Virginia Tech MASTERS OF INFORMATION TECHNOLOGY

# CS 5044
# Object-Oriented Programming with Java

## Q&A Session

## Comparisons

- Java relational operators are similar to those of most other languages:

| | | | | | |
|---|---|---|---|---|---|
| <= | < | == | > | >= | != |

  - Compatible with all non-boolean primitives (byte, short, char, int, long, float, double)
    - However, beware of using `==` and `!=` with double and float (see Section 5.2.2)
    - boolean primitives can only work with `==` and `!=`
- Comparing objects requires special consideration
  - Using `==` and `!=` will compare whether **references** point to the same object or not
    - Also useful for checking whether a reference is (or is not) null
      - Note: `null` is an explicit value used when a reference doesn't point to *any* object
    - These operators don't consider whether the **objects** are **equivalent** or not
      - Note: `enum` values of the same type are compared via `==` and `!=` (see Special Topic 5.4)
  - We use the `equals()` method to compare the **equivalence** of **objects**
    - Use of relational operators is a common mistake, especially with String objects:

```
String a = "a";
String b = "b";
boolean x = "ab".equals(a + b);       // x is true, as expected
boolean y = "ab" == (a + b);          // y is false (probably unexpected!)
```

    - NOTE: The `equals()` method won't (yet!) work as expected for your own classes

## Conditionals

- Conditionals are just boolean expressions, enclosed within parentheses
  - If you already have a boolean, just use it (or *NOT* it) directly, without any comparison
    - *Always* use `(valid)` or `(!valid)` rather than `(valid == true)` or `(valid == false)`
  - Avoid confusion: Be consistent, rearrange, and use parentheses when at all in doubt

```
(inStock && count == 0 || -1 != 3 + type)           // valid, but potentially confusing

((inStock && (count == 0)) || ((type + 3) != -1))     // much better equivalent
```

  - Short-circuits apply to `&&` and `||` operators, always from left to right

```
(perfect || goodEnough)                       // goodEnough is only considered if perfect is false

(perfect || isGoodEnough())                   // accessor is only called if perfect is false

(perfect || fixSomething())                   // mutator is only called if perfect is false (CAUTION!)
```

- Notable differences from some other programming languages:
  - Conditionals *must* evaluate to a formal boolean (Java primitive type)
    - All relational operators (previous slide) return a boolean
    - Common typos such as `(a = 3)` as boolean instead of `(a == 3)` are errors (that's good!)
      - Beware of boolean typos: `(valid = true)` changes `valid` to true, then evaluates as true!
  - Neither boolean expressions nor conditionals are valid as stand-alone statements
    - Some languages allow/encourage this: `perfect || fixSomething(); // not valid in Java`

## Branches: if () statements

- Branches can appear alone or in any combinations
  - The following patterns are common (names aren't standardized, nor are they important)
    - Chain:

```
if (...) { /*...*/ }
else if (...) { /*...*/ }
else if (...) { /*...*/ }
else { /*...*/ }
```

      - In certain cases, this can become a `switch()` statement (but normally better left as-is)
    - Ladder:

```
if (...) { /*...*/ }
if (...) { /*...*/ }
if (...) { /*...*/ }
```

      - Mutually exclusive sub-sequences can (and should) be converted to a chain
    - Nest:

```
if (...) {
    //...
    if (...) {
        //...
        if (...) {
            //...
        }
        //...
    }
    //...
}
```

    - Nests more than 2-3 levels deep are very hard to read, and generally discouraged

## Loops: while (), for (), and do-while () statements

- `while () { ... }` is the most fundamental loop structure
  - Conditional is re-evaluated before every entry into the loop body
    - Loop body is never executed if the conditional begins as false
    - Continues executing body statements "while" the conditional remains true
- `for () { ... }` is just a special case of a while () loop, in its own scope
  - `for (A; B; C) { ... }` is exactly equivalent to: `{ A; while (B) { ... C;} }`
  - Typically used for counted iterations (see idiomatic usage in Programming Tip 6.1)
  - It's never "wrong" to use a while () loop instead, but if the usage is appropriate...
    - ...then a for () loop is more convenient (and less surprising to other developers)
- `do { ... } while ();` is just a while () loop, with the conditional evaluated at the end
  - Guaranteed to execute the loop body at least one time
  - Much less common than the others, but quite useful in certain situations
- As with branches, nested loops are also naturally expected on occasion
  - Nests more than 2-3 levels deep are very hard to read, and generally discouraged

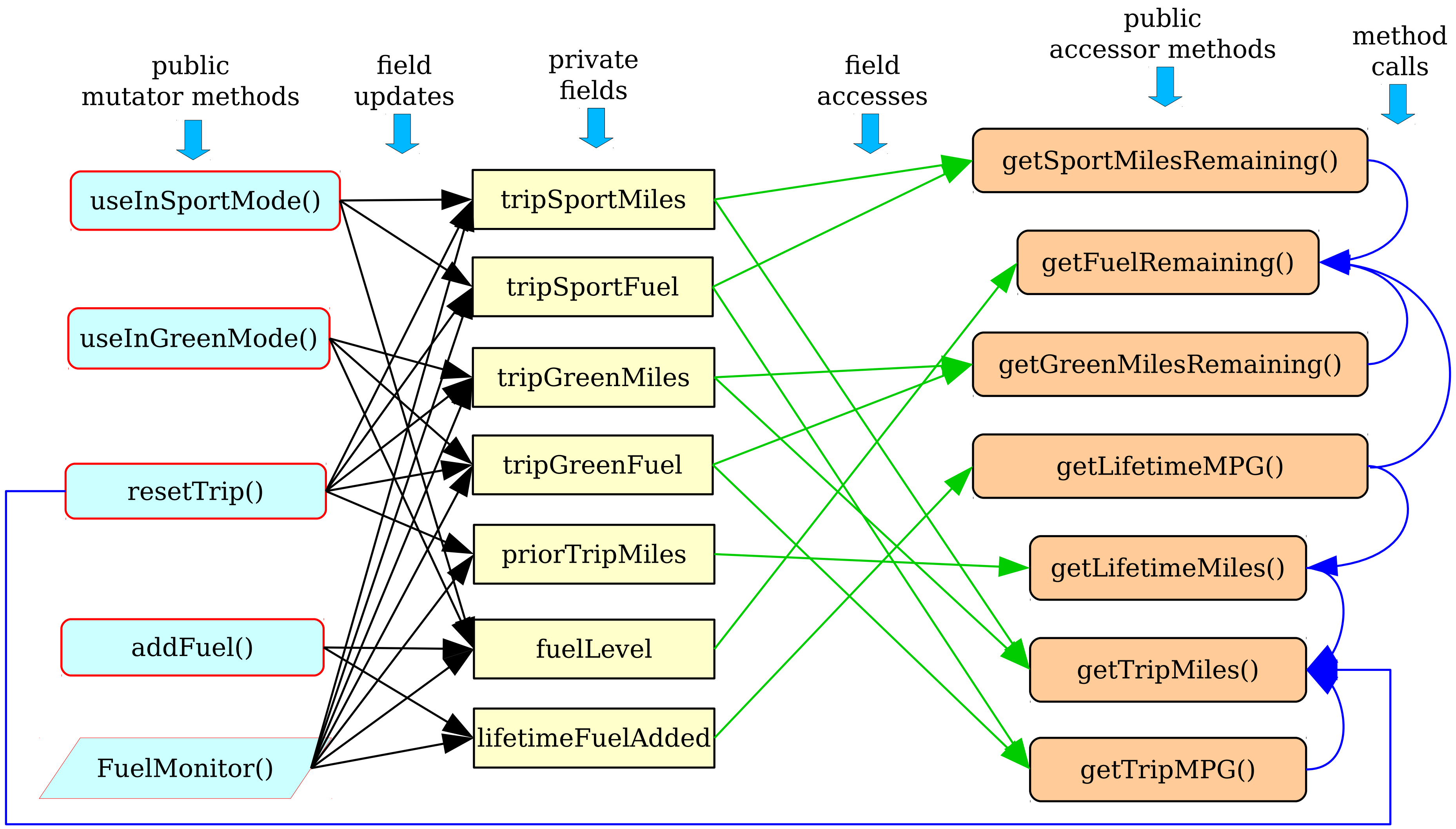
## Other considerations for branches/loops

- *Always* use braces to enclose the loop or branch body
  - Java will allow you to omit the braces, if the body contains only one statement
    - This was only done for consistency with other popular languages when Java was new
  - Please don't *ever* omit the braces!

```
while (!done)
{
    doSomething();
}
```

  - Note: Web-CAT treats this as "best practice" issue, rather than a Checkstyle rule
- Both `break` and `continue` are perfectly acceptable
  - The textbook implies that these are improper and to be avoided
    - Feel free to use them, as long as they simplify your code
  - However, don't use labels, nor the labeled-break; these are actually considered improper

# Project 1 - Programming the model


public mutator methods
field updates
private fields
field accesses
public accessor methods
method calls
useInSportMode()
useInGreenMode()
resetTrip()
addFuel()
FuelMonitor()
tripSportMiles
tripSportFuel
tripGreenMiles
tripGreenFuel
priorTripMiles
fuelLevel
lifetimeFuelAdded
getSportMilesRemaining()
getFuelRemaining()
getGreenMilesRemaining()
getLifetimeMPG()
getLifetimeMiles()
getTripMiles()
getTripMPG()