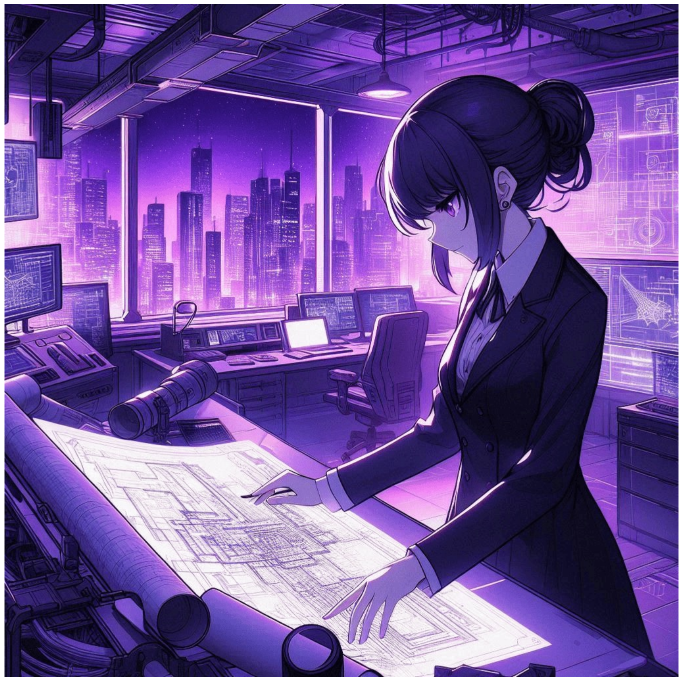


CSE 331

Specifications & Testing

James Wilcox and Kevin Zatloukal

The Need For Software Design

- Large scale software must be correct and…
  - easy to **test**
  - easy to **understand**
  - easy to **change**

most work in industry is changing existing code

- It does not naturally have these properties
  - naturally tends toward "spaghetti code"
  - if anything, AI makes this worse

- Latter properties are provided via **abstraction**

Abstraction

- An <u>abstraction</u> hides details in part of the code
  - "high level" description, avoiding "low level" details
- Makes the code more **understandable**
  - client does not need to learn those details
- Makes the code more **changeable**
  - implementer can change details hidden from client

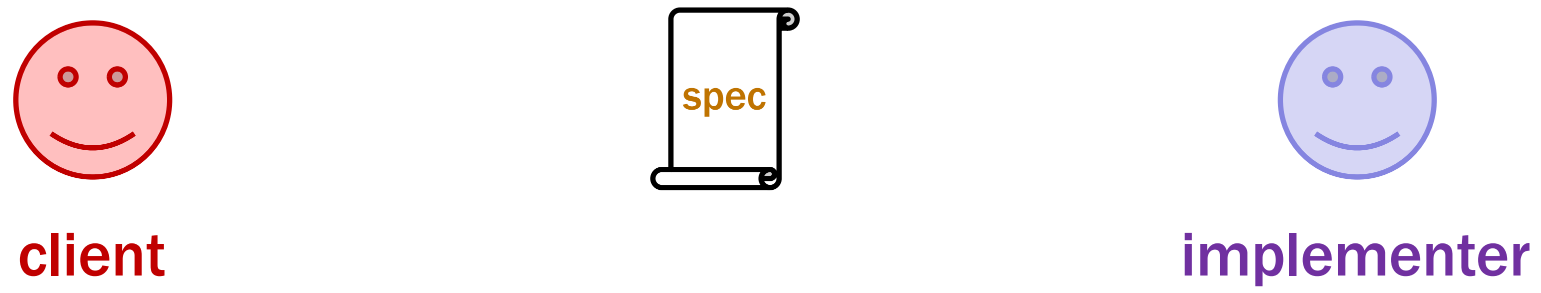
client

implementer

Procedural Abstraction

- Provided for a method via its <u>specification</u>
  - must describe the input/output behavior
  - need not describe every detail of how it works

- Usually called "procedural abstraction"
  - method aka function aka procedure aka routine
  - we will see other kinds in the future...

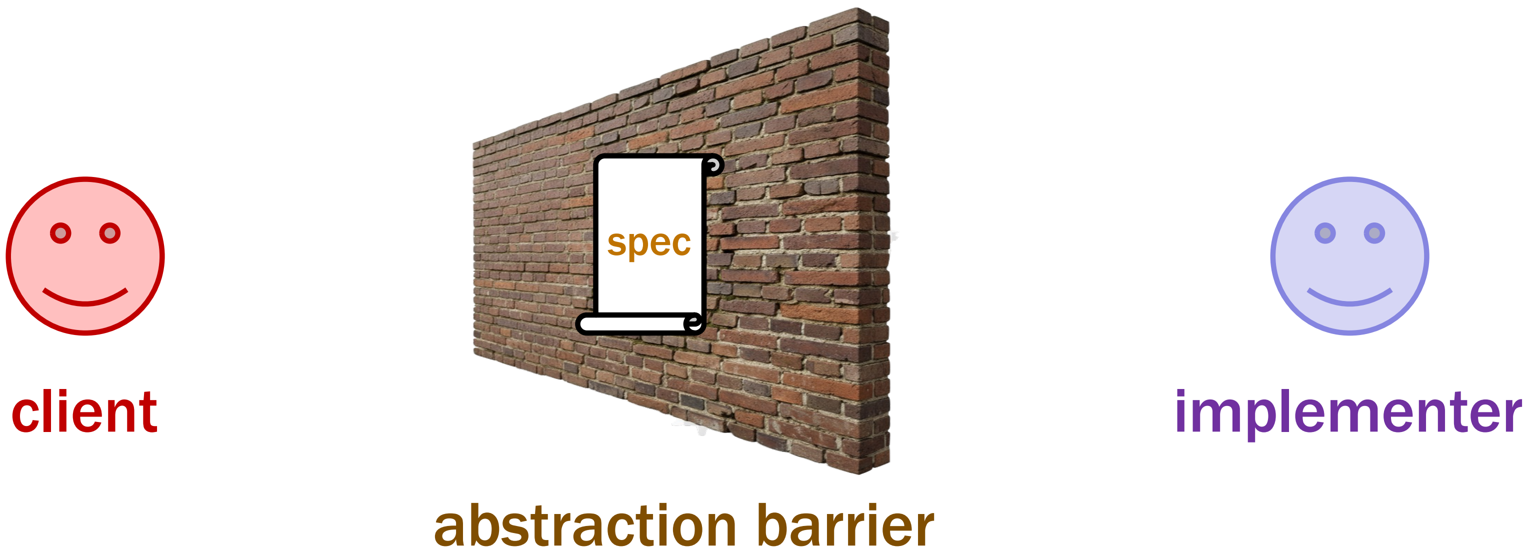


Abstraction Barrier

- "Abstraction barrier" between client & implementer
  - implementer promises code satisfies the specification

    free to choose any implementation within those constraints
  - client promises not to depend on hidden details

    will depend only on details included in the specification



Specifications

- A good specification is <u>precise</u>
  - cannot have confusion about required behavior

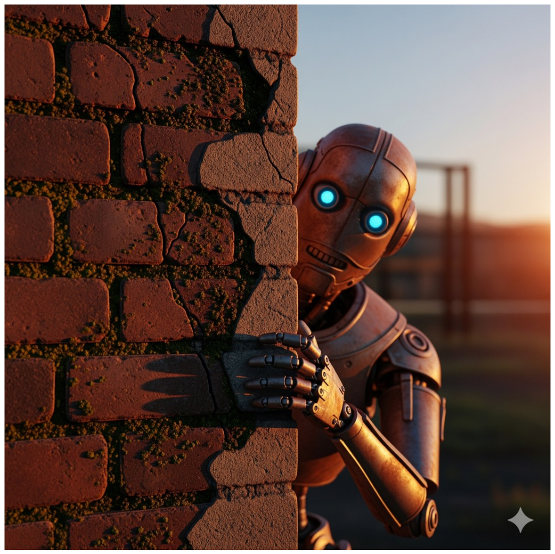
- A good specification…
  - hides details that may **change**
  - hides details that are hard to **understand**
  - provides enough detail to be **useful**

- Creating abstractions requires judgement
  - no sure-fire formula for how to do this
  - none of us can perfectly foresee future changes

AI and Abstraction

- At present, AI does not respect abstraction barriers
  - it should be possible to fix this in the future
  - for now, you have to police this yourself

- Writing specs is necessary
  - must distinguish details that are incidental vs essential
  - AI cannot read your mind!

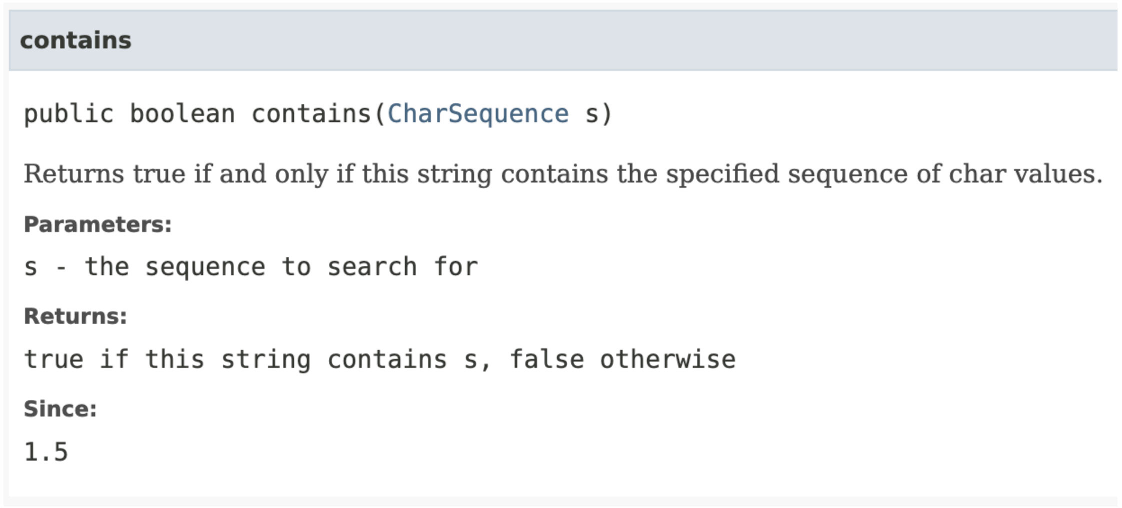


Specifications in Java

Writing Method Specifications in Java

- Java writes method specs in special comments
  - immediately before the method
  - using `/** .. */` comment format

- Produces HTML documentation from comments



contains

`public boolean contains(CharSequence s)`

Returns true if and only if this string contains the specified sequence of char values.

Parameters:

`s - the sequence to search for`

Returns:

`true if this string contains s, false otherwise`

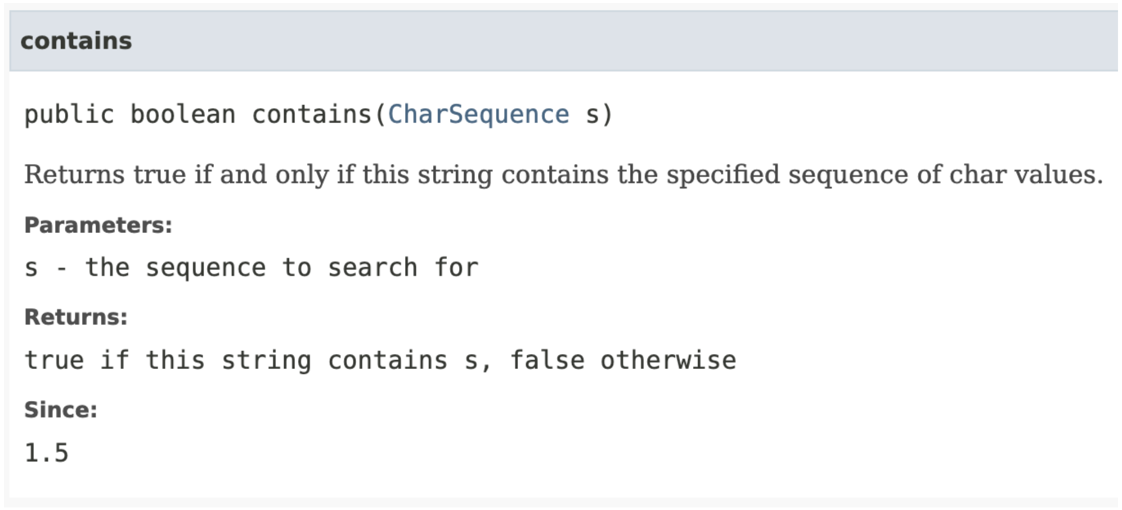
Since:

`1.5`

incredibly important feature of Java

Writing Method Specifications in Java

- Each Javadoc comment includes
  - an overview sentence
  - explanations of each parameter
  - explanations of what is returned
  - any exceptions thrown and, if so, under what conditions



contains

```
public boolean contains(CharSequence s)
```

Returns true if and only if this string contains the specified sequence of char values.

Parameters:

`s - the sequence to search for`

Returns:

`true if this string contains s, false otherwise`

Since:

`1.5`

Writing Method Specifications in Java

- Overview comment is the first sentence
  - shown in the IDE when you hover over method name
  - followed by any amount of additional explanation

- Other parts are included with `@tag` syntax
  - **`@param`** `name` explains what name is for
  - **`@return`** explain in detail what return values mean
  - **`@throws`** `E` explain when exception `E` is thrown

- We will add some others…

Method Specifications in General

- Specification consists of two parts
  - precondition says what inputs are allowed
  - postcondition says what result for allowed inputs
- Client promises only to pass allowed inputs
  - inputs will satisfy the precondition
- Implementer promises results will be as expected
  - outputs will satisfy the postcondition
  - no promises if the precondition does not hold!

Writing Method Specifications in Java

- Preconditions on individual parameters
  - often included in the `@param`

```
@param n Name to look for. Must be non-null
```

- Preconditions on multiple parameters
  - do not fit well in standard Javadoc tags
  - we will add our own `@requires`

```
@param i An index in the array
@param A The array to look in
@requires 0 <= i < A.length
```

Writing Method Specifications in Java

- Method call either returns or throws an exception
  - one or the other, not both
  - describe in `@return` and `@throws`

```
@param n The number to look for
@return an index i such that A[i] = n
@throws NotFound if n is not in A
```

- Postcondition only specified for allowed inputs
  - contradictory to disallow and then say what happens

```
@requires 0 <= i < A.length
@throws InvalidArg if 0 < i or A.length <= i
```

Writing Method Specifications in Java

- Every input falls in one of three cases:
  1. input is disallowed
  2. input is allowed and will return something
  3. input is allowed and will throw something

- Item 1 is the precondition
  – explained in `@param` and `@requires`

- Items 2-3 are the postcondition
  – explained in `@return` and `@throws`

Example 1

```
/**
 * Returns the index of a number in the list.
 * @param A The list to look in. Must be non-null
 * @param v The value to look for
 * @return an index i such that A[i] = v
 * @throws NotFound if no such index exists
 */
int indexOf(List<Integer> A, int v);
```

What is are other reasonable specifications?

Example 2

```
/**
 * Returns the concatenation of two lists.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null
 * @return a list containing the elements of A
 *     followed by all the elements of B.
 */
List<Integer> concat(
    List<Integer> A, List<Integer> B);
```

What is another reasonable specification?

Example 3

```
/**
 * Adds the elements in two lists.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null
 * @requires A.size() = B.size()
 * @return list C of the same length, where
 *    C's value is the sum of those at the
 *    corresponding indexes in A and B
 */
List<Integer> addLists(
    List<Integer> A, List<Integer> B);
```

Is it 100% clear what this returns?

Comparing Specifications

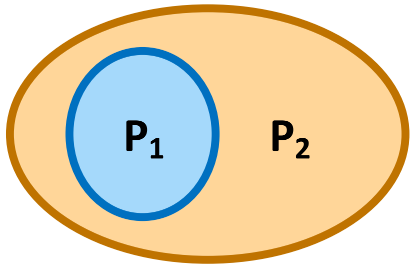
Propositions

- Precondition & Postcondition are "propositions"
  - T/F claims about values in the program

- We say that P_1 is stronger than P_2...
  - whenever P_1 is true, P_2 is also true
  - values satisfying P_1 are a subset of those for P_2

- We say that P_2 is weaker than P_1

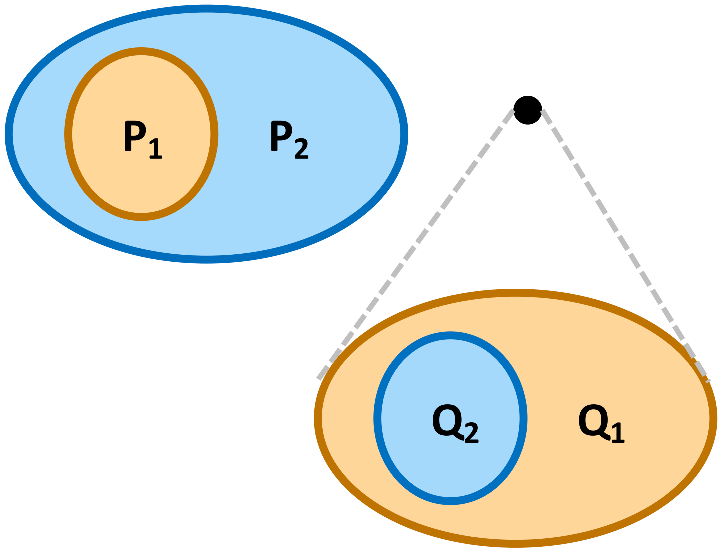
311 alert: P_1 implies P_2



Comparing Specifications

- Specification S_1 is stronger than S_2 if it has...
  - a stronger postcondition   and the same precondition
  - a weaker precondition   and the same postcondition
  - (or both)

precondition



postcondition
(for a **fixed** input)

Comparing Specifications in Java

- Specification S_1 is stronger than S_2 if it has...

- A stronger postcondition:
  - smaller subset of allowed outputs for a given input
  - e.g., return value "is between 1 and 100"
    is stronger than "is positive"

- A weaker precondition:
  - larger subset of allowed inputs
  - e.g., allowing all integer values instead of positive ones

Comparing Specifications

- Specification S_1 is stronger than S_2 if it has…
  - a stronger postcondition
  - a weaker precondition
  - (or both)

- Not all specifications are stronger or weaker
  - all others are called "incomparable"
  - most pairs of specifications are incomparable

    stronger and **weaker** are special cases

Example 4

```
int indexOf(List<Integer> A, int v);

// Specification A
// @requires value v occurs somewhere in A
// @return an index i such that A[i] = v

// Specification B
// @requires value v occurs somewhere in A
// @return the smallest i such that A[i] = v
```

How does A relate to B?

Example 5

```
int indexOf(List<Integer> A, int v);

// Specification A
// @requires value v occurs somewhere in A
// @return an index i such that A[i] = v

// Specification C
// @return an index i such that A[i] = v
//     if v appears in A and otherwise -1
```

How does A relate to C?

Example 6

```
int indexOf(List<Integer> A, int v);

// Specification B
// @requires value v occurs somewhere in A
// @return the smallest i such that A[i] = v

// Specification C
// @return an index i such that A[i] = v
//      if v appears in A and otherwise -1
```

How does B relate to C?

Why Do We Care?

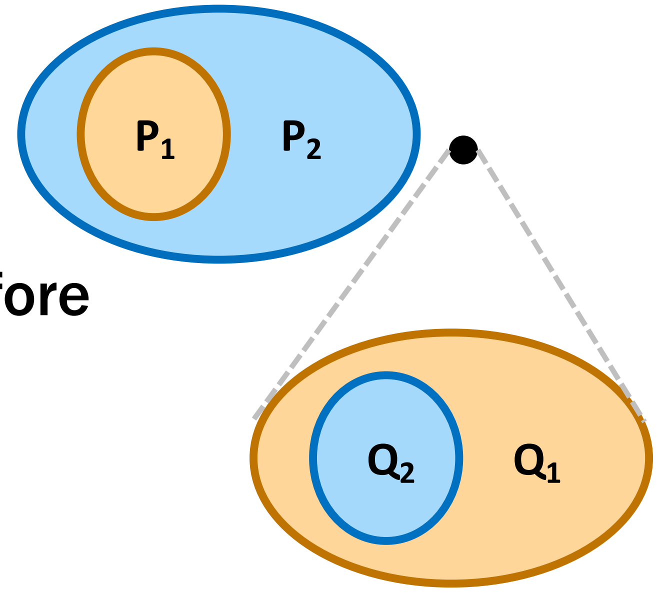
- Specification S_1 is stronger than S_2 if it has...
  - a weaker precondition
  - a stronger postcondition

- Changing from S_2 to S_1 (strengthening)...
  - cannot break any clients!
  - only allows more inputs

    old inputs in P_2 are still in P_1
  - all outputs were also allowed before

    clients' code was ready to handle all Q_2

    this includes all Q_1 also



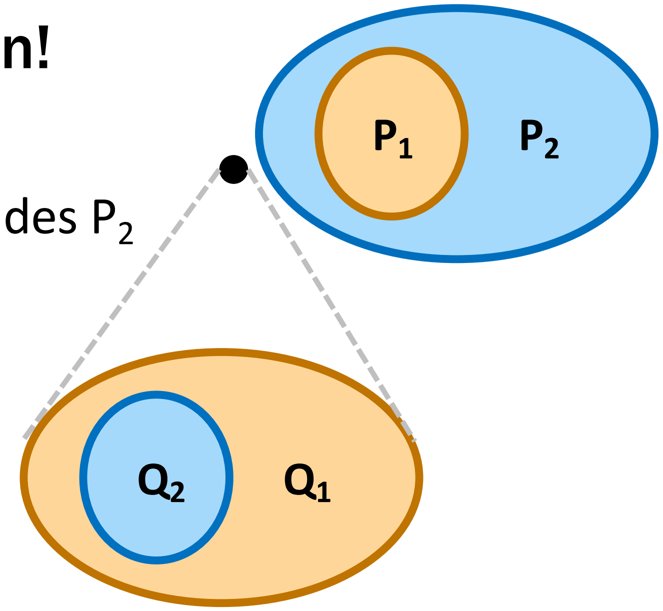
Why Do We Care?

- Specification S_1 is stronger than S_2 if it has...
  - a weaker precondition
  - a stronger postcondition

- Changing from S_1 to S_2 (weakening)...
  - cannot break the implementation!
  - allows fewer inputs

    current code handled all P_1, which includes P_2
  - allows more outputs

    current code returns values in Q_1, which is still inside of Q_2



Why Do We Care?

- Specification S_1 is stronger than S_2 if it has...
  - a weaker precondition
  - a stronger postcondition
- Changing from S_1 to S_2 (weakening)...
  - cannot break the implementation!
- Changing from S_2 to S_1 (strengthening)...
  - cannot break any clients
- Tells us what we need to fix with this change

Which Is Better?

- Changing from S_1 to S_2 (weakening)…
  - cannot break the implementation!
- Changing from S_2 to S_1 (strengthening)…
  - cannot break any clients!

- In principle, neither stronger nor weaker is better

- With few clients, weakening is easier
- With many clients, strengthening is easier
  - with 1,000 clients, weakening is *impossible*

Varieties of Specifications

Kinds of Specifications

- **Declarative** specification says <u>what</u> the answer looks like
  - does not say how to calculate it
  - up to us to ensure that our code satisfies the spec

- **Imperative** specification says <u>how</u> to calculate the answer
  - lays out an exact steps to perform to get the answer

    may **not** be the steps used by the implementation!

- Can implement a *different* set of steps than in the spec
  - up to us to ensure that our code satisfies the spec

Example: Imperative Specification

- Absolute value: $|x| = x$ if $x \geq 0$ and $-x$ otherwise
  - definition is an "if" statement

```
int abs(int x) {
  if (x >= 0) {
    return x;
  } else {
    return -x;
  }
}
```

just translating math to Java

Example: Declarative Specification

- **Subtraction** ($a - b$): return x such that $b + x = a$
  - can see that $b + (a - b) = b + a - b = a$

```
int subtract(int a, int b) {

    ??

}
```

we are left to figure out how to do this…
and convince ourselves it satisfies the spec

Example: Declarative Specification

- Square root of x is number y such that $y^2 = x$
  - not all positive integers have integer square roots, so… let's round up
  - $(y - 1)^2 \leq x \leq y^2$

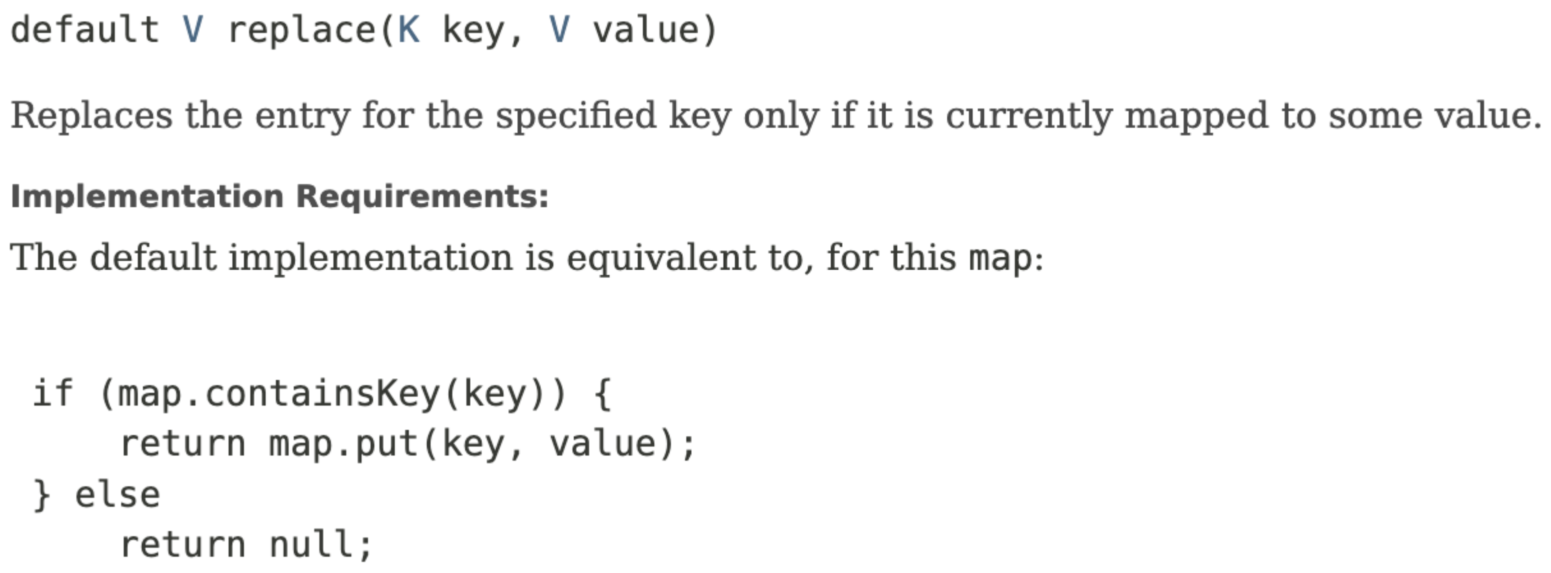
    smallest integer y such that $x \leq y^2$

```
int sqrt(int x) {

    ??

}
```

we are left to figure out how to do this…
and convince ourselves it satisfies the spec

Examples from the Java APIs

`java.util.Map` — set of (key, value) pairs



```
default V replace(K key, V value)
```

Replaces the entry for the specified key only if it is currently mapped to some value.

Implementation Requirements:

The default implementation is equivalent to, for this `map`:

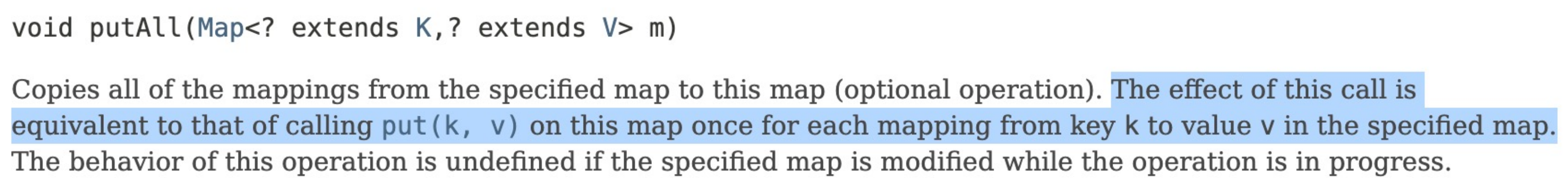
```
 if (map.containsKey(key)) {
     return map.put(key, value);
 } else
     return null;
```

Imperative or declarative?

Imperative

Examples from the Java APIs

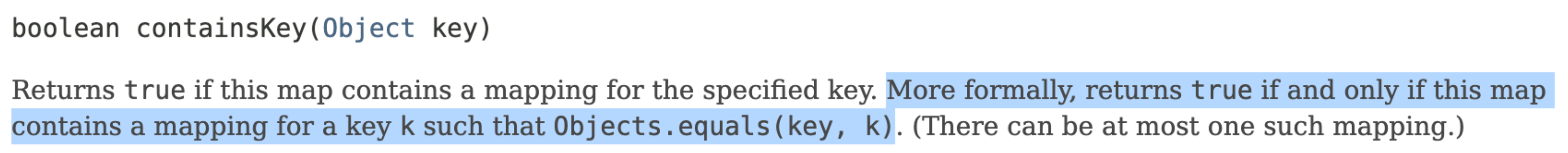
`java.util.Map` — set of (key, value) pairs



```
void putAll(Map<? extends K,? extends V> m)
```

Copies all of the mappings from the specified map to this map (optional operation). The effect of this call is equivalent to that of calling `put(k, v)` on this map once for each mapping from key `k` to value `v` in the specified map. The behavior of this operation is undefined if the specified map is modified while the operation is in progress.

Imperative



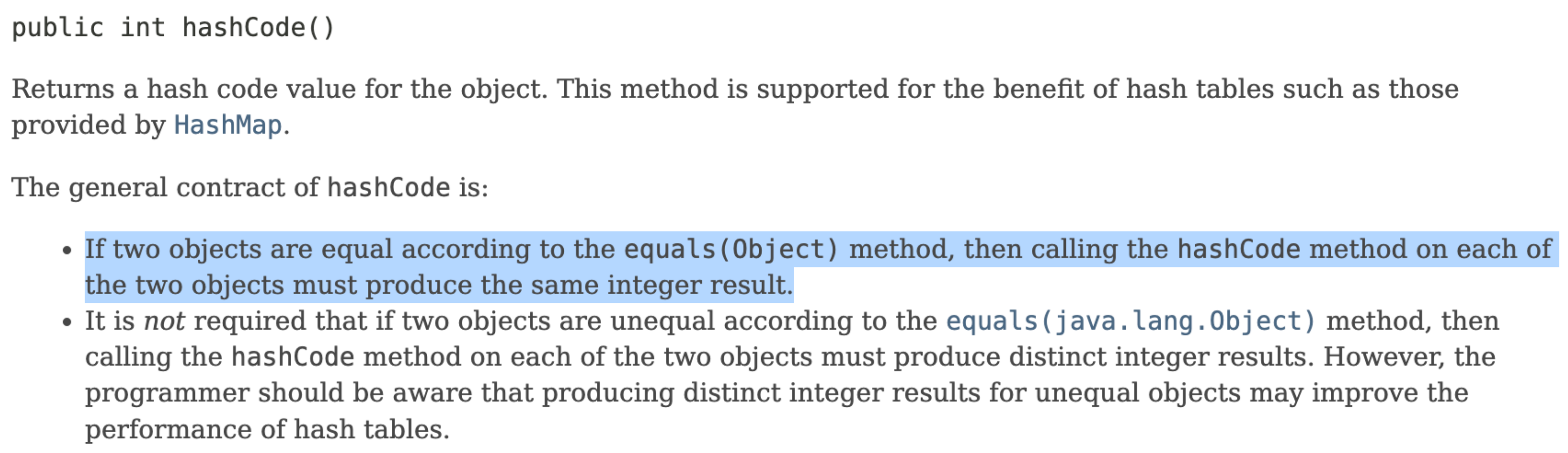
```
boolean containsKey(Object key)
```

Returns `true` if this map contains a mapping for the specified key. More formally, returns `true` if and only if this map contains a mapping for a key `k` such that `Objects.equals(key, k)`. (There can be at most one such mapping.)

Declarative

Examples from the Java APIs

`java.util.Object`



`public int hashCode()`

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of `hashCode` is:

- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

Declarative

Recall: Example 1

```
/**
 * Returns the index of a number in the list.
 * @param A The list to look in.
 * @param v The value to look for
 * @return an index i such that A[i] = v
 * @throws NotFound if no such index exists
 */
int indexOf(List<Integer> A, int v);
```

Is this imperative or declarative?

Declarative

Which is Better?

- **Like stronger vs weaker, one is not always better**

- **Generally speaking...**
  - **imperative are easier**

    easier for implementers (if they implement the algorithm as is)

    easier for clients to reason about
  - **declarative gives more room for change**

    making the code easy to change is often important
  - **but neither of these is always true**

Formal Specifications

Next Up…

- Specifications must be <u>precise</u>
  - can’t have disagreement about what is correct

- Informal descriptions (English) are often imprecise
  - normal practice is to use English when it is clear,
    but switch to formal description when confusion is possible

    requires practice and judgement
  - we will start out completely formal to make it easier
  - can't talk about English ambiguity without unambiguous language

    gives you a toolkit for when the English is unclear

- Start out by describing our formal language
  - will cover data and code
  - programming language independent (not Java specific)

Math Notation

Basic Data Types in Math

- In math, the basic data types are “sets”
  - sets are collections of objects called **elements**
  - write x $\in$ S to say that “x” is an element of set “S”, and x $\notin$ S to say that it is not.

- Examples:

$x \in \mathbb{Z}$	x is an integer
$x \in \mathbb{N}$	x is a non-negative integer (natural)
$x \in \mathbb{R}$	x is a real number
$x \in \mathbb{B}$	x is T or F (boolean)

non-standard name

we will often write
$x : \mathbb{Z}$ instead of $x \in \mathbb{Z}$

Compound Types In Math

- Compound types combine multiple data types
  - multiple ways build them

- **Record Types** $\{x : \mathbb{N},\ y : \mathbb{N}\}$
  - record with fields “x” and “y” each containing a number
  - e.g., $\{x: 3, y: 5\}$

- Note that $\{x: 3, y: 5\} = \{y: 5, x: 3\}$ in math
  - field names matter, not order
  - note that these are not "$==$" in Java

    in math, “=“ means same values

    in JavaScript, "==" is reference equality

Compound Types In Math

- **Record Types** $\{x : \mathbb{N},\ y : \mathbb{N}\}$
  - record with fields “x” and “y” each containing a number
  - e.g., $\{x: 3, y: 5\}$
- **Tuple Types** $\mathbb{N} \times \mathbb{N}$
  - pair of two natural numbers, e.g., $(5, 7)$
  - can do tuples of 3, 4, or more elements also
- **Mostly equivalent alternatives**
  - both let us put parts together into a larger object
  - record distinguishes parts by name
  - tuple distinguishes parts by order

Retrieving Part of a Tuple

- To refer to tuple parts, we must give them names

- Tuple Types $\mathbb{N} \times \mathbb{N}$

Let $(a, b) := t$.

":=" means a definition

Suppose we know that $t = (5, 7)$

Then, we have $a = 5$ and $b = 7$

Simple Functions in Math

- **Simplest function definitions are single expressions**

- **Will write them in math like this:**

$$\mathrm{double} : (\mathbb{N}) \to \mathbb{N}$$

$$\mathrm{double}(n) := 2n$$

  - **first line declares the type of double function**

    takes a natural number input to a natural number output
  - **second line shows the calculation**

    know that "n" is a natural number from the *first* line
  - **will often put the type in the text before the definition, e.g.,**

    The function $\mathrm{double} : (\mathbb{N}) \to \mathbb{N}$ is defined by…

$$\mathrm{double}(n) := 2n$$

Simple Functions in Math

- Another example:

$$\text{dist} : (\{x : \mathbb{R}, y : \mathbb{R}\}) \to \mathbb{R}$$

$$\text{dist}(p) := (p.x^2 + p.y^2)^{1/2}$$

  - first line tells us that "p" is a record and "$p.x$" is a real number

- Can define short-hand for types in math also

$$\textbf{type } \text{Point} := \{x : \mathbb{R}, y : \mathbb{R}\}$$

$$\text{dist} : (\text{Point}) \to \mathbb{R}$$
$$\text{dist}(p) := (p.x^2 + p.y^2)^{1/2}$$

Complex Functions in Math

- **Most interesting functions are not simple expressions**
  - **need to use different expressions in different cases**

- **Can use side-conditions to split into cases**

$$\mathrm{abs} : (\mathbb{R}) \to \mathbb{R}$$

$$\begin{array}{ll} \mathrm{abs}(x) := x & \text{if } x \geq 0 \\ \mathrm{abs}(x) := -x & \text{if } x < 0 \end{array}$$

  - **conditions must be <u>exclusive</u>**

    we do not want to require on *order* to determine which applies
  - **not always <u>exhaustive</u>: missing cases are undefined**
  - **there is a better way to do this in many cases…**

Pattern Matching

- Can also define functions by “pattern matching”

$$\text{double} : (\mathbb{N}) \to \mathbb{N}$$

$$\begin{aligned}\text{double}(0) &:= 0 \\ \text{double}(\text{n}+1) &:= \text{double}(\text{n}) + 2\end{aligned}$$

  - first case matches only 0
  - second case matches numbers 1 more than some $\text{n} : \mathbb{N}$ ...

    double(6) = double(5+1) so it matches with n = 5

    since n $\geq$ 0, we have n+1 $\geq$ 1, so it matches 1, 2, 3, …
  - pattern “$\text{n}+2$” would match 2, 3, 4, …

- Simplifies the math in multiple ways…

Pattern Matching on Natural Numbers

- **Pattern matching definition**

$$\begin{aligned} \text{double}(0) &:= 0 \\ \text{double}(n+1) &:= \text{double}(n) + 2 \end{aligned}$$

is simpler than using side conditions

$$\begin{aligned} \text{double}(n) &:= 0 && \text{if } n = 0 \\ \text{double}(n) &:= \text{double}(n-1) + 2 && \text{if } n > 0 \end{aligned}$$

  - **e.g., need to explain why** $\text{double}(n-1)$ **is legal**

    easy in this case, but it gets harder

- **We will prefer pattern matching whenever possible**

Pattern Matching on Booleans

- Booleans have only two legal values: T and F

- Can pattern match just by listing the values:
  - the function $\text{not} : (\mathbb{B}) \to \mathbb{B}$ is defined as follows:

$$\text{not}(\text{T}) := \text{F}$$
$$\text{not}(\text{F}) := \text{T}$$

  - negates a boolean value
  - no simpler way to define this function!

Pattern Matching on Records

- Can pattern match on individual fields of a record

```
type Steps := {n : ℕ, fwd : 𝔹}

change : (Steps) → ℤ

change({n: m, fwd: T}) := m
change({n: m, fwd: F}) := –m
```

  - clear that the rules are exclusive and exhaustive

- Can match on multiple parameters
  - e.g., change({n: m+5, fwd: T}) := 2m
  - just make sure the rules are exclusive (and exhaustive)

Code Without Mutation

- Saw all types of code without mutation:
  - straight-line code
  - conditionals
  - recursion
- This is all we need, mathematically
  - can write anything computable with just these
- Code that only uses these properties is "functional"

Inductive Data Types

Inductive Data Types

- **Previously saw records and tuples**
  - **very useful but limited**

    can only create types that are “small” in some sense
  - **missing one more way of defining types**

    arguably the most important

- **One critical element is missing: recursion**

  Java classes can have fields of same type, but records cannot

- **Inductive data types are defined recursively**
  - **combine union with recursion**

Inductive Data Types

- Describe a set by ways of creating its elements
  - each is a “constructor”

$$\textbf{type}\ \mathrm{T} := \ \mathrm{C}(\mathrm{x} : \mathbb{Z}) \ \mid \ \mathrm{D}(\mathrm{x} : \mathbb{Z},\ \mathrm{y} : \mathrm{T})$$

  - second constructor is recursive
  - can have any number of arguments (even none)

    will leave off the parentheses when there are none

- Examples of elements

$\mathrm{C}(1)$

$\mathrm{D}(2, \mathrm{C}(1))$

$\mathrm{D}(3, \mathrm{D}(2, \mathrm{C}(1)))$

in math, these are **not** function calls

Inductive Data Types

- Each element is a description of how it was made

$$\mathrm{C}(1)$$
$$\mathrm{D}(2, \mathrm{C}(1))$$
$$\mathrm{D}(3, \mathrm{D}(2, \mathrm{C}(1)))$$

- Equal when they were made *exactly* the same way
  - $\mathrm{C}(1) \neq \mathrm{C}(2)$
  - $\mathrm{D}(2, \mathrm{C}(1)) \neq \mathrm{D}(3, \mathrm{C}(1))$
  - $\mathrm{D}(2, \mathrm{C}(1)) \neq \mathrm{D}(2, \mathrm{C}(2))$
  - $\mathrm{D}(1, \mathrm{D}(2, \mathrm{C}(3))) = \mathrm{D}(1, \mathrm{D}(2, \mathrm{C}(3)))$

Natural Numbers

$$\textbf{type}\ \mathbb{N} := \text{zero} \mid \text{succ}(n : \mathbb{N})$$

Only possible to make **non-negative** integers

- Inductive definition of the natural numbers

zero	0
succ(zero)	1
succ(succ(zero))	2
succ(succ(succ(zero)))	3

The most basic set we have is defined inductively!

Lists

$$\textbf{type}\ \text{List} := \text{nil} \mid \text{cons}(x : \mathbb{Z},\ L : \text{List})$$

- **Inductive definition of lists of integers**

nil

$\text{cons}(3, \text{nil})$

$\text{cons}(2, \text{cons}(3, \text{nil}))$

$\text{cons}(1, \text{cons}(2, \text{cons}(3, \text{nil})))$

Our most important data type!

$$\begin{tikzcd} \boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{3} \end{tikzcd}$$

Shorthand Notation for Lists

type List := nil | cons(x : $\mathbb{Z}$, L : List)

- **We will use:**
  - "x :: L" **to mean** "cons(x, L)"
  - "[1, 2, 3]" **to mean** "1 :: 2 :: 3 :: nil"

- **Examples:**

nil	nil	[]
cons(3, nil)	3 :: nil	[3]
cons(2, cons(3, nil))	2 :: 3 :: nil	[2, 3]
cons(1, cons(2, cons(3, nil)))	1 :: 2 :: 3 :: nil	[1, 2, 3]

Functions Defined on Inductive Data Types

- We need recursion to define interesting functions

- Inductive types fit esp. well with *pattern matching*
  - every object is created using some constructor
  - match based on which constructor was used

Example 7: List Length

type List := nil | cons(hd: $\mathbb{Z}$, tl: List)

- **Mathematical definition of list length:**

len : (List) $\to \mathbb{N}$

len(nil) := 0
len(x :: L) := 1 + len(L)

  - **any list is either** nil **or** x :: L **for some** x **and** L
  - **cases are exclusive and exhaustive**

Example 7: List Length

- **Mathematical definition of list length:**

$$\text{len} : (\text{List}) \to \mathbb{N}$$

$$\begin{aligned}
\text{len}(\text{nil}) &:= 0 \\
\text{len}(x :: L) &:= 1 + \text{len}(L)
\end{aligned}$$

- **Check this on an example:**

$$\begin{aligned}
&\text{len}(1 :: 2 :: 3 :: \text{nil}) \\
&= 1 + \text{len}(2 :: 3 :: \text{nil}) && (2^{\text{nd}} \text{ line}) \\
&= 2 + \text{len}(3 :: \text{nil}) && (2^{\text{nd}} \text{ line}) \\
&= 3 + \text{len}(\text{nil}) && (2^{\text{nd}} \text{ line}) \\
&= 3
\end{aligned}$$

Example 8: Swap Adjacent

- **Function that swaps adjacent elements in a list:**

$$\mathrm{swap} : (\mathrm{List}) \to \mathrm{List}$$

$$\begin{aligned}
\mathrm{swap}(\mathrm{nil}) &:= \mathrm{nil} \\
\mathrm{swap}(x :: \mathrm{nil}) &:= x :: \mathrm{nil} \\
\mathrm{swap}(x :: y :: L) &:= y :: x :: \mathrm{swap}(L)
\end{aligned}$$

  - **any list is either** nil **or** $x :: \mathrm{nil}$ **or** $x :: y :: L$ **for some** x, y **and** L
  - **cases are exclusive and exhaustive**

Structural Recursion

- Examples only recurse on *parts* of the input

$$\mathrm{len}(x :: L) \;:=\; 1 + \mathrm{len}(L)$$

  – call on $x :: L$ recurses on L

$$\mathrm{swap}(x :: y :: L) \;:=\; y :: x :: \mathrm{swap}(L)$$

  – call on $x :: y :: L$ recurses on L
  – such cases are called "structural recursion"

- Guarantees no infinite recursion!
  – one argument gets *strictly smaller* on each call
  – restrict ourselves to structural recursion in math

Exercise 1: List Sum

- Write a mathematical definition of sum:

$$\text{sum} : (\text{List}) \to \mathbb{N}$$

Answer 1: List Sum

- **Mathematical definition of sum:**

$$\text{sum} : (\text{List}) \to \mathbb{N}$$

$$\begin{aligned} \text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L) \end{aligned}$$

- **Check that this works on the examples…**

$$\begin{aligned} &\text{sum}(1 :: 2 :: 3 :: \text{nil}) \\ &\quad = 1 + \text{sum}(2 :: 3 :: \text{nil}) && (\text{2nd line}) \\ &\quad = 1 + 2 + \text{sum}(3 :: \text{nil}) && (\text{2nd line}) \\ &\quad = 1 + 2 + 3 + \text{sum}(\text{nil}) && (\text{2nd line}) \\ &\quad = 1 + 2 + 3 && (\text{1st line}) \end{aligned}$$

Notes on Lists Posted on the Website

- Shorter version of everything we've discussed

- In addition:
  1. Defines a few more useful list functions
  2. Mentions important properties of concat:
     – operator notation "$+\!\!+$"
     – associativity and identity
  3. Mentions important applications of lists
     – maps *are* lists of (key, value) pairs
     – sets can be defined defined as lists

- Lists are our most important data type!

Formal Specifications

Example 10: List Concatenation

- **Mathematical definition of list concatenation**

$$\text{concat} : (\text{List}, \text{List}) \to \text{List}$$

$$\begin{aligned} &\text{concat}(\text{nil}, R) && := R \\ &\text{concat}(x :: L, R) && := x :: \text{concat}(L, R) \end{aligned}$$

- **Check that this matches examples…**

$$\begin{aligned} &\text{concat}(1 :: 2 :: \text{nil}, 3 :: 4 :: \text{nil}) && \\ &\quad = 1 :: \text{concat}(2 :: \text{nil}, 3 :: 4 :: \text{nil}) && (2^{\text{nd}}\ \text{line}) \\ &\quad = 1 :: 2 :: \text{concat}(\text{nil}, 3 :: 4 :: \text{nil}) && (2^{\text{nd}}\ \text{line}) \\ &\quad = 1 :: 2 :: 3 :: 4 :: \text{nil} && (1^{\text{st}}\ \text{line}) \end{aligned}$$

Recall: Example 2

```
/**
 * Returns the concatenation of two lists.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null.
 * @return a list containing the elements of A
 *     followed by all the elements of B.
 */
List<Integer> concat(
    List<Integer> A, List<Integer> B);
```

Can formalize "followed by" with our math definition...

Formal Version of Example 2

```
/**
 * Returns the concatenation of two lists.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null.
 * @return concat(A, B), where
 *     concat(nil, R) := R
 *     concat(x :: L, R) := x :: concat(L, R)
 */
List<Integer> concat(
    List<Integer> A, List<Integer> B);
```

No ambiguity about what this means.

Recall: Example 3

```
/**
 * Adds the elements in two lists.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null.
 * @requires A.size() = B.size()
 * @return A list C of the same length, where
 *    C's value is the sum of those at the
 *    corresponding indexes in A and B
 */
List<Integer> addLists(
    List<Integer> A, List<Integer> B);
```

Can formalize "corresponding indexes" with our math notation...

Example 11: List Addition

- **Mathematical definition of list addition**

  add-lists : (List, List) → List

add-lists(nil, nil)	:= nil
add-lists(x :: L, y :: R)	:= (x+y) :: add-lists(L, R)

  - **not defined on lists of different lengths**
  - **check this on an example:**

  add-lists(1 :: 2 :: nil, 3 :: 4 :: nil)

= 4 :: add-lists(2 :: nil, 4 :: nil)	(2nd line)
= 4 :: 6 :: add-lists(nil, nil)	(2nd line)
= 4 :: 6 :: nil	(1st line)

Formal Version of Example 3

```
/**
 * Adds the elements in two lists.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null.
 * @requires A.size() = B.size()
 * @return add-lists(A, B), where
 *    add-lists(nil, nil) := nil
 *    add-lists(x :: L, y :: R) :=
 *        (x+y) :: add-lists(L, R)
 */
List<Integer> addLists(
    List<Integer> A, List<Integer> B);
```

No ambiguity about what this means.

Next Up…

- How do we know that these are correct?
  - concat, add-lists, etc. could have mistakes
- Generally, all we can do is **test** them on examples
  - tried one example of each above
  - this is not enough…
- Well-chosen set of tests is likely to find most bugs
- Testing also matters for implementation
  - remaining slides will matter to both sides

Varieties of Testing

Unit vs Integration Tests

- A <u>unit test</u> checks one function
  - ideally, without testing anything else (not always possible)
- An <u>integration test</u> makes sure units work together
  - many (most?) bugs in practice are here
- An <u>end-to-end test</u> exercises almost all the code

Unit vs Integration Tests

- A unit test checks one function
- An integration test makes sure units work together
- An end-to-end test exercises almost all the code

- You will be expected to write unit tests in industry

- There will also be integration and end-to-end tests
  - someone will write them, but maybe not you
  - (requires understanding the whole system)

- We will focus on unit testing

Unit Testing

- **Even individual functions might be too big…**

```
int f(List<Integer> vals) {
  Map<Integer, List<Integer>> M = …;
  for (int v : vals) {
    …
  }

  int s = 0;
  for (int v : vals)  {
    for (int w : M.get(v))
      s += …
  }
}
```

Unit Testing

- Even individual functions might be too big…

```
int f(List<Integer> vals) {
  for (int v : vals)
    …
  for (int v : vals)
    …
}
```

- Multiple loops often should be multiple functions

- Purposefully design the code to be testable
  – important part of programming in practice

“Manual” vs Programmatic Tests

- Usually possible to run the code by hand (“manually”)
  - open the terminal and execute it
  - start the application and look at it (UI)

- No downside… unless the code changes
  - then, you need to do the tests again

- Programmatic tests are code that tests other code
  - easy to run them again whenever the code changes
  - these are generally preferred

“Manual” vs Programmatic Tests

- Usually possible to run the code by hand (“manually”)
  - open the terminal and execute it
  - start the application and look at it (UI)

- No downside… unless the code changes
  - then, you need to do the tests again

- For UI, manual testing is still common
  - written tests are hard to write and imperfect

    need to see it on the screen to be sure that it looks right
  - non-UI functions and all server code tested programmatically

Unit Testing

Writing a Programmatic Test

1. Choose an input
   - description of the inputs is the “test case”

2. Think through what the answer should be
   - look at the specification for the correct answer
   - if you run the code to get the answer, you are not testing

AI often does this
(and worse)

Writing a Programmatic Test

1. Choose an input / configuration
   - description of the inputs / configuration is the “test case”

2. Think through what the expected answer is
   - if you run the code to get the answer, you are not testing

3. Write code that
   a) calls the function that input
   b) compares the actual answer to the expected one
   c) throws an error if they do not match
   - useful libraries for doing this…

Writing a Programmatic Test

```
public class NumberUtil {

  // Determines whether n is a prime number.
  public static boolean isPrime(int n) {
    …
  }

  // Returns the greatest common divisor of a and b.
  public static int gcd(int a, int b) {
    …
  }
}
```

Writing a Programmatic Test with JUnit

```
import org.junit.*;
import static org.junit.Assert.*;

public class NumberUtilTest {
  @Test
  public void testIsPrime() {
    assertEquals(true, isPrime(2));
    assertEquals(true, isPrime(3));
    assertEquals(false, isPrime(4));
  }

  @Test
  public void testGcd() {
    assertEquals(1, gcd(3, 2));
    assertEquals(3, gcd(9, 3));
    assertEquals(4, gcd(12, 8));
  }
}
```

`assertEquals` will throw an exception and fail the test if the two are not equal

use `gradlew test` to run all the tests in the Java project

Ground Rules for Testing

1. Only need to test inputs allowed by the spec
   - there is no correct answer for other inputs

```
// Determines whether a positive integer is prime.
public static boolean isPrime(int n) {
  if (n <= 0)
    throw new IllegalArgumentException("negative n");
  …
}
```

good use of defensive programming
to check that the input is valid

Ground Rules for Testing

1. Only need to test inputs allowed by the spec
   – there is no correct answer for other inputs

2. Choose tests for each function individually
   – pick tests to do a good job of testing that one function

```
// Determines whether a positive integer is prime.
public static boolean isPrime(int n) {
  if (n <= 0)
    throw new IllegalArgumentException("negative n");

  int m = intSqrt(n);  // integer square root of n
  …
}
```

`intSqrt` has its own tests!

Ground Rules for Testing

1. Only need to test inputs allowed by the spec
   – there is no correct answer for other inputs

2. Test each function individually
   – assume anything it calls is correct (its own tests will check)

3. Test code should be **simple**
   – any loops in tests need their own tests!

How Many Tests are Necessary?

- Consider the following function:

```
// Allows inputs 0 <= a, b, c <= 10,000 …
public static int f(int a, int b, int c) {
  …
};
```

- How many tests needed guarantee correctness?
  - 1 trillion!
  - "just write a loop and ..."

    the code in that loop could also be wrong
  - cannot **think** through even 1000 tests

    most code we write cannot be *exhaustively* tested

Ground Rules for Testing

1. Only need to test inputs allowed by the spec
   - there is no correct answer for other inputs

2. Test each function individually
   - assume anything it calls is correct (its own tests will check)

3. Test code should be **simple**
   - any loops in tests need their own tests!

4. If there are fewer than 10 allowed inputs,
   then test them all!
   - take advantage of the easy case

Choosing Test Cases

```
// Returns true iff n is a prime number
public static boolean isPrime(int n) { … }
```

- How about if we test 2, 3, 4, 7, 12, 97, 99?
  - seems okay?

Choosing Test Cases

```
// Returns true iff n is a prime number
public static boolean isPrime(int n) {
  if (n < 100) {
    return PRIME_CACHE[n];  // precomputed answers
  } else {
    for (int k = 2; k*k <= n; k++) {
      if (n % k === 0)
        return false;
    }
    return true;
  }
};
```

Cases 2 .. 100 are table lookups!

We didn't test the loop at all!

Impossible to know this without looking at the actual code.

Clear-Box Testing

- We need to look at the code to know what to test
  - this is called "clear-box testing"
  - it will be our **primary** heuristic
- In this class, I want a clear rule for how many tests
  - want homework and tests to have clear right/wrong answers
- Outside of class, these rules are also good
  - most programmers will be familiar with these concepts

Statement Coverage

- Simplest metric is "statement coverage"
  - what percentage of the statements in the code are executed by *at least one* test

    in our math notation, think of each **case** in the definition as a statement
  - this should be nearly 100%

```
// Determines whether a positive integer is prime.
public static boolean isPrime(int n) {
  if (n <= 0)
    throw new IllegalArgumentException("negative n");

  …
```

- The "`throw`" is not executed by any *allowed* input
  - we only test the allowed inputs

Statement Coverage

- Simplest metric is "statement coverage"
  - what percentage of the statements in the code are executed by *at least one* test
- Must test 100% of code reachable on allowed inputs
  - cannot send code to users that you didn't even try!
  - we will refer to this as having "full statement coverage"
- Are we done?

Statement Coverage

- Consider the following function:

```
// Returns the smaller of a and b.
public static int min(int a, int b) {
  int m = a;
  if (a <= b)
    m = a;
  return m;
};
```

  - testing on a=1 b=2 gives full statement coverage
  - what is the bug?

    gives the wrong answer whenever a > b
  - we never tested the case where the "`if`" doesn't execute

Conditionals

Conditionals are "`if`" statements

```
if (n > 0) {
  x = 2*(n - 1);
} else {
  x = 0;
}
```

Every conditional has two branches (“then” and “else”)

Conditionals

Conditionals are "`if`" statements

```
if (n > 0) {            =     if (n > 0) {
  x = 2*(n - 1);        =       x = 2*(n - 1);
}                       =     } else {
                        =     }
```

Every conditional has two branches (“then” and “else”)

- missing "`else`" still has an empty else branch

Branch Coverage

- Next metric is "branch coverage"
  - for what percentage of the conditionals, are both branches executed by some test

- Must test all branches reachable on allowed inputs
  - can ignore branches that are unreachable

    i.e., the ones that `throw new Exception` on bad inputs

Branch Coverage

- Consider the following function:

```
/** Returns the smaller of a and b. */
public static int min(int a, int b) {
  int m = a;
  if (a <= b)
    m = a;
  return a;
};
```

  - problem only arises when "`if`" falls through to code after
  - if every branch ends with `return` / `throw`,
    then statement coverage = branch coverage

    always true for code without mutation of local variables

Branch Coverage

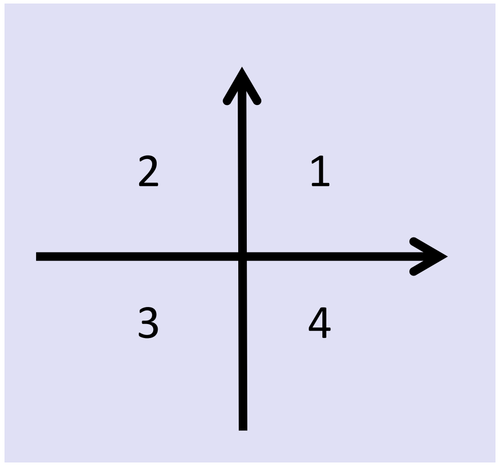
- Next metric is "branch coverage"
  - for what percentage of the conditionals, are both branches executed by some test
- Must test all branches reachable on allowed inputs
  - can ignore branches that are unreachable

    i.e., the ones that `throw new Error` on bad inputs
- Are we done?

Branch Coverage

- Consider the following function:

```
// Returns quadrant containing (x, y).
public static int quad(float x, float y) {
  int answer;
  if (x >= 0) {
    answer = 4;
  } else {
    answer = 3;
  }
  if (y >= 0)
    answer = 1;
  return answer;
};
```



  - testing on (1, -1) and (-1, 1) gives full branch coverage
  - this code is wrong... it never returns 2!

How Many Tests Are Required?

- More advanced metrics could fix this
  - "path coverage" would require 4 tests
  - #paths can grow exponentially in #branches

- For straight-line code and conditionals, we will only require branch coverage

- What about loops / recursion?

How Many Tests Are Required?

- Consider the following function:

```
public static int binarySearch(String s, String[] A) {
  int lo = 0;
  int hi = A.length;
  while (lo < hi) {  // s could be in A[lo .. hi-1]
    int m = (lo + hi) / 2;
    if (s < A[m]) {
      hi = m - 1;
    } else if (s > A[m]) {
      lo = m + 1;
    } else {
      return m;
    }
  }
  return hi;
};
```

Testing on s="a"/"b"/"c" A=["b"] gives full statement coverage

But the code is wrong.

In general, values written inside the loop are not read until the next time around, so you need 2+ iterations to test them.

How Many Tests Are Required?

- Our last metric is "loop coverage" (not 100% standardized)
  - what percent of loops are executed 0, 1, and many (2+) times by some test case

- Same idea applies to recursion
  - some arguments passed to recursive calls may not be read until the second recursive call
  - full loop coverage means every recursive call is executed 0, 1, and many times by some test

    will need this for **specifications** written in our math notation

Summary of testing requirements

- At least two tests for any function (non-UI)

- Must have full coverage of *reachable*
  - <u>statements</u>: must be executed
  - <u>branches</u>: must execute both branches
  - <u>loops</u> / <u>recursion</u>: must run 0, 1, & many times

Example 12

$$f: (\mathbb{N}) \to \mathbb{R}$$

$$f(0) \quad := 0$$

$$f(n+1) \quad := \sin\left(\left(n + \tfrac{1}{2}\right)\pi\right)$$

How many tests? Which ones?

- 0 (top branch) and 1 (bottom branch)

  statement coverage = branch coverage in functional code

Example 13

```
// n must be a non-negative integer
public static int f(int n) {
  if (n < 3) {
    return 0;
  } else if (n < 10) {
    return (n - 3) / 10;
  } else {
    return 1;
  }
}
```

How many tests? Which ones?

– 2 (top), 6 (middle), and 10 (bottom)

Example 14

```
// m and n must be a non-negative
public static int f(int m, int n) {
  if (m > n)
    m = n;
  return Math.abs(m);
}
```

How many tests? Which ones?

- $m=2$, $n=1$ gives full statement coverage
- adding $m=1$, $n=2$ gives branch coverage

Example 15

```
// n must be a non-negative integer
public static int f(int n) {
  if (n <= 1) {
    return 0;
  } else {
    return 1 + f(n / 2);
  }
}
```

How many tests? Which ones?

- 1 (0 recursive calls)
- 2 (1 recursive call)
- 5 (2 recursive calls)

Example 16

$f: (\mathbb{Z}) \to \mathbb{N}$

$f(1)$	$:= 0$	
$f(n+1)$	$:= 1 + 2\, f(n)$	**if** $1 \leq n \leq 9$

- **only defined on 1, 2, ..., 10**

  we have $1 \leq n \leq 9$ exactly when $2 \leq n+1 \leq 10$

How many tests? Which ones?

- only 10 inputs, so... all of them

Other Heuristics

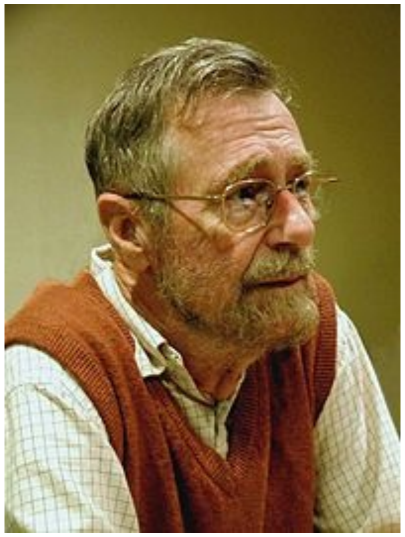
Not mandatory for 331 but useful in practice:

- Make sure every argument value is changed
- Look at special values
  - null, undefined, NaN, empty array, etc. often have bugs
- Look at the specification for branches
  - maybe the code doesn't split inputs where it should!
  - e.g., spec splits into "if $x \geq 0$" but code is "`if (x > 0)`"

Summary of testing requirements

- At least two tests for any function (non-UI)
- Must have full coverage of *reachable*
  - statements: must be executed
  - branches: must execute both branches
  - loops / recursion: must run 0, 1, & many times
- Concise summary of these linked on the Topics page
  - includes the other heuristics as well
- Are we done?
  - no!

What Can We Learn From Testing?



“Program testing can be used to show the presence of bugs, but never to show their absence!”

Edsgar Dijkstra

Notes on Structured Programming, 1970

“Beware of bugs in the above code;
I have only proved it correct, not tried it.”

Donald Knuth, 1977