Correctness and Robustness

Topics

- identifying correct behavior
  - method contracts
  - javadoc notation
  - pre- and postconditions
  - invariants
- ensuring correct behavior
  - testing
  - reasoning about correctness
  - assertions
- debugging strategies
- avoiding errors
  - defensive programming
  - clean code practices
- robustness
  - checking preconditions
  - handling input
  - dealing with error cases in subroutines
- exceptions

Correctness and Robustness

A *correct* module does what it is supposed to do when it is used as intended.

- wrong behavior is the fault of the module's code
- correctness is always essential

A *robust* module behaves reasonably in all situations, even when used incorrectly or when there's a problem outside of the module's control.

- bad situation is not the module's fault (but allowing badness to propagate is)
- how robust a module must be depends on its purpose and application

What does program correctness primarily refer to?

×	The ability of a program to handle unexpected user inputs gracefully.	1	6%
✓	The program producing the expected output for all valid inputs.	15	83%
×	The program running efficiently with minimal resource usage.	2	11%
×	The ease with which the program can be modified and maintained.	0	0%

these are all good things, but *correctness* has a specific meaning

Which of the following best describes a robust program?

✗	A program that executes quickly and uses minimal memory.	1	6%
✗	A program that is easy to read and maintain.	2	11%
✓	A program that continues to function correctly even with invalid or unexpected inputs.	15	83%
✗	A program that passes all predefined test cases but crashes with unexpected input.	0	0%

these are good things, but not what *robustness* refers to

Correctness

- strategies for achieving correctness
  - testing
    - each test only verifies correct behavior in a specific situation
    - with careful reasoning, a particular test can be expanded into a test case; need a thorough set of test cases
  - careful reasoning about program operation
    - formal proofs assure correctness in all cases, but can be difficult and impractical
    - can combine reasoning with runtime checks
  - error prevention
    - language features that prevent or highlight mistakes
    - coding practices to avoid or reduce problems (*defensive programming*)
- attention to correctness should occur throughout the development process

Specifying Correct Behavior

Achieving correctness requires specifying what the correct behavior is.

We usually start with a description of the method or class –

- content
  - methods – what the method does, what its parameters are for, what it returns (if anything)
  - classes – what the class represents
- written in comments
- "javadoc style" provides some structure
  - helps you remember all the pieces
  - allows for automatic generation of API documentation

javadoc-Style Comments

- for methods and classes

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argument
 * @return      the image at the specified URL
 * @see         Image
 */
public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
}
```

don't worry about @link, @see, or HTML tags like <p>

include preconditions, often in the @param description

for class comments, include @author *authorname*

Which of the following denote Javadoc comments? Choose all that apply.

✗	/* This is a Javadoc comment */	5	22%
✗	// This is a Javadoc comment	3	13%
✓	/** This is a Javadoc comment */	15	65%
✗	(* This is a Javadoc comment *)	0	0%

these are all valid comment styles in Java, but the javadoc tool only processes /** */ comments

What elements of a Java program can have Javadoc comments?

✓	a class definition	17	33%
✓	a method declaration	16	31%
✓	an instance variable	13	25%
✗	a local variable inside a method	5	10%

but typically only write javadoc style comments for public elements such as constants

/** */ is a valid comment syntax so it's not illegal inside a method, but the javadoc tool doesn't process it there

A Javadoc comment can go ... (Choose all that apply.)

✓	immediately before the element it describes	18	53%
✗	on the same line as the variable declaration	3	9%
✗	inside the method or class body	7	21%
✗	immediately after the element it describes	6	18%

/** */ is a valid comment syntax so it can be used anywhere, but for the javadoc tool it can only go immediately before the element it describes

Which of the following is a properly-formatted Javadoc comment for the method below?

```
public double calculatePower(double base, int exponent) {
    return Math.pow(base, exponent);
}
```

order of elements
– description first (first sentence should give an overview, additional sentences / paragraphs provide detail)
– `@param` next, in the order the parameters are declared
– `@return` last

format of `@param` –
`@param` *`name description`*

○
```
/**
 * @param base the base number
 * @param exponent the exponent to raise the base to
 * @return the result of the base raised to the power of the exponent
 *
 * Computes the power of a number.
 */
```

●
```
/**
 * Computes the power of a number.
 *
 * @param base the base number
 * @param exponent the exponent to raise the base to
 * @return the result of the base raised to the power of the exponent
 */
```

○
```
/**
 * Computes the power of a number.

 * @return the result of the base raised to the power of the exponent
 * @param base the base number
 * @param exponent the exponent to raise the base to
 */
```

○
```
/**
 * Computes the power of a number.
 */
```

○
```
/**
 * Computes the power of a number.
 *
 * @param the base number
 * @param the exponent to raise the base to
 * @return the result of the base raised to the power of the exponent
 */
```

`@param` omits parameter name

Questions

As long as we're conveying all the information, do we have to use Javadoc-style comments specifically?

- yes
- it's also a good idea
  - the structure of Javadoc-style comments helps remind you of the elements to include
  - programmers are familiar with the style – makes it easier to locate key information
  - can automatically generate API documentation

Public Preconditions

"...does the right thing *when used correctly*..."

- address correct usage of public methods
- state expectations for the public values the method works with
  - method's parameters
  - global variables or other state used by the method body
- only include conditions which could be violated at runtime
  - e.g. for an integer parameter, a precondition could be that the value must be > 0
  - that the value of a parameter or variable must match the declared type is not a precondition – type mismatches won't compile
  - that initialization done by a constructor has been done is not a precondition – constructor can't not have been called
- where do they go?
  - include in the method comments, often in @param tag

Which of the following are preconditions for the following method? Choose all that apply.

```
public double divide(double numerator, double denominator) {
    return numerator / denominator;
}
```

✕	numerator > 0	1	3%
✓	denominator != 0	16	50%
✕	denominator > numerator	0	0%
✕	The result must be a whole number.	0	0%
✕	numerator and denominator must be doubles	15	47%

this is a valid form for a precondition but it's not appropriate here – it's not illegal to have a numerator ≤ 0

true, but it isn't useful to state something that can't be violated at runtime

Questions

Do preconditions include things about the return value?

- no, preconditions are about what is required in order for the operation to run correctly
- postconditions address what is expected to be the case at the end

Questions

What does this syntax mean?

```
public int max(int a, int b) {
    return (a > b) ? a : b;
}
```

equivalent to

```
if ( a > b ) {
  return a;
} else {
  return b;
}
```

- ?: is called the *conditional operator* or *ternary operator*

  condition ? *expression1* : *expression2*

  - the value of this expression is *expression1* if *condition* is true and *expression2* if *condition* is false

Testing

Testing refers to running the program and verifying that the expected output is produced for the input provided.

There are many levels and types of testing.

Unit testing focuses on testing individual units of code – such as methods.

- small pieces are easier to debug
- the whole program won't work if the individual parts don't

It doesn't replace overall *system testing*, but it helps localize errors and ensure code coverage.

Test Cases

A *test case* for a method has four components –

- a *descriptive name* which concisely identifies what the test case is testing
  - each test necessarily only tests a specific input, but that input can represent a category of specific inputs for which the program carries out the same steps
- a *starting state*
  - the state of the object the method is invoked on, as well as the state of anything else relevant to the test (other than parameters) – everything that needs to be set up so the method being tested can be called
- the *input*
  - values of the method's parameters, and anything else the method might take in from outside (the method or program) as it executes (e.g. user input)
- the *expected result*
  - the return value, printed output, instance variable values, etc that should result from the method's correct execution given the starting state and input

Example

```
/**
 * Determine if the specified element is in the
 * array.
 *
 * @param array the array
 * @param elt element to look for
 * @return true if elt is in the array, false
 *   otherwise
 */
public static boolean contains ( int[] array,
                                 int elt ) { … }
```

name	contains
starting state	n/a
input	array – [30, 10, 40, 20] elt – 10
expected result	true

Identifying Test Cases

- cover all of the different behaviors that the method may have
  - *black box testing* – test cases are derived from the specifications of the method only, without looking at the method body
  - tests the abstraction
- cover every line of code
  - *white box testing* – test cases are based on analyzing the code itself
  - tests the implementation

Example – Black Box Test Cases

```
/**
 * Determine if the specified element is in the
 * array.
 *
 * @param array the array
 * @param elt element to look for
 * @return true if elt is in the array, false
 *   otherwise
 */
public static boolean contains ( int[] array,
                                 int elt ) { … }
```

- contains – present
  - input
    - array – [30, 10, 40, 20]
    - elt – 10
  - expected result – true
- contains – not present
  - input
    - array – [30, 10, 40, 20]
    - elt – 50
  - expected result – false

Example – White Box Test Cases

```
public static boolean contains ( int[] array,
                                 int elt ) {
  for ( int i = 0 ; i < array.length ; i++ ) {
    if ( array[i] == elt ) { return true; }
  }
  return false;
}
```

- loop body never executes
  - 0 iterations – loop condition is false the first time
- loop body executes at least once
  - `if` condition is true – loop exits because `array[i] == elt`
  - `if` condition is never true – loop exits because `i == array.length`

- empty array (length 0)
  - input – array [], elt 50
  - expected result – false
- contains – present
  - input – array [30, 10, 40, 20], elt 10
  - expected result – true
- contains – not present
  - input – array [30, 10, 40, 20], elt 50
  - expected result – false

Being Thorough

- for black-box testing, cover all of the possible behaviors as identified in the method contract
  - e.g. for a boolean method, cover both the `true` result and the `false` result
- for white-box testing, cover all of the code e.g.
  - loops: 0 repetitions (condition is false the first time), at least 1 repetition (condition is true the first time, loop body is run at least once)
  - each case in an 'if' statement, including the case of none of them (if possible)
  - each combination of values in boolean expressions containing && and ||

Being Thorough

More is not automatically better – redundant test cases just waste time.

Testing correct things is also a waste of time – focus on test cases for what is likely to fail.

- balance the consequences of missing a bug with the effort of unnecessary testing

- *don't* write test cases for simple things where you can confidently reason about the correctness of the code
  - but bugs can still creep in to simple things, and simple things may become less simple as development continues
- *do* write test cases if checking if something worked or not is easier than reasoning about the correctness of the code
  - but testing is not a perfect substitute for reasoning about correctness
- *do* test special cases and trouble spots where bugs often arise
  - black box tests covering cases which often require unique code paths

Being Thorough

- include the typical case(s) – e.g.
  - middle element in a non-empty collection

```
public static boolean contains ( int[] array,
                                 int elt ) { … }
```

- contains – present middle
  - input
    - array – [30, 10, 40, 20]
    - elt – 10
  - expected result – true

Being Thorough

- typical special cases – e.g.
  - empty collections or collections with only one element
  - end conditions – involving first or last element
  - off-the-end conditions – before the beginning or after the end

```
public static boolean contains ( int[] array,
                                 int elt ) { … }
```

- contains – empty array
- contains – first element
  - input
    - array – [30, 10, 40, 20]
    - elt – 30
  - expected result – true
- contains – last element
  - input
    - array – [30, 10, 40, 20]
    - elt – 20
  - expected result – true

Being Thorough

- typical bugs and trouble spots – e.g.
  - off-by-one in counting loops – 1 repetition, max repetitions
  - where `null` values can arise

```
public static boolean contains ( int[] array,
                                 int elt ) {
  for ( int i = 0 ; i < array.length ; i++ ) {
    if ( array[i] == elt ) { return true; }
  }
  return false;
}
```

- contains – not present
- contains – single element (present)
  - input
    - array – [30]
    - elt – 30
  - expected result – true

Summary

- contains – present (middle)
  - input
    - array – [30, 10, 40, 20]
    - elt – 10
  - expected result – true
- contains – empty array
  - input
    - array – []
    - elt – 50
  - expected result – false
- contains – first element
  - input
    - array – [30, 10, 40, 20]
    - elt – 30
  - expected result – true
- contains – not present
  - input
    - array – [30, 10, 40, 20]
    - elt – 50
  - expected result – false
- contains – single element
  - input
    - array – [30]
    - elt – 30
  - expected result – true
- contains – last element
  - input
    - array – [30, 10, 40, 20]
    - elt – 20
  - expected result – true

Designing for Testing

Implementing test cases for class methods may require access to private instance variables to set up starting state or check the expected result.

- add what is needed, but try to grant as little extra access as possible
  - for testing code in the same package, use the default (no keyword) access modifier rather than `public`
  - add a getter method or constructor rather than making instance variables less `private`
  - consider returning a “safe” representation rather than granting direct access
    - e.g. to `String()` to return a string version of the contents instead of returning the array of elements
  - consider implementing the check (at least partially) within the class rather than in the tester