Programming by Contract

Defensive Programming

Ch 3.6-3.7
Topics

• What can go wrong with using the following?
  ```java
double squareRoot(double n) {
    ...
    // compute x
    return x;
}
```

• So, why do your classes interact correctly?
  Options:
  - Magic!
  - Your client code agrees to..
  - Your classes check all arguments and operations for correctness
Programming by Contract

- **Programming by Contract:** Each *method* and *class* has a *contract*.
  - Client code..
  - Class..

- What the client ensures before calling the method.

- What the class ensures when method finishes.

```java
/**
 * Returns the real number x, such that x * x = n
 * Precondition:
 * Input n is 0 or greater.
 */

double squareRoot(double n) {
    // compute x
    ..
    return x;
}
```
Example

- The method assumes the client enforces the contract
  - ..
  - Client code's responsibility to ensure contract preconditions are not violated

```java
/**
 * Removes top element from the stack
 * @pre stack is not empty
 * @post stack is not full,
 * @post top element removed,
 * @post size decreased by one
 */
public void pop(){
    elements.remove(0);
}
```

- Client must be..

Example:
Stack must have an `isEmpty()` method.
Driving Analogy

- **Driving could be a contract:**
  - Given the preconditions that everyone else obeys the law, you will be safe.

- **Defensive Driving:**
  - You are never sure what other drivers will do, so always..

- **Example:**
  - **Shoulder check** when making a **left turn** to make sure nobody is illegally passing you on the left
  - Staying out of a car's **blind spot** to avoid getting hit if they fail to shoulder check while changing lanes
Defensive Programming

- A class is responsible for..
  - All input values and actions are checked for correctness.
    - ex: prevent adding a duplicate element to a "set"
    - ex: prevent adding an element to a full array.
- Brian's "Defensive Programming"
  - Find bad inputs/actions and..
  - How?..
**Assert Basics**

- **Assert (basics)**
  - **Usage:**
    ```java
    assert condition;
    ```
  - If the condition is false, ..
    (throws an `AssertionError` exception)

- **Example Statement:**
  ```java
  assert age >= 0;
  ```

- **Example Method:**
  ```java
  public void pop() {
    assert !isEmpty();
    elements.remove(0);
  }
  ```
Comparison

- Should a square-root method check that the input is non-negative?
  - Design by Contract:...
  - Defensive Programming:...
    client may call us with a bad value we should check.

- Benefit of Design by Contract
  - ..
    - otherwise client & class check for valid values.
    - Duplicate checks make system more complicated.

- Benefit of Defensive Programming
  - ..
    - Should use for all calls accessible by untrusted code.
Error Handling Options

1. - BAD idea!
   - EX: sqrt() w/o any checking or documentation

2. - Programming by contract
   - Works best with language support.
   - EX: sqrt() w/o any checking, but with documentation

3. (assert) - Check for programmer errors
   - EX: sqrt() w/ assert

4. - EX: sqrt() w/ exception

5. (null, -1, ...)
   - EX: sqrt() w/ return -1

6. - Given incorrect input, try to correct it as best as possible.
   Ex: sqrt() w/ abs(x) call to make positive.
Asserts:
Sanity tests in your code
Assertions

- **Assert statements**
  - Trigger a runtime error if a condition is false
  - ..

- **Example Usage**
  ```java
double rSquared = getCircleArea() / Math.PI;
assert rSquared >= 0;
double r = squareroot(rSquared);
```

- **Assertion failure**
  - Displays source file & line number via an exception.

```
Exception in thread "main" java.lang.AssertionError
at ca.sfu.cmpt213.AssertDemo.assertRadius(AssertDemo.java:14)
at ca.sfu.cmpt213.AssertDemo.main(AssertDemo.java:9)
```
Enabling Assertions

- **Enabling Assertions**
  - Turned on/off at runtime by JVM
  - Pass `-ea` argument to the JVM
  - `-ea` means..

- **In IntelliJ**
  Run --> Edit Configurations in VM options: add `-ea`
• Assertions check for.. which should crash the program.

• **Guide to using Asserts**
  - Assert the expectations you place..
    • **Ex:** Calling `pop()` on a non-empty stack.
  - Don't assert things that could reasonably be false.
    • **Ex:** Don't assert a user's input is > 0 because they may have typed in -1.
  - Must check for and handle these errors.
String getDescription(Car car) {
    assert car != null;
    String str = car.toString();
    return str;
}

switch(productType) {
    case SOFTWARE:
        // ...
        break;
    case HARDWARE:
        // ...
        break;
    default:
        assert false;
}

If car is null, it will generate an exception on it's own.
int age = getUserAge();
if (age < 50) {
    // ...
} else if (age >= 50) {
    // ...
} else {
    assert false;
}
Problems with Assert

• Too many asserts can..
  - Ex: in a graphics engine for a game.
  - Solution: disable them at runtime.

• Too many asserts can..
  - Solution: only use where they will help.

• Not for handling errors at runtime
  - Ex: Asserts can be disable at runtime; ..

  - Solution:
    • assert for programmer errors or “invalid” conditions.
    • use error handling for "possible" errors (file not found)
Summary

- **Programming by Contract**
  - Class states the contract
  - Client enforces it meets preconditions.

- **Defensive Programming**
  - Class ensures it's always in a valid state.
  - It validates all actions and values.

- **Use asserts to validate assumptions**
  - Check for programmer errors, not “possible” errors.
  - Asserts must be enabled in JVM (-ea)