Correctness and Robustness

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

CPSC 225: Intermediate Programming • Spring 2025

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

Topics

 identifying correct behavior robustness checking preconditions

handling input

subroutines

exceptions

dealing with error cases in

- 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

CPSC 225: Intermediate Programming . Spring 2025

Vha	t does program correctr	ness primarily refer	to?	
×	The ability of a program to handle unexpected user inputs gracefully.	1	6%	these are all good things, but <i>correctness</i> has a specific meaning
~	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%	

Nhi	ich of the following best	describes a robus	t program?		
×	A program that executes quickly and uses minimal memory.	1	6%		these are good things, but not wh
×	A program that is easy to read and maintain.	2	11%	ſ	robustness refers
~	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%		

Specifying Correct Behavior

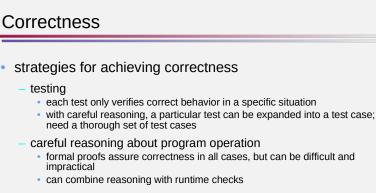
Achieving correctness requires specifying what the correct behavior is.

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

CPSC 225: Intermediate Programming • Spring 2025

- "javadoc style" provides some structure
 - helps you remember all the pieces
 - allows for automatic generation of API documentation



- 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

CPSC 225: Intermediate Programming • Spring 2025

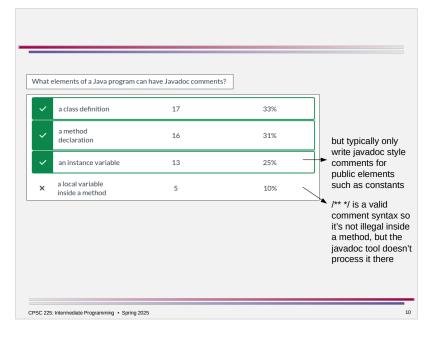
javadoc-Style Comments

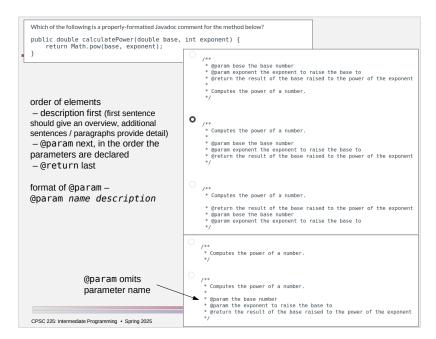
• for methods and classes

} authorname

CPSC 225: Intermediate Programming • Spring 2025







A Javadoc comment can go ... (Choose all that apply.) immediately before the element it 18 53% describes on the same line as 3 9% × the variable declaration inside the method × 7 21% or class body immediately after × the element it 6 18% describes

/** */ 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

11

CPSC 225: Intermediate Programming • Spring 2025

CPSC 225: Intermediate Programming • Spring 2025

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

CPSC 225: Intermediate Programming . Spring 2025

- 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

Which of the following are preconditions for the following method? Choose all that apply. public double divide(double numerator, double denominator) { return numerator / denominator; this is a valid form for a precondition numerator > 0 × 1 3% but it's not appropriate here -16 50% denominator != 0 it's not illegal to have a numerator ≤ 0 denominator > × 0 0% numerator The result must be a × 0 0% whole number. true, but it isn't useful numerator and to state something denominator must 15 47% × that can't be violated be doubles at runtime CPSC 225: Intermediate Programming • Spring 2025

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

Questions

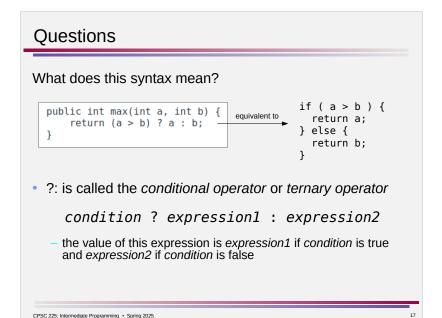
13

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

CPSC 225: Intermediate Programming • Spring 2025

4



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

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.

Example

CPSC 225: Intermediate Programming . Spring 202

/**

 $\overset{*}{}$ 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
- */

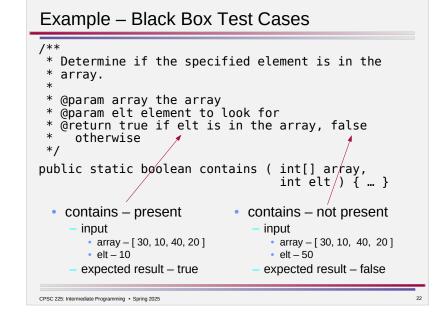
name	contains	
starting state	n/a	
input	array – [30, 10, 40, 20] elt – 10	
expected result	true	
CPSC 225: Intermediate Programming • Spri	ng 2025	20

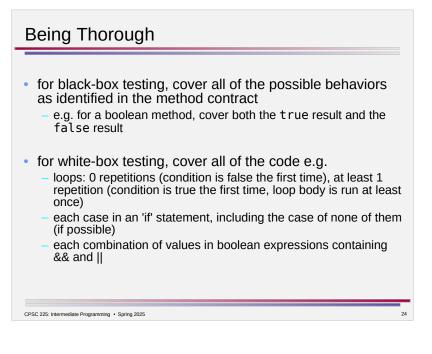
Identifying Test Cases

- 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

CPSC 225: Intermediate Programming • Spring 2025

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; } empty array (length 0) loop body never executes input – array [], elt 50 0 iterations - loop condition is false expected result - false the first time contains – present loop body executes at least input - array [30, 10, 40, 20], once elt 10 if condition is true - loop exits expected result – true because array[i] == elt if condition is never true – loop contains – not present exits because i == array.length input - array [30, 10, 40, 20], elt 50 expected result - false CPSC 225: Intermediate Programming • Spring 2025



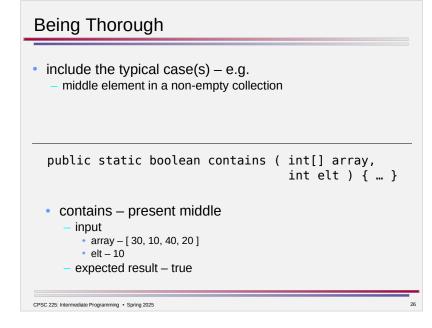


Being Thorough

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

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

Being Thorough

typical bugs and trouble spots – e.g.

off-by-one in counting loops – 1 repetition, max repetitions

– where null values can arise

Summary	
 contains – present (middle) 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
 contains – empty array input array – [] elt – 50 expected result – false 	 contains – single element input array – [30] elt – 30 expected result – true
 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
CPSC 225: Intermediate Programming • Spring 2025	29

acċ	lementing test cases for class methods may require ess to private instance variables to set up starting stat check the expected result.
	dd what is needed, but try to grant as little extra acces s 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. toString() 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

30

CPSC 225: Intermediate Programming • Spring 2025