CS 5044
Object-Oriented Programming with Java

Q&A Session

Arguments and parameters

- Special Topic 8.1 might be better titled as "Terminology Nightmare"
  - The caller passes *arguments*; the called method receives *parameters*
    - This distinction hardly ever matters, except to explain certain best practices
  - Choice between "Call by Reference" and "Call by Value" was crucial in other languages
    - Horstmann is trying to explain and reconcile the terminology, but it's futile
      - The same terms mean different things in different languages
- The terminology is mostly irrelevant, as long as you understand what happens:
  - Primitive arguments: the *value* of the primitive is passed to the method
    - Reassignments inside the method are seen only within the method
      - This is highly discouraged as a bad practice anyway
  - Object arguments: the *value* of the reference is passed to the method
    - Reassignments inside the method are seen only within the method
      - This is highly discouraged as a bad practice anyway
    - Mutations of the referenced object made inside the method are seen globally
      - This is perfectly acceptable, however it should be documented
      - Javadocs and method names should make this behavior clear

Reference values vs. primitive values

- Example of differences between primitive and reference arguments:

```
public void sampleMethod(int x) {
    x = x + 10; // note warning from IDE
    System.out.print("Within method: ");
    System.out.println(x);
}

public void demo() {
    int a = 500;
    sampleMethod(a);
    System.out.print("After method: ");
    System.out.println(a);
}

Output:
Within method: 510
After method: 500
```

```
public void sampleMethod(BankAccount x) {
    x.deposit(10);
    System.out.print("Within method: ");
    System.out.println(x.getBalance());
}

public void demo() {
    BankAccount a = new BankAccount(500);
    sampleMethod(a);
    System.out.print("After method: ");
    System.out.println(a.getBalance());
}

Output:
Within method: 510.0
After method: 510.0
```

- Why is any of this important?
  - Mostly because we generally strive to minimize side-effects (See 8.2.4)
    - We always want to avoid unexpected mutation, especially of any *argument* object
    - Be sure to name all methods clearly, using appropriate conventions
      - Fully specify any side-effects in the Javadoc comments
  - This is closely related to distinguishing between accessors and mutators (8.2.3)

Static fields and static methods

- Static members (methods and fields) in general belong to the class itself
  - They're shared among all objects of this type ever instantiated
  - Note that this is very different from the "static" keyword of some other languages
- Static methods:
  - Stateless utility operations, intended to be useful without any instance
    - Examples:
      - `Math.min(int a, int b)`
      - `Character.isDigit(char c)`
      - `Double.parseDouble(String d)`
- Static fields:
  - Conventionally only used in very specific cases:
    - Declared as `private static final` to be used as *internal constants*, or...
    - Declared as `public static final` to be used as *global constants*
      - Any `final` fields must be assigned exactly once (no more; no less)
        - » It's actually only "constant" for primitives and immutable objects
        - » A mutable object referenced by a `final` field can still be mutated!
    - The textbook example of a static field as a counter is an extremely bad practice!
      - With parallel processing, this can lead to undetectable and irreversible data corruption
        - » The details are outside the scope of this course, but we'll cover this briefly later

Packages

- Note that `import` in Java is unlike in many other languages
  - Only lets short class names act as aliases for long (fully-qualified) class names
    - If you always used long class names, you would never need to import anything
- Source file directory structures are managed entirely by the IDE
  - Package names are NOT hierarchical, so the structure is very misleading!
    - Package `edu.vt` and package `edu.vt.cs5044` are completely unrelated to each other
- Developing code within a named package is always a best practice
  - Using the default (unnamed) package is allowed, but highly discouraged
- Other than naming conflicts, why do we care about multiple packages?
  - See Special Topic 8.4 (although the rationale given there is overly contrived)
  - We can grant "package private" access to all classes within a package
    - This is more restrictive than `public` but less restrictive than `private`
    - In practice, this is only used within moderately large/complex systems
  - We won't ever need to declare anything as "package private" this semester
    - Later we'll cover one other access level, but we'll probably never use it either!
    - We can always safely stick with just `public` and `private` for everything

Project 2: Test coverage of conditionals

- Every possible branch of each conditional must be exercised:

```
if (number > 0) {
    // handle positive
}
```

  - You must ensure your test cases encounter this conditional **twice**:
    - Once where the conditional is true (number > 0)
    - Once where the conditional is false (number <= 0)
- When conditionals contain redundancy, coverage can become impossible:

```
if (number > 0) {
    // handle positive
} else if (number <= 0) {
    // handle non-positive
}
```

  - Where's the redundancy? The *else* already ensures that number <= 0
    - The second conditional must be removed to achieve full coverage:

```
if (number > 0) {
    // handle positive
} else {
    // handle non-positive
}
```

    - This now only requires encountering the conditional twice, as above

Project 2: Coverage of complex conditionals

- Conditionals with `&&` and `||` connecting terms can be a lot more difficult to cover
  - With N terms, N+1 test cases are required to achieve full coverage
    - Consider `(A && B)` where N=2, meaning 3 test cases are required:
      - Case 1: A is false
      - Case 2: A is true, B is false
      - Case 3: A is true, B is true (entire conditional is true)
    - Consider `(A || B)` instead (N=2, so again need 3 cases):
      - Case 1: A is true (entire conditional is true)
      - Case 2: A is false, B is true (entire conditional is true)
      - Case 3: A is false, B is false
    - Pattern continues for larger conditionals: `(A && B && C && D)` (N=4, so 5 cases):
      - Case 1: F
      - Case 2: T, F
      - Case 3: T, T, F
      - Case 4: T, T, T, F
      - Case 5: T, T, T, T (entire conditional is true)
- Especially in if-else chains, it's very easy to unintentionally introduce redundancy
  - Such redundancy can make coverage challenging (or even impossible!)