CSCI 136: Data Structures & Advanced Programming

Today: Object Oriented Programming

Admin

- Lab out!
- Any questions about lab or class?

OOP: OBJECT ORIENTED PROGRAMMING

What is an object?

- First, let's recall methods

- Methods: a way to group together code that performs a single task

Why use methods?

- Organization
- Avoiding Repetition
- Encapsulation

```
String[] studentNames = {"Bill", "Sam", "Cathy", "Dev"};
char[] studentGrades = {'B', 'C', 'A', 'A'};
String course = “CS136”;
for(int i =0; i < studentGrades.length; i++) {
   for(int j = i; j > 0 && studentGrades[j-1] > studentGrades[j]; j--){
       String tempName = studentNames[j];
       int tempGrade = studentGrades[j];
       studentNames[j] = studentNames[j-1];
       studentGrades[j] = studentGrades[j-1];
       studentNames[j-1] = tempName;
       studentGrades[j-1] = tempGrade;
   }
}
System.out.println(course);
for(int i = 0; i < studentNames.length; i++)
     System.out.println(studentNames[i] + ": " + studentGrades[i]);
```

```
String[] studentNames = {"Bill", "Sam", "Cathy", "Dev"};
char[] studentGrades = {'B', 'C', 'A', 'A'};
String course = “CS136”;


sortStudentsByGrade(studentNames, studentGrades);
System.out.println(course);
printStudents(studentNames, studentGrades);
```

Why use methods?

- Organization
  - Easier to read, easier to change
- Avoiding Repetition
  - Can sort other arrays; or can sort multiple times
- Encapsulation
  - Methods only* affect variables that are arguments to the method

Generalizing to Objects

- Objects group together methods and data
- All the benefits of methods
- Same benefits apply to data as well!
- Before: a program can be built up by defining a number of methods that interact with each other.
- In Java, we build up our programs by defining a number of objects that interact with each other

Classes, objects, and interfaces

Classes, objects, and interfaces

- Classes let us define our own types.

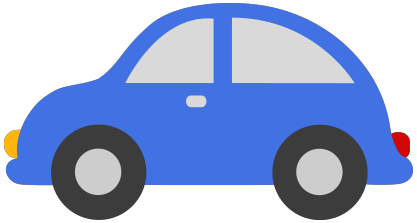
Classes, objects, and interfaces

- Classes let us define our own types.
- Objects are instances of class types

Classes, objects, and interfaces

- Classes let us define our own types.
- Objects are instances of class types
- *Example*: Think about the abstract concept of a car. Here are three instances of a car:

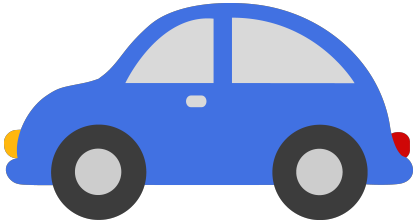




Classes, objects, and interfaces

- Classes let us define our own types.
- Objects are instances of class types
- *Example*: Think about the abstract concept of a car. Here are three instances of a car:





- Conceptually, all these cars have the same high-level interface (wheels, doors, color, transmission, top speed, etc.) but individual cars differ in their details
  - In OOP paradigm, we could *define* a car class, and then *instantiate* that class to create individual car objects.

Instructor class

- What kind of data does a 136 instructor have?
  - Email address
  - Office Hours
- I could define a class of 136Instructors
- Sam, Dan, and Lida are specific instances of 136 Instructors
  - We all have that data, but it's different for each of us
  - …in a Java context we would be objects of type 136Instructor

Object-Oriented Programming

Object-Oriented Programming

- Objects are building blocks of Java software

Object-Oriented Programming

- Objects are building blocks of Java software

- Programs are collections of interacting objects
  - Cooperate to complete tasks
  - Represent the "state" of the program
  - Communicate by sending messages to each other
    - Through *method invocation*

Object-Oriented Programming

- With enough creativity, objects can model almost anything:

Object-Oriented Programming

- With enough creativity, objects can model almost anything:
  - Physical items – cars, dice, book, students

Object-Oriented Programming

- With enough creativity, objects can model almost anything:
  - Physical items – cars, dice, book, students
  - Concepts – time, relationships

Object-Oriented Programming

- With enough creativity, objects can model almost anything:
  - Physical items – cars, dice, book, students
  - Concepts – time, relationships
  - Processing – sort, simulation, gameplay

Object-Oriented Programming

- With enough creativity, objects can model almost anything:
  - Physical items – cars, dice, book, students
  - Concepts – time, relationships
  - Processing – sort, simulation, gameplay
- Objects contain:

Object-Oriented Programming

- With enough creativity, objects can model almost anything:
  - Physical items – cars, dice, book, students
  - Concepts – time, relationships
  - Processing – sort, simulation, gameplay
- Objects contain:
  - State (instance variables)

Object-Oriented Programming

- With enough creativity, objects can model almost anything:
  - Physical items – cars, dice, book, students
  - Concepts – time, relationships
  - Processing – sort, simulation, gameplay
- Objects contain:
  - State (instance variables)
  - Functionality (methods)

Object Support in Java

- Java supports the creation of programmer-defined types called *class types*

Object Support in Java

- Java supports the creation of programmer-defined types called *class types*
- A *class declaration* defines data components and functionality of a type of object

Object Support in Java

- Java supports the creation of programmer-defined types called *class types*
- A *class declaration* defines data components and functionality of a type of object
  - Data components: *instance variable declarations*

Object Support in Java

- Java supports the creation of programmer-defined types called *class types*
- A *class declaration* defines data components and functionality of a type of object
  - Data components: *instance variable declarations*
  - Functionality: *method declarations*
    - *Constructor(s)*: special method(s) that describe the steps needed to create an object (*instance*) of this class type

```
String[] studentNames = {"Bill", "Sam", "Cathy", "Dev"};
char[] studentGrades = {'B', 'C', 'A', 'A'};
String course = “CS136”;


sortStudentsByGrade(studentNames, studentGrades);
System.out.println(course);
printStudents(studentNames, studentGrades);
```

```
String[] studentNames = {"Bill", "Sam", "Cathy", "Dev"};
char[] studentGrades = {'B', 'C', 'A', 'A'};
String course = “CS136”;


sortStudentsByGrade(studentNames, studentGrades);
System.out.println(course);
printStudents(studentNames, studentGrades);
```

- It’s very dangerous to store data this way

```
String[] studentNames = {"Bill", "Sam", "Cathy", "Dev"};
char[] studentGrades = {'B', 'C', 'A', 'A'};
String course = “CS136”;


sortStudentsByGrade(studentNames, studentGrades);
System.out.println(course);
printStudents(studentNames, studentGrades);
```

- It’s very dangerous to store data this way
- We want to store all student data in one place

```
String[] studentNames = {"Bill", "Sam", "Cathy", "Dev"};
char[] studentGrades = {'B', 'C', 'A', 'A'};
String course = “CS136”;


sortStudentsByGrade(studentNames, studentGrades);
System.out.println(course);
printStudents(studentNames, studentGrades);
```

- It’s very dangerous to store data this way
- We want to store all student data in one place
- How? Define a Student class

Two tasks towards OOP

- Define a Student class:
  - Tell Java what a Student is
  - What data does a Student have?
  - What methods do we want to associate with each Student?
    - How do we want to access Student data?

- Create a Student object
  - Then we can sort, print, etc., these objects

A Simple Class

Task: Define a type that stores information about a student: name, age, and a single grade.

A Simple Class

Task: Define a type that stores information about a student: name, age, and a single grade.

- Java class types should always (always) start with a capital letter
  - Not enforced. (But really always do this)

A Simple Class

Task: Define a type that stores information about a student: name, age, and a single grade.

- Java class types should always (always) start with a capital letter
  - Not enforced. (But really always do this)
- Java class types must be stored in a java file of the same name
  - In this case: Student.java
  - The compiler will check this!

A Simple Class

Task: Define a type that stores information about a student: name, age, and a single grade.

- Let's list these variables on the board

A Simple Class

Task: Define a type that stores information about a student: name, age, and a single grade.

- Declare a Java class called `Student` with data components (*fields/instance variables*):

```
String name;
int age;
char grade;
```

A Simple Class

Task: Define a type that stores information about a student: name, age, and a single grade.

- Declare a Java class called `Student` with data components (*fields/instance variables*):

```
String name;
int age;
char grade;
```

Instance variables are "attributes" in python!

- and methods for accessing/modifying fields:
  - "Getters": `getName`, `getAge`, `getGrade`
  - "Setters": `setAge`, `setGrade`

A Simple Class

Task: Define a type that stores information about a student: name, age, and a single grade.

- Declare a Java class called `Student` with data components (*fields/instance variables*):

```
String name;
int age;
char grade;
```

- and methods for accessing/modifying fields:
  - "Getters": `getName`, `getAge`, `getGrade`
  - "Setters": `setAge`, `setGrade`
- Declare a constructor, also called `Student`

```
class Student {
```

```
class Student {
        // instance variables
        int age;
        String name;
        char grade;
```

```
class Student {
        // instance variables
        int age;
        String name;
        char grade;

        // A constructor
        Student(int theAge, String theName,
                        char theGrade) {
                age = theAge;
                name = theName;
                grade = theGrade;
        }

        // Methods for accessing/modifying objects
        // ...see next slide...
```

```
} // end of class declaration from previous slide
```

```
        int getAge() { return age; }

        String getName() { return name; }

        char getGrade() { return grade; }

        void setAge(int theAge) {
                age = theAge;
        }

        void setGrade(char theGrade) {
                grade = theGrade;
        }
} // end of class declaration from previous slide
```

How Methods work with Data

- Methods of a class can access any instance variables
- So: an instance variable can be accessed by any method, even if it's not a parameter
  - If you've seen "global variables,"

Constructors

- Used to create ("construct") new objects of a certain class type

- Always have same name as the class
- Never have a return type
- Can have any arguments you want
  - Can have multiple constructors with different arguments

Constructors

*Principle: Use constructors to initialize the **state** of an object, nothing more.*

Constructors

*Principle: Use constructors to initialize the **state** of an object, nothing more.*

- What is state? instance variables

Constructors

*Principle: Use constructors to initialize the **state** of an object, nothing more.*

- What is state? instance variables

- Frequently constructors are short simple methods

Constructors

*Principle: Use constructors to initialize the **state** of an object, nothing more.*

- What is state? instance variables

- Frequently constructors are short simple methods
- More complex constructors will typically use helper methods. Why?

Constructors

*Principle: Use constructors to initialize the **state** of an object, nothing more.*

- What is state? instance variables

- Frequently constructors are short simple methods
- More complex constructors will typically use helper methods. Why?
  - A class may have more than one constructor!

Constructors

*Principle: Use constructors to initialize the **state** of an object, nothing more.*

- What is state? instance variables

- Frequently constructors are short simple methods
- More complex constructors will typically use helper methods. Why?
  - A class may have more than one constructor!
  - Your constructors can call other constructors or helper methods in order to reuse code

Constructors

*Principle: Use constructors to initialize the **state** of an object, nothing more.*

- What is state? instance variables

- Frequently constructors are short simple methods
- More complex constructors will typically use helper methods. Why?
  - A class may have more than one constructor!
  - Your constructors can call other constructors or helper methods in order to reuse code
    - **Never copy/paste code!!!** (ok, almost never ☺)

Creating Objects

- “new” keyword
- Tells Java to create a new object of a given class type
- Arguments are the same as the constructor arguments

```
Student s1 = new Student(32, “Sam”, ‘A’);
```

Creating Objects

```
Student s1 = new Student(32, "Sam", 'A');
```

When running this line, Java will:

- Make enough space for a new Student
- Call the constructor we wrote:

```
Student(int theAge, String theName, char theGrade) {
                age = theAge;
                name = theName;
                grade = theGrade;
}
```

Using Objects

- Use a period to accesses a variable or method of an object

```
Student s1 = new Student(32, "Sam", 'A');
//Output: Sam
System.out.println(s1.getName());
//Output: Sam
System.out.println(s1.name);
s1.name = "Sam M.";
```

Using Objects

- Use a period to accesses a variable or method of an object

```
Student s1 = new Student(32, "Sam", 'A');
//Output: Sam
System.out.println(s1.getName());
//Output: Sam
System.out.println(s1.name);
s1.name = "Sam M.";
```

Don't do this! Use getter/setter methods

IMPROVING THE STUDENT CLASS

Access Modifiers

- **`public`**, **`private`**, and **`protected`** are called *access modifiers*

Access Modifiers

- **`public`**, **`private`**, and **`protected`** are called *access modifiers*
  - They control access of other classes to instance variables and methods of a given class
    - `public` : Accessible to all other classes
    - `private` : Accessible only to the class declaring it
    - `protected` : Accessible to the class declaring it and its subclasses

Access Modifiers

- **`public`**, **`private`**, and **`protected`** are called *access modifiers*
  - They control access of other classes to instance variables and methods of a given class
    - `public` : Accessible to all other classes
    - `private` : Accessible only to the class declaring it
    - `protected` : Accessible to the class declaring it and its subclasses
- Data-Hiding Principle (encapsulation)
  - Make instance variables `private`
  - Use `public` methods to access/modify object data

```
public class Student {
        // instance variables
        private int age;
        private String name;
        private char grade;

        // A constructor
        public Student(int theAge, String theName,
                       char theGrade) {
                age = theAge;
                name = theName;
                grade = theGrade;
        }

        // Methods for accessing/modifying objects
        // ...see next slide...
```

```
        public int getAge() { return age; }

        public String getName() { return name; }

        public char getGrade() { return grade; }

        public void setAge(int theAge) {
                age = theAge;
        }

        public void setGrade(char theGrade) {
                grade = theGrade;
        }
} // end of class declaration from previous slide
```

TESTING THE STUDENT CLASS

Always test your code!

- You should never write more than 10-20 lines without testing
- 4-5 is better
- Let's test out our Student class
  - See some examples of making objects
  - How classes interact

Testing the Student Class

Testing the Student Class

```
public class TestStudent {

   public static void main(String[] args) {
      Student a = new Student(18, ”Sam", 'B');
      Student b = new Student(19, ”Bill L", 'A');
      // Some code to nicely print student details
      System.out.println(a.getName() + ", " +
         a.getAge() + ", " + a.getGrade());
      System.out.println(b.getName() + ", " +
         b.getAge() + ", " + b.getGrade());
   }
}
```

Worth Noting

- We can create as many Student objects as we need, including arrays of Students

Worth Noting

- We can create as many Student objects as we need, including arrays of Students

```
Student[] section = new Student[3];
section[0] = new Student(18, ”Huey", 'A’);
section[1] = new Student(20, ”Dewey", 'B’);
section[2] = new Student(21, ”Louie", 'A');
```

Worth Noting

- We can create as many Student objects as we need, including arrays of Students

```
Student[] section = new Student[3];
section[0] = new Student(18, ”Huey", 'A’);
section[1] = new Student(20, ”Dewey", 'B’);
section[2] = new Student(21, ”Louie", 'A’);
```

```
Student[] studentArray = new Student[4];
studentArray[0] = new Student(18, "Bill", 'B');
studentArray[1] = new Student(19, "Sam", 'C');
studentArray[2] = new Student(24, "Cathy", 'A');
studentArray[3] = new Student(20, "Dev", 'A');

//sort students
for(int i =0; i < studentArray.length; i++) {
  for(int j = i; j > 0 && studentArray[j-1].getGrade()
              > studentArray[j].getGrade(); j--){
    Student temp = studentArray[j];
    studentArray[j] = studentArray[j-1];
    studentArray[j-1] = temp;
  }
}
//print students
for(int i = 0; i < studentArray.length; i++)
  System.out.println(studentArray[i].getName() + ": " +
                     studentArray[i].getGrade());
```

SOME MORE DETAILS

Testing the Student Class

```
public class TestStudent {

   public static void main(String[] args) {
      Student a = new Student(18, ”Sam", 'B');
      Student b = new Student(19, ”Bill L", 'A');
      // Some code to nicely print student details
      System.out.println(a.getName() + ", " +
         a.getAge() + ", " + a.getGrade());
      System.out.println(b.getName() + ", " +
         b.getAge() + ", " + b.getGrade());
      // Ugly (not useful) printing (calls default toString())
      System.out.println(a);
      System.out.println(b);
   }
}
```

"Special" Methods

- Everything "inherits" from the class `java.lang.Object`
- In particular, we'll take advantage of a few methods repeatedly in this course:
  - `String toString()`
  - `boolean equals(Object other)`
  - `int hashCode()`
- Today, let's just look at `toString()`

toString()

- Every object has a toString() method whether you make one or not
  - `public String toString()`
  - You should usually make one!
- The toString() method gives a String version of the object
  - Useful!
- Why it's REALLY useful: System.out.println() calls this automatically
- Let's look at an example

Comment on scope

```
public class Student {
        // instance variables
        private int age;
        private String name;
        private char grade;

        // A constructor
        public Student(int age, String name,
                       char grade) {
                // What would age, name, grade
                // refer to here...?
        }
```

For clarity, can use ‘this’

```
public class Student {
        // instance variables
        private int age;
        private String name;
        private char grade;

        // A constructor
        public Student(int age, String name,
                       char grade) {
                this.age = age;
                this.name = name;
                this.grade = grade;
        }
        public String getName() { return this.name; }
```

this

- (like self in python)
- Some people always use this when accessing member variables
- I think it's OK to leave out
- But, helps with redundancy when accessing instance variables

INTERFACES: A WAY TO STANDARDIZE BEHAVIOR

Keeping track of a course

- Let's say we want to keep track of a course
- Course consists of Students and TeachingAssistants
- Students have:
  - int age, String name, char grade
- TeachingAssistants have:
  - int age, String name, int numHours

```
public class Student {
        // instance variables
        private int age;
        private String name;
        private char grade;

        // A constructor
        public Student(int theAge, String theName,
                       char theGrade) {
                age = theAge;
                name = theName;
                grade = theGrade;
        }

        // Methods for accessing/modifying objects
        // ...see next slide...
```

```
        public int getAge() { return age; }

        public String getName() { return name; }

        public char getGrade() { return grade; }

        public void setAge(int theAge) {
                age = theAge;
        }

        public void setGrade(char theGrade) {
                grade = theGrade;
        }
} // end of class declaration from previous slide
```

```
public class TeachingAssistant {
        // instance variables
        private int age;
        private String name;
        private int numHours;

        // A constructor
        public TeachingAssistant(int theAge, String
                theName, int theNumHours) {
                age = theAge;
                name = theName;
                numHours = theNumHours;
        }

        // Methods for accessing/modifying objects
        // ...see next slide...
```

```
        public int getAge() { return age; }

        public String getName() { return name; }

        public int getNumHours() { return numHours; }

        public void setAge(int theAge) {
                age = theAge;
        }

} // end of class declaration from previous slide
```

A Simple Task

- Let's say I want to go through all class participants (both students and TAs), print out everyone who has age = 20
- How can I do that?
  - Loop through students, check if age is 20, print if so
  - Do the same for Tas

- Let's try it

Redundancy!

- The loops are exactly the same

Redundancy!

- The loops are exactly the same
- All we're doing is getName() and getAge(). Why can't we do that in one loop?

Redundancy!

- The loops are exactly the same
- All we're doing is getName() and getAge(). Why can't we do that in one loop?
- Need a way to put both types of objects in *one array*. All we care about is having a getName() and getAge() method
  - Create an array of "things that have a getName() and getAge() method"

Interfaces

Interfaces

- We've used the term interface to colloquially describe the way that we interact with objects, but a Java `interface` is a contract

Interfaces

- We've used the term interface to colloquially describe the way that we interact with objects, but a Java `interface` is a contract
  - Defines methods (name, parameters, return types) that a class *must* implement
  - Kind of like a "class recipe"

Interfaces

- We've used the term interface to colloquially describe the way that we interact with objects, but a Java `interface` is a contract
  - Defines methods (name, parameters, return types) that a class *must* implement
  - Kind of like a "class recipe"
- Multiple classes can *implement* the same interface, and we are guaranteed that they all implement the required methods

How can we use it here?

- Students and TeachingAssistants both are people—so they both have getName() and getAge() methods
- Let's write a Person interface; a contract for these methods
- Then, let's tell Java that Students and TeachingAssistants both *implement* Person
- Try it out, and see what javac says

Removing redundancy

- Let's refactor our code to have one loop
- What is our array type?
  - Our array stores things that have getName() and getAge
  - So…it stores People!
  - Let's try it

Interfaces

- A class can *implement* an interface by providing code for each required method.

Interfaces

- A class can *implement* an interface by providing code for each required method.
- If we have code that depends only on the functionality described in the interface, that code can work for objects of any class that implements the interface!
  - Recall our eternal goal: write code exactly once
- If the methods aren't all implemented, Java gives an error

(NO) STATIC

Static Variables

Static Variables

- Variables can either be “attached” to the class or to instances of the class.

Static Variables

- Variables can either be “attached” to the class or to instances of the class.
  - Static variables **are not** associated with any one object’s state. They are usually properties or definitions.

Static Variables

- Variables can either be “attached” to the class or to instances of the class.
  - Static variables **are not** associated with any one object’s state. They are usually properties or definitions.
  - Non-static variables are called instance variables because they are tied to exactly one instance of an object. They can be accessed with the keyword ‘this’.

Static Variables

- Variables can either be “attached” to the class or to instances of the class.
  - Static variables **are not** associated with any one object’s state. They are usually properties or definitions.
  - Non-static variables are called instance variables because they are tied to exactly one instance of an object. They can be accessed with the keyword ‘this’.
  - Ask yourself: Is it possible that the value of this variable will vary across different objects?

Static Variables

- Variables can either be “attached” to the class or to instances of the class.
  - Static variables **are not** associated with any one object’s state. They are usually properties or definitions.
  - Non-static variables are called instance variables because they are tied to exactly one instance of an object. They can be accessed with the keyword ‘this’.
  - Ask yourself: Is it possible that the value of this variable will vary across different objects?
    - Consider a `Rectangle` class :
      - `numSides;`
      - `height;`

Static Variables

- Variables can either be “attached” to the class or to instances of the class.
  - Static variables **are not** associated with any one object’s state. They are usually properties or definitions.
  - Non-static variables are called instance variables because they are tied to exactly one instance of an object. They can be accessed with the keyword ‘this’.
  - Ask yourself: Is it possible that the value of this variable will vary across different objects?
    - Consider a `Rectangle` class :
      - `numSides;` static (all rectangles have 4 sides)
      - `height;`

Static Variables

- Variables can either be “attached” to the class or to instances of the class.
  - Static variables **are not** associated with any one object’s state. They are usually properties or definitions.
  - Non-static variables are called instance variables because they are tied to exactly one instance of an object. They can be accessed with the keyword ‘this’.
  - Ask yourself: Is it possible that the value of this variable will vary across different objects?
    - Consider a `Rectangle` class :
      - `numSides;` static (all rectangles have 4 sides)
      - `height;` not static (rectangles can have different dimensions)

Static Methods

Static Methods

- Methods can either be “attached” to the class or to instances of the class.

Static Methods

- Methods can either be “attached” to the class or to instances of the class.
  - Static methods **do not** depend on the state of the object. They can be answered without anything that could reference the keyword “this”. Called using the class name.

Static Methods

- Methods can either be “attached” to the class or to instances of the class.
  - Static methods **do not** depend on the state of the object. They can be answered without anything that could reference the keyword “this”. Called using the class name.
  - Non-static methods rely on an object’s state, often depending on the values of instance variables. Called on an instance.

Static Methods

- Methods can either be “attached” to the class or to instances of the class.
  - Static methods **do not** depend on the state of the object. They can be answered without anything that could reference the keyword “this”. Called using the class name.
  - Non-static methods rely on an object’s state, often depending on the values of instance variables. Called on an instance.
  - Ask yourself: Does this method depend on the state of the object, or is it always the same regardless?

Static Methods

- Methods can either be "attached" to the class or to instances of the class.
  - Static methods **do not** depend on the state of the object. They can be answered without anything that could reference the keyword "this". Called using the class name.
  - Non-static methods rely on an object's state, often depending on the values of instance variables. Called on an instance.
  - Ask yourself: Does this method depend on the state of the object, or is it always the same regardless?
    - Consider a `Rectangle` class:
      - `getArea();`
      - `calculateArea(int h, int w);`

Static Methods

- Methods can either be "attached" to the class or to instances of the class.
  - Static methods **do not** depend on the state of the object. They can be answered without anything that could reference the keyword "this". Called using the class name.
  - Non-static methods rely on an object's state, often depending on the values of instance variables. Called on an instance.
  - Ask yourself: Does this method depend on the state of the object, or is it always the same regardless?
    - Consider a `Rectangle` class:
      - `getArea();`
      - `calculateArea(int h, int w);` static (formula; all info provided as inputs)

Static Methods

- Methods can either be "attached" to the class or to instances of the class.
  - Static methods **do not** depend on the state of the object. They can be answered without anything that could reference the keyword "this". Called using the class name.
  - Non-static methods rely on an object's state, often depending on the values of instance variables. Called on an instance.
  - Ask yourself: Does this method depend on the state of the object, or is it always the same regardless?
    - Consider a `Rectangle` class:
      - `getArea();` not static (depends on a particular rectangle's dims)
      - `calculateArea(int h, int w);` static (formula; all info provided as inputs)

More Gotchas

```
public static void main(String[] args) {
     // try to access a student's age
     System.out.println(getAge());




}
```

More Gotchas

```
public static void main(String[] args) {
     // try to access a student's age
     System.out.println(getAge());
     // Wrong! Which student? getAge is not static,
     // so we need to call it on a particular object

     // try to access a student's age (correctly)
     Student s = new WilliamsStudent(18, "Ron", 'C');
     System.out.println(s.getAge());
}
```