CS 5044
Object-Oriented Programming with Java

Q&A Session

Design and language concepts

- Develop classes (singular nouns: "Vehicle") which become custom language types
  - State is maintained as a set of fields (instance variables) of any type
    - It may help to consider state as a bunch of named attributes:
      - *engineOn*: true, *speedMph*: 50, and so forth
  - Direct access to the internal state should be prohibited (by declaring fields `private`)
    - Indirect access is made available (or not) by exposing `public` methods
- Develop methods (verbs: "accelerate") within classes to allow interactions
  - Strict public interface ensures consistency and integrity of the state data
  - Methods perform controlled operations in response to requests from external code
    - **Constructors** to define the initial state
    - **Accessors** to reveal artifacts of the current state
      - Note: Artifacts may be fields or computed from fields
    - **Mutators** to allow the potential modification of state
- Construct and manipulate new instances ("objects") of class types
  - We can use our own custom class types exactly like the built-in class library types
    - A class acts as a template upon which new objects are based
  - We also have primitive types, with basic arithmetic/logic operations

Object life cycle

- Objects are constructed (via the `new` operator) as instances of a class
  - The constructor (a somewhat special method) is called to initialize the state fields
  - The `new` operator returns the address of the newly-created object
    - This address acts as a "reference" (a pointer, in other languages) to the object
- Objects can then be accessed and/or mutated by their public methods
  - Method calls are always requests; we ask the object to do something on our behalf
    - Accessors ask objects for current state information
    - Mutators ask objects to make changes (which may be refused or altered)
  - The state fields, and internal operation of each method, are hidden from other objects
- Objects cannot be deleted explicitly, but they can become unreachable
  - Unreachable: no in-scope variable, or other reachable object's field, references the object
  - Once unreachable, objects become immediately eligible for garbage collection
    - Garbage collection eventually removes the unreferenced objects from the heap

The stack and the heap

- There are two distinct storage areas within the JVM
  - The **heap** stores all object instances, including all of their fields
    - The heap changes when any objects are created, mutated, or become unreferenced
  - The **stack** stores the values of **local** variables, meaning those declared **within methods**
    - The stack changes:
      - For *primitive* variables, whenever a different *value* is assigned to them
      - For *reference* variables, whenever a different *object* (or *null*) is assigned to them
- The stack and the heap usually don't need to be considered while programming
  - However, understanding the difference between primitives and objects is vital, so...
    - ...it sometimes helps to recall what's happening behind the scenes, thus...
    - ...we may ask about these differences in quizzes, homeworks, and exams
- Please review Dr. K's Module 2 recordings for more details

Properties of Java primitives

- Numeric primitive types:
  - Two categories, all of which are "signed" (meaning positive and negative values allowed):
    - Whole numbers: `byte`, `short`, `int`, `long` (`int` is assumed for literals)
    - Floating point numbers: `float`, `double` (`double` is assumed for literals)
  - **Lengthening** operations are done automatically by default for arithmetic operations
    - Any `byte` or `short` is automatically expanded to `int` in all calculations
    - Any `int` is expanded to `long` when combined with a `long`
    - Any `int` or `long` is expanded to `float` or `double` when combined with `float` or `double`
  - **Shortening** must be done via explicit cast, due to possible overflow and/or precision loss
- Non-numeric primitive types:
  - Primitive `boolean` type
    - Values are either `true` or `false` (only)
    - Only logical operations are allowed (`&&`, `||`, and `!` represent *AND*, *OR*, and *NOT*)
    - Neither arithmetic operations nor conversions to/from numeric types are possible
  - Primitive `char` type
    - Represents one Unicode *unit* (literals are defined within single quotes, as in `'a'`)
    - Normally used in conjunction with `String` objects and/or the `Character` class
    - Arithmetic operations allowed (via UTF-16) but are very highly discouraged

Some notes about the String class

- The `String` class has special support within the Java language
  - Literals (in double quotes, such as `"Hello"`) are implicitly constructed as new String objects
  - Concatenation (via the `+` operator) of `String` objects with other entities is supported
    - Concatenation with primitives use well-defined canonical string representations
      - Numeric primitives and `char` work as expected; `boolean` is either `"true"` or `"false"`
    - Concatenation with another object implicitly calls that object's `toString()` method
      - Process then works exactly as expected for concatenation with other `String` objects
      - We'll eventually learn how to develop our own custom `toString()` methods
- The `String` class happens to be *immutable*, meaning there are no public mutators
  - Once you create a `String` object, that particular object will never change
    - Any method that manipulates a `String` object returns a *new* independent `String` object

```
String string1 = "Hello";                  // Variable string1 references a String, which is immutable
String string2 = string1.toLowerCase();    // This line of code does NOT mutate string1
```

  - Classes that happen to be immutable allow some very important optimizations
    - We'll cover these concepts extensively, much later in the semester

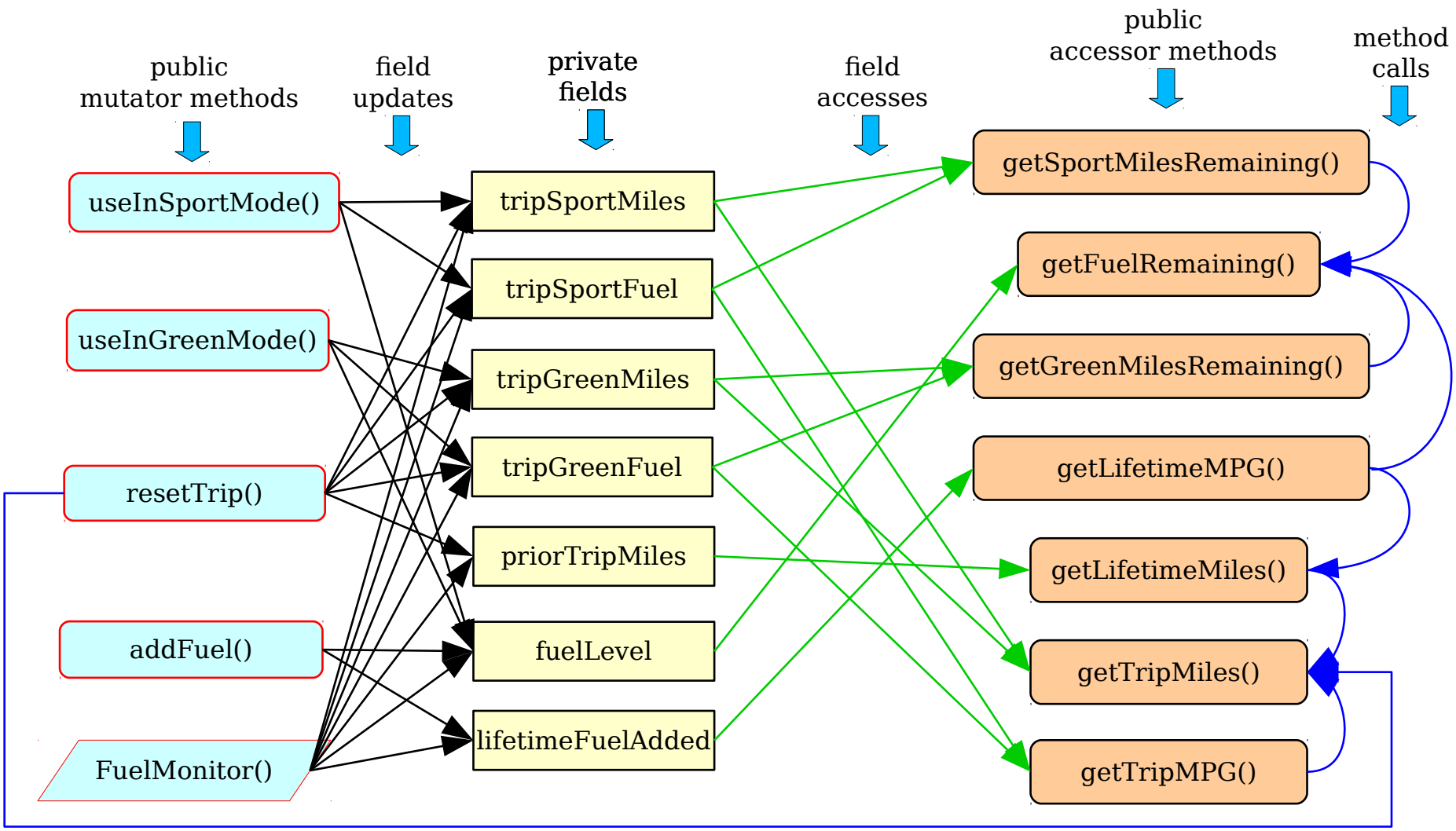
Project 1 - Review of the system requirements

- Some basic algebra is very useful:
  - `MPG = milesDriven / fuelUsed`
  - `fuelUsed = milesDriven / MPG`
  - `milesDriven = MPG * fuelUsed`
- Assumptions:
  - We receive miles driven and fuel used in each mode (sport mode or green mode)
  - We are notified when fuel is added, and when the trip meter is reset
  - No validation of parameters should be done (for this assignment only!)
    - You may safely assume that all values will be reasonable
- Other hints/guidance:
  - Truncation applies only after all calculations have been performed normally, as `double`
    - More on this in a few slides...
  - Math libraries are not needed (nor allowed)
  - Branches are not needed (nor allowed)

Project 1 - Other notes and design

- Starting with this project, a human review is part of the grading process
  - Project 1 goals and objectives:
    - Develop a single class with private fields and public accessors/mutators
    - Satisfy all the functional requirements, within the academic constraints
    - Develop test cases to exercise your code and demonstrate correctness
    - Strive for overall simplicity rather than complexity
      - A straightforward implementation is always preferred
      - Below (after the model) is a brief discussion regarding redundancies
  - See the project grading rubric in Canvas for more details
- What's so bad about code complexity? Complex code is:
  - Verbose: challenging to test adequately (much more on this, in a few weeks)
  - Fragile: easy to break when making any changes
  - Confusing: difficult to maintain over time
- Let's look at one possible design for this class
  - Your implementation does NOT need to follow this design

Project 1 - One possible sample model



Project 1 - Regarding truncation

- Two types of truncation needed:
  - Truncate downward to highest multiple of 10
    - Examples:
      - Truncate 990 to 990, 372.34 to 370, 209.99 to 200, and 9.87 to 0
    - Applies to methods:
      - `getGreenMilesRemaining()` and `getSportMilesRemaining()`
  - Truncate downward to highest multiple of 0.1
    - Examples:
      - Truncate 9.87 to 9.8, 24.75 to 24.7, 20.00 to 20.0, and 5.19 to 5.1
    - Applies to methods:
      - `getTripMPG()` and `getLifetimeMPG()`
  - Both of these can be achieve with a combination of casting, multiplication, and division
    - See Horstmann 4.2 for more details about casting and integer multiplication/division
- What is casting?
  - A type cast, in general, coerces a value of one type into another (compatible) type
  - Cast can be used to "shorten" a `double` to an `int` by removing any decimal portion
    - ```
      double x = 1.75;
      int i = (int)x; // a type in parentheses is a cast, so i is now just the int value of 1
      ```
    - This is *much* more efficient than using the equivalent `Math.floor()`
      ```

Project 1 - Regarding redundancies

- Ideally, to eliminate redundancies, we should create two private "helper" methods
  - Methods `truncateToTens()` and `truncateToTenths()` make the code more straightforward overall
  - Each truncation algorithm is written once, but is used in two places
    - This is *highly* encouraged (but not required) in Project 1
    - Note that eliminating such redundancies will be *required* in Project 2 and beyond!
- Strive to minimize (within reason) the number of fields
  - You may prefer to declare a few more fields than the model above
    - This will generally *increase* the overall amount of code you must develop
  - If you find yourself declaring significantly more fields, please reconsider your approach
    - Your implementation will become far more complex than necessary
- Please ask for help in Piazza if you're having any troubles at all!