Lab 3: Multiplayer Game

For this lab you will implement the MIT 6.102 (2025) Memory Scramble lab.

You can find the starter code that they provide to the students in this Github repo.

You are free to use any HTTP and unit test libraries.

Structure of your implementation

The provided skeleton (in TypeScript) already implements the web API server, your job remains to implement the Board ADT and complete the functions in "commands.ts" by calling your Board's methods.

You may use any language you want. However, you must still follow the structure imposed in the requirements and in the skeleton. Basically, you should convert the provided skeleton to your language of choice and go from there. More explicitly, you must retain the following structure:

- A mutable Board ADT with defined rep invariants and a checkRep() function
- A "commands" module implementing the given specification;

Module commands
F
Flip
Function flip
```
flip(
   board: Board,
   playerId: string,
   row: number,
   column: number,
): Promise<string>
```
Tries to flip over a card on the board, following the rules in the ps4 handout. If another player controls the card, then this operation waits until the flip either becomes possible or fails.

Parameters
board: Board
a Memory Scramble board

playerId: string
ID of player making the flip; must be a nonempty string of alphanumeric or underscore characters

row: number
row number of card to flip; must be an integer in [0, height of board), indexed from the top of the board

column: number
column number of card to flip; must be an integer in [0, width of board), indexed from the left of the board

Returns Promise<string>
the state of the board after the flip from the perspective of playerId, in the format described in the ps4 handout

Throws
an error (in a rejected promise) if the flip operation fails as described in the ps4 handout.

F
Look
Function look
look(board: Board, playerId: string): Promise<string>
Looks at the current state of the board.

Parameters
board: Board
a Memory Scramble board

playerId: string
ID of player looking at the board; must be a nonempty string of alphanumeric or underscore characters

Returns Promise<string>
the state of the board from the perspective of playerId, in the format described in the ps4 handout

F
Map
Function map
```
map(
   board: Board,
   playerId: string,
   f: (card: string) => Promise<string>,
): Promise<string>
```
Modifies board by replacing every card with f(card), without affecting other state of the game.

This operation must be able to interleave with other operations, so while a map() is in progress, other operations like look() and flip() should not throw an unexpected error or wait for the map() to finish. But the board must remain observably pairwise consistent for players: if two cards on the board match each other at the start of a call to map(), then while that map() is in progress, it must not cause any player to observe a board state in which that pair of cards do not match.

Two interleaving map() operations should not throw an unexpected error, or force each other to wait, or violate pairwise consistency, but the exact way they must interleave is not specified.

f must be a mathematical function from cards to cards: given some legal card c, f(c) should be a legal replacement card which is consistently the same every time f(c) is called for that same c.

Parameters
board: Board
game board

playerId: string
ID of player applying the map; must be a nonempty string of alphanumeric or underscore characters

f: (card: string) => Promise<string>
mathematical function from cards to cards

Returns Promise<string>
the state of the board after the replacement from the perspective of playerId, in the format described in the ps4 handout

F
Watch
Function watch
watch(board: Board, playerId: string): Promise<string>
Watches the board for a change, waiting until any cards turn face up or face down, are removed from the board, or change from one string to a different string.

Parameters
board: Board
a Memory Scramble board

playerId: string
ID of player watching the board; must be a nonempty string of alphanumeric or underscore characters

Returns Promise<string>
the updated state of the board from the perspective of playerId, in the format described in the ps4 handout

- An HTTP API that only calls the functions from the “commands” module (it can’t call the Board’s methods directly)

Grading

Implementation:

- **(10 points)** The game works correctly according to all the rules
- **(10 points)** You have unit tests for the Board ADT covering all the rules, the tests are readable and documented (and passing)

- **(4 points)** You have a script that simulates multiple players making random moves with random timeouts. The goal is to check that the game never crashes (and it doesn't). Requirements: 4 players, timeouts between 0.1ms and 2ms, no shuffling, 100 moves each.

Design and documentation:

- **(6 points)** You followed the required structure of the modules (especially the "commands" module)
- **(6 points)** Representation invariants (text attached lower), safety from rep exposure arguments, for every ADT
- **(8 points)** Specifications (text attached lower) for every method (function signature, preconditions, postconditions)

Deadline and presentation:

- **20 points** if you present before November 15, **minus 5** for every week after that
- **(12 points)** you understand how it all works

Specifications:

Reading 4: Specifications

Praxis Tutor exercises

Keep making progress on TypeScript by completing the following categories in the Praxis Tutor:

✓Comments & Assertions1/1

✓Exceptions1/1

Software in 6.102

Safe from bugs	Easy to understand	Ready for change

Correct today and correct in the unknown future. Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

Understand preconditions and postconditions in function specifications, and be able to write correct specifications

Be able to write tests against a specification

Understand how to use exceptions

Introduction

Specifications are the linchpin of teamwork. It's impossible to delegate responsibility for implementing a function without a specification. The specification acts as a contract: the implementer is responsible for meeting the contract, and a client that uses the function can rely on the contract. In fact, we'll see that like real legal contracts, specifications place demands on both parties: when the specification has a precondition, the client has responsibilities too.

In this reading we'll look at the role played by specifications of functions. We'll discuss what preconditions and postconditions are, and what they mean for the implementer and the client of a function. We'll also talk about how to use exceptions, an important language feature found not only in TypeScript, but also Python, Java, and many other modern languages, which allows us to make a function's interface safer from bugs and easier to understand.

Behavioral equivalence

Suppose you are working on a program containing this function, which finds the index of an integer in an array:

```
function find(arr: Array<number>, val: number): number {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === val) return i;
  }
  return -1;
}
```

This find function has many clients in the program (places where the function is called). You've just realized that frequently in this program, when find is called with a large array, the value it finds is likely to be either close to the start of the array (which is very fast to find), or close to the end (which is very slow, because it requires checking almost the entire array). So you have the clever idea to speed things up by searching from both ends of the array at the same time:

```
function find(arr: Array<number>, val: number): number {
  for (let i = 0, j = arr.length-1; i <= j; i++, j--) {
    if (arr[i] === val) return i;

    if (arr[j] === val) return j;

  }

  return -1;

}
```

Is it safe to replace find with this new implementation? Can we make this change without introducing bugs? To determine behavioral equivalence, we ask whether we could substitute one implementation for the other without affecting correctness.

Not only do these implementations have different performance characteristics, they actually have different output behavior for certain inputs. If val happens to appear more than once in the array, the original find always returns the lowest index at which it occurs. But the new find might return the lowest index or the highest index, whichever it finds first.

When val occurs at exactly one index of the array, however, the two implementations behave the same: they both return that index. It may be that the clients never rely on the behavior outside of that case. Whenever they call the function, they will be passing in an array with exactly one element matching val. For such clients, these two versions of find are the same, and we could switch from one implementation to the other without issue.

The notion of behavioral equivalence is in the eye of the beholder — that is, the client. In order to make it possible to substitute one implementation for another, and to know when this is acceptable, we need a specification that states exactly what the client depends on.

In this case, a specification that would allow these two implementations to be behaviorally equivalent might be:

find(arr: Array<number>, val: number): number

requires:

val occurs exactly once in arr

effects:

returns index i such that arr[i] = val

As we will see in more detail later, the requires clause specifies conditions that must be true for a legal call to find, and the effects clause specifies how the implementation will behave when the client has made a legal call.

reading exercises

Something's missing

Order matters

Behave nicely

Best behavior

Why specifications?

Our find example showed how a specification can help make a program both ready for change and safe from bugs. Many of the nastiest bugs in programs arise because of misunderstandings about behavior at the interface between two pieces of code. Although every programmer has specifications in mind, not all programmers write them down. As a result, different programmers on a team may have different specifications in mind. When the program fails, it's hard to determine where the error is. Precise specifications in the code let you apportion blame (to code fragments, not people!), and can spare you the agony of puzzling over where a bug fix should go.

Specifications are good for the client of a module because they help make the module easier to understand, like a label on the outside of a black box. Having a specification lets you understand what the module does without having to read the module's code. If you're not convinced that reading a spec is easier than reading code, compare our spec for find on the left, with its tricky implementation on the right:

find(arr: Array<number>, val: number): number

requires:

val occurs exactly once in arr

effects:

returns index i such that arr[i] = val

```
function find(arr: Array<number>, val: number): number {
  for (let i = 0, j = arr.length-1; i <= j; i++, j--) {
    if (arr[i] === val) return i;
    if (arr[j] === val) return j;
  }
  return -1;
}
```

Specifications are good for the implementer of a function because they give the implementer freedom to change the implementation without telling clients. Specifications can make code faster, too. We'll see that a specification can rule out certain states in which a function might be called. Restricting the inputs might allow the implementer to skip an expensive check that is no longer necessary and use a more efficient implementation. We will see an example of this later in this reading.

The contract acts as a firewall between client and implementer. It shields the client from the details of the workings of the module: as a client, you don't need to read the source code of the module if you have its specification. And it shields the implementer from the details of the usage of the module: as an implementer, you don't have to ask every client how they plan to use the module. This firewall results in decoupling, allowing the code of the module and the code of a client to be changed independently, so long as the changes respect the specification — each obeying its obligation under the contract.

Specification structure

Abstractly speaking, a specification of a function has several parts:

a function signature, giving the name, parameter types, and return type

a requires clause, describing additional restrictions on the parameters

an effects clause, describing the return value, exceptions, and other effects of the function

Taken together, these parts form the precondition and the postcondition of the function.

The precondition is an obligation on the client (the caller of the function). It is a condition on the state in which the function is invoked. One aspect of the precondition is the number and types of the parameters in the function signature, which TypeScript can statically check. Additional conditions are written down in the requires clause, for example:

narrowing a parameter type (e.g. x is an integer >= 0 to say that a number parameter x must actually be a nonnegative integer)

interactions between parameters (e.g., val occurs exactly once in arr)

The postcondition is an obligation on the implementer of the function. TypeScript can statically check some parts of the postcondition, notably the return type. Additional conditions are written down in the effects clause, including:

how the return value relates to the inputs

which exceptions are thrown, and when

whether and how objects are mutated

In general, the postcondition is a condition on the state of the program after the function is invoked, assuming the precondition was true before.

The overall structure is a logical implication: if the precondition holds when the function is called, then the postcondition must hold when the function completes.

If the precondition does not hold when the function is called, the implementation is not bound by the postcondition. It is free to do anything, including never returning, throwing an exception not mentioned in the spec, returning arbitrary results, making arbitrary mutations, etc.

reading exercises

Logical implication

Logical implementation

Specifications in TypeScript

Some languages (notably Eiffel) incorporate preconditions and postconditions as a fundamental part of the language, as expressions that the runtime system (or even the compiler) can automatically check to enforce the contracts between clients and implementers.

TypeScript does not go quite so far, but its static type declarations are effectively part of the precondition and postcondition of a function, a part that is automatically checked and enforced by the compiler. The rest of the contract — the parts that we can't write as types — must be described in a comment preceding the function, and generally depends on human beings to check it and guarantee it.

A common convention, originally designed for Java but now also used by TypeScript and JavaScript, is to put a documentation comment before the function, in which:

parameters are described by @param clauses

results are described by the @returns clause

Preconditions generally go into @param where possible, and postconditions into @returns. So a specification like this:

```
find(arr: Array<number>, val: number): number

requires:

val occurs exactly once in arr

effects:

returns index i such that arr[i] = val
```

… might be rendered in TypeDoc like this:

```
/**
 * Find a value in an array.
 * @param arr array to search, requires that val occurs exactly once
 *          in arr
 * @param val value to search for
 * @returns index i such that arr[i] = val
 */
function find(arr: Array<number>, val: number): number
```

Documenting your specifications using this TypeDoc format allows Visual Studio Code to show you (and clients of your code) useful information, and allows you to generate HTML documentation automatically.

reading exercises

TypeDoc

Concise TypeDoc specs

Back to Python

What a spec may talk about

The specification of a function can talk about the parameters and return value of the function, but it should never talk about local variables of the function, or helper functions defined inside the function body. You should consider the implementation invisible to the reader of the spec. It's behind the firewall as far as clients are concerned.

The source code of the function may even be unavailable to the reader of your spec, because the TypeDoc tool extracts just the spec comments from your code and renders them as HTML. This separation is a good thing: a spec frees the client from worrying about (or inappropriately depending on!) the details of the function's implementation. The client is only obliged to pay attention to the spec.

Avoid null

Many languages allow a variable to take on the special value like None or null, which means that the variable doesn't point to an object. By default, TypeScript behaves this way, so that any variable can be null regardless of the variable's type:

```
let word: string = null;  // legal by default!
```

This is an unfortunate hole in the static type system, because null is not truly a legal string value. You can't call any methods or use any properties with this reference:

```
word.toLowerCase() // throws TypeError

word.length        // throws TypeError
```

Note in particular that null is not the same as an empty string "" or an empty array [ ]. On an empty string or empty array, you can call methods and access instance variables. The length of an empty array or an empty string is 0. The length of a string variable that points to null isn't anything: accessing length throws a TypeError.

Also note that an array might be non-null but contain null as a value:

```
let words: Array<string> = [ null ];
```

These nulls are likely to cause errors as soon as someone tries to use the contents of the array.

Null values are troublesome and unsafe, so much so that good programming tries to avoid them. As a general convention, null values are disallowed in parameters and return values unless the spec explicitly says otherwise. So every function has a precondition on the object and array types in its parameters that they be non-null – including elements of collections like arrays, sets, and maps. Every function that can return object or array types implicitly has a postcondition that their values are non-null, again including elements of collection types.

When TypeScript has strict null checking enabled (which is the 6.102 configuration), the static type checker guarantees this:

```
let word: string = null;  // compile error when strict null checking is on

let words: Array<string> = [ null ]; // compile error when strict null checking is on
```

If a function allows null in a parameter or return value, it needs to explicitly declare null in the type (e.g. string|null). But this should be done sparingly. Avoid null.

Google has their own discussion of null in Guava, the company's core Java libraries. The project explains:

Careless use of null can cause a staggering variety of bugs. Studying the Google code base, we found that something like 95% of collections weren't supposed to have any null values in them, and having those fail fast* rather than silently accept null would have been helpful to developers.

Additionally, null is unpleasantly ambiguous. It's rarely obvious what a null return value is supposed to mean — for example, Map.get(key) [in Java] can return null either because the value in the map is null, or the value is not in the map. Null can mean failure, can mean success, can mean almost anything. Using something other than null makes your meaning clear.*

(*Boldface added for emphasis.)

Include emptiness

Make sure to understand the difference between null and emptiness.

Recall that in Python, None is not the same as the empty string "", the empty array [ ], or the empty dictionary { }. These empty objects like these are valid objects that simply happen to contain no elements. But you can use them with all the usual operations allowed by the type. For example, len("") returns 0, and "" + "a" returns "a". That's not true of None: len(None) and None + "a" both produce errors.

The same idea translates to TypeScript. The null reference is not a valid string, or array, or map, or any other object. But the empty string "" is a valid string value, and the empty array [ ] is a valid array value.

The upshot of this is that empty values are always allowed as parameter or return values unless a spec explicitly disallows them.

reading exercises

Vacuous statements

Intervals

Combining no things

Testing and specifications

In testing, we talk about black box tests that are chosen with only the specification in mind, and glass box tests that are chosen with knowledge of the actual implementation (Testing). But it's important to note that even glass box tests must follow the specification. Your implementation may provide stronger guarantees than the specification calls for, or it may have specific behavior where the specification is undefined. But your test cases should not count on that behavior. Test cases must be correct, obeying the contract just like every other client.

For example, suppose you are testing this specification of find, slightly different from the one we've used so far:

find(arr: Array<number>, val: number): number

requires:

val occurs in arr

effects:

returns index i such that arr[i] = val

This spec has a strong precondition in the sense that val is required to be found; and it has a fairly weak postcondition in the sense that if val appears more than once in the array, this specification says nothing about which particular index of val is returned. Even if you implemented find so that it always returns the lowest index, your test case can't assume that specific behavior:

```
let array: Array = [7, 7, 7];

let i = find(array, 7);

assert.strictEqual(i, 0);  // bad test case: assumes too much, more than the postcondition promises

assert.strictEqual(array[i], 7);  // correct
```

Similarly, even if you implemented find so that it (sensibly) throws an exception when val isn't found, instead of returning some arbitrary misleading index, your test case can't assume that behavior, because it can't call find() in a way that violates the precondition.

So what does glass box testing mean, if it can't go beyond the spec? It means you are trying to find new test cases that exercise different parts of this particular implementation, but assert the behavior in an implementation-independent way, following the spec.

Testing units

Recall the search engine example from Testing with these functions:

```
/**

 * @returns the contents of the file

 */

function load(filename: string): string { ... }


/**

 * @returns the words in string s, in the order they appear,

 *       where a word is a contiguous sequence of

 *       non-whitespace and non-punctuation characters
 */
function extract(s: string): Array<string> { ... }


/**
 * @returns an index mapping a word to the set of filenames
 *        containing that word, for all files in the input set
 */
function index(filenames: Set<string>): Map<string, Set<string>>  {
   ...
   for (let file of files) {
      let doc = load(file);
      let words = extract(doc);
      ...
   }
   ...
}
```

We talked then about unit testing, the idea that we should write tests of each module of our program in isolation. A good unit test is focused on just a single specification. Our tests will nearly always rely on the specs of JavaScript library functions, but a unit test for one function we've written shouldn't fail if a different function fails to satisfy its spec. As we saw in the example, a test for extract() shouldn't fail if load() doesn't satisfy its postcondition.

Good integration tests, tests that use a combination of modules, will make sure that our different functions have compatible specifications: callers and implementers of different functions are passing and returning values as the other expects. Integration tests cannot replace systematically-designed unit tests. From the example, if we only ever test extract by calling index, we will only test it on a potentially small part of its input space: inputs that are possible outputs of load. In doing so, we've left a place for bugs to hide, ready to jump out when we use extract for a different purpose elsewhere in our program, or when load starts returning documents written in a new format, etc.

reading exercises

gcd 1

gcd 2

All tests must follow the spec

We cannot test for behavior when a precondition is violated: for example, we cannot check that a function fails fast as opposed to returning a garbage value on some illegal input, because all tests must follow the spec. If the precondition is one that the implementer can reasonably check, then it often makes sense to change the spec: remove the precondition, and instead specify the behavior in the postcondition. We'll discuss two ways to do that below (exceptions and special results) and dive deeply into spec-writing decisions like this in the next reading.

Specifications for mutating functions

We discussed mutable vs. immutable objects in an earlier reading (Static Checking). But our specification examples thus far haven't illustrated how to describe side-effects — changes to mutable objects or input/output state — in the postcondition.

So here's a specification that describes a function that mutates an object:

addAll(array1: Array<string>, array2: Array<string>): boolean

requires:

array1 and array2 are not the same object

effects:

modifies array1 by adding the elements of array2 to the end of it, and returns true if and only if array1 changed as a result of call

First, look at the postcondition. It gives two constraints: the first telling us how array1 is modified, and the second telling us how the return value is determined.

Second, look at the precondition. It tells us that array1 cannot be the same object as array2. The behavior of the function if you attempt to add the elements of an array to itself is undefined. You can easily imagine why the implementer of the function would want to impose this constraint: it's not likely to rule out any useful applications of the function, and it makes it easier to implement. The specification allows a simple implementation in which you take an element from array2 and add it to array1, then go on to the next element of array2 until you get to the end.

If array1 and array2 are the same array, this simple algorithm will not terminate, as shown in the sequence of snapshot diagrams on the right — or practically speaking it will throw a memory error when the array object has grown so large that it consumes all available memory. Either outcome, infinite loop or crash, is permitted by the specification because of its precondition.

Here is another example of a mutating function:

sort(array: Array<string>): void

requires:

nothing

effects:

puts array in sorted order, i.e. array[i] ≤ array[j] for all 0 ≤ i < j < array.length

And an example of a function that does not mutate its argument:

toLowerCase(array: Array<string>): Array<string>

requires:

nothing

effects:

returns a new array t, with same length as array, where t[i] = array[i].toLowerCase()

Just as null is implicitly disallowed unless stated otherwise, programmers also assume that mutation is disallowed unless stated otherwise. The spec of toLowerCase could explicitly state as an effect that “array is not modified”, but in the absence of a postcondition describing mutation, we demand no mutation of the inputs.

When writing a specification in TypeDoc, mutations that affect a parameter can be described in the corresponding @param clause:

```
/**

 * Sorts an array.

 * @param array - mutated into sorted order, so that
 *    array[i] ≤ array[j] for all 0 ≤ i < j < array.length.
 */
function sort(array: Array<string>): void { ... }
```

reading exercises

Code review

Exceptions

Now that we're writing specifications and thinking about how clients will use our functions, let's discuss how to handle exceptional cases in a way that is safe from bugs and easy to understand.

Since an exception is a possible output from a function, it may have to be described in the postcondition for the function. An exception can be documented with a @throws clause in the documentation comment. This section discusses when to include an exception in a specification, and when not to.

Exceptions for signaling bugs

You've probably already seen some exceptions in your Python or TypeScript programming so far. For example, in Python:

IndexError is thrown when an array index array[i] is outside the valid range for the array array

KeyError is thrown when a dictionary lookup dict[key] finds no matching key

TypeError is thrown for a variety of dynamic type errors, such as trying to call a method on None

TypeScript also has TypeError for dynamic type errors, but most dynamic errors are signaled with a generic Error class.

These kinds of exceptions generally indicate bugs – in either the client or the implementation – and the information displayed when the exception is thrown can help you find and fix the bug.

Exceptions that signal bugs are not part of the postcondition of a function, so they should not appear in @throws. For example, TypeError normally should never be mentioned in a spec. The types of the arguments are part of the precondition, which means that a valid implementation is free to throw TypeError without any warning if those types are violated (for example, if a client manages to pass a null value). So this spec, for example, never mentions TypeError:

```
/**
 * @param array array of strings to convert to lower case
 * @returns new array t, same length as array,
 *        where t[i] is array[i] converted to lowercase for all valid indices i
 */
function toLowerCase(array: Array<string>): Array<string>
```

Exceptions for anticipated failures

Exceptions are not just for signaling bugs. They can be used to signal anticipated sources of failure, in such a way that the caller can catch them and respond to them. Exceptions that signal anticipated failures should be documented with a @throws clause, specifying the conditions under which that failure occurs. For example:

```
/**
 * Compute the integer square root.
 * @param x integer value to take square root of
 * @returns square root of x
 * @throws NotPerfectSquareError if x is not a perfect square
 */
function integerSquareRoot(x: number): number
```

And here's a new version of the addAll spec:

```
/**
 * If the array1 !== array2, adds the elements of array2 to the end of array1.
 * @returns true if array1 changed as a result
 * @throws AliasingError if array1 === array2
 */
function addAll(array1: Array<string>, array2: Array<string>): boolean
```

In the original addAll above, the spec included array1 !== array2 as a precondition: the behavior of the function was undefined if that requirement was violated. In this version of the spec, the precondition is removed, and instead we have a postcondition that defines the behavior both when the arrays are different objects ("adds the elements…" and the @returns clause) and when they are the same object (the @throws clause).

reading exercises

Throw all the things!

Special results

Exceptions are the best way to communicate a problem that the caller is unlikely to be able to handle, for example because they provided an illegal input: either the caller or someone who called them has a bug.

For unusual or exceptional results that the caller should be prepared to handle, different programming languages take different approaches. In the Java language, for example, the implementer can declare exceptions that the caller is statically required to handle: if they don't either catch the exception or add it to their own specification as a thrown exception, the compiler gives a static error.

TypeScript does not provide static checking for exception handling, so using exceptions for special-case results can be a source of bugs. For example, suppose we use integerSquareRoot with inputs taken from a file or entered by the user, and we neglect to try ... catch the NotPerfectSquareError. An invalid number will crash the program, when it would be better to produce a clear error message or prompt the user to try again. But TypeScript lacks the static checking of exceptions that would help us see and account for the problem ahead of time.

A good alternative is to make the return type of the function a union type, a type whose set of values is the union of the sets of values defined by two (or more) other types:

```
/**
 * Compute the integer square root.
 * @param x integer value to take square root of
 * @returns square root of x if x is a perfect square, undefined otherwise
 */
function integerSquareRoot(x: number): number|undefined
```

undefined is very similar to null: in JavaScript, declared but un-initialized variables have the value undefined instead of null. Without strict null checking, any variable can be undefined or null. With strict checking on, both are excluded.

By convention, null is used to mean "a value that is no value," and it should be avoided at all times. In contrast, undefined is used to mean "no value here at all," and as long as strict null checking is on, we can use it in a union type to signal a special case result. In our example, trying to use the updated integerSquareRoot without accounting for the possibility that it might return undefined is in many cases a static error:

```
let twice = integerSquareRoot(input) * 2; // static error

if (integerSquareRoot(input) > 4) { ... } // static error
```

… although we must still exercise care:

```
console.log('Your lucky number is:', integerSquareRoot(5)); // no error
```

As a client, how can we use integerSquareRoot() correctly? We need to check for the possibility of the special result. For example:

```
let root = integerSquareRoot(input); // type of `root` is `number|undefined`

if (root === undefined) {

   // in this branch, TypeScript knows that `root` has type `undefined`

   ...

} else {

   // in this branch, TypeScript knows that `root` has type `number`

   let twice = root * 2; // now legal

   ...

}
```

If the special result would indicate a bug, then one common simplification is to simply assert it:

```
let root = integerSquareRoot(input); // type of `root` is `number|undefined`

assert(root !== undefined);
```

```
// if we got past the assertion, TypeScript knows that `root` has type `number`
let twice = root * 2; // now legal
```

This is often combined using the nullish coalescing operator:

```
let root = integerSquareRoot(input) ?? assert.fail('root is undefined');
// `root` now simply has type `number`
let twice = root * 2; // now legal
```

Missing Map keys

If you have a Python dictionary, what happens when you try to look up a key that is not in the dictionary?

```
zoo = { 'Tim': 'beaver' }
zoo['Tim']     # => 'beaver'
zoo['Bao Bao'] # => raises KeyError: 'Bao Bao'
```

Python considers this an exception. TypeScript makes a different choice:

```
const zoo = new Map([['Tim', 'beaver']]);
zoo.get('Tim');     // => 'beaver'
zoo.get('Bao Bao'); // => undefined
```

In Python, the postcondition specifies a KeyError. In TypeScript, for a Map with values of type V, the postcondition specifies a return type of V|undefined.

Illegal array indices

What if we keep our zoo in an array instead of a map?

```
const zoo = [ 'Tim' ];
for (let i = 0; i <= 1; i++) {
  let name: string = zoo[i];
  console.log(`Hello, ${name}!`);
}
```

```
// => prints:
//    Hello, Tim!
//    Hello, undefined!
```

Oh no. The compiler allows zoo[i] to be assigned to name, even though name must be a string and zoo[i] might not be! This is another hole in the static type system, and it requires another bit of TypeScript configuration: turning on no unchecked index access will change the return type of the [..] indexing operator and make it a union with undefined. So far in 6.102 that option has been off. We'll be turning it on in future exercises and problem sets. Once we do, we'll have to account for the possibility that the index might be illegal in code like this, or change the code to avoid using indices.

You can edit this example in the TypeScript Playground with noUncheckedIndexedAccess on.

Note that this example is doing out-of-bounds indexing (beyond the end of the array). A different reason that an array index can return undefined is when the array is sparse, but sparse arrays should be avoided.

Modules

This reading has focused on specifications for functions. But TypeScript/JavaScript and Python (and many other languages) also support modules, which are an abstraction above the function level, allowing a group of related functions to be collected together.

(This specific use of the term module, as a language feature, unfortunately collides with the more general computer-science term module, which we previously discussed. Python and TS/JS modules are indeed modules in the general CS sense, but so are other units of code like functions, classes, libraries, etc.)

In modern TypeScript/JavaScript, each file is a module. A module can export a particular set of functions (or constants, or user-defined types) by declaring them with the keyword export. A client of a module must import from the module, either specifying particular functions to import, or simply importing all the exported functions it wants to use.

The specification of a module is the combination of its exported function specifications (or constants, or user-defined types). Functions and variables that are not exported are private to the module's implementation of the module, much like local variables and internal helper functions are private to a function body. They are not part of the specification, because clients cannot access them or depend on them.

Python modules work similarly, except that Python has no explicit export declaration, and instead controls what is nominally exported from a module using naming conventions (functions and variables starting with _ are considered internal to the implementation) and __all__.

Summary

Before we wrap up, check your understanding with one last example:

reading exercises

Scrabble 1

Scrabble 2

A specification acts as a crucial firewall between the implementer of a module and its client. It makes separate development possible: the client is free to write code that uses the module without seeing its source code, and the implementer is free to write the code that implements the module without knowing how it will be used.

Let’s review how specifications help with the main goals of this course:

Safe from bugs. A good specification clearly documents the mutual assumptions that a client and implementer rely on. Bugs often come from disagreements at the interfaces, and the presence of a specification helps avoid those disagreements. Using machine-checked language features in your spec, like static typing and exceptions rather than just a human-readable comment, can reduce bugs still more.

Easy to understand. A short, simple spec is easier to understand than the implementation itself, and saves other people from having to read the code.

Ready for change. Specs establish contracts between different parts of your code, allowing those parts to change independently as long as they continue to satisfy the requirements of the contract.

Representation invariants:
Reading 7: Abstraction Functions & Rep Invariants

Software in 6.102

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

Today's reading introduces several ideas:

invariants

representation exposure

abstraction functions

representation invariants

In this reading, we study a more formal mathematical idea of what it means for a class to implement an ADT via the notions of abstraction functions and rep invariants. These mathematical notions are eminently practical in software design. The abstraction function gives us a way to cleanly define the equality operation on an abstract data type (which we'll discuss in more depth in a future class). The rep invariant makes it easier to catch bugs caused by a corrupted data structure.

Invariants

Resuming our discussion of what makes a good abstract data type, the final, and perhaps most important, property of a good abstract data type is that it preserves its own invariants. An invariant is a property of a program that is always true, for every possible runtime state of the program, given it was true at the beginning. You may recall this concept of a preserved invariant from 6.1200 [formerly 6.042]. Immutability is one crucial invariant that we've already encountered: once created, an immutable object should always represent the same value, for its entire lifetime. But there are many other kinds of invariants, including types of variables (i: number means that i is always a number), or relationships between variables (e.g. when i is used to loop over the indices of an array, then 0 <= i < array.length is an invariant within the body of the loop).

Saying that the ADT preserves its own invariants means that the ADT is responsible for ensuring that its own invariants hold. This is done by hiding or protecting the variables involved in the invariants (e.g., using language features like private), and allowing access only through operations with well-defined contracts.

When an ADT preserves its own invariants, reasoning about the code becomes much easier. If you can count on the fact that strings are immutable, for example, then you can rule out the possibility that a string has been mutated when you're debugging code that uses strings – or when you're trying to establish an invariant for another ADT that uses strings. Contrast that with a mutable string type, which can be mutated by any code that has access to it. To reason about a bug or invariant involving a mutable string, you'd have to check all the places in the code where the string might be accessible.

That argument applies in all situations, even when you're programming by yourself. But when you are programming in a team, or writing software to be used by developers outside your team, ADTs that preserve their own invariants are safer from bugs too. There won't be nearly as high a chance of other teams in the company misusing your code, or real-world users accidentally breaking your software.

reading exercises

Statically-checked invariants

Immutability

Later in this reading, we'll see many interesting invariants. Let's focus on immutability for now. Here's a specific example:

```
/**
 * This immutable data type represents a tweet from Twitter.
 */
class Tweet {

    public author: string;
    public text: string;
    public timestamp: Date;

    /**
```

```
     * Make a Tweet.
     * @param author    Twitter user who wrote the tweet
     * @param text      text of the tweet
     * @param timestamp date/time when the tweet was sent
     */
    public constructor(author: string, text: string, timestamp: Date) {
        this.author = author;
        this.text = text;
        this.timestamp = timestamp;
    }
}
```

How do we guarantee that these Tweet objects are immutable – that, once a tweet is created, its author, message, and date can never be changed?

The first threat to immutability comes from the fact that clients can — in fact must — directly access its fields. So nothing's stopping us from writing code like this:

```
let t: Tweet = new Tweet("justinbieber",
                "Thanks to all those beliebers out there inspiring me every day",
                new Date());
t.author = "rbmllr";
```

This is a trivial example of representation exposure, meaning that code outside the class can modify the representation directly. Rep exposure like this threatens not only invariants, but also representation independence. We can't change the implementation of Tweet without affecting all the clients who are directly accessing those fields.

Fortunately, TypeScript gives us language mechanisms to deal with this kind of rep exposure:

```
class Tweet {
```

```
private readonly author: string;
private readonly text: string;
private readonly timestamp: Date;

public constructor(author: string, text: string, timestamp: Date) {
   this.author = author;
   this.text = text;
   this.timestamp = timestamp;
}

/**
 * @returns Twitter user who wrote the tweet
 */
public getAuthor(): string {
   return this.author;
}

/**
 * @returns text of the tweet
 */
public getText(): string {
   return this.text;
}

/**
 * @returns date/time when the tweet was sent
 */
public getTimestamp(): Date {
   return this.timestamp;
```

```
    }


}
```

The private and public keywords indicate which fields and methods are accessible only within the class and which can be accessed from outside the class. private is used for the object's internal state and internal helper methods, while public indicates methods and constructors that are intended for clients of the class. The observers getAuthor(), getText(), and getTimestamp() allow clients to still access a Tweet object's properties.

readonly is used to indicate which of the object's instance variables can be reassigned and which can't. Note that TypeScript uses readonly only for instance variables, and const only for local and global variables, but both readonly and const mean the same thing: unreassignable.

But that's not the end of the story: the rep is still exposed! Consider this perfectly reasonable client code that uses Tweet:

```
/**
 * @returns a tweet that retweets t, one hour later
 */
function retweetLater(t: Tweet): Tweet {
    const d: Date = t.getTimestamp();
    d.setHours(d.getHours()+1);
    return new Tweet("rbmllr", t.getText(), d);
}
```

retweetLater takes a tweet and should return another tweet with the same message (called a retweet) but sent an hour later. The retweetLater method might be part of a system that automatically echoes funny things that Twitter celebrities say.

What's the problem here? The getTimestamp call returns a reference to the same Date object referenced by tweet t. t.timestamp and d are aliases to the same mutable object. So when that date object is mutated by d.setHours(), this affects the date in t as well, as shown in the snapshot diagram.

Tweet’s immutability invariant has been broken. The problem is that Tweet leaked out a reference to a mutable object that its immutability depended on. We exposed the rep, in such a way that Tweet can no longer guarantee that its objects are immutable. Perfectly reasonable client code created a subtle bug.

We can patch this kind of rep exposure by using defensive copying, a strategy we’ve seen before: making a copy of a mutable object to avoid leaking out references to the rep. Here’s the code:

```
public getTimestamp(): Date {
   return new Date(this.timestamp.getTime());
}
```

Mutable types often have a copy constructor that allows you to make a new instance that duplicates the value of an existing instance. In this case, Date’s copy constructor uses the timestamp value, measured in milliseconds since January 1, 1970. As another example, Map’s copy constructor takes an array of key-value pairs, which can be extracted from the map you want to copy by calling entries().

So we’ve done some defensive copying in the return value of getTimestamp. But we’re not done yet! There’s still rep exposure. Consider this (again perfectly reasonable) client code:

```
/**
 * @returns an array of 24 inspiring tweets, one per hour today
 */
function tweetEveryHourToday(): Array<Tweet> {
   const array: Array<Tweet> = [];
   const date: Date = new Date();
   for (let i = 0; i < 24; i++) {
      date.setHours(i);
      array.push(new Tweet("rbmllr", "keep it up! you can do it", date));
```

```
    }
    return array;
}
```

This code intends to advance a single Date object through the 24 hours of a day, creating a tweet for every hour. But notice that the constructor of Tweet saves the reference that was passed in, so all 24 Tweet objects end up with the same time, as shown in this snapshot diagram.

Again, the immutability of Tweet has been violated. We can fix this problem, too, by using judicious defensive copying, this time in the constructor:

```
public constructor(author: string, text: string, timestamp: Date) {
    this.author = author;
    this.text = text;
    this.timestamp = new Date(timestamp.getTime());
}
```

In general, you should carefully inspect the argument types and return types of all your ADT operations. If any of the types are mutable, make sure your implementation doesn't return direct references to its representation. Returning a reference to a mutable rep object causes rep exposure.

You may object that this seems wasteful. Why make all these copies of dates? Why can't we just solve this problem by a carefully written specification, like this?

```
/**
 * Make a Tweet.
 * @param author    Twitter user who wrote the tweet
 * @param text      text of the tweet
 * @param timestamp date/time when the tweet was sent. Caller must never
 *                  mutate this Date object again!
 */
public constructor(author: string, text: string, timestamp: Date) {
```

This approach is sometimes taken when there isn't any other reasonable alternative – for example, when the mutable object is too large to copy efficiently. But the cost in your ability to reason about the program, and your ability to avoid bugs, is enormous. In the absence of compelling arguments to the contrary, it's almost always worth it for an abstract data type to guarantee its own invariants, and preventing rep exposure is essential to that. We used similar reasoning when we talked about designing specifications as well.

An even better solution is to prefer immutable types. If Date were an immutable type, then we would have ended this section after talking about public and private. No further rep exposure would have been possible.

reading exercises

Given:

```
class Zoo {
  private animals: Array<string>;

  public constructor(animals: Array<string>) {
    this.animals = animals;
  }

  public getAnimals(): Array<string> {
    return this.animals;
  }
}
```

Aliasing 1

Aliasing 2

Rep exposure

Sign here

Although the signed classes in the previous exercise don't actually exist in JavaScript/TypeScript, they are indeed a feature of Java, and would you believe it: Java 1.1 had precisely this security bug. The fix was to return a copy of the array.

Rep invariant and abstraction function

We now take a deeper look at the theory underlying abstract data types. This theory is not only elegant and interesting in its own right; it also has immediate practical application to the design and implementation of abstract types. If you understand the theory deeply, you'll be able to build better abstract types, and will be less likely to fall into subtle traps.

In thinking about an abstract type, it helps to consider the relationship between two spaces of values.

The space of abstract values consists of the values that the type is designed to support, from the client's point of view. For example, an abstract type for unbounded integers, like TypeScript's BigInt, would have the mathematical integers as its abstract value space.

The space of representation values (or rep values for short) consists of the objects that actually implement the abstract values. For example, a BigInt value might be implemented using an array of digits, represented as number values. The rep space would then be the set of all such arrays.

In simple cases, an abstract type will be implemented as a single object, but more commonly a small network of objects is needed. For example, a rep value for Array might possibly be a linked list, a group of objects linked together by next and previous pointers. So a rep value is not necessarily a single object, but often something rather complicated.

Now of course the implementer of the abstract type must be interested in the representation values, since it is the implementer's job to achieve the illusion of the abstract value space using the rep value space.

Suppose, for example, that we choose to use a string to represent a set of characters:

```
class CharSet {
  private s: string;
  ...
```

}

the abstract space and rep space of CharSet

Then the rep space R contains strings, and the abstract space A is mathematical sets of characters. We can show the two value spaces graphically, with an arrow from a rep value to the abstract value it represents. There are several things to note about this picture:

Every abstract value has some rep value that maps to it. The purpose of implementing the abstract type is to support operations on abstract values. Presumably, then, we will need to be able to create and manipulate all possible abstract values, and they must therefore be representable.

Multiple rep values can map to the same abstract value. This happens because the representation isn't a tight encoding. There's more than one way to represent an unordered set of characters as a string.

Not all rep values map to an abstract value. In this case, the string "abbc" doesn't map to anything, because we have decided that the rep string should not contain duplicates. This will allow us to terminate the remove method when we hit the first instance of a particular character, since we know there can be at most one.

In practice, we can only illustrate a few elements of the two spaces and their relationships; the graph as a whole is infinite. So we describe it by giving two things:

1. An abstraction function that maps rep values to the abstract values they represent:

AF : R → A

The arrows in the diagram above show the abstraction function. In the terminology of functions, the properties we discussed above can be expressed by saying that the function is surjective (also called onto), not necessarily injective (one-to-one) and therefore not necessarily bijective, and often partial.

2. A rep invariant that maps rep values to booleans:

RI : R → boolean

A rep invariant is a predicate over rep values.

For a rep value r, RI(r) is true if and only if r is mapped by AF. In other words, RI tells us whether a given rep value is well-formed. Alternatively, you can think of RI as a set: it's the subset of rep values on which AF is defined.

the abstract space and rep space of CharSet using the NoRepeatsRep

For example, the diagram at the right showing a rep for CharSet that forbids repeated characters:

RI("a") = true

RI("ac") = true

RI("acb") = true

but

RI("aa") = false

RI("abbc") = false

Rep values that obey the rep invariant are shown in the green part of the R space, and must map to an abstract value in the A space. Rep values that violate the rep invariant are shown in the red zone, and have no equivalent abstract value in the A space.

Both the rep invariant and the abstraction function should be documented in the code, right next to the declaration of the rep itself:

```
class CharSet {
   private s: string;
   // Rep invariant:
   //   s contains no repeated characters
   // Abstraction function:
   //   AF(s) = {s[i] | 0 <= i < s.length}
   ...
}
```

A common confusion about abstraction functions and rep invariants is that they are determined by the choice of rep and abstract value spaces, or even by the abstract value space alone. If this were the case, they would be of little use, since they would be saying something redundant that's already available elsewhere.

The abstract value space alone doesn't determine AF or RI: there can be several representations for the same abstract type. A set of characters could equally be represented as a string, as above, or as a bit vector, with one bit for each possible character. Clearly we need two different abstraction functions to map these two different rep value spaces.

It's less obvious why the choice of both spaces doesn't determine AF and RI. The key point is that defining a type for the rep, and thus choosing the values for the space of rep values, does not determine which of the rep values will be deemed to be legal, and of those that are legal, how they will be interpreted. Rather than deciding, as we did above, that the strings have no duplicates, we could instead allow duplicates, but at the same time require that the characters be sorted, appearing in nondecreasing order. This would allow us to perform a binary search on the string and thus check membership in logarithmic rather than linear time. Same rep value space – different rep invariant:

the abstract space and rep space of CharSet using the SortedRep

```
class CharSet {
  private s: string;
  // Rep invariant:
  //   s[0] <= s[1] <= ... <= s[s.length-1]
  // Abstraction function:
  //   AF(s) = {s[i] | 0 <= i < s.length}
  ...
}
```

Even with the same type for the rep value space and the same rep invariant, we might still interpret the rep differently by using different abstraction functions. Suppose the rep invariant admits any string of characters. Then we could define the abstraction function, as above, to interpret the array's elements as the elements of the set. But there's no a priori reason to let the rep decide the interpretation. Perhaps we'll interpret consecutive pairs of characters as subranges, so that the string rep "acgg" is interpreted as two range pairs, [a-c] and [g-g], and therefore represents the set {a,b,c,g}. Here's what the AF and RI would look like for that representation:

the abstract space and rep space of CharSet using the SortedRangeRep

```
class CharSet {
    private String s;
    // Rep invariant:
    //   s[0] <= s[1] <= ... <= s[s.length-1]
    // Abstraction function:
    //   AF(s) = union of { c | s[2i] <= c <= s[2i+1] }
    //           for all 0 <= i < s.length/2
    ...
}
```

(What does this abstraction function do when s.length is odd, i.e. when there is an unpaired character at the end of the string?)

The essential point is that implementing an abstract type means not only choosing the two spaces – the abstract value space for the specification and the rep value space for the implementation – but also deciding which rep values are legal, and how to interpret them as abstract values.

It's critically important to write down these assumptions in your code, as we've done above, so that future programmers (and your future self) are aware of what the representation actually means. Why? What happens if different implementers disagree about the meaning of the rep?

You can look at example code for three different CharSet implementations.

reading exercises

Exploring a rep

Who knows what?

Rep invariant pieces

Trying to implement without an AF/RI

Trying to implement without an AF/RI #2

Trying to implement without an AF/RI #3

Example: rational numbers

the abstraction function and rep invariant of RatNum

Here’s an example of an abstract data type for arbitrary-precision rational numbers, implemented using BigInt in the rep. Look closely at its rep invariant and abstraction function. It would be completely reasonable to design another implementation of this same ADT with a more permissive RI. With such a change, some operations might become more expensive to perform, and others cheaper.

```
class RatNum {

    private readonly numerator: bigint;
    private readonly denominator: bigint;

    // Rep invariant:
    //   denominator > 0
    //   numerator/denominator is in reduced form,
    //       i.e. gcd(|numerator|,denominator) = 1

    // Abstraction function:
    //   AF(numerator, denominator) = numerator/denominator

    /**
     * Make a new RatNum = (n / d).
     * @param n numerator
     * @param d denominator
     * @throws Error if d = 0
     */
    public constructor(n: bigint, d: bigint) {
        // reduce ratio to lowest terms
        const g = gcd(BigInt(n), BigInt(d));
        const reducedNumerator = n / g;
        const reducedDenominator = d / g;


        // make denominator positive
        if (d < 0n) {
          this.numerator = -reducedNumerator;
          this.denominator = -reducedDenominator;
        } else if (d > 0n) {
          this.numerator = reducedNumerator;
          this.denominator = reducedDenominator;
        } else {
          throw new Error('denominator is zero');
        }
        this.checkRep();
    }


}
```

(Note that bigint literals are integers suffixed with n.)

reading exercises

RatNum

Checking the rep invariant

The rep invariant isn't just a neat mathematical idea. If your implementation asserts the rep invariant at run time, then you can catch bugs early. Here's a method for RatNum that tests its rep invariant:

```
// assert the rep invariant
private checkRep(): void {
  assert(this.denominator > 0n);
  assert(gcd(abs(this.numerator), this.denominator) === 1n);
}
```

You should certainly call checkRep() to assert the rep invariant at the end of every operation that creates or mutates the rep – in other words, creators, producers, and mutators. Look back at the RatNum code above, and you'll see that it calls checkRep() at the end of the constructor.

Observer methods don't normally need to call checkRep(), but it's good defensive practice to do so anyway. Why? Calling checkRep() in every method, including observers, means you'll be more likely to catch rep invariant violations caused by rep exposure.

Why is checkRep private? Who should be responsible for checking and enforcing a rep invariant – clients, or the implementation itself?

reading exercises

Checking the rep invariant

No null values in the rep

Recall from the Specifications reading that null and undefined values are troublesome and unsafe, so much so that we try to remove them from our programming entirely. Preconditions and postconditions implicitly require that objects be non-null. Turning on strict null-checking in TypeScript enforces this with static type checking.

We extend that prohibition to the reps of abstract data types. The rep invariant implicitly includes x !== null and x !== undefined for every reference x in the rep, including references inside arrays, maps, etc. So if your rep is:

```
class CharSet {
   s: string;
}
```

then its rep invariant automatically includes s !== null && s !== undefined, and you don't need to state that in a rep invariant comment. And if you have strict null checking enabled (which is the 6.102 TypeScript configuration), the static type checker guarantees it.

If some field in the rep is allowed to be null, then its type should say so explicitly. A union type like string|undefined is an example of how to do that. But think twice before designing a rep like this, because allowing null and undefined to leak into your code leads to headaches down the road.

reading exercises

Null check?

Benevolent side-effects

Recall that a type is immutable if and only if a value of the type never changes after being created. With our new understanding of the abstract space A and rep space R, we can refine this definition: the abstract value should never change. But the implementation is free to mutate a rep value as long as it continues to map to the same abstract value, so that the change is invisible to the client. This kind of change is called a benevolent side-effect.

Here's a simple example using an alternative rep for the RatNum type we saw earlier. This rep has a weaker rep invariant that doesn't require the numerator and denominator to be stored in lowest terms:

```
class RatNum {

   private numerator: bigint;
   private denominator: bigint;

   // Rep invariant:
   //   denominator != 0

   // Abstraction function:
   //   AF(numerator, denominator) = numerator/denominator

   /**
    * Make a new RatNum = (n / d).
    * @param n numerator
    * @param d denominator
```

```
     * @throws Error if d = 0
     */
    public constructor(n: bigint, d: bigint) {
        if (d === 0n) throw new Error("denominator is zero");
        this.numerator = n;
        this.denominator = d;
        checkRep();
    }

    ...
}
```

This weaker rep invariant allows a sequence of RatNum arithmetic operations to simply omit reducing the result to lowest terms. But when it’s time to display a result to a human, we first simplify it:

```
    /**
     * @returns a string representation of this rational number
     */
    public toString(): string {
        const g = gcd(this.numerator, this.denominator);
        this.numerator /= g;
        this.denominator /= g;
        if (this.denominator < 0n) {
            this.numerator = -this.numerator;
            this.denominator = -this.denominator;
        }
        checkRep();
        return (this.denominator > 1n) ? (this.numerator + "/" + this.denominator)
                                      : (this.numerator + "");
    }
```

Notice that this toString implementation reassigns the private fields numerator and denominator, mutating the representation – even though it is an observer method on an immutable type! But, crucially, the mutation doesn't change the abstract value. Dividing both numerator and denominator by the same common factor, or multiplying both by -1, has no effect on the result of the abstraction function, AF(numerator, denominator) = numerator/denominator. Another way of thinking about it is that the AF is a many-to-one function, and the rep value has changed to another that still maps to the same abstract value. So the mutation is harmless, or benevolent.

We'll see other examples of benevolent side-effects in future readings. This kind of implementer freedom often permits performance improvements like caching, data structure rebalancing, and lazy cleanup.

Documenting the AF, RI, and safety from rep exposure

It's good practice to document the abstraction function and rep invariant in the class, using comments right where the private fields of the rep are declared. We've been doing that above.

When you describe the rep invariant and abstraction function, you must be precise.

It is not enough for the RI to be a generic statement like "all fields are valid." The job of the rep invariant is to explain precisely what makes the field values valid or not.

As a function, if we take the documented RI and substitute in actual field values, legal or not, we should obtain a boolean result. Note that if we're tempted to refer to the abstract value in our RI, we've put the cart before the horse: illegal rep values don't have an abstract value, and we need to reject them in the RI by talking about the rep values alone.

It is not enough for the AF to offer a generic explanation like "represents a set of characters." The job of the abstraction function is to define precisely how the concrete field values are interpreted.

As a function, if we take the documented AF and substitute in actual legal field values, we should obtain out a complete description of the single abstract value they represent. For example, for the CharSet abstraction function AF(s) = {s[i] | 0 <= i < s.length}, we can substitute a specific (legal) rep value like s="abbc", and get:

AF("abbc") = { "abbc"[i] | 0 <= i < "abbc".length } = { 'a', 'b', 'c' }.

But if we instead had a vague and ill-defined abstraction function like AF(s) = a set of characters, then substituting for s would not affect the righthand side at all. This definition does not say precisely which abstract set of characters corresponds to the rep value s="abbc".

A precise statement can be made either symbolically (like {s[i] | 0 <= i < s.length}) or in English (like “the set of all characters found in s“). You have written both kinds of statements in your math classes, and you should take the same mathematical perspective here. Symbolic expressions may seem more precise, and English may seem fuzzier, but that’s just a matter of perception: both approaches can be done either carefully or sloppily. As a rule of thumb, use symbolic expressions when they can be concise and simple, and use English to convey something complicated, but in both cases, aim to be precise.

Another piece of documentation that 6.102 asks you to write is a rep exposure safety argument. This is a comment that examines each part of the rep, looks at the code that handles that part of the rep (particularly with respect to parameters and return values from clients, because that is where rep exposure occurs), and presents a reason why the code doesn’t expose the rep.

Here’s an example of Tweet with its rep invariant, abstraction function, and safety from rep exposure fully documented:

```
// Immutable type representing a tweet.
class Tweet {

  private readonly author: string;
  private readonly text: string;
  private readonly timestamp: Date;

  // Rep invariant:
  //   author is a Twitter username (a nonempty string of letters, digits, underscores)
  //   text.length <= 280
  // Abstraction function:
  //   AF(author, text, timestamp) = a tweet posted by author, with content text,
  //                               at time timestamp
  // Safety from rep exposure:
  //   All fields are private;
  //   author and text are Strings, so are guaranteed immutable;
  //   timestamp is a mutable Date, so Tweet() constructor and getTimestamp()
  //       make defensive copies to avoid sharing the rep's Date object with clients.

  // Operations (specs and method bodies omitted to save space)
  public constructor(author: string, text: string, timestamp: Date) { ... }
  public getAuthor(): string { ... }
  public getText(): string { ... }
  public getTimestamp(): Date { ... }
}
```

Notice that we don't have any explicit rep invariant conditions on timestamp (aside from the conventional assumption that timestamp is not null, which we have for all object references). But we still need to include timestamp in the rep exposure safety argument, because the immutability property of the whole type depends on the abstract value remaining unchanged, and the abstraction function uses timestamp to map to the abstract value. Not only does the argument include what TypeScript can statically check (private fields and types), but it also references what the implementer actively does to enforce the rep invariant.

Compare the argument above with an example of a broken argument involving mutable Date objects.

Here are the arguments for RatNum.

```
// Immutable type representing a rational number.
class RatNum {
    private readonly numerator: bigint;
    private readonly denominator: bigint;
    // Rep invariant:
    //   denominator > 0
    //   numerator/denominator is in reduced form, i.e. gcd(|numerator|,denominator) =
1
    // Abstraction function:
    //   AF(numerator, denominator) = numerator/denominator
    // Safety from rep exposure:
    //   All fields are private, and all types in the rep are immutable.

    // Operations (specs and method bodies omitted to save space)
    public constructor(n: bigint, d: bigint) { ... }
    ...
}
```

Notice that an immutable rep is particularly easy to argue for safety from rep exposure.

You can look at the full code for RatNum.

reading exercises

Arguing against rep exposure

What an ADT specification may talk about

Since we've just discussed where to document the rep invariant and abstraction function, now is a good moment to update our notion of what a specification may talk about.

The specification of an abstract type T – which consists of the specs of its operations – should only talk about things that are visible to the client. This includes parameters, return values, and exceptions thrown by its operations. Whenever the spec needs to refer to a value of type T, it should describe the value as an abstract value, i.e. mathematical values in the abstract space A.

The spec should not talk about details of the representation, or elements of the rep space R. It should consider the rep itself (the private fields) invisible to the client, just as method bodies and their local variables are considered invisible. That's why we write the rep invariant and abstraction function as ordinary comments within the body of the class, rather than TypeDoc comments above the class. Writing them as TypeDoc comments would commit to them as public parts of the type's specification, which would interfere with rep independence and information hiding.

ADT invariants replace preconditions

Now let's bring a lot of pieces together. An enormous advantage of a well-designed abstract data type is that it encapsulates and enforces properties that we would otherwise have to stipulate in a precondition. For example, instead of a spec like this, with an elaborate precondition:

```
/**
 * @param set1 is a sorted set of characters with no repeats
 * @param set2 is likewise
 * @returns characters that appear in one set but not the other,
 *  in sorted order with no repeats
 */
static exclusiveOr(set1: string, set2: string): string;
```

We might instead use an ADT that captures the desired property:

```
/**
 * @returns characters that appear in one set but not the other
 */
static exclusiveOr(set1: SortedCharSet, set2: SortedCharSet): SortedCharSet;
```

This hits all our targets:

it's safer from bugs, because the required condition (sorted with no repeats) can be enforced in exactly one place, the SortedCharSet type, and because static checking comes into play, preventing values that don't satisfy this condition from being used at all, with an error at compile-time.

it’s easier to understand, because it’s much simpler, and the name SortedCharSet conveys what the programmer needs to know.

it’s more ready for change, because the representation of SortedCharSet can now be changed without changing exclusiveOr or any of its clients.

Many of the places where we used preconditions earlier in this course would have benefited from a custom ADT instead.

reading exercises

Encapsulating preconditions in ADTs

How to establish invariants

An invariant is a property that is true for the entire program – which in the case of an invariant about an object, reduces to the entire lifetime of the object.

To make an invariant hold, we need to:

make the invariant true in the initial state of the object; and

ensure that all changes to the object keep the invariant true.

Translating this in terms of the types of ADT operations, this means:

creators and producers must establish the invariant for new object instances; and

mutators, observers, and producers must preserve the invariant for existing object instances.

The risk of rep exposure makes the situation more complicated. If the rep is exposed, then the object might be changed anywhere in the program, not just in the ADT’s operations, and we can’t guarantee that the invariant still holds after those arbitrary changes.

So the full rule for proving invariants is as follows:

If an invariant of an abstract data type is established by creators and producers;

preserved by mutators, observers, and producers; and

no representation exposure occurs,

then the invariant is true of all instances of the abstract data type.

This rule applies structural induction over the abstract data type.

reading exercises

Structural induction

Recipes for programming

Abstract data types are at the heart of software construction for correctness, clarity, and changeability.

Recall the test-first programming approach for:

Writing a method

Spec. Write the spec, including the method signature (name, argument types, return types, exceptions), and the precondition and the postcondition as a documentation comment.

Test. Create systematic test cases and put them in test file so you can run them automatically.

You may have to go back and change your spec when you start to write test cases. Just the process of writing test cases puts pressure on your spec, because you're thinking about how a client would call the method. So steps 1 and 2 iterate until you've got a better spec and some good test cases.

Make sure at least some of your tests are failing at first. A test suite that passes all tests even when you haven't implemented the method is not a good test suite for finding bugs.

Implement. Write the body of the method. You're done when the tests are all passing.

Implementing the method puts pressure on both the tests and the specs, and you may find bugs that you have to go back and fix. So finishing the method may require changing the implementation, the tests, and the specs, and bouncing back and forth among them.

Let’s broaden this to a recipe for:

Writing an abstract data type

Spec. Write specs for the operations of the data type, including method signatures, preconditions, and postconditions.

This includes defining the abstract value space, but not the abstraction function (since we don’t have a rep yet).

Test. Write test cases for the ADT’s operations.

Again, this puts pressure on the spec. You may discover that you need operations you hadn’t anticipated, so you’ll have to add them to the spec.

Implement. For an ADT, this part expands to:

Choose rep. Write down the private fields of a class (or, as we’ll see in future readings, choose and define a different kind of rep). Write down the rep invariant and abstraction function as a comment.

Assert rep invariant. Implement a checkRep() method that enforces the rep invariant. This is critically important if the rep invariant is nontrivial, because it will catch bugs much earlier.

Implement operations. Write the method bodies of the operations, making sure to call checkRep() in them. You’re done when the tests are all passing.

And let’s broaden it further to a recipe for:

Writing a program

(consisting of ADTs and functions)

Choose data types. Decide which ones will be mutable and which immutable.

Choose functions. Write your top-level main function and break it down into smaller steps.

Spec. Spec out the ADTs and functions. Keep the ADT operations simple and few at first. Only add complex operations as you need them.

Test. Write test cases for each unit (ADT or function).

Implement simply first. Choose simple, brute-force representations. The point here is to put pressure on the specs and the tests, and try to pull your whole program together as soon as possible. Make the whole program work correctly first. Skip the advanced features for now. Skip performance optimization. Skip corner cases. Keep a to-do list of what you have to revisit.

Iterate. Now that it’s all working, make it work better. Reimplement, optimize, redesign if necessary.

Summary

An invariant is a property that is always true of an ADT object instance, for the lifetime of the object.

A good ADT preserves its own invariants. Invariants must be established by creators and producers, and preserved by observers and mutators.

The rep invariant specifies legal values of the representation, and should be checked at runtime with checkRep().

The abstraction function maps a concrete representation to the abstract value it represents.

The rep of an immutable ADT may change, as long as the abstract value it represents doesn’t change.

Representation exposure threatens both representation independence and invariant preservation.

An invariant should be documented, by comments or even better by assertions like checkRep(), otherwise the invariant is not safe from bugs.

The topics of today’s reading connect to our three properties of good software as follows:

Safe from bugs. A good ADT preserves its own invariants, so that those invariants are less vulnerable to bugs in the ADT’s clients, and violations of the invariants can be more easily isolated within the implementation of the ADT itself. Stating the rep invariant explicitly, and checking it at runtime with checkRep(), catches misunderstandings and bugs earlier, rather than continuing on with a corrupt data structure.

Easy to understand. Rep invariants and abstraction functions explicate the meaning of a data type’s representation, and how it relates to its abstraction.

Ready for change. Abstract data types separate the abstraction from the concrete representation, which makes it possible to change the representation without having to change client code.