# Reasoning about Specifications

# So Far

- We discussed reasoning about code
  - Forward reasoning and backward reasoning
- Hoare Logic
  - Hoare Triples
  - Rule for assignment
  - Rule for sequence
  - Rule for if-then-else
  - Rule for method call
  - Reasoning about loops

# Specifications


- A specification consists of a precondition and a postcondition
  - Precondition: conditions that hold before method executes
  - Postcondition: conditions that hold after method finished execution (if precondition held!)

# Specifications

- A specification is a contract between a method and its caller
  - Obligations of the method (implementation of the specification): agrees to provide postcondition **if precondition holds**
  - Obligations of the caller (user of specification): agrees to meet the precondition and not expect more than the promised postcondition
    - If the preconditions is violated, postcondition is not guaranteed
    - Bad things like unexpected exceptions, crashes etc. can happen

- A specification is an abstraction
  - An abstract description of the behavior of the method

# Example Specification

Precondition: `a != null && len ≥ 0 && a.length == len`

Postcondition: `result = a[0]+…+a[a.length-1]`

```
int sum(int[] a, int len) {
   int sum = 0;
   int i = 0;
   while (i < len) {
      sum = sum + a[i];
      i = i+1;
   }
   return sum;
}
```

For our purposes, we will be writing specifications that are not especially formal. Mathematical rigor is welcome, but not always necessary.

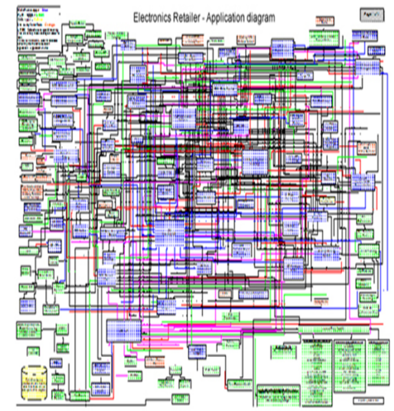
# Benefits of Specifications

- Precisely documents method behavior
  - Imagine if you had to read the <u>code</u> of the Java libraries to figure what they do!
- Promotes modularity
  - Modularity is key to the development of correct and maintainable software
  - Specifications help organize code into small modules, with clear-cut boundaries between those modules

# Benefits of Specifications

- Promote abstraction…
  - Client relies on description in specification, no need to know the implementation
  - Method must support specification, but its implementation is free otherwise!
  - Method and client code can be built simultaneously
- Enables reasoning about correctness
  - “Does code do the right thing?” means “Does code conform to its specification?”
  - Confirmed by testing and verification
    - Reasoning about the code

# So, Why Not Just Read Code?

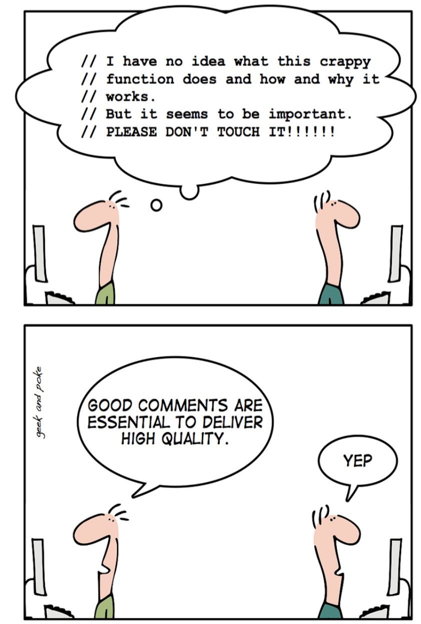



```
boolean sub(List<T> src, List<T> part)
   int part_index = 0;
   for (Object o : src) {
      if (o.equals(part.get(part_index))) {
          part_index++;
          if (part_index == part.size()) {
              return true;
          }
      } else {
          part_index = 0;
      }
   }
   return false;
}
```

# So, Why Not Just Read Code?

- Code is complicated
  - Gives a lot more detail than client needs
  - Understanding code is a cognitive burden
  - Big code is very difficult to understand
- Code is ambiguous
  - Often unclear what is essential and what is incidental
    - Incidental code
      - Result of an optimization
      - Dead code
- Client needs to know what the code does, not how it does it!
  - Even in mathematical/scientific software where a specific method is required, user doesn’t need to know all of the implementation details

# What About Comments?




// method checks if **`part`** appears as
// subsequence in **`src`**

```
boolean sub(List<T> src, List<T> part) {
…
}
```

- Comments are important, but insufficient.
- They are informal and often ambiguous or worse misleading.
  - Specifications should be precise and concise, carrying relevant information.
- Comments are sometimes not updated when code is updated.

# Specifications Are Concise and Precise

- Unfortunately, lots of code lacks specification
  - Programmers guess what code does by reading the code or running it
  - This results in bugs and/or complex code with unclear behavior

- It would be nice to generate code from specs
  - Inference of code from specifications is an active area of research

# So, What’s in **sub**’s Specification?

What happens if part is null?

```
boolean sub(List<T> src, List<T> part) {
    int part_index = 0;
    for (Object o : src) {
        if (o.equals(part.get(part_index))) {
            part_index++;
            if (part_index == part.size()) {
                return true;
            }
        } else {
            part_index = 0;
        }
    }
    return false;
}
```

# So, What's in **`sub`**'s Specification?

Choice 1:

// **`sub`** returns true if **`part`** is a subsequence of **`src`**; it returns false otherwise.

Choice 2:

// **`src`** must be non-null

// If **`src`** is the empty list, then **`sub`** returns false

// Requirements on **`part`** too… part must be non-null

// If there is a partial match in the beginning, **`sub`** returns false, even
// if there is a full match later. E.g. **`sub`**([1, 2, 1, 2, 1, 3], [1, 2, 1, 3]) is
// false.

# What’s in **`sub`**’s Specification?

- A complex specification is a red flag
- Rule: it is better to simplify design and code rather than try to describe complex behavior!
- If you end up writing a complicated specification, redesign and rewrite your code --- either something is wrong or code is extremely complex and hard to reason about
  - Software designers are in a constant battle with complexity
  - Good specs help reduce programmer's cognitive burden

# Goal of Principles of Software

- **One of our goals is to establish the discipline of writing concise and precise specifications**

- We need some specification conventions
    - JavaDoc Style
    - PSoft style
    - JML
    - Dafny
    - There are many more ….

# Javadoc

- Javadoc convention
  - Method's type signature
  - Text description of what method does
  - Parameters: text description of what gets passed
  - Return: text description of what gets returned
  - Throws: list of exceptions that may get thrown

# Example: Javadoc for String.substring

**`public String substring(int beginIndex)`**

Returns a string that is a substring of **`this`** string. The substring begins with the character at the specified index and extends to the end of this string.

Parameters:

**`beginIndex`** --- the beginning index, inclusive.

Returns:

the specified substring.

Throws:

**`IndexOutOfBoundsException`** --- if **`beginIndex`** is negative or larger than the length of this String object.

# Specifications

- Principles of Software (PSoft) specification conventions
  - Included in the code as comments
- The precondition
  - requires: clause spells out constraints on client code
- The postcondition
  - modifies: lists objects that may be modified by the method. Any object not listed under this clause is guaranteed untouched
  - effects: describes final state of modified objects
  - throws: lists possible exceptions
  - returns: describes return value
- A few brief comments describing the function along with the specification helps.

# Specifications

- Method signatures are part of contract with client.
  - Argument and return types
  - We don’t usually consider them preconditions, because code will fail at compile time if client code doesn’t match signature.

# Example 0

**`String substring(int beginIndex)`**

requires: none

modifies: none

effects: none

returns: a string with same value as the substring beginning at **`beginIndex`** and extending until the end of this string

throws: **`IndexOutOfBoundsException`** --- if **`beginIndex`** is negative or greater than the length of this String object.

# Example 1

**`static <T> int change(List<T> lst, T old, T newelt)`**

requires: **`lst`**, **`old`**, **`newelt`** are non-null. **`old`** occurs in **`lst`**.

modifies: **`lst`**

effects: replace first occurrence of **`old`** in **`lst`** with **`newelt`**. makes no other changes.

returns: position of element in **`lst`** that was **`old`** and is now **`newelt`**

```
static <T> int change(List<T> lst, T old, T newelt) {
   int i = 0;
   for (T curr : lst) {
       if (curr == old) {
              lst.set(i, newelt);
              return i;
       }
       i = i + 1;
   }
   return -1;
}
```

Is there a problem with this spec?

# Example 1 – another try

**`static <T> int change(List<T> lst, T old, T newelt)`**

requires: **`lst`**, **`old`**, **`newelt`** are non-null.

modifies: **`lst`**

effects: replace first occurrence of **`old`** in **`lst`** with **`newelt`**. makes no other changes.

returns: position of element in **`lst`** that was **`old`** and is now **`newelt`** or -1 if old was not found.

```
static <T> int change(List<T> lst, T old, T newelt) {
   int i = 0;
   for (T curr : lst) {
       if (curr == old) {
              lst.set(i, newelt);
              return i;
       }
       i = i + 1;
   }
   return -1;
}
```

Maybe a better version

# Example 2

static List<Integer> listAdd(List<Integer> lst1,  List<Integer> lst2)

requires: **lst1**, **lst2** are non-null.
**lst1** and **lst2** are same size.

modifies: none

effects: none

returns: a new list of the same size as **lst1**, whose i-th element is the sum of the i-th elements of **lst1** and **lst2**

```
static List<Integer> listAdd(List<Integer> lst1,
                             List<Integer> lst2) {
 List<Integer> res = new ArrayList<Integer>();
 for (int i = 0; i < lst1.size(); i++)
   res.add(lst1.get(i) + lst2.get(i));
 return res;
}
```

# Aside: Autoboxing and Unboxing

- Boxed primitives. E.g., **`int`** vs. **`Integer`**
- Autoboxing: automatic conversion from primitive to boxed type
  - Java generics require reference type arguments

```
ArrayList<Integer> al = new …
for (int i=0; i<10; i++) al.add(i);
```

- Unboxing: automatic conversion from boxed type to primitive

```
res.add(lst1.get(i) + lst2.get(i))
```

# Details

- Sometimes hard to capture all potentially relevant details in the spec
  - in Example 1, should we specify that **`old`** is found using reference equality, and thus **`change`** may work unexpectedly when T is not a simple type?
  - Rule: add more but stay concise and precise. If spec is too complex, redesign!

# Example 3

```
static void listAdd2(List<Integer> lst1,
                     List<Integer> lst2)
```

requires: ??

modifies: ??

effects: ??

returns: ??

```
static void listAdd(List<Integer> lst1,
                    List<Integer> lst2) {
 for (int i = 0; i < lst1.size(); i++) {
   lst1.set(i,lst1.get(i) + lst2.get(i));
 }
}
```

# Example 3

```
static void listAdd2(List<Integer> lst1,
                     List<Integer> lst2)
```

requires: lst1, lst2 non-null; lst1.size() = lst2.size()

modifies: lst1

effects: i-th element of lst1 is replaced by the sum of i-th elements of lst1 and lst2

returns: none

```
static void listAdd(List<Integer> lst1,
                    List<Integer> lst2) {
 for (int i = 0; i < lst1.size(); i++) {
   lst1.set(i,lst1.get(i) + lst2.get(i));
 }
}
```

# Example 4

**`static void uniquefy(List<Integer> lst)`**

requires: ?

modifies: ?

effects: ?

returns: ?

```
static void uniquefy(List<Integer> lst) {
  for (int i = 0; i < lst.size()-1; i++)
   if (lst.get(i).intValue() == lst.get(i+1).intValue())
         lst.remove(i);
}
```

# Example 4

**`static void uniquefy(List<Integer> lst)`**

requires: lst non-null

modifies: lst

effects: removes first of a pair of consecutive duplicates in lst. E.g., <1,1,2,2,2,3> becomes <1,2,2,3>

returns: none

```
static void uniquefy(List<Integer> lst) {
  for (int i = 0; i < lst.size()-1; i++)
   if (lst.get(i).intValue() == lst.get(i+1).intValue())
         lst.remove(i);
}
```

There is a problem here.

# Example 4 – maybe a better way?

```
static void uniquefy(List<Integer> lst)
```

requires: lst non-null

modifies: lst

effects: removes duplicate elements from lst, maintains order. E.g., <1,1,2,2,2,3,1> becomes <1,2,3>

returns: none

```
static void uniquefy(List<Integer> lst) {
      // make a copy without duplicates
      Set<Integer> dup = new LinkedHashSet<Integer>(lst);
      lst.clear();
      lst.addAll(dup); // add the items back
}
```

# Example 4 – a slightly different way

**`static void uniquefy(List<Integer> lst)`**

requires: lst non-null

modifies: lst

effects: removes duplicate elements from lst, maintains order. E.g., <1,1,2,2,2,3,1> becomes <1,2,3>

returns: none

```
static void uniquefy(List<Integer> lst) {
        List<Integer> lst2 = lst.stream().distinct().collect(Collectors.toList());
        lst.clear();
        lst.addAll(lst2);
}
```

# Example 5

**`private static void swap(int[] a, int i, int j)`**

requires: ??

modifies: ??

effects: ??

returns: ??

```
static void swap(int[] a, int i, int j) {
      int tmp = a[j];
      a[j] = a[i];
      a[i] = tmp;
}
```

# Example 5

```
private static void swap(int[] a, int i, int j)
```

requires: a non-null, 0 <= i, j <a.length()

modifies: a

effects: swaps i-th and j-th element of a

returns: none

```
static void swap(int[] a, int i, int j) {
      int tmp = a[j];
      a[j] = a[i];
      a[i] = tmp;
}
```

# Example 5 – another way

```
private static void swap(int[] a, int i, int j)
```

requires: none

modifies: a

effects: swaps i-th and j-th element of a

returns: none

throws: NullPointerException, ArrayIndexOutOfBoundsException

```
static void swap(int[] a, int i, int j) {
      int tmp = a[j];
      a[j] = a[i];
      a[i] = tmp;
}
```

# Example 6

**`private static void selection(int[] a)`**

requires: ?

modifies: ?

effects: ?

returns: ?

```
static void selection(int[] a) {
   for (int i = 0; i < a.length; i++) {
      int min = i;
      for (int j = i+1; j < a.length; j++)
         if (a[j] < a[min]) min = j;
      swap(a, i, min);
   }
}
```

# Example 6

**`private static void selection(int[] a)`**

requires: a non-null

modifies: a

effects: a is sorted; a[i-1] <= a[i] for 0 < i < a.length;

returns: none

```
static void selection(int[] a) {
  for (int i = 0; i < a.length; i++) {
     int min = i;
     for (int j = i+1; j < a.length; j++)
        if (a[j] < a[min]) min = j;
     swap(a, i, min);
  }
}
```

# Javadoc for java.util.Arrays.binarySearch

**`public static int binarySearch(int[] a,int key)`**

Searches the specified array of ints for the specified value using the binary search algorithm. The array must be sorted (as by the sort(int[]) method) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.

Parameters:

**`a`** - the array to be searched.

**`key`** - the value to be searched for.

# Javadoc for java.util.Arrays.binarySearch

Returns:

index of the search key, if it is contained in the array; otherwise, **`(-(`*`insertion point`*`) - 1)`**. The *insertion point* is defined as the point at which the key would be inserted into the array; the index of the first element greater than the key, or **`a.length`** if all elements in the array are less than the specified key. Note that this guarantees that the return value will be **`>=`** **`0`** if and only if the key is found.

So, what is wrong with this spec?

# Alternative binarySearch Specification

**`public static int binarySearch(int[] a,int key)`**

Precondition:

requires: **`a`** is sorted in ascending order

Postcondition:

modifies: none

effects: none

returns:

**i** such that **a[i] = key** if such an **i** exists

a negative value otherwise

throws: NullPointerException

# Better binarySearch Specification

```
public static int binarySearch(int[] a,int key)
```

Precondition:

requires: **`a`** is sorted in ascending order

Postcondition:

modifies: none

effects: none

returns:

**i** such that **a[i] = key** if such an i exists

**(- i - 1)** s.t. inserting **key** at index **i** would keep **a** sorted, otherwise

throws: NullPointerException

# Even better binarySearch Specification

```
public static int binarySearch(int[] a,int key)
```

Precondition:

requires: **a** is sorted in ascending order

Postcondition:

modifies: none

effects: none

returns:

**i** such that **a[i] = key**, if such an **i** exists

**(- i - 1)** s.t. (i = 0 v a[i-1] < key) ^ (i = a.length v key < a[i]), otherwise

throws: NullPointerException

# Shall We Check requires Clause?

- If client (i.e., caller) fails to provide the preconditions, method can do anything --- throw an exception or pass bad value back
- It is polite to check
- Checking preconditions
    - Makes an implementation more robust
    - Provides feedback to the client
    - Avoids silent errors

# Rule

- If private method, don't have to check

- If public method, do check unless such a check is expensive
  - E.g., requires: **lst** is non-null. Check
  - E.g., requires: **lst** is sorted in ascending order. Don't check

- Example 1: requires: **`lst`**, **`old`**, **`new`** are non-null. **`old`** occurs in **`lst`**. Check?
- Example 2: requires: **`lst1`**, **`lst2`** are non-null. **`lst1`** and **`lst2`** are same size. Check?

- binarySearch: requires: **`a`** is sorted. Check?

# The JML Convention

- Javadocs and PSoft specifications are not “machine-checkable”
  - PSoft – Principles of Software

- JML (Java Modeling Language) is a formal specification language
  - Builds on the ideas of Hoare logic
  - End goal is automatic verification
    - Does implementation obey contract? Given a spec S and implementation I, does I conform to S?
    - Does client obey contract? Does it meet preconditions? Does it expect what method provides?

# The JML Spec for binarySearch

Precondition:

```
requires: a != null
  && (\forall int i;
        0 < i && i < a.length;
        a[i-1] <= a[i];)
```

Postcondition:

```
ensures: // complex statement…
```

- Difficult problem and an active research area.
- Sir Tony Hoare’s Grand Challenge
  - https://www.cs.ox.ac.uk/files/6187/Grand.pdf
  - A Verifying Compiler
    - https://dl.acm.org/citation.cfm?id=1538814

# Dafny Spec for binarySearch

```
method BinarySearch(a: array<int>, key: int)  returns (index: int)
  requires a != null && sorted(a);
  ensures 0 <= index ==> index < a.Length && a[index] == key;
  ensures index < 0 ==>  forall k :: 0 <= k < a.Length ==> a[k] != key;
```

# Better Dafny Spec for binarySearch

```
method BinarySearch(a: array<int>, key: int)  returns (index: int)
  requires a != null && sorted(a);
  ensures 0 <= index ==> index < a.Length && a[index] == key;
  ensures index < 0 ==> ((-index-1 <= a.Length) &&
                  (forall k :: 0 <= k < -index-1 ==> a[k] < key) &&
                  (forall k :: -index-1 <= k < a.Length ==> key < a[k]))
```

# Better Dafny Spec for binarySearch

```
method BinarySearch(a: array<int>, key: int)  returns (index: int)
  requires a != null && sorted(a);
  ensures 0 <= index ==> index < a.Length && a[index] == key;
  ensures index < 0 ==> ((-index-1 <= a.Length) &&
                  (index == -1 || a[-index-2] < key) &&
                  (index == -a.Length-1 || key < a[-index-1]))
```

# Specifications and Dynamic Languages

In statically-typed languages (C/C++, Java) type signature is a form of specification:

```
List<Integer> listAdd(List<Integer> lst1,
                      List<Integer> lst2)
```

In dynamically-typed languages (Python, JavaScript), there is no type signature!

```
def listAdd(lst1, lst2):
   i = 0
   res = []
   for item in lst1:
      res.append(item+lst2[i])
      i=i+1
   return res
```

# Example 5

**`def listAdd(lst1, lst2):`**

requires: ?
modifies: ?
effects: ?
returns: ?

```
def listAdd(lst1, lst2):
    i = 0
    res = []
    for j in lst1:
        res.append(j+lst2[i])
        i=i+1
    return res
```

# Example 5

**`def listAdd(lst1, lst2):`**

requires: len(lst1) <= len(lst2); lst1, lst2 must contain a type supporting the "+" operator; both lists must contain the same type elements at corresponding list positions

modifies: none

effects: none

returns: a new list of the same size as lst1, whose i-th element is the sum/concatenation of the i-th elements of **lst1** and **lst2**

```
def listAdd(lst1, lst2):
    i = 0
    res = []
    for j in lst1:
        res.append(j+lst2[i])
        i=i+1
    return res
```

# Specification Style

- A method is called for its side effects (effects clause) or its return value (return clause)
    - It is usually bad style to have both effects and return
    - There are exceptions. Can you think of one?
        - E.g., Map.put returns the previous value
- Main point of spec is to be helpful
    - Being overly formal does not help
    - Being too informal does not help either

# Specification Style

- A specification should be
  - Concise: not too many cases
  - Informative and precise
  - Specific (strong) enough: to make guarantees
  - General (weak) enough: to permit efficient implementation
    - Too weak a spec imposes too many preconditions and gives too few guarantees
    - Too strong a spec imposes too few preconditions and gives many guarantees, placing the burden on the implementation (e.g., is input array sorted?); may hinder efficiency
  - Like a class, methods should do one thing.
    - A complex spec is a warning about the design

# Specification Strength

- Sometimes, we need to compare specifications (we'll see why a little later)
- "A is stronger than B" means
  - For every implementation *I*
    - "*I* satisfies A" implies *"I* satisfies B"
    - If the implementation satisfies the stronger spec (A), it satisfies the weaker spec (B)
    - The opposite is not necessarily true
  - For every client *C*
    - "*C* meets the obligations of B" implies "*C* meets the obligations of A"
    - If C meets the weaker spec (B), it meets the stronger spec (A)
    - The opposite is not necessarily true
- In general, we want strong specifications
- A **larger world** of implementations satisfy the weaker spec B than the stronger spec A
- Consequently, it is easier to implement a weaker spec!
  - Weaker specs require *more* AND/OR Weaker specs guarantee (promise) *less*

# Specification Strength

- A weaker specification is easier to satisfy! And implement…
- A stronger specification is easier for clients to use.

- Example: spec A requires x > -1, spec B requires x > 0, A is stronger all else being the same.
- An implementation that meets A will certainly meet B.
- If Client satisfies B, it satisfies A.

# Specification Strength

```
int find(int[] a, int value) {
   for (int i=0; i<a.length; i++) {
      if (a[i] == value) return i;
   }
   return -1;
}
```

- Specification 1:
  - requires: **`a`** is non-null and **`value`** occurs in **`a`**
  - returns: **`i`** such that **`a[i] = value`**
- Specification 2:
  - requires: **`a`** is non-null
  - returns: **`i`** such that **`a[i] = value`** or **`i = -1`** if value is not in **`a`**

# Specification Strength

- Which one is STRONGER?
  - Spec 2.
  - If implementation I satisfies Spec 2, then I will satisfy Spec 1 as well.
- The reverse does not hold.

- A client built against Spec 2 CAN give array a and a value, such that value is in a, or it is not in a, and expect an index or -1.
  - But a client built against Spec 1 MUST GIVE a and value such that value is in a and expect an index.
  - Client built against Spec 1 will work with Spec 2.

# Strengthening and Weakening Specifications

- Strengthen a specification
  - Require less of client: fewer conditions in requires clause
    - Weaken precondition
  - Promise more to client: effects, modifies, returns
    - Strengthen postcondition
    - Effects/modifies affect fewer objects
    - Stronger requirements on returned values
- Weaken a specification
  - Require more of client: add conditions to requires
    - i.e., strengthen precondition
  - Promise less to client: effects, modifies, returns clauses are weaker, easier to realize in code
    - Weaken postcondition

# Input and Results – Strength Spec

- Method inputs
  - Argument types may be replaced with supertypes
    - E.g. replace Integer with Number
    - Contravariance
    - Doesn’t place any extra demand on client
- Method results
  - Result type may be replaced with a subtype
    - E.g. return Integer rather than Number
    - Covariance
    - Doesn't violate client's expectations
- No new exceptions for values in the domain
    - Domain in this case is the set of argument values
    - Violates client expectations
    - Existing exceptions may be replaced by subtypes of exceptions
- Basic Rule – No surprises

# Ease of Use by Client; Ease of Implementation

- Stronger specification is easier to use
  - Client has fewer preconditions to meet
  - Client gets more guarantees in postconditions
  - But stronger spec is harder to implement
- Weaker specification is easier to implement
  - Larger set of preconditions, relieve implementation from burden
  - Easier to guarantee less in postcondition
  - But weaker spec is harder to use

# Specification Strength

- Specification A consists of precondition $P_A$ and postcondition $Q_A$
- Specification B consists of precondition $P_B$ and postcondition $Q_B$
- A is stronger than B if and only if
  - $P_B$ is stronger than $P_A$ (this means, stronger specifications require less):
    - $P_A$ has a weaker precondition
    - Requires less of client
    - Preconditions are contravariant
  - $Q_A$ is stronger than $Q_B$ (stronger specifications promise more):
    - $Q_A$ has a stronger postcondition
    - Promises more to client
    - Postconditions are covariant
- In other words, A is stronger than B if and only if $P_B$ => $P_A$ ^ $Q_A$ => $Q_B$

## Specification Strength

- $P_B \Rightarrow P_A \wedge Q_A \Rightarrow Q_B$ is a necessary and sufficient condition, e.g.:
  - $P_B \Rightarrow P_A$ and $Q_A = Q_B$, A is stronger than B
  - $Q_A \Rightarrow Q_B$ and $P_A = P_B$, A is stronger than B
- $P_B \Rightarrow P_A$ and $Q_B \Rightarrow Q_A$, we can't say which spec A or B is stronger
- Some conditions are not related by implication (i.e., neither P => Q nor Q => P), similarly some specifications are not related by the "stronger than" relationship.

# Exercise: Order by Strength

Spec A: requires: a non-negative int argument
returns: an int in [1..10]

Spec B: requires: int argument
returns: an int in [2..5]

Spec C: requires: true
returns: an int in [2..5]

Spec D: requires: an int in [1..10]
returns: an int in [1..20]

# Substitutability

- Sometimes we use a method where another one is expected. Where?
  - X x; … x.m().
  - A subclass of X, e.g. Y, may have its own implementation of m, Y.m
  - In order for Y.m() to work correctly where X.m() was expected, the spec of Y.m() must be stronger than the spec of X.m()!
- Liskov Principle of Substitutability
  - An object with a stronger specification can be substituted for an object with a weaker one without altering desirable properties like correctness, tasks performed, etc.