6.170 L1

- Attend lectures and recitations
- Complete readings in advance
- Do problem sets weekly

R1

Specification

i.e. static int findMinIndex(int[] a)

Requires a ≠ null, perhaps also that a.length>0
(or don't require that and throw IllegalArgumentException)

Modifies: nothing

Returns i s.t. 0≤i<a.length and for all j | 0≤j<a.length => a[i] ≤ a[j]

—

double sqrt(double x)

A
@requires x>=0
@return y such that |y^2-x|<=1

B
@return y such that |y^2-x|<=1
@throws IllegalArgumentException if x<0

C
@requires x>=0
@return y such that |y^2-x|<=0.1

B is stronger than A as it requires less
A is weaker than C because promises less
B and C are incompatible

A weaker specification isn't always worse, sometimes it's needed

L2

Vector is like an array, but it can grow and shrink dynamically

```
Vector v = new Vector();
v.lastElement()
↑ receiver
String a = "zeep"
a.toUpperCase()      ← returns a new string object
v.add(a)             ← mutates the receiver
Vector k = v;
(print) k.lastElement()  ← prints "zeep" because k is an alias of v (point to the same vector object)
```

aliasing adds complexity, we check reference equality by using ==

Garbage collector recycles objects only if they are not referenced

```
String a="zeeb"
String b="zeeb"
```

same object, as Java only allocates one object if it can tell that two strings are the same.

a.equals(b) and b.equals(a) can produce different results when one of 'em is null — You cannot rely on == for strings, you better use String.equals().

String a = null → value can be taken by any object as it points to nothing

a.toUpperCase(), throws NullPointerException

the receiver of a method can never be null cause it has to be some object.

It is good practice to avoid creating nulls on the first place

For static methods (no objects) like println it works fine when we use null reference as it has no receiver.

L2 ...

```
class Trans {
    int amount;
    Date date;
}
```

fields / instance vars

int don't hold references but integers themselves

```
int i = 5;
Integer obj_i = new Integer(i);
i = obj_i.intValue();
```

to object ... and back

```
Trans t = new Trans()
t.amount = 20;
t.Date = new Date();
```

creates obj with default values (0 for int and null for Date)

setter!

just for setting the actual date

✗ BAD!

But that is not what we would want for a transaction.

We'd like to do that inside of the class, as the program grow bigger, you will want as much modularity you can get.

```
Trans(int a, Date d) { amount = a; date = d; }
```

You could also add checks that way.

```
Trans t = new Trans(20, new Date());
```

L3 the keyword extends indicates that the class extends another one by adding new behaviors.

```
AccountPlus extends Account
```

AccountPlus ← superclass of AccountPlus

Subclass of Account

AccountPlus must have its own constructor but all other methods and fields are implicitly already in.

If we need a different method body, we use override override that one. which method is called depends on the object type that called it (dynamically resolved).

runtime object allocation (fields are in compile time)

```
AccountPlus = new AccountPlus() is same as
Account = new AccountPlus()
```

Suppose we want to handle a collection of accounts, we can have a Bank implementation

```
class Bank { Account[] Accounts; void MonthlyFee() }
```

arrays, like Vector, but cannot grow and shrink

this code is polymorphic since monthlyFee works with both Account and AccountPlus.

The array can hold both Account and AccountPlus objects.

Arrays don't connect to program well, however vectors are not part of the language itself and there no build syntax (call elementAt(i)). Also we can't say a Vector is a vector of some type.

instead, we can downcast calls to 'elementAt(i) like that (Account) acc.elemAt(i);

Vectors returns an object type, so we could use if we're evaluating Account or AccountPlus method calls to checkTrans and post will be made to objects without these methods

That kind of code will be rejected by the compiler! (Java is a safe language)

downcasting happens in runtime, it
checks if the object belongs to that
class or subclasses of it does,
execution continues normally, else
exception. These are different from
Typecasts! Downcast is a test!

—

Not every type is a class. Interfaces
for example are not executable code.
It is just a collection of method signatures.
A class that satisfies that specification is
said to implement it.
Interfaces can be part of a type hierarchy,
but they are never the runtime type of an object
~~object~~

Java is statically typed

L4 Specifications in software to catalog reusable components,
like they do in the Java collection framework.
And to ~~regulate~~ regulate connections between
modules in a design, how each module every
~~module~~ other.
They are the design of the software

consider these:

FindA (j)
for i in a
if i == j return i
else a.length

FindB (j)
for i in a
if i == j return i
else -1
return

They behave the same, except for when the
value is not found.
To know if this is acceptable, we need
a specification ~~that~~ states what the
client depends on.

requires val occurs in a
effects returns result such
that a[result] == val.

A spec consists of: precondition (requires), postcondition (effects)
frame condition (modifies).
precondition, state in which the method is called
postcondition, given that precondition holds, the method
~~must~~ must obey the postcondition by
by throwing exceptions, returning values, modifying objects.

frame condition, allows more succinct specification, saying
the reachable objects may or may not change
those ~~objects~~ objects are mutable. It is an
assertion for the objects not mentioned is
that ~~it~~ not change

omitted clauses say, if you omit pre, default value is
true (no obligation for the caller), the method is said
to be "total". Else, if not true, it's "partial".
omitting frame condition means that does not modify
anything.
You never omit postcondition

—

modifies, (name of vars modified)
we'll specify in "postcondition" how they're
modified
precondition says that the behaviour is not
constrained if called with ~~null~~ null arguments

requires v != null and ! this
modifies this
effects add elem to v, returns true if this changed

—

Specs can be operational and declarative. Operational
gives the steps the methods perform (pseudocode
for example), declarative is what we've seen thus far.

If we want to substitute a method A with B, we
can check the spec is as strong as A.
It is if the precondition of A isn't stronger than B's
and if A's postcondition isn't weaker than B's

5.9 on good specs. /lecture-notes/lec4.pdf

L5

Testing proves the presence of errors, not the absence you can use it to increase confidence in software

Prove correctness, prove that the specs are met.

Run on all possible inputs, if we check every possible input, then the program is correct. Checking output is done by using assertions. Desired outputs are generated by hand or by simpler implementations already verified.
This is infeasible for the large space of inputs

Run the program on some inputs, it doesn't guarantee correctness. However if we carefully select testcases, this will give high confidence in the program working on all inputs.
The idea is to partition the space of all possible inputs in groups such that the program behaves the same for each member of each group
We must find the groups, that's the hard part, we can guess the groups and then add more test cases to guarantee that those members are part of the same group.

Three ways of heuristics are:
coverage, the key only aims to make 1 test case for some to execute each code or specification construct.
Complete coverage is wholly impossible.
There are different types of coverage we want to check

Boundary cases, when drawing equivalence boundaries, even though the general idea is correct, boundaries may get different. Test minimal, maximal with typical "center" values.

Duplicates, programs often behave differently for inputs with relationships to one another. Aliasing, or multiple references to the same object is the problem.

These are the heuristics, usable in black-box and clear-box testing.

Black box test cases are generated by examining the specs. The test are testing the program's functionality, but not it's implementation.
They are representation-independent and can be re-used even with a new implementation.

Clear box cases are generated by looking at the implementation and they are chosen by considering the code structure.
They are good with special cases that are localized differently by the code or with optimizations.

L7

A type is defined by the operations you can make with it
You may want your types to be immutable, sometimes mutable types are needed in your ADT.
operations in ADTs are creators (t → T), producers (T,t → T), mutators (T,t → void), observers (T,t → t)

T → ADT
t = other type

creates objects

take objs of the ADT and return a diff type (like size())

(T,t → T) ↓ make new objs from old objs

Iterator is an object that provides methods for iterating over a collection