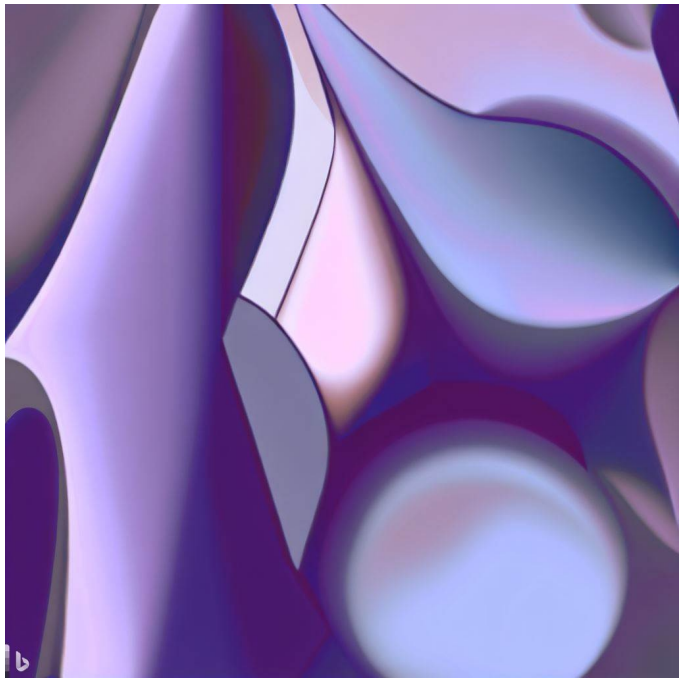


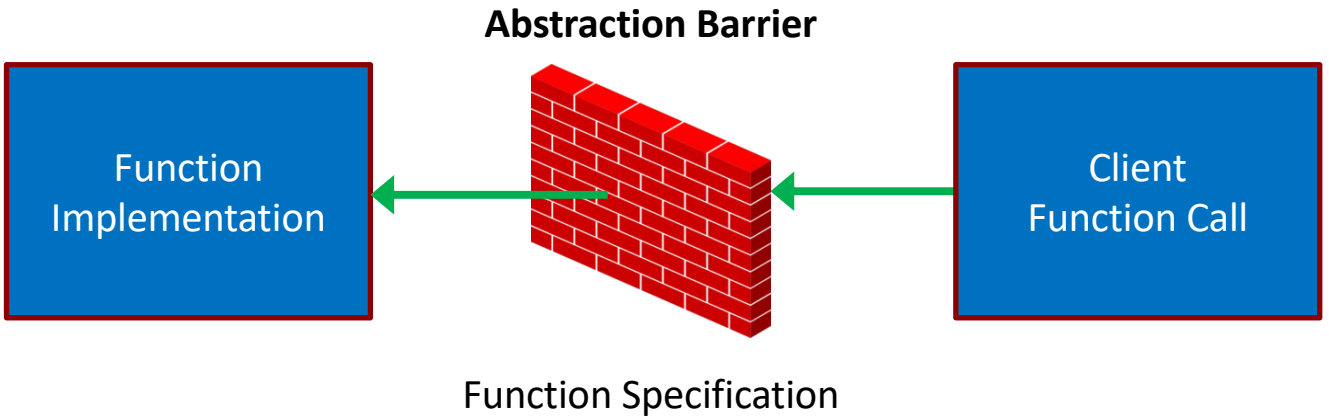
CSE 331

Data Abstraction

Kevin Zatloukal

Abstraction Barrier

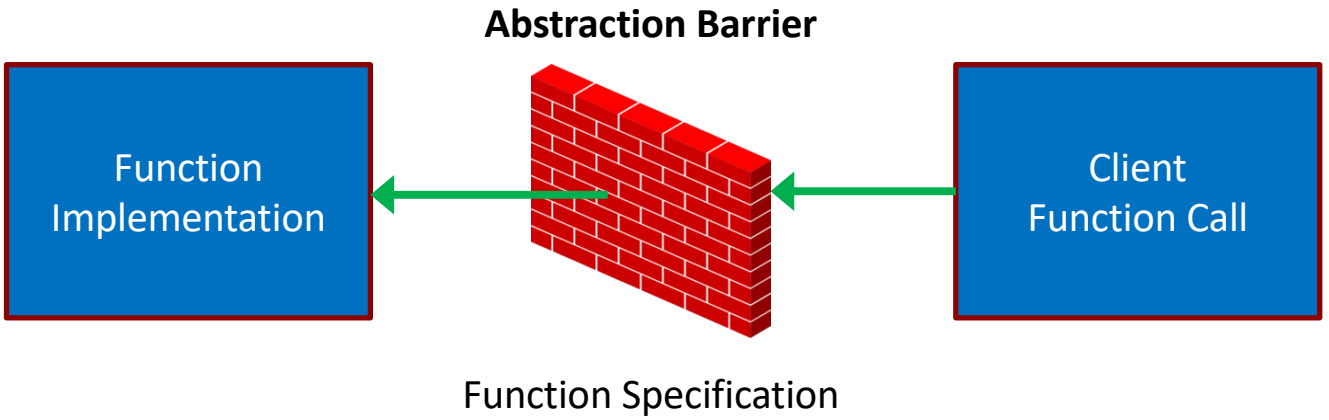
- Last time, we saw *procedural* abstraction



  - specification is the "barrier" between the sides
  - clients depend only on the spec
  - implementer can write any code that satisfies the spec

Abstraction Barrier

- Last time, we saw *procedural* abstraction



- Specifications improve
  - understandability (client)
  - changeability (implementation)
  - modularity

correctness is impossible without specifications

Performance Improvements

- Last time, we saw rev-acc, which is faster than rev
  - faster *algorithm* for reversing a list
  - rare to see this
- Most perf improvements change ***data structures***
  - different kind of abstraction barrier for data
- Let's see an example...

Data Abstraction

Last Element of a List

$$
\begin{array}{lll}
\textbf{func}\ \text{last(nil)} & := \text{undefined} & \\
\quad \text{last(cons(x, nil))} & := \text{x} & \text{for any x} : \mathbb{Z} \\
\quad \text{last(cons(x, cons(y, L))} & := \text{last(cons(y, L))} & \text{for any x, y} : \mathbb{Z} \text{ and any L : List}
\end{array}
$$

- Runs in $\Theta(n)$ time
  - walks down to the end of the list
  - no faster way to do this on a list

- We could cache the last element
  - new data type just dropped:

```
type FastLastList = {list: List, last: number|undefined}
```

empty list has undefined last

Fast-Last List

```
type FastLastList = {list: List, last: number|undefined}
```

- How do we switch to this type?
  - change every `List` into `FastLastList`
- Will still have functions that operate on List
  - e.g., len, sum, concat, rev
- Suppose `F` is a `FastLastList`
  - instead of calling `rev(F)`, we have call `rev(F.list)`
  - cleaner to introduce a helper function

Fast-Last List

```
type FastLastList = {list: List, last: number|undefined}

function toList(F: FastLastList): List<number> {
  return F.list;
}
```

- How do we switch to this type?
  - change every `List` into `FastLastList`
  - replace `F` with `toList(F)` where a `List` is expected
- What happens if we need to change it again?
  - do it all over again!

Another Fast List

- Suppose we often need the 2nd to last item and the 3rd to last, etc. How can we make it faster?
  - store the list in ***reverse*** order!

```
type FastList = List<number>;

function getLast(F: FastList): number|undefined {
  return (F === nil) ? undefined : F.hd;
}

function getSecondToLast(F: FastList): number|undefined {
  return (F === nil) ? undefined :
         (F.tl === nil) ? undefined : F.tl.hd;
}

function toList(F: FastList): List<number> {
  return rev(F);
}
```

Another Fast List

```
type FastList = List<number>;

function getLast(F: FastList): number|undefined {
  return (F === nil) ? undefined : F.hd;
}

function toList(F: FastList): List<number> {
  return rev(F);
}
```

- Problems with this solution...
  - no type errors if someone forgets to call `toList`!

```
const F: FastList = …;
return concat(F, cons(1, nil));  // bad!
```

Another Fast List — Take Two

```
type FastList = {list: List<number>};

function getLast(F: FastList): number|undefined {
  return (F.list === nil) ? undefined : F.list.hd;
}

function toList(F: FastList): List<number> {
  return rev(F.list);
}
```

- Problems with this solution...
  - no type errors if someone grabs the field directly

```
const F: FastList = …;
return concat(F.list, cons(1, nil));  // bad!
```

Another Fast List — Take Three

```
const F: FastList = …;
return concat(F.list, cons(1, nil));  // bad!
```

- Only way to completely stop this is to hide `F.list`
  - do not give them the data, just the functions

```
type FastList = {
  getLast(): number|undefined,
  toList(): List<number>
};
```

  - the only way to get the list is to call `F.toList()`
  - seems weird… but we can make it look familiar

Another Fast List — Take Three

```
interface FastList {
  getLast(): number|undefined;
  toList(): List<number>;
};
```

- In TypeScript, “interface” is synonym for “record type”
- You’ve seen this in Java

```
interface FastList {
  int getLast() throws EmptyList;
  List<Integer> toList();
}
```

Java interface is a record where field values are functions (methods)

Data Abstraction

- Give clients only operations, not data
  - operations are “public”, data is “private
- We call this an Abstract Data Type (ADT)
  - invented by Barbara Liskov in the 1970s
  - fundamental concept in computer science

    built into Java, JavaScript, etc.
  - data abstraction via procedural abstraction

- Critical for the properties we want
  - easier to change data structure
  - easier to understand (hides details)
  - more modular

How to Make a `FastList` — Attempt One

```
function makeFastList(list: List<number>): FastList {
  const last = last(list);
  return {
    getLast: () => { return last; },
    toList: () => { return list; }
  };
}
```

- Values in `getLast` and `toList` fields are functions
  - "=>" syntax is an expression that produces a function
- There is a cleaner way to do this
  - will also look more familiar

How to Make a `FastList`

```
class FastLastListImpl implements FastList {
  last: number|undefined;  // should be "readonly"
  list: List<number>;

  constructor(list: List<number>) {
    this.last = last(list);
    this.list = list;
  }

  getLast = () => { return this.last; }
  toList = () => { return this.list; }
}
```

- Can create a new record using "new"
  - each record has fields `list`, `last`, `getLast`, `toList`
  - bodies of functions use "`this`" to refer to the record

How to Make a `FastList`

```
class FastLastListImpl implements FastList {
  last: number|undefined;  // should be "readonly"
  list: List<number>;

  constructor(list: List<number>) {
    this.last = last(list);
    this.list = list;
  }

  getLast = () => { return this.last; }
  toList = () => { return this.list; }
}
```

- **Can create an instance using "`new`"**
  - **all four assignments are executed on each call to "`new`"**
  - `getLast` **and** `toList` **are always the same functions**

How to Make a `FastList`

```
class FastLastListImpl implements FastList {
  last: number|undefined;  // should be "readonly"
  list: List<number>;

  constructor(list: List<number>) {
    this.last = last(list);
    this.list = list;
  }

  getLast = () => { return this.last; }
  toList = () => { return this.list; }
}
```

- Implements the **`FastList`** interface
  - i.e., it has the expected `getLast` and `toList` fields
  - (okay for records to have more fields than required)

Another Way to Make a `FastList`

```
class FastListImpl implements FastList {
  list: List<number>;  // stored in reverse order

  constructor(list: List<number>) {
    this.list = rev(list);
  }

  getLast = () => {
    return (this.list === nil) ?
        undefined : this.list.hd;
  };

  toList = () => { return rev(this.list); }
}
```

- **Might be better if we had more operations**
  - `secondToLast`, `thirdToLast`, **etc.**, `rev` **(no op)**

How Do Clients Get a `FastList`

```
function makeFastList(list: List<number>): FastList {
  return new FastLastListImpl(list);
}
```

- **Export only `FastList` and `makeFastList`**
  - **completely hides the data representation from clients**

- **This is called a "factory function"**
  - **another design pattern**
  - **can change implementations easily in the future**

    becomes `FastListImpl` with a one-line change

- **Difficult to add to the list with this interface**
  - **requires three calls: `toList`, `cons`, `makeFastList`**

Another Way To Do It

```
interface FastList {
  cons(x: number): FastList;
  getLast(): number|undefined;
  toList(): List<number>;
};

function makeFastList(): FastList {
  return new FastListImpl(nil);
}
```

- **New method `cons` returns list with `x` in front**
  - **now, we only need to make an empty `FastList`**
    anything else can be built via `cons`
  - **example of a "producer" method (others are "observers")**
    produces a new list for you

Specifications for ADTs

Specifications for ADTs

- **Run into problems when we try to write full specs**
  - **for example, what goes after `@return`?**

    don't want to say returns the `.list` field (or reverse of that)

    we want to hide those details from clients

```
interface FastList {
  /**
   * Returns the "underlying" list of items
   * @return ??
   */
  toList(): List<number>
};
```

- **Need some terminology to clear up confusion**

ADT Terminology

New terminology for specifying ADTs

Concrete State / Representation (Code)

actual fields of the record and the data stored in them

Last example: `{list: List, last: number|undefined}`

Abstract State / Representation (Math)

how clients should *think* about the object

Last example: List (i.e., nil or cons)

- **We've had different abstract and concrete types all along!**
  - **in our math, List is an inductive type (abstract)**
  - **in our code, `List` is a string or a record (concrete)**

List Is (Almost) an ADT

New terminology for specifying ADTs

Concrete State / Representation (Code)

actual fields of the record and the data stored in them

Last example: `"nil" | {kind: "cons", hd: number, tl: List}`

Abstract State / Representation (Math)

how clients should *think* about the object

Last example: List (i.e., nil or cons)

- **Doesn't *precisely* follow the design pattern we will use for ADT**
  - **"`cons`" is a function rather than a method**
  - **fields "`hd`" and "`tl`" are accessed by clients**

ADT Terminology

New terminology for specifying ADTs

Concrete State / Representation (Code)

actual fields of the record and the data stored in them

Last example: `{list: List, last: number|undefined}`

Abstract State / Representation (Math)

how clients should *think* about the object

Last example: List (i.e., nil or cons)

- Term "object" (or "obj") will refer to abstract state
  - "object" means mathematical object
  - "obj" is the mathematical value that the record represents

Specifying FastList

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
export interface FastList {
  /**
   * Returns the last element of the list (O(1) time).
   * @returns last(obj)
   */
  getLast(): number | undefined;
```

- "obj" **refers to the abstract state (the list, in this case)**
  - **actual state will be a record with fields** `last` **and** `list`

Specifying FastList

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
export interface FastList {
  …
  /**
   * Returns the object as a regular list of items.
   * @returns obj
   */
 toList(): List<number>
```

- In math, this function does nothing ("`@returns obj`")
  - two concrete representations of the same math idea
  - details of the representations are ***hidden*** from clients

Specifying FastList

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
export interface FastList {
  …
  /**
   * Returns a new list with x in front of this list.
   * @returns cons(x, obj)
   */
  cons(x: number): FastList;
```

- **Producer method: makes a new list for you**
  - **"obj" above is a list, so** $\mathrm{cons}(\mathrm{x}, \mathrm{obj})$ **makes sense in math**

Specifying FastList

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
export interface FastList {
  …
  /**
   * Returns a new list with x in front of this list.
   * @returns cons(x, obj)
   */
  cons(x: number): FastList
```

- **Specification does not talk about fields, just "obj"**
  - **fields are *hidden* from clients**

Documenting the ADT Implementation

Documenting an ADT Implementation

- We also need to document the ADT implementation
  - for this, we need two new tools

Abstraction Function

defines what abstract state the field values currently represent

- Maps the field values to the object they represent
  - output is math, so this is a *mathematical* function

    there is no such function in the code

    this is a tool for reasoning
  - will usually write this as an *equation*

    obj = ... **right-hand side uses the fields**

Documenting the FastList ADT

```
class FastListImpl implements FastList {
  // AF: obj = this.list
  readonly last: number | undefined;
  readonly list: List<number>;
  …
}
```

- **Abstraction Function (AF) gives the abstract state**
  - obj **= abstract state**
  - this **= concrete state**

    "this" is the record, which has fields last and list
  - **AF relates abstract state to the current concrete state**

    okay that "last" is not involved here
  - **specifications only talk about "obj", not "this"**