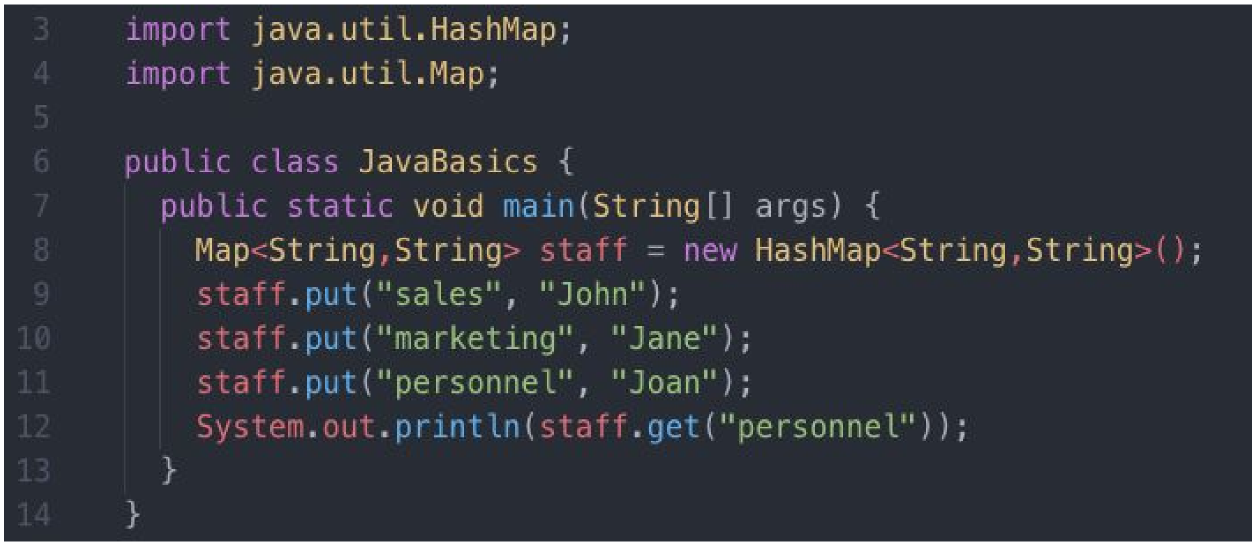
- Read this page on the HashSet class: https://www.geeksforgeeks.org/hashset-in-java/

- Stack

- Implements Collection and List interfaces (among others) and extends the Vector class
- https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html
- Adding/removing items is restricted to last-in-first-out pattern
  - push( ) method adds an item to the top of the stack while the pop( ) method removes the item at the top of the stack and returns the item
  - The peek( ) method allows users/applications to see which item is next to be removed without actually removing the item
- Common uses include undo functions,

- Map
  - An interface which is used to create classes which maintain lists of key-value pairs
    - Keys must be unique
    - Order of the map depends on the implementation
  - https://docs.oracle.com/javase/tutorial/collections/implementations/map.html
  - https://docs.oracle.com/javase/8/docs/api/java/util/Map.html
  - https://www.geeksforgeeks.org/map-interface-java-examples/
- HashMap
  - Implementation of the Map interface
  - Stores information using key-value pairs
  - Aka, dictionary, object, or associative array in other languages
  -

- Example:

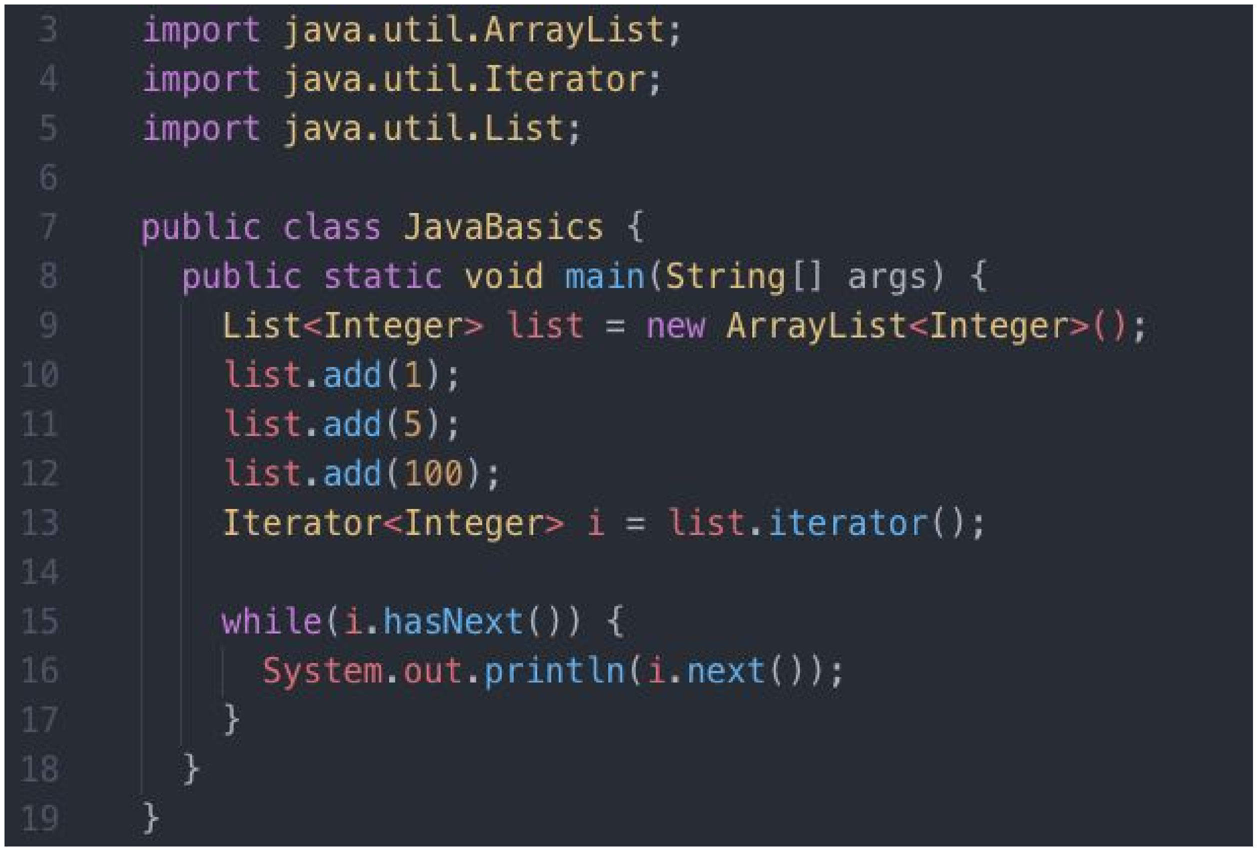


```
import java.util.HashMap;
import java.util.Map;

public class JavaBasics {
  public static void main(String[] args) {
    Map<String,String> staff = new HashMap<String,String>();
    staff.put("sales", "John");
    staff.put("marketing", "Jane");
    staff.put("personnel", "Joan");
    System.out.println(staff.get("personnel"));
  }
}
```

- Read this page on the HashMap class:
  - https://www.geeksforgeeks.org/java-util-hashmap-in-java/

- TreeMap
  - Similar to a HashMap, but ordered/sorted
  - https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html
  - https://www.geeksforgeeks.org/hashmap-treemap-java/
- Iterators
  - Used to iterate over a collection
  -

- Example:



```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class JavaBasics {
  public static void main(String[] args) {
    List<Integer> list = new ArrayList<Integer>();
    list.add(1);
    list.add(5);
    list.add(100);
    Iterator<Integer> i = list.iterator();

    while(i.hasNext()) {
      System.out.println(i.next());
    }
  }
}
```

Output:

```
1
5
100
```

- Read the following two web pages on iterators and use the third link as a reference:
- https://www.geeksforgeeks.org/iterators-in-java/
- https://www.geeksforgeeks.org/how-to-use-iterator-in-java/
- https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html