~~Static typing~~

Advanced Software Construction in Java

course goals ↓

Good code is
- safe from bugs (or correct)
- easy to understand
- ready to be changed in the future

↑ There are other important software properties but those generally are put foremost!

Static typing and checking

Hailstone sequence:
- start with n
- if n is even, the next number is n/2, else 3n+1
- the sequence ends when it reaches 1

Static typing

Types of the variables have to be known before the program runs. The compiler can then deduce the types of all expressions as well

dynamic typing

this type of checking is deferred until runtime

Static typing is a type of static checking, which means checking for bugs at compile time.

Static checking prevents:
- syntax errors (extra punctuation, spurious words
- wrong names (Math.sine(3) instead of Math.sin(3))
- wrong number of arguments
- wrong argument types
- wrong return types

dynamic checking prevents:
- illegal argument values (x/y when y is 0)
- unrepresentable return values (the returned value cannot be represented in the type)
- out-of-range indexes
- calling a method on null

Static checking is about the types, independently from the specific values

dynamic checking is about errors caused by specific values

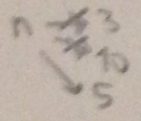
Primitive numeric types have corner cases that do not behave like integers and real numbers. As a result some errors that should be dynamically checked are not checked at all

- Integer division, 5/2 returns a truncated integer
- Integer overflow, when you reach a type capacity, the computation quietly overflows (wraps around) and returns an integer within the type capacity, but not the right answer
- Special values in floating point types, they are NaN, POSITIVE_INFINITY and NEGATIVE_INFINITY.

When you apply certain operations on a double such as divide by zero, or square root of a negative number, you get one of those values instead of getting a dynamic error. If you keep computing it, you'll end up with a bad answer!

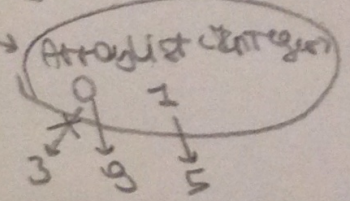
When you assign to a variable, you're changing what the variable's arrow points. You can point it to a different value

```
int n=3;
n= 3*n+1
n= 2/n
```



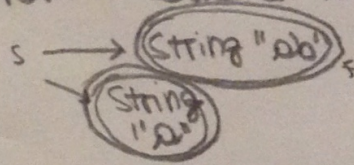
When you change the contents of a mutable value, such as an array or list, you're changing references inside the value
This is called mutating the value

```
List<Integer> list = new ArrayList<Integer>();
list.add(3);
list.add(5);
list.set(0,9);
```



For example if we have a String variable s, we can reassign it from a value of "a" to "ab"

```
String s = "a"
s += "b";
```

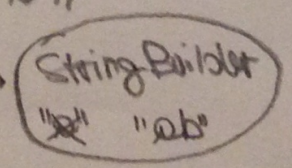


double border means immutable

is an immutable type, it can never change once created

By contrast StringBuilder is a mutable object that represents a string of characters. It has methods that change the value of the object

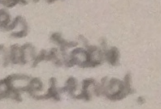
```
StringBuilder sb = new StringBuilder("a");
sb.append("b");
```



Immutability makes our code safer from bugs!

Immutable references
... are variables that are assigned but never reassigned. To make a reference immutable, we declare it Final

Final int n=5



The compiler will throw an error if it isn't sure your final variable is only assigned once

Notice that we can have an immutable reference to a mutable variable. Final StringBuilder sb

the StringBuilder value can change, but sb's variable's arrow cannot change

Also a mutable reference to an immutable variable
String s ← s can change as it can be repointed to a different object

Using Final is good practice.
Use it where possible
- statically checked by compiler
- useful to the reader of the code

Documenting assumptions

Writing down the type of a variable is a form of documentation on assumption about it

Declaring a variable final is also a kind of documentation

There are assumptions that javac doesn't check automatically

```
/**
 ...
 * @param n ... Assumes n > 0
 */
```

Our program is full of assumptions, others might have to guess those and that leads to inconsistency and bugs

Write your program for
- communicating with the computer
- communicating with other people (making it easy to understand, so that when somebody has to fix it, improve it or adapt it, they can do it without introducing bugs

Why Java?

Safety, Java has static checking (primarily type checking, but other kinds of static checks too)

Ubiquity, widely used, runs on many platforms

A programmer must be multilingual, programming languages are tools and you must use the right tool for the job

Java has a wide array of libraries! (as a result of its ubiquity)

However Java is words, has many features, it's inconsistent (final for example means different things in different contexts).

It's weighted on the baggage of C/C++ (primitive types or switch) and it has no interpreter!

Code Review

... is careful, systematic study of source code by people who are not the original author of the code

- Improve the code, Find bugs, clarity and checking for consistency with style standards
- Improving the programmer, programmer can learn and teach each other

Code review is widely practiced in industry & open source projects

Style standards

Most companies and large projects have style standards. These can get pretty detailed to the point of specifying whitespace (how deep indent) and where curly braces and parenthesis [redacted] should go.

Follow the conventions of the project you're working on. [redacted] IF you're the programmer that reformats every module you touch with your [redacted] personal style, your [redacted] teammates will hate you [redacted], and rightly so. Be a team player

But there are some rules that are rules that are quite sensible and target our big three properties in a stronger way than placing curly braces

- DRY, DON'T REPEAT YOURSELF
- Comments when needed
- Fail fast
- Avoid magic numbers

...

- One purpose for each variable
- Use good names
- No global variables
- Return results, don't print them
- Use whitespace for [redacted] readability

Don't repeat yourself

Duplicated code can be buggy; some maintainer could fix one place but not the other

Avoid duplication like [redacted] you'd avoid crossing the street without looking

=

commenting

Good comments

- make code easier to understand
- Safer from bugs
- Ready for change

spec appears the method/class and documents the behavior of [redacted] that method/class

javadoc comment

```
/** specs...
 * @params
 *      ...to on
 */
```

Another use for comments: [redacted] is to document where that code came from

```
// does something
// see http://stackoverflow...
```

[redacted] it helps us avoid violations of copyright or the code can fall out of date

Bad comments:

[redacted] ← unnecessary

- Simply transliterates the code into english
- If there's code that is obscure, [redacted] it's good to say what the code is intended to do

Fail Fast

The code should reveal bugs as early as possible

The earlier we find out there's a bug (closer to the cause in the code) the easier it's going to be to [redacted] Find and Fix

Static checking: as fast as you can fail, find the bug before you even run the code

dynamic checking: is a little bit later

With no checking you may have to do a lot of debugging [redacted] to actually find out the bug

Avoid magic numbers

All numbers and constants that appear by themselves without names

The code doesn't explain what they mean, what they represent and where they came from

Don't compute constants by hand and hardcode them into your code!

One purpose for each variable

Don't reuse parameters and don't reuse variables

You'll confuse your reader with variables that used to mean one thing, suddenly starts meaning something different a few lines down

Method parameters should generally be left unmodified!

Use Final for method parameters, where possible

Use good names

Best names are long and self-descriptive, if you have good names, you can often avoid writing a comment entirely

In general names like tmp, temp, data are awful symptoms of laziness.

Every variable is temporary, and every variable is data, so those names are generally meaningless

[redacted]

- methodsAreNamedWithCamelCaseLikeThis
- variablesAreAlsoCamelCase
- CONSTANTS_ARE_IN_ALL_CAPS_WITH_UNDERSCORES
- packages are lowercase and [redacted] by dots separated

methods are usually verb phrases while variable names are noun phrases

Choose short words, and be concise, but avoid abbreviations

Another thing that makes your code more readable is good use of whitespace

Put spaces between code lines to make them easy to read

Use only spaces when possible, set your editor to put spaces instead of a tabs when pressing tab key. That's because different tools interpret tabs differently. Even worse if you use a mix of spaces and tabs, in another editor where the tab is different, your indentation will be completely screwed up

Use spaces, unless the team you're working on has the exact opposite policy

Don't use global variables

Return, don't print

```
public static int ...
```
↖ No Final, can change
↑ global
↑ not associated with object
← Global variable Really bad!

In general, change global variables into parameters and return values or put them inside objects that you're calling methods on.

If you're printing things out, then the code you're writing isn't ready for change. It isn't ready for reuse in some other context.

It's ok to print debugging information, but that information shouldn't be part of your design. It shouldn't be a part of how the function is supposed to work, it should only be part of how you debug your design

Testing

is an example of a more general process called validation, which is about uncovering problems in your program and hopefully increasing your confidence that is correct

Validation includes

- Formal reasoning, also called verification, where you construct a formal PROOF that a program is correct. Tedious to do by hand and automated tools are still in area of research. Nevertheless small crucial pieces might be formally verified
- Code review, we talked about it
- Testing, running a program on carefully selected inputs and checking that it does the right thing

Even will the very best validation, it's very hard to achieve perfect quality in software

Why software testing is hard?

- Exhaustive testing, is infeasible. The space of possible test cases is generally too big to cover exhaustively
  (a multiplication between floating points needs $2^{32} * 2^{32}$ test cases)
- Haphazard testing, (just try it and see if it works) is less likely to find bugs, unless the program is so buggy that an arbitrary chosen input is more likely to fail than to succeed. Also it doesn't increase our confidence in program correctness.

Random or statistical testing:

Software varies discontinuously and discretely across the space of possible inputs.
The system may seem to work fine across a broad range of inputs, and then abruptly fail at a single boundary point.
The famous pentium division bug affected 1 in 9 billion divisions.
Stack overflows, out of memory errors and numeric overflows, tend to happen abruptly and always in the same way.

Instead, test cases must be chosen carefully and systematically!

Test-First Programming

When testing your goal is to make the program fail. The best kind of test is one that finds a bug

Don't treat your code as a precious thing and test it really lightly just to get the reward of seeing it work ... instead be brutal and beat that program where it might be vulnerable, so that these vulnerabilities can be eliminated!

In test-first programming, you write tests before you even write any code. For a single function:

- Write a specification to the function
- Write tests that exercise the specification
- Write actual code, when your code passes all tests you're done.

Writing tests first is a good way to understand the specification. The specification can be buggy ... incorrect, incomplete, ambiguous, missing corner cases. Trying to write test cases can uncover these problems early, before you've wasted time implementing a buggy spec.

Creating good tests is an interesting design problem. We want our set of test cases to be small enough to run quickly, yet large enough to validate the program

... To do so, we divide the input space in subdomains, which taken together they completely cover the input space so that every input lies in at least one subdomain. Then we choose one test case from each subdomain

The idea behind subdomains is that we want to partition the input space into sets of similar inputs on which the program has similar behaviour

Include boundaries within subdomains, in the partition, which is where bugs usually occur. For example

- 0 is boundary between positive and negative numbers
- The maximum and minum value of numeric types
- Emptyness for collections
- The first and last element of a collection

Programmers often make off-by-one errors (like > instead of >= or initialize a counter to 0 instead of 1). Another is that some boundaries may be places of discontinuity

eg when int overflows it abruptly becomes negative!

After partitioning our input space, we can choose how exhaustive we want the test suite to be

- Full cartesian product, every legal combination of the partition dimensions is covered by one test case
- Cover each part, Every part of each dimension is covered by at least one test case, but not necessarily every combination

Often we strike a compromise between these two extremes!

Blackbox and whitebox testing

Recall a specification is a description of a function's behaviour

- Blackbox testing means that you're choosing those test cases just by looking at the specification. That's what we've been doing: we partitioned and looked for boundaries without looking at the actual code

- Whitebox testing (also called glass box testing) means choosing test cases with knowledge of how the function is actually implemented.

For example if the implementation selects different algorithms depending on the input, then you should partition these domains. If the implementation keeps an internal cache that remembers the answers to the previous inputs, then you should test repeated inputs.

When doing whitebox testing you must be careful that your test cases don't require specific implementation behaviour that isn't actually called for by the spec.

Your testcase should be general to preserve the implementer's freedom

Coverage

One way to judge a test suite is to see how thoroughly it exercises the program. This idea is called coverage

There are three common kinds of coverage (sort of increasing level of exhaustiveness)

statement coverage, is every statement run by some test case?

branch coverage, every if or while statement, are both boolean directions taken by some test case?

path coverage, is every possible combination of branches (path through the program) explored by some test case?

100% statement coverage is a common goal in industry, but hardly achieved due to unreachable defensive code (like "should never get here" assertions)

100% branch coverage is highly desirable. Safety critical code has even more strenuous criteria (MCDC modified decision/condition coverage).

100% path coverage is infeasible, requiring exponential-size test suites to achieve

A standard approach to testing is to add tests until the test suite achieves adequate statement coverage.

Statement coverage is usually measured with a code coverage tool.

Documenting your testing strategy

```
/**
 * Reverses the end of a string
 *
 * For example: whitespace here
 *    reverseEnd("Hello World", 6)
 *    returns "Hello dlroW"
 *
 * with start == 0, reverses the entire text.
 * with start == text.length, reverses nothing
 *
 * @param text non-null string that will
 *             have its end reversed
 *
 * @param start the index at which the
 *              remainder of the
 *              input is reversed,
 *              requires 0 <= start <= text.length()
 *
 * @return input with the substring from
 *         start to the end of the
 *         string reversed
 */
```

→ Document the strategy at the top of the test class

```
/**
 * Testing strategy
 *
 * Partition the input as follows:
 * text.length() : 0, 1, >1
 * start:          0, 1, 1 < start < text.length(),
 *                 text.length()-1, text.length()
 *
 * text.length() - start: 0, 1, even > 1, odd > 1
 *
 * Include even- and odd-length intervals because
 * only odd has a middle element that doesn't
 * move
 *
 * Exhaustive Cartesian coverage of partitions.
 */
```

→ Document how each test case was chosen, including whitebox tests

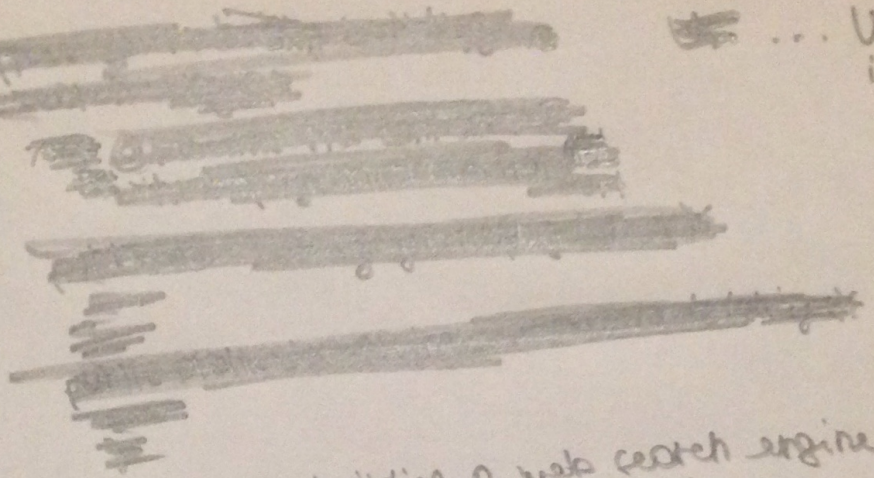
```
// covers text.length() = 0,
//        start = 0 = text.length(),
//        text.length - start = 0
@Test public void testEmpty() {
    assertEquals("", reverseEnd("", 0));
}

...other test cases...
```

Unit testing and stubs

A well tested program will have tests for each individual module, where a module might be a method or a class. A test that tests one individual module is called a unit test. Testing modules in isolation leads to much easier debugging as you can be confident that the bug is in a specific module, when an unit test fails.

The opposite of a unit test is an integration test, which tests a combination of modules, or even the entire program. If all you have are integration tests, when a test fails, you have to hunt for the bug. It might be anywhere in the program. Integration tests are still important as the program can fail at the connection between modules. One cost of later writing on a module was make an assumptions about another module that they're using that aren't correct



Suppose we're building a web search engine. Two of our modules might be getWebPage(), which downloads web pages, and extractWords(), which splits a page into its component words

These methods might be used by another module makeIndex() as a part of the web crawler that makes the search engine's index

In our test suite we would want
- Unit tests for just getWebPage() that test it on various urls
- Unit tests for just extractWords() that test it on various strings
- Unit tests for makeIndex() that test it on various set of urls

One mistake programmers make sometimes is writing test cases for extractWords() in such a way that the test cases depend on getWebPage() to be correct. It's better to think about and test extractWords() in isolation, and partition it.

...

... Using test partitions that involve web page content might be reasonable, because that's how extractWords() is actually used in the program. But don't actually call getWebPage() from the test case because getWebPage() might be buggy! Instead, store web page content as a literal string, and pass it directly to extractWords(). That way you're writing an isolated test and if that fails you can be confident that the bug is in extractWords()

...

...

Note that unit tests for makeIndex() can't be easily isolated that way.
When testing makeIndex(), you're testing the correction of the methods called by makeIndex() as well as makeIndex() itself.
If the test fails, the bug might be found in any of those methods. That's why we want separate tests for getWebPage() and extractWords() to increase our confidence in those modules individually and localize the problem to the makeIndex() code.

Isolating a higher-level module like makeIndex() is possible if we make stub versions of the modules that it calls.
For example getWebPage(), a stub for it would return a mock web page no matter the URL. A stub for a class is often called a mock object!

A good unit test isolates the method it is testing rather than calling other modules of the program. When testing your method you can compute results by hand and see if it matches the output or run another module needed for that result, and copy its output, and you're confident the other module works, but do not call that module from your unit test so it might break in the future and it will be harder to find out where's the bug

Automated testing and regression testing

The combination of these two, automated regression testing, is one of the best practices in software engineering.

Automated testing means running the tests and checking their results automatically. A test driver should not prompt the user for inputs and print out the results for you to manually check, instead it should invoke the module itself on fixed test cases and automatically check that the results are correct

Automated testing frameworks, like JUnit, make it easy to run tests, but you still have to come up with good tests yourself (automatic test generation is still a subject of active research)

Re run your tests whenever you change your code. It prevents the program from regressing! Introducing other bugs when you Fix new bugs / add a feature.
Running tests after every change is called regression testing

Whenever you Fix a bug, take the input that elicited that bug and add it to your automated test suite as a test case. This kind of test case is called a regression test.
"A test case is good if it elicits a bug!"

Saving regression tests also protects against reversions that reintroduce the bug

...

This also leads to test-first debugging: when a bug arises, immediately write a test case for it and add it to your test suite. Once you've fixed the bug and all your tests will be passing, you'll be done debugging, and have a regression test for that bug.

Regression testing can be only practical if the tests can be run often, automatically. Therefore the name automated regression testing!

Specifications!

It's impossible to think about delegating responsibility for implementing a method unless you have a clear specification that acts as a contract. The implementer is responsible for meeting that contract and the client that uses the method can rely on that contract

Many of the nastiest bugs in programs arise because of misunderstandings about behaviour at the interface between the caller and the implementation of the function and the implementation of it

Different programmers on a team might have different specifications in mind. So when the program fails it's hard to figure out which part of code to blame. Writing specs can let you figure out where you should put your fix

Specs are good for the client as well cause they spare the cost of actually reading the code, that might be hard to understand and might require to look at other modules used in it.

Specifications are good for the implementer cause the give freedom to change the implementation without even telling the client, without then changing the client code at all.

Sometimes a weak specification makes it possible to use a much more efficient implementation.

A precondition may rule out certain states in which a method might have been invoked that would have incurred an expensive check that is no longer necessary

[redacted]

Specification structure

specification of a method has two parts:

- There's the precondition, indicated by the keyword requires
- and the postcondition, implicated by the keyword effects

The precondition is an obligation on the client and it's a condition over the state in which the method is invoked.

The postcondition is an obligation on the implementer of the method. It says what has to be true, after the method completes, IF the precondition held when the method started

IF the precondition is not met, the method is not bound by the postcondition

static type declarations are effectively part of the preconditions and postconditions of a method

Javadoc comments. parameters are described in @param clauses and result is described in @return and @throws clauses

Preconditions should be put in @param when possible and postconditions in @return and @throws

```
/**
 * Finds a value in an array
 *
 * @param arr an array to search, requires that
 *                 val actually exists in arr
 * @param val value to search for
 * @return index i such that arr[i] = val
 */
```

In Java, every objects and arrays can also take a special value null, which means that the reference doesn't point to anything

You'll get NullPointerExceptions at runtime because you can't call any methods or use any fields with one of these references

Null values are troublesome and unsafe.

In 6.005 and most good Java programming, we're going to disallow null as a possible parameter and return value

If a method allows null it should be explicitly stated. But that's almost never a good idea.

"Much better to fail fast rather than silently ignoring and propagating null"

What a specification may talk about

It may talk about parameters and return value of the method, but it should never talk about local variables of the method or private fields of the method's class. You should consider the implementation invisible to the reader.

It's important that even glassbox tests must follow the specification. Your implementation may have stronger guarantees than the specification calls for or it may have specific behaviour where the specification is undefined. But your test cases should not count on that behaviour. Test cases must obey the contract, just like any other client.

For example:

```
static int find(int[] arr, int val)
requires: val occurs in arr
effects: returns index i such that arr[i]=val
```

This spec has a strong precondition and a fairly weak postcondition: if val appears more than once, the spec says nothing about which index is returned.

Even if you implemented find so that always returns the lowest index, your test case can't assume that specific behaviour ...

```
int[] array = new int[] {7,7,7};   // bad test case violates the spec
assertEquals(0, find(array,7));
assertEquals(7, array[find(array,7)]);   // correct!
```

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

Glassbox testing means you are trying to find new test cases that exercise different parts of the implementation, but still checking those test cases in an implementation-independent way.

Testing Units

A good unit test is focused on just a single specification. A unit test should only fail if a different method fails to satisfy its spec.

Good integration tests will make sure that our different methods have compatible specifications: callers and implementors of different methods are passing and returning values as the other expects.

Integration tests cannot replace systematically designed unit tests. In our web search example, if we only call extractWords by calling makeIndex, we will only test it on a potentially small partition of its input space (inputs that are possible outputs of getWebPage). In doing so we've left a place for bugs to hide, ready to jump out when we use extractWords for a different purpose or getWebPage starts returning web pages written in a different format.

Specs for mutable methods

Describing side effects, changes to mutable data, in the postconditions

```
static boolean addAll(List<T> list1, List<T> list2)
requires: list1 != list2
effects: modifies list1 by adding the elements of
         list2 to the end of it, and returns true
         if list1 changed as a result of call
```

It's not likely to rule out any useful applications of the method and it makes it easier to implement

more examples ...

```
static void sort(List<String> list)
  requires: nothing
  effects: [redacted]
```

puts list in sorted order,
i.e. list[i] <= list[j] for all 0 <= i < j < list.size()

does not mutate its argument

```
static List<String> toLowerCase(List<String> list)
  requires: nothing
  effects: returns a new list t where t[i] = list[i].toLowerCase()
```

Like null, mutation is disallowed unless stated otherwise!

Exceptions!

Exceptions can signal bugs in your code, things that you don't expect to happen and don't want to happen.
They can also be used to improve the structure of your code that involves procedures with special results

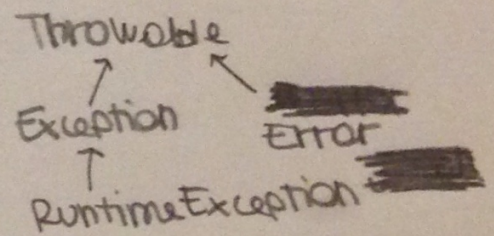
Java libraries are often designed like this: you get -1 when expecting a positive integer or a null reference when expecting an object.
This approach is OK if used sparingly but it's tedious to check the return value and it gets easy to forget to do it! Fail fast!

Exceptions should be used instead of special results. For those results you'll want to use checked exceptions and unchecked, to signal bugs!
exceptions

checked exceptions are checked by the compiler
- If a method might throw a checked exception, the compiler will make sure that we declared that on the method's signature
- If a method calls another method that might throw a checked exception, the compiler is going to make sure that we deal with that possible exception by catching it or by declaring that exception in the calling method's signature

That ensures that exceptions that we expect to occur are handled

Unchecked exceptions do not need to be declared because we don't expect them to happen in a correct program, and do not need a try catch block.
You can still write it, but that has no effect and is not recommended



RuntimeException and Error subclasses are unchecked exception. Although Error is only used by the Java runtime system
Exception and Throwable subclasses are checked exceptions (except those subclasses above & cousins)

Exception design considerations

Exceptions in Java aren't as lightweight as they might be

Aside from performance penalty, exceptions are a pain for the programmer to use in Java

- If you design a method to have its own (new) kind of exception, you have to create a new class for the exception
- If you call a method that can throw a checked exception, you have to wrap it in a try-catch statement (even if you know the exception will never be thrown)

More refined rule:

- Use an unchecked exception for ... or if you expect that clients will usually write code that ensures the exception will not happen, because there is a convenient and inexpensive way to avoid the exception
- Otherwise use a checked exception

eg queue.pop() throws an unchecked exception for an empty queue because it's easy for the caller to avoid this (queue.isEmpty())

URL.getWebPage() throws a checked exception when it can't retrieve a webpage because it's not easy for the caller to prevent this

integerSquareRoot(int x) throws a checked exception when x has no perfect square root because testing whether x is a perfect square is as hard as finding the actual square root, so it's not feasible to expect the caller to prevent it

The cost of using exceptions in Java is the reason why many Java APIs use the null reference as a special value unfortunately

Designing specifications:

We'll look at different specs for similar behaviours and look at three dimensions for comparing them

- How deterministic it is. Does the spec define a single output for every input or does it allow the implementer to choose from legal outputs for any given input
- How declarative it is. Does the spec just characterize what the output should be or does it explicitly say how to compute the output
- How strong it is. Does the spec have a small set of legal implementations, or a large set?

Not all specs are equally useful and we'll explore what makes some specs better than others

...

...

Deterministic vs underdetermined specs

```
static int FindFirst(int[] att, int val) {
    for (int i=0; i<att.length; i++) {
        if (att[i]==val) return i;
    }
    return att.length;
}

static int FindLast(int[] att, int val) {
    for (int i=att.length-1; i>=0; i--) {
        if (att[i]==val) return i;
    }
    return -1;
}
```

(just to distinguish them)

One possible spec for Find:

```
static int FindExactlyOne(int[] att, int val)
requires: val occurs exactly once
effects: returns index i such that att[i]=val
```

This spec is deterministic. For every state that satisfies the precondition, the outcome is completely determined. There are no valid inputs for which there is more than one valid output

Both FindFirst and FindLast satisfy the spec we could use either one

A slightly different spec

```
static int FindOneOrMore, AnyIndex(int[] att, int val)
requires: val occurs in att
effects: returns index i such that att[i]=val
```

This spec is not deterministic. It doesn't say which index is returned if val occurs more than once. This spec allows multiple valid outputs for the same input

Not deterministic is not the same as nondeterministic, as that term is used to describe code that sometimes behaves one way and sometimes another, even with the same inputs

We'll refer to not deterministic specs as underdetermined

So the spec above is underdetermined. A client can't rely on which index will be returned if val appears more than once

Declarative vs operational specs

Operational specs, give a series of steps that the method performs.

Pseudocode descriptions are operational

Declaratis specs don't give details of intermediate steps. Instead they just give properties of the final outcome, and how it's related to the initial state

Almost always, declarative specs are preferable. They're shorter, easier to understand, they don't expose implementation details that a client may rely on (that may no longer hold when the implementation has changed)

One reason for operational specs may be explaining the implementation for a maintainer. DON'T DO THAT You have clients and maintainers, to communicate with the maintainer you mostly want to use comments within the body of the method

For a given spec, there may be many ways to express it declaratively. It's up to you to choose the clearest specification for clients and maintainers

stronger vs weaker specs

Suppose you want to change a method; either its implementation or specs. There are already depending on the current specs, how do you decide whether it's safe to change the spec? or equal to

A spec S2 is stronger than S1 if

- S2's precondition is weaker than or equal to S1's
- S2's postcondition is stronger than or equal to S1's, for the states that satisfy S1's precondition

Then every specification that satisfies S2 can be used to satisfy S1 as well, and it's safe to replace S1 with the stronger spec S2.

so you can always weaken the precondition (make fewer demands) and strengthen the postcondition (make more promises) For example

static int FindExactlyOne(int[] arr, int val)
requires: val occurs exactly one in arr
effects: returns index i such that arr[i]=val

can be replaced with:

static int FindOneOrMore, AnyIndex(int[] arr, int val)
requires: val occurs at least one in arr ← weaker!
effects: returns lowest index i such that arr[i]=val ← stronger!

Designing good specifications

Designing a good method means primarily writing a specification

It should obviously be succint, clear, and well-structured, so that it's easy to read

The content is hard to prescribe as there are no infallible rules, but there are some useful guidelines:

The spec should be coherent

It shouldn't have lots of different cases. Long argument lists, deeply nested if statements, and boolean flags are all signs of trouble

static int sumFind(int[] a, int val)
effects: returns the sum of all indices in array a and b at which val appears

That procedure is NOT well-defined. It's incoherent, since it does several things (finding in two arrays and summing the indices) that are not really related. It would be better to use a different procedure for finding indices, call it twice and sum the result

Separating responsibilities in different methods will make them simpler and more useful in other contexts

...

The result of a call should be informative

```
static V put(Map<K,V> map,
             K key, V val)
```

requires: val may be null, and map may contain null values

effects: inserts (key,val) into the mapping, overriding any existing mapping for key, and returns old value for key, unless none, in which case it returns null

It's impossible to distinguish between the case where the key wasn't found or where the key was found and the value that was in the map was null

We want to make sure our specs are strong enough so that the client can do something useful in the general case

For example we don't want to throw an exception when we get a bad argument, but allowing arbitrary mutations, because a client won't be able to determine what mutations have been made

```
static void addAll(List<T> list1, List<T> list2)
```

effects: adds the elements of list2 to list1 unless it encounters a null, at which point it throws a NullPointerException

If a NullPointerException is thrown, the client is left to figure out which elements of list2 have made it to list1

The spec should also be weak enough

```
static File open(String filename)
```

effects: opens a file named filename

This is a bad specification, as it lacks details: is the file opened for reading or writing? Does it already exist or is it created? It's too strong since there's no way it can guarantee to open a file. Instead, it should say something weaker: that it attempts to open a file, and if it succeeds, the file has certain properties

The spec should use abstract types where possible

Writing specs with abstract types gives more freedom to both client and the implementor. In Java, this often means using an interface type, like Map, Reader or List. Do that when the behavior doesn't depend on a specific abstract type implementation.

Precondition or postcondition?

Use preconditions to demand a property precisely because it would be hard or expensive for the method to check it.

A non-trivial precondition is an inconvenience to the client. Users don't like preconditions. The Java API designers tend to specify (as a postcondition) that they throw unchecked exceptions when arguments are inappropriate

This approach makes it easier to find the bug or incorrect assumption in the other code that led to passing bad arguments (Fail Fast!)

Sometimes it's not feasible and that's where we want to use a precondition to avoid making the method unacceptably slow.

If we wanted to implement find using binary search, we would require the array to be sorted. Forcing the method to check the array is sorted would defeat the purpose of binary search

↑ to obtain logarithmic not linear time

If the method it's only used locally in a class, the precondition can be discharged by carefully checking all the sites that call the method. But if the method is public, and used by other developers, it would be less wise to use a precondition. Instead, like the Java API, you should throw an exception

About access control

If you make all your methods public - including helper methods - then other parts of the program may come to depend on them, which will make it harder for you to change the internal representation of your class in the future. It will also add clutter to the visible interface your class offers.

About static vs instance methods

```
static int Find(int[] arr, int val)
requires: val occurs in arr
effects: returns index i such that
         arr[i] = val

int Find(int val)
requires: val occurs in this array
effects: returns index i such that
         the value at index this
         array is val
```

Avoid Debugging!

The best defense is to make bugs impossible by design!

Static checking eliminates many bugs by catching them at compile time as we already saw

...And dynamic checking for runtime errors

Immutability is another design principle that prevents bugs

Immutable types that cannot be changed after construction

Java also gives immutable references (variables declared with Final) use Final as much as you can!

Final only makes the variable immutable, not the object that it points to

=

IF we can't prevent bugs, we can try to localize them to a small module or a method so that we don't have to look too hard for the cause (Fail Fast)

```
/**
 * @param x requires x>=0
 * @return approximation to
 *         square root of x
 */
public double sqrt(double x) {...}
```

Suppose somebody calls sqrt with a negative argument. Since the caller didn't respect the precondition, the postcondition is no longer valid and the method could do anything. However since the bad argument indicates a bug in the caller, the most useful behaviour would be to point out the bug as soon as possible

We do it by inserting an assertion:

```
public double sqrt(double x) {
    if (!(x>=0)) throw new
                    AssertionError();
    ...
}
```

The effects of the caller bug are now prevented from propagating

Checking preconditions is an example of defensive programming, which offers a way to mitigate the effects of the bugs even if you don't know where they are!

...

Assertions ...

assert (x >= 0); ← parenthesis are optional

Assertions document assumptions about the program.

It throws an AssertionError if the assertion fails.

It may also include a description expression, which is printed in an error message if the assertion fails:

assert (x >= 0): "x is " + x;

Checking assertions can be costly in performance and are off by default.

For example a procedure with binary search has a requirement that the array be sorted.

Asserting this will turn your procedure from logarithmic to linear time.

You should pay this cost during testing, since it makes it much easier to debug, but not after the program is released to users.

For most applications, however, assertions are not expensive compared to the rest of the program

Note that the Java assert statement is a different mechanism than the JUnit assertTrue, assertEquals, etc.

JUnit assert... () should be used in JUnit tests, to check the result of a test only

What to assert?

Method argument requirements, like we saw for sqrt

Method return value requirements. This type of assertion is sometimes called self check. For example:

```
public double sqrt (double x) {
    assert x >= 0;
    double r;
    ... // compute result r
    assert
        Math.abs(r*r - x) < .0001;
    return r;
}
```

Covering all cases. If a conditional statement or switch does not cover all possible cases, it is good practice to use an assertion to block the illegal cases:

```
switch (vowel) {
    case 'a': ...
    case 'e': ...
    case 'i':
    case 'o':
    case 'u':
    default: assert fail();
```

Write assertions as you write code, as you have the invariants in mind!

What not to assert?

Runtime assertions are not free. They can clutter your code, so they must be used judiciously. Avoid trivial assertions just like you would avoid trivial comments!

If an assertion is obvious from its local context, leave it out!

Never use assertions to test conditions that are external to your program! (e.g. existence of files, availability of a network or correctness of input typed by a human user).

Assertions test internal state of your program to ensure that it is within the bounds of your specification. An assertion failure indicates the program has run off rails, in some state in which it was not designed to function properly.

Thus indicate bugs.

External failures are not bugs and there is no change you can make to your program to prevent them from happening

The correctness of your program should never depend on whether assertions are executed and asserted expressions should not have side effects

Incremental development, modularity and encapsulation

A great way to localize bugs to a tiny part of your program is incremental development. Build only a bit of your program at a time, and test that bit thoroughly before you move on. That way when you discover a bug it's very likely to be in the part you just wrote

Our class on testing talked about unit testing and regression testing. Those help with incremental development

you can localize bugs by better software design

Modularity: Dividing the system into modules each of which can be designed, implemented, reasoned about, and reused separately from the rest of the system.
The opposite is monolithic - big and with all of its pieces tangled up and dependent on each other -

Encapsulation. Once you have modules you can encapsulate them. You can build walls around them so that the module is responsible for its own internal behavior and other parts of the system can't threaten its integrity

One kind of encapsulation is access control. Keeping things private as much as possible, especially variables, limits the code that could inadvertently cause bugs (if that private variable value is wrong, then you should look for the bug just inside this class)

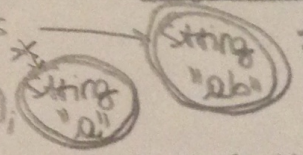
Another kind of encapsulation comes from variable scope. Keeping variable scopes as small as possible makes it much easier to reason where a bug might be in a program

Minimize the scope of variables by:
- Always declaring a loop variable
  For (int i=0; i<100; i++)
- Declaring variables only when you first need it, and in the innermost curly-brace block that you can (related to the rule above)
  Note that languages without static type declarations, the scope of the variable is the entire function anyway
- Avoiding global variables entirely (variables in the form public static ...). They are often unwisely used as a shortcut to provide a parameter to several parts of your code, but it's much better to just pass the parameter into the code that needs it

Mutability and immutability

String is immutable. Once created, a String object always has the same value.

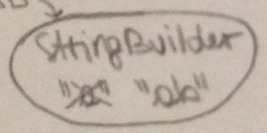
String s="a"; s = s.concat("b");



s += "b"; also calls concat!

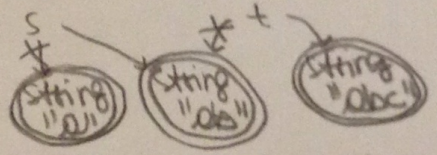
StringBuilder is a mutable type. It has methods that change the value of the object, rather than just returning new values

```
StringBuilder sb =
    new StringBuilder("a");
sb.append("b")
```

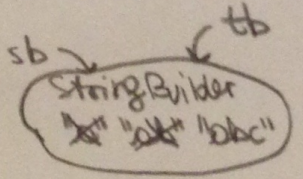


The difference between mutability and immutability doesn't matter much when there's only one reference to the object. But there are big differences in how they behave when there are multiple references to the object

```
String t = s;
t = t + "c";
```




```
StringBuilder tb = sb;
tb.append("c");
```



Changing t creates a fresh string and has no effect on what s means.

But append on tb changes the object value. So sb sees the same change

StringBuilder is still useful to concatenate strings together without making lots of copies

```
String s = "";
for(int i=0; i<n; ++i){
    s = s+n;
}
```

the first number of the string ("0") is actually copied n times while we build the final string, the second n-1 times and so on. It takes O(n²) time just to copy, even though we concatenated n elements

```
StringBuilder sb = new StringBuilder();
for (int i=0; i<n; ++i) {
    sb.append( String.valueOf(n));
}
String s = sb.toString();
```

Getting good performance is one reason why we use mutable objects.

Another is convenient sharing: two parts of your program can communicate more conveniently by sharing a common mutable data structure

Risks of mutation

immutable types are safer from bugs, easier to understand, and more ready for change!

Mutability makes it harder to understand what your program is doing, and much harder to enforce contracts

...

... Here a few examples that illustrate why

#1 Passing a mutable value into a method

```
/**@return the sum of the members in the list*/
public static int sum(List<Integer> list) {
    int sum=0;
    for(int x: list) sum+=x;
    return sum;
}
```

Suppose we need a method that sums the absolute values and reuse sum() above following good practice

```
/**@return the sum of the absolute
          values of the numbers
          in the list
*/
public static int sumOfAbsoluteValues(List<Integer> list){
    for(int i=0; i<list.size(); ++i){
        list.set(i, Math.abs(list.get(i)));
    }
    return sum(list);
}
```

This method does its job by mutating the list directly.

It seemed reasonable to the implementer because it's more efficient to reuse the list.

rather than generating a new one with the absolute values

...

But the resulting behaviour will be surprising to anyone who uses it

```
// meanwhile, somewhere else in the code
public static void main(String[] args) {
    //...
    List<Integer> myData = Arrays.asList(-5,-3,2);
    System.out.println(sumOfAbsoluteValues(myData));

    System.out.println(sum(myData));
```

It will print 10 in both cases!

It's easy to blame the implementer of sumOfAbsoluteValues() for going beyond the spec. But really, passing mutable objects around is a latent bug. It's just waiting for some programmer to inadvertedly mutate that list, often with very good intentions like reuse of performance but resulting in a bug that is very hard to track down

#2 returning mutable values

Suppose we write a method that determines the first day of spring. It uses Date which is mutable

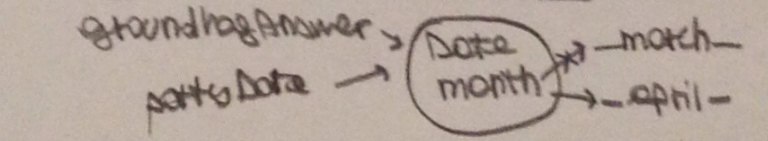
```
/**@return the first day of spring this year */
public static Date startOfSpring() {
    return ost=Groundhog();
}

// somewhere else in the code...
public static void partyPlanning() {
    Date partyDate = startOfSpring();
    //... }
```

the implementer of startOfSpring() rewrites the code to ask groundhog and cache the result

```
/**@return the first day of spring
        this year */
public static Date startOfSpring() {
    IF (groundhogAnswer == null)
        groundhogAnswer = ask Groundhog();
    return groundhogAnswer; }
private static Date groundhogAnswer=null;

//somewhere else in the code...
public static partyPlanning() {
    //let's have a party the month
    // after spring starts'.
    Date partyDate = startOfSpring();
    partyDate.setMonth(partyDate.getMonth()
                        +1);
```



Worst, the bug will probably be discovered in some piece of code that subsequently calls startOfSpring()

IF getMonth returns 11 (December cause it's 0-based), the code will then make the invalid call setMonth(12)

Aliasing is what makes mutability types tricky

In both previous examples, the problem would have been completely avoided if list and date had been immutable types.

In fact, you should never use Date! Use one of the classes in java.time: LocalDateTime, Instant, etc. All guarantee in their specs that they are immutable

The simplest solution to this bug, is for startOfSpring() to always return a copy of the cached groundhog's answer:

return new Date(groundhogAnswer.getTime());

This pattern is defensive copying. The defensive copy means partyPlanning() can freely stomp all over the returned date without affecting startOfSpring()'s cached date. But that forces startOfSpring() to do extra work and use extra space for every client, even if most clients never need to mutate the date. We may end up with lots of copies of the first day of spring throughout memory. If we used an immutable type instead, then different parts of the program could safely share the same value in memory!

=

Using mutable objects is just fine if you are using them entirely locally within a method, and with one reference to the object. The problem is with multiple references, in different parts of the code. These references are called aliases:

- In the list example, the same list was pointed by both list (in sumOfAbsoluteValues()) and myData (main())
- In the Date example, there are two variable names that point to the Date object, groundhogAnswer and partyDate

=

At this point it should be clear that when a method performs mutation, it is really important to include that mutation in the method's spec

(However, we've seen that even when a particular method doesn't mutate an object, that object's mutability can still be a source of bugs)

Here's an example of a mutating method:

```
static void sort(List<String> list)
    requires: nothing
    effects: puts list in sorted order
        i.e list[i] <= list[j]
        for all 0 <= i < j < list.size()
```

And an example of a method that does not mutate its arguments:

```
static List<String> toLowerCase(List<String> list)
    requires: nothing
    effects: returns a new list + where
        +[i] = list[i].toLowerCase()
```

If the effects does not implicitly say that an input can be mutated, then in 6.005 we assume mutation of the input is implicitly disallowed!

Iterating over arrays and lists

An iterator is a mutable object that steps through a collection of elements and returns the elements one by one.
They are used behind the scenes when using a for loop (for(x : list))

```
List<String> list = ... ;
for (String str : list) System.out.println(str);
```

↓ is rewritten like this

```
List<String> list = ...;
Iterator iter = list.iterator()
while (iter.hasNext()) {
    String str = iter.next();
    System.out.println(str);
}
```

next() is a mutator method, it changes what iterator mean, its value

There are many different kinds of collections with different internal representations. The iterator concept allows a single uniform way to access them all, so that client code is simpler and the collection implementation can change without changing the client code that iterates over it. Most modern languages have the notion of an iterator. It's a very effective design pattern (well tested solution to a common design problem)

Iterators can be broken by mutating the underlying list of strings while you're iterating over it, and the for loop that's secretly under for it has the same problem. By doing

```
for (String str : list) list.remove(str);
```

you get a ConcurrentModificationException

TO Fix it use the remove() of Iterator so that the iterator adjusts its index appropriately

```
Iterator iter = list.iterator();
while (iter.hasNext()) {
    if(iter.next().startsWith("...")) {
        iter.remove();
    }
}
```

But if there are other iterators active on the same list they won't all be informed

Mutable objects make simple contracts very complex

Multiples aliases rely on the object to remain consistent so contracts can't be enforced in just one place anymore (between the client of a class and the implementer of a class). Contracts involving mutable objects depend on the good behaviour of everyone who has a reference to the mutable object

...

Mutable objects reduce changeability

Mutable objects make the contract between a client and an implementer more complicated, and reduce the freedom of the client and implementer to change.

For example

```
/**
 *@param username [redacted] username
 *                [redacted] of person to look up
 *@return the 9-digit MIT identifier
 *                for username
 *@throws NoSuchUserException if nobody with
 *                username is in
 *                MIT's database
 */
public static char getMitId(String username)
                throws NoSuchUserException
```

Now suppose the client writes

```
char[] id = getMitId("bitdiddle");
System.out.println(id);
```

Now both implementer and client decide to make a change

- The client is worried about user privacy, and decides to delete the first 5 digits of the id
- The implementer is worried about the speed and load on the database, [redacted] introduces a cache that remembers usernames and that have been looked up

```
client
...
char[] id = getMitId("bitdiddle");
for(int i=0; i<5; i++) id[i] = '*';
...
implementer
...
if (cache.containsKey(username)) {
    return cache.get(username);
} ... look up!
cache.put(username, id);
return username;
...
```

When the client looks up "bitdiddle" and gets back a char array, now both client and implementer's cache are pointing to the same [redacted] char array (it is aliased). That means the client's deleting code is actually overwriting the identifier in the cache. It is hard to blame someone here

Here's one way we could've clarified the spec

```
public static char[] getMitId(String username) throws
                                        NoSuchUserException
    requires: nothing
    [redacted]effects: returns an array containing the 9-digit MIT identifier
                of username or throws NoSuchUserException if nobody
                with username is found. Caller may never modify
                the returned array
```

This is a bad way to fix it the contract has to be enforced for the whole rest of the program. You can think of preconditions et particularities but you don't have to reason about what would happen for the rest of the time. Another spec, more client-favorable

```
    effects: returns a new array containing the 9-digit MIT
                identifier of username or throws ...
```

This doesn't entirely fix the [redacted] problem as it doesn't say whether the implementer can hold onto an alias to that fresh array... A much better spec

```
public static String getMitId(String username) throws NoSuch
                                                UserException
    requires: nothing
    effects: returns the 9-digit MIT identifier of username, or
                throws NoSuchUserException if nobody with
                username is found
```

Immutability gives a guarantee that implementer and client will never step on each other the way they could with char arrays. It also gives the implementer the freedom to use that cache!

Useful immutable types

Since immutable types avoid so many pitfalls, let's look at some of the commonly used immutable types in the Java API

- Primitive types and their wrappers are immutable - BigInteger and BigDecimal are also immutable
- Don't use mutable Dates, use the appropriate immutable type from java.time based on the granularity needed
- List, Set, Map implementations are all mutable, but they have utilities methods to create unmodifiable views of these mutable collections (Collections.unmodifiableList)

You can think of the unmodifiable view as a wrapper of the underlying list/set/map, and every mutation will trigger an UnsupportedOperationException

Before passing a mutable collection to another part of your program it's a great idea to wrap it in one of these unmodifiable wrappers. But be careful at that point to forget about the reference to the mutable collection or you will have aliases to the mutable part floating around

Collections also provides methods for obtaining empty collections, because there's nothing worse than discovering what you thought was empty has suddenly been made not empty by some other part of the program

Debugging!

systematic debugging

sometimes you have no choice but to debug. It may be hard to find where it is. For those situations we have a systematic strategy that can make debugging more effective

1. Reproduce the bug!

Start by finding a small, repeatable test case that produces the failure. If the bug was found by regression testing you already have that test case, otherwise it may take some effort to reproduce the bug. For graphical user interfaces or multithreaded programs, a bug may be hard to reproduce consistently if it depends on timing of events or thread execution.

Any effort you can make on making that test case small and repeatable will pay off, because you'll have to run it over and over while you search for the bug and try to fix it. And once you fix it this small repeatable test case will make a great regression test. Once you have a test case for the bug, making this test work becomes your goal

```
/**
 * Finds the most common word in a string
 * @param text string containing zero or
 *             more words
 * @return a word that occurs more often than
 *         in text, ignoring alphabetic case
 */
public String mostCommonWord(String text)
```

A user pasted the whole text of Shakespeare's plays and got back "x"

The input is too large and we can't print or breakpoint and debug it. Instead reduce the size of the buggy input

- Does the first half of Shakespeare show the same bug (Binary search)
- Does a single play have the same bug)
- Does a single speech have the same bug?

Once you've found a small test case, find and fix the bug using that smallest test case, and then go back to the original buggy input and confirm that you fixed the same bug

Understand the location and cause of the bug

You can use the scientific method:

- Study the data, look at the test input, the incorrect results and the stack trace.
- Hypothesize, [redacted]... with consistencies to your data, about where the bug might be or [redacted] be, it cannot
  It's good to make a general hypothesis at first
- Experiment, to test your hypothesis.
  It's good to make the experiment an observation at first (disturb the system as little as possible.)
- Repeat, add data collected from previous experiments and make a fresh hypothesis
  Hopefully you have ruled out some possibility and narrowed down the possible locations [redacted] and reasons for the bug

[redacted]

↓ A good experiment is a gentle observation of the system [redacted] without disturbing it much. It might be
- run a different test case
- insert a print statement or an assertion in [redacted]
- Breakpoint, then single step through the code and look at variable and object values

It's tempting to try to insert fixes to the hypothesized bug instead of mere probes. But that is [redacted] almost always the wrong thing to do. It leads to a kind of ad-hoc guess-and-test programming, which produces awful, complex, hard-to-understand code. Your fixes may just mask the true bug without actually removing it

Other tips

Bug localization by binary search. Divide the workflow in half [redacted] insert probes where the bug should be. Then check the results. From the answer to that experiment, you would divide further in half

Prioritize your hypotheses. Different parts of the system have different likelihood of failure. Old well-tested code is probably more trustworthy than recently-added code. Java libraries code is more trustworthy than yours because more tried and tested. Likewise your [redacted] Java compiler, OS, hardware, [redacted] than should be trusted until you've [redacted] found a good reason not to

Swap components. If you have another implementation that satisfies the same interface, and you suspect the module, then [redacted] one experiment you can do is [redacted] to the swapping in the alternative. eg. you suspect your binarySearch(), then substitute with a simpler linearSearch() you can waste a lot of time by swapping components so don't do this unless you have a good reason to suspect a component

Make sure your source code and target code are up-to-date. Pull the latest version, delete your binary files and recompile everything (Project → Clean)

Get help. It often helps to explain the problem to someone else

Sleep on it. If you're too tired, then you won't be an effective debugger. Trade latency for efficiency

ooo

Fix the bug...

Once you've found the bug, you must devise a fix for it. Avoid the temptation to slap a patch on it and move on. Ask yourself whether the bug was a coding error, or was a design error. Design errors may suggest that you step back and think about your design, or at the very least consider all the clients of the failing interface to see if they suffer from the bug too.

Think also whether the bug has any relatives. If you just found a divide-by-zero error, have you in the GUI code? Try to make the code safe from future bugs like this. Also consider what effects your fix will have. Will it break any other code?

After you've applied the fix, add the bug's test case to regression suite and run all the tests to show yourself that you fixed it and didn't introduce new bugs.

Abstract Data Types

Abstract data types are an instance of a general principle that goes by many names with slightly different shades of meaning. Abstraction, omitting or hiding low level details with a simpler high-level idea.

Modularity, dividing a system into components or modules, each of which can be designed, implemented, tested, reasoned about, and revised separately from the rest of the system.

Encapsulation, building walls around each module (a hard shell or a capsule) so that the module is responsible for its own internal behavior, and bugs in other parts of the system can't damage the integrity of that internal behavior.

Information hiding, hiding details of a module implementation from the rest of the system, so that those details can be changed later without changing the rest of the system.

Separation of concerns, making a feature (or "concern") the responsibility of a single module, rather than spreading it widely across multiple modules.

The fundamental purpose of all of these ideas is to help achieve safety from bugs, ease of understanding, readiness for change.

User-defined types

In the early days of computing, a programming language came with built-in procedures and types. Users could define their own procedures.

A major advance in software development was the idea of abstract types, that one could define types too.

Researchers that helped are Dahl, Hoare, Parnas, at MIT Liskov and Guttag did seminal work on the specification of abstract types.

The key idea of data abstraction is that a type is only characterized by the operations you can perform on it and not by the data structure you used to represent it.

eg a number is something that you can add or multiply, a string is something that you can concatenate and take substrings of. In a sense, users could already define types in the early days of programming languages by creating a record data type for example with integer fields for day, month and year and define procedures on that type. What made abstract types new and different was this focus on operations.

...

...

the user of a type would not need to worry about how it's values were actually stored, in the same way that a programmer can ignore how the compiler actually stores primitive types like integers.

All that matters is what operations there are and their specs

[redacted]

Classifying types and operations

Types, whether built-in or user-defined, can be either mutable or immutable. Mutable types provide operations that which when executed cause the results of other operations on the same object to give different results.

Date is mutable as when we call setMonth we can observe the change with getMonth.

String is immutable because its operations that seem to change a string actually create a new string object and leave the existing value unchanged

The operations of an abstract type can also be classified:

- Creators create a new object of the type. It may take an object as an argument but not an object of the type being constructed
- Producers create new objects from old objects of the type. The concat method in String for example
- Observers take objects of the abstract type and return properties of that object which has a different type. Size in List, it is an observer for list that returns an int
- Mutators change objects. The add method of list mutates the list

In summary: T abstract type
regex notation t other types

creator: $t^* \to T$
producer: $T+, t^* \to T$
observer: $T+, t^* \to t$
mutator: $T+, t^* \to$ void/t/T

A creator operation is often implemented as a constructor called by using new. But it doesn't have to be, a creator can simply be a static method instead, like Arrays.asList().
A creator implemented as a static method is often called a Factory method.
Another example is String.valueOf

Mutators are often signaled by a void return type. A method that returns void must be called for some side effect.
But not all mutators return void, eg Set.add returns boolean, or [redacted].add() returns the object itself so that we can add another .add() and chain calls together

From java's graphical user interface toolkit, component.add()

Abstract Data Type Examples

int is a primitive type, it is immutable since has no mutators
creators: the numeric literals 0, 1, 2, 3...
producers: arithmetic operators + - * /
observers: comparison operators ==, !=, <, >
mutators: none (it's immutable)

List is mutable, it is an interface
creators: ArrayList and LinkedList constructors, Collections.singletonList
producers: Collections.unmodifiableList
observers: size, get
mutators: add, remove, addAll, Collections.sort

String is immutable
creator: String constructors
producers: concat, substring, toUpperCase
observers: length, charAt
mutators: none (it's immutable)

This classification gives some useful terminology for thinking about operations and types, it is not perfect, there may be an operation that's both a producer and mutator

[redacted]

Designing an abstract type

Designing an abstract data type involves choosing good operations and determining how they should behave:

It's better to have a few, simple operations that can be combined in powerful ways, rather than lots of complex operations.

Each operation should have a well defined purpose and it should have a coherent behaviour rather than a lot of special cases. We probably shouldn't add a sum operation to List as might help for lists of integers but lists of strings or lists of lists are special cases that make sum hard to understand and use

The set of operations should be adequate. There must be enough to do the kinds of computations clients are likely to want to do. A good test is to check that every property of an object can be extracted. For example with no get operation we would not be able to find out what the elements of the list are.

Basic information should not be inordinately difficult to obtain. For example we could call get on increasing indices until we get a failure to know size, but that's inefficient and inconvient.

The type may be generic (a list or a set, or a graph) or it could be domain specific (street map, phonebook), but you should not mix generic and domain specific features

Representation independent

Critically, a good abstract data type must be representation independent. This means that the use of an abstract data type is independent of its representation (the actual data structure or data fields used to implement it), so that changes in representation have no effect on the abstract type itself. For example the operations of list are independent of whether list is represented as a linked list or as an array

User won't be able to change the representation of an abstract data type at all unless operations are fully specified with preconditions and postconditions, so that clients know what to depend on and you know what you can safely change

Realizing abstract data type concepts in java

There are several ways to do it and it's important to understand the big idea, like a creator pattern and different ways to achieve that idea in practice

Creator operation	constructor	ArrayList()
	static (factory) method	Collections.singletonList()
	constant	BigInteger.ZERO
Observer operation	instance method	list.get()
	static method	collections.max()
Producer operation	instance method	String.trim()
	static method	Collections.unmodifiableList()
Mutator operation	instance method	List.add()
	static method	Collections.copy()

Representation private fields

Testing an abstract data type

We build a test for an abstract data type by creating tests for each of its operations. These tests inevitably interact with each other as the only way to test creators, producers and mutators is by calling observers on the object that result, and likewise, the only way to test and an observer is by creating objects for them to observe

Abstraction Functions and rep invariants

Invariants!

Last reading we talked about what makes a good abstract data type, and the final 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.

Immutability is an invariant we've already seen.

Saying that an abstract data type preserves its own invariant means that it is responsible for ensuring that its own invariants hold. It doesn't depend on good behaviour from its clients

When an ADT preserves its own invariants, reasoning about the code becomes much easier. If you can count on the fact that Strings never change, you can rule out that possibility when debugging, or when you're trying to establish an invariant for another ADT that uses Strings.

Immutability

```
public class Tweet { public String author, text;
  public Date timestamp;
  public Tweet (String author, String text, Date timestamp) {
      this.author = author; this.text = text;
      this.timestamp = timestamp;
  }
}
```

How do we guarantee that tweet objects are immutable?

↙ that all its fields can never be changed once a Tweet is created

The first threat to immutability is that the client can directly access its fields. In fact they have to

So nothing's stopping to write code like:

```
Tweet t = new Tweet ("max" "hello!!",
t.author = "tom!!";    new Date());
```

This is a trivial example of representation exposure, meaning that code outside the class is allowed to modify the representation directly. Rep exposure threatens invariants and representation independence, i.e we can't change the implementation of Tweet without affecting all the clients that are directly accessing these fields

```
public class Tweet {
  private final String author, text;
  private final Date timestamp;
  public Tweet (...) { ... }
  public String getAuthor () {
      return author; }
  public String getText () {
      return text; }
  public Date timestamp () {
      return timestamp;
```

That's better but the rep is still exposed

```
public retweetLater (Tweet t) {
  Date d = t.getTimestamp();
  d.setHours (d.getHours()+1);
  return new Tweet ("tom!!",
      t.getText(), d);
...
```

d.setHours changes the same object that t was depending on. Tweet leaked out a reference to a mutable object that its immutability depended on.

We talked about this before, we can patch this code by doing defensive copying

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

mutable objects have a copy constructor that allows you to do that.

Another way to copy a mutable object is clone(), supported by some types

...

...

We're not done yet, there's still rep exposure!

Consider this (reasonable) code

```
public static List<Tweet>
        tweetEveryHourDay() {
    Date date = new Date();
    for (int i=0; i<24; i++) {
        date.setHour(i);
        list.add(new Tweet("Hmm!",
            "keep it up!", date));
    } return list;
}
```

Every tweet object points to the same date object. All tweet objects end up at the same time (hour 23).

Again the immutability of tweet is violated.

We can fix it too by judicious defensive copying

```
public Tweet(...) {
    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 [redacted] return direct references to its representation!

But why make all these copies of dates when we can solve by a carefully written spec like:

```
@param timestamp date/time when the Tweet was sent.
       Caller must never mutate this Date
       object again!
```

The spec approach is sometimes taken when there isn't any other reasonable alternative - e.g. the mutable object is too large to copy. 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.

An even better solution to that problem is to use immutable types. If we used ZonedDateTime instead of Date, then we would've ended the section a long time ago as no further rep exposure would've been possible.

Immutable wrappers around mutable data types

The Java collections classes offer an interesting compromise: immutable wrappers!

Collections.unmodifiableList() takes a (mutable) list and wraps it with an object that looks like a list but whose mutators throw exceptions. So you can construct a list using mutators and then seal it up in an [redacted] unmodifiable wrapper (and then throw away your reference to the original mutable list), and get an immutable list.

The downside is that you get immutability at runtime, but not compile time. But that's still better than nothing as using immutable collections can be a very good way to reduce the risk of bugs.

Rep Invariant and Abstraction Function

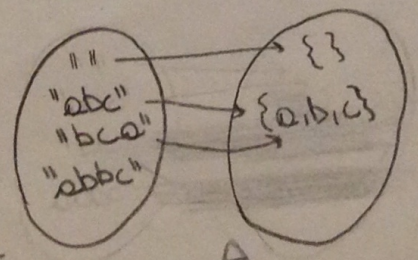
In thinking about an abstract type, it helps to consider the relationship between two spaces of values:

- The space of the representation values (or rep values for short) [redacted] consists of the actual implementation entities. In simple cases, an abstract type will be implemented as a single object, but more commonly a small network of objects is needed. For now [redacted] we'll see it as a mathematical value

- The space of abstract values. Consists of the values our type is designed to support. Those are platonic entities that don't exist as described, but they're the way we want to [redacted] view the elements of the abstract type as a type
  eg an abstract type for unbounded integers might be implemented as an array of primitive integers, but that's not relevant to the user.

The implementer must be interested in the rep values, since it's job is to achieve the illusion of the abstract value space using the rep value space

```
public CharSet {
    private String s;
    ...
}
```



- Every abstract value is mapped to by some rep value. The purpose of implementing the abstract type is to support operations on abstract values. All [redacted] possible abstract values [redacted] must be [redacted] representable

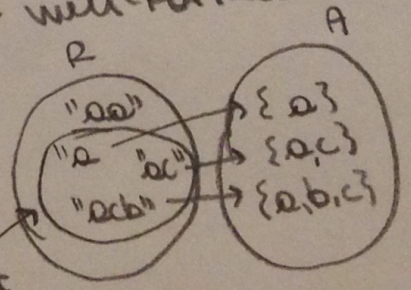
- Some abstract values [redacted] are mapped to by more than one rep value. This [redacted] happens cause the representation isn't a tight encoding.

- Not all rep values are mapped

The mapping is described by:

- an abstraction FUNCTION AF: R → A
  It [redacted] maps the rep values to the abstract values they represent

- a rep invariant that maps rep values to a boolean RI: R → boolean.
  For a rep value r, RI([redacted] r) is true if and only if r is mapped by AF. ie it tells us whether a given rep value is well-formed



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

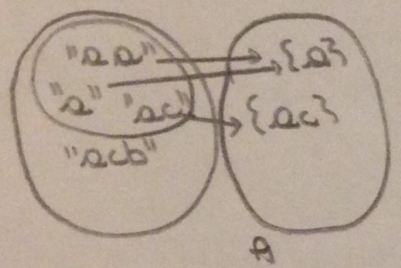
```
public CharSet {
    private String s;
    // Rep invariant:
    //   s contains no repeated
    //   characters
    // Abstraction Function:
    //   represents the set of
    //   characters found in s
    //
    ...
}
```

A common confusion for AF on RI:
There can be several [redacted] representations for the same abstract type. A set of characters could equally be represented as a string, as above, or as a bit [redacted] vector, with one bit for each possible character. Clearly we would need [redacted] two different abstraction functions to map these two different rep value spaces

It's less obvious why the choice of both spaces [redacted] doesn't determine AF and RI. The key point is that defining a type for [redacted] a rep, and thus choosing the space for rep values, [redacted] does not determine which of the rep values will be deemed to be legal, and those that are legal, how they will be interpreted.
eg [redacted] We could allow duplicates (in the example above) but require that the characters are sorted, allowing binary search and checking membership in log time

ppp

```
...
public class CharSet {
    private String s;
    // rep invariant:
    //   s[0] <= s[1] <= ... s[s.length()-1]
    // Abstraction Function:
    //      ...
}
```



Even with the same type for the rep value space and the same rep invariant RI we still might interpret the rep differently, with a diffent AF. Suppose RI admits any string of characters, then we could define AF, as above, to represent strgs's elements as elements of the set. But there's no a priori reason to let rep decide the interpretation. If we could interpret consecutive pair of characters as subranges, so that "acgg" is interpreted as two range pairs [a-c] and [g-g] therfore representing {a,b,c,g}

```
public CharSet {
    private String s;
    // Rep invariant:
    //      s.length is even
    //      s[0] <= s[1] <= ... <= s[s.length()-1]
    // Abstraction Function:
    //      represents the union of the ranges
    //      {s[i]... s[i+1]} for each adjacent pair
    //      of characters in s

}
```

Designing an abstract type means not only choosing two spaces (rep value space and abstract value space) but also deciding what rep values to use and how to interpret them!

It's critically important to write these assumptions in your code so that you and other programmers are aware of what the representation actually means

Check the RI

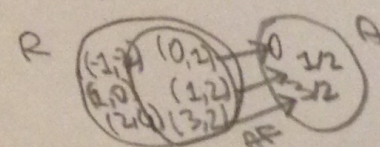
```
public class RatNum {
    private final int numer;
    private final int denom;

    // Rep invariant:
    //    denom > 0
    //    numer/denom is in reduced form

    // Abstraction Function:
    // represents the rational number
    //    numer/denom

    /** Make a new RatNum == n
     */
    public RatNum(int n){...}

    // Make a new RatNum == (n/d)
    public RatNum(int n, int d){...}
```



~~The rep invariant~~, if your implementation detects the rep invariant at runtime then it can fail fast and catch bugs earlier

```
// Check that the rep invariant is true
private void checkRep() {
    assert denom > 0;
    assert gcd(numer, denom) == 1;
}
```

You should call checkRep at the end of every operation that creates or mutates the rep (creators, producers, mutators).

Observers don't usually need to call checkRep, but it's good practice as you will be more likely to catch rep invariant violations caused by rep exposure

No null values in the rep

Like we did with method specs, in 6005 the rep invariant implicitly includes x != null for every reference x in the rep that has object type (including references in arrays or lists)

In checkRep, you should still implement the != null check sometimes is not need when calling a method on that object in checkRep, an exception is thrown if null

Documenting the AF, RI and safety from rep exposure

optional

It's good practice to document the AF and RI in the class using comments right where the private fields of the rep are declared. We've seen that another piece of documentation that 6005 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 constructors, parameters and return values from clients) and presents a reason why the code doesn't expose the rep

```
public class Tweet {...}
```

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 rep's Date with clients

an immutable rep is particularly easy to argue for safety. Doesn't need that last part (about Date)

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

So specifically
- creators and producers must establish the invariant for new object instances and
- mutators and observers must preserve the invariant

If the rep is exposed the object might change anywhere in the program and we can't guarantee that the invariant is still held after arbitrary changes

Structural induction
If an invariant of an ADT is
- established by creators/producers
- preserved by mutators/observers
- No rep exposure occurs

then the invariant is true of all instances of the ADT

ADT Invariants Replace Preconditions

An enormous advantage of a well designed ADT is that it encapsulates and enforces properties that we would otherwise state in a precondition

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

We can instead use an ADT that captures the desired property

```
/** @return characters that appear in one set but not the other
 */
static SortedSet<Character> exclusiveOr(SortedSet<Character> set1,
                                        SortedSet<Character> set2);
```

This easier to understand as the name of the ADT conveys what the programmer needs to know. It's also safer from bugs, because Java static type checking comes into play and the required condition can be enforced in the type

Many of the places where we used preconditions would've benefited from a custom ADT instead

Important: specs in implementation of interface B are written in terms of A. It would be malformed to mention details of any particular implementation of A with particular private fields. The spec should apply to any valid A implementation

Interfaces

Java's interfaces are useful to express ADTs. An interface in Java is a list of signatures, no method bodies. A class implementing it must provide all method bodies for that interface. So one way to define an ADT in Java is as an interface

One advantage of this approach is that the interface specifies the contract for the client and nothing more. The interface is all a client needs to read to understand the ADT. The client can't create inadvertent dependencies on the ADT's rep, because instance variables can't be put in the interface at all. The implementation is kept well and truly separated, in a different class altogether.

Another advantage is that multiple different representations of the same ADT can coexist in the same program. When an ADT is represented just as a single class it's harder to have multiple representations

Java static type checking catches many mistakes in implementing an ADT's contract. But there's no check that the code adheres the specs of those methods.

Subtyping

Recall that a type is a set of values. The Java List is an interface so all List values are objects of another class implementing List. A subtype is a subset of the supertype

"B is subtype of A", every B is an A. In spec terms "every B satisfies the spec for A"

This means that B is a subtype of A if B's spec is at least as strong as A's spec (B satisfies all requirements put on A)

The compiler can't check that we haven't weakened the spec in other ways when defining B:

- strengthen precondition/weaken postcondition
- weakening a guarantee that the interface abstract type advertises to clients (such as immutability)

If you implement an interface, then you must ensure that the subtype's spec is at least as strong as the supertype's

Why Interfaces?

There are a few good reasons to bring an interface into your code:

- documentation for both the compiler and humans
  - they help the compiler catch ADT implementation bugs and easier for humans to read than a concrete implementation. Such an implementation intersperses ADT-level types and specs with implementation details
- Allowing performance tradeoffs.
  Different ADT implementations can provide methods with different performance. Different implementations may work better with different choices, but we would like these to be representation independent. It should be possible to drop in any new implementation with simple localized code changes
- Optional methods. List from the Java library makes all mutators as optional. We can provide immutable lists. Although some operations are hard to implement with good performance on immutable lists, so we want mutable implementations, too.
- Methods with intentionally undetermined specs.
  An ADT for finite sets could leave unspecified the element order you get back when converting to a list. Some implementations might manage to keep the list sorted, by allowing quick conversion to sorted list. Others might not bother to support conversions to sorted lists
- Multiple views of one data.
  A drop-down list is natural to view as both a widget and a list. The class for this widget could implement both interfaces. I.e. we don't implement an interface multiple times because we're storing multiple data structures, but we make multiple implementations when different objects may be seen as special cases of the ADT, among other useful perspectives
- More and less trustworthy implementations.
  Another reason to implement an interface multiple times might be that it is easy to build a simple implementation that you believe it's correct, while you can work on a faster version that is more likely to contain bugs.

We can now extend our Java toolbox of ADT concepts from the First ADTs reading!

ADT concept	Ways to do it in Java	Examples
Abstract data type	Single class interface + class(es)	String List and ArrayList
Creator operation	constructor static (Factory) method constant	ArrayList() Collections.singletonList() Arrays.asList() BigInteger.ZERO
Observer operation	instance methods static methods	List.get() Collections.max() String.trim() → unmodifiable List()
Producer operation	instance method static method	Collections.[redacted] List() List.add() Collections.copy()
Mutator operation	instance method static method	

Representation: private fields

Equivality

Formally we can digord equality in an ADT in several ways:

- Using the abstraction function.
  a equals b iff F(a) = F(b)
- Using a relation.
  An equivalence is a relation E⊆T×T that is:
  - reflexive: $E(t,t) \forall t \in T$
  - symmetric: $E(t,u) \Rightarrow E(u,t)$
  - transitive: $E(t,u) \wedge E(u,v) \Rightarrow E(t,v)$

  To use E as a definition for equality we would a equals b iff E(a,b)

The two notions above are equivalent. An equivalence relation induces an abstraction function. The relation induced by an abstraction function is an equivalence relation

- Using observation
  We can say that two objects are equal when they cannot be distinguished by observation - every operation we can apply produces the same result for both objects
  eg consider sets {1,2,3}, {2,1,2} with observer operations cardinality |...| and membership ∈. Then
  - $|\{1,2,3\}| = 2$ and $|\{2,1,2\}| = 2$
  - $1 \in \{1,2,3\}$ is true, $1 \in \{2,1,2\}$ is true
  - $2 \in \{1,2,3\}$ is true, $2 \in \{2,1,2\}$ is true
  - $3 \in \{1,2,3\}$ is false, $3 \in \{2,1,2\}$ is false
  - ... and so on

In terms of ADT, "observations" means calling operations on the objects. Two objects are equal iff they cannot be distinguished by calling any operations of the ADT

Example: Duration

```
public class Duration {
    private final int mins;
    private final int secs;
    // rep invariant:
    //   mins >= 0, secs >= 0
    // abstraction function:
    //   represents the span of time
    //   of mins minutes and secs seconds

    /** Make a duration lasting for
     * m minutes and s seconds
     */
    public Duration(int m, int s) {
        mins = m; secs = s;
    }

    /** @return length of this duration
     *          in seconds
     */
    public long getLength() {
        return mins * 60 + secs;
    }
}
```

Equality of immutable types

Like many languages, Java has two different operations for testing equality, with different semantics:

- The == compares references. More precisely it tests referential equality: Two references are == if they point to the same storage in memory
- The equals() operation compares the object contents. In other words object equality, in the sense we've been talking about in this reading. The equals operation has to be defined appropriately for every ADT

Java	equals()
objective-C	isEqual:
C#	Equals()
python	== ← Flips in python
javascript	n/a

The equals() method is defined Object and its default implementation is

```
public boolean equals(Object that) {
    return this == that;
}
```

Which is the same as referential equality. For immutable data types this is almost always wrong. So you have to override equals() and replace it with your own implementation

```
public class Duration {          // Every class implicitly extends Object
    public boolean equals(Duration that) {
        return this.getLength() == that.getLength();
    }
}
```

There's a subtle problem here

```
Duration d1 = new Duration(1,2);
Duration d2 = new Duration(1,2);
Object o2 = d2;
d1.equals(d2); → true
d1.equals(o2); → false
```

It turns out that Duration has overloaded the equals() method, because the method signature was not identical to other we actually have two equals. Equality has become inconsistent in Duration

Use @Override to avoid such issues

```
right way to implement equals()
public boolean equals(Object thatObject) {
    if (!(thatObject instanceof Duration)) {
        return false;
    }
    Duration thatDuration =
        (Duration) thatObject;
    return this.getLength() ==
        thatDuration.getLength();
}
```

...

instanceOF

instanceOF tests whether an object is an instance of a particular type. It performs dynamic type checking.

In general, using instanceOF in OOP is bad smell. In 6.005 and most java programming.

instanceOF is disallowed everywhere except for implementing equals.

This prohibition also includes other ways of inspecting object's runtime types like calling getClass()

The spec of the Object class is so important that it's often referred to as the Object contract. equals states that:

- equals must define an equivalence relation.
- equals cannot be consistent. Repeated
- For a non-null reference x, x.equals(null) should return false
- hashCode must provide the same result for two objects that are deemed equal by the equals() method

We have to make sure that the definition of equality implemented by equals() is actually an equivalence relation as defined earlier. If it isn't, then operations that depend on equality (like sets, searching) will not behave as expected

Here's an example of how an innocent attempt to make equality more flexible can go wrong

```
private static final int CLOCK_SKEW=5;

@Override
public boolean equals(Object thatObject){
    if (!(thatObject instanceOF Duration)){
        return false;
    }
Duration thatDuration=(Duration)
                        thatObject;
return Math.abs(this.getLength()-
    thatDuration.getLength()) <=CLOCK_SKEW;
}
```

This equals violates transitivity

```
d_0_57 = new Duration(0,57)
d_1_00 = new Duration(1,0)
d_1_03 = new Duration(1,3)
d_0_57.equals(d_1_00) →true
d_1_00.equals(d_1_03) →true
d_0_57.equals(d_1_03) →false
```

not transitive!

Creating Hash Tables

To understand the part of the contract relating the hashCode() method, you'll need to have some idea of how hash tables work.

HashSet and HashMap depend on the hashCode() method to be implemented correctly for the objects stored in the set and used as keys in the map

A hash table is an ADT that maps keys to values and offer constant time lookup. Keys don't have to be ordered or any particular property, except for offering equals() and hashCode()

Here's how it works. Contains an array initialized to a size corresponding to the number of elements that we expect to be inserted

When a key and a value are presented for insertion, we compute the hash code of the key and convert it to an index in the array's range. The value is then inserted at that index

The rep invariant of a hashTable includes that fundamental constraint that keys are in the slots determined by their hash codes.

Hashcodes are designed so that keys will be spread evenly. But a collision can occur and keys can end up at the same index. So instead of holding a single value at an index, a hash table holds a list of key/value pairs, usually called a hash bucket.

Now it should be clear why the Object contract requires equal objects to have the same hash code. If two equal objects have distinct hash codes they may be placed in different slots. So if you attempt to lookup a value using a key equal to the one with which it was inserted, the lookup may fail

000

Object's default hashCode() implementation is consistent with its default equals()

```
public Object {
    public boolean equals(Object that) {
        return this == that;
    }
    public hashCode() { return memory address of this; }
```

IF a==b then the addition of a == the addition of b. So the object contract is satisfied

But immutable objects need a different representation of hashCode, because they override the equals() method.

For duration, since we haven't overridden the default hashCode() we're actually breaking the object contract

A simple and drastic way to ensure that the contract is met for hashCode is to always return some constant value, so every object's hashCode is the same. That would have a disastrous performance effect since every key will be stored in the same slot, and every look will become linear time.

The standard way, that still satisfies the contract is to compute the hash code for each component of the object that is used in the determination of equality and and then combining these throwing in a few arithmetic operations. For duration is based on the the abstract value of the class is int

```
@Override
public int hashCode() { return(this).getLength(); }
```

There are different hashing techniques. Objects.hash makes it easier to implement a hash code involving multiple fields

Make as long as you satisfy the requirement that equal objects have the same hashcode value, then the particular hashing implementation doesn't make a difference to the correctness of your code. It may affect effect its performance by creating unnecessary collisions between different objects, but even a poorly-performing hash function is better than one that breaks the contract.

If you have overridden equals() but not hashCode() that means that you have almost certainly violated the contract

Always override hashCode() when you override equals()

Always use @Override to make sure you're not creating a new method of a type, or overloading because of a slightly different signature.

Equality of mutable types

Recall our definition that two objects are equal when they cannot be distinguished by observation. With mutable objects there two ways to interpret this:

- When they cannot be distinguished by observation that doesn't change the state of the object ie by calling observer, producer and creator methods. This is often called observational equality, since tests whether two objects look the same in the current state of the program
- When they cannot be distinguished by any observation, even state changes. This interpretation allows by calling any methods on the objects including mutators. This is often called behavioral equality, since it tests whether the two objects will behave the same, in this and all future states

For immutable objects, observational and behavioral equality are identical

For mutable objects it's tempting to implement strict observational equality. Java uses observational equality for most of its data types eg if two distinct lists contain the same sequence of elements then equals() reports they're equal

But using observational equality can lead to subtle bugs, in fact it allows us to break the rep invariant of other collections & ADTs

Suppose we make a list and drop it into a set

```
List<String> list = new ArrayList<>();
list.add("a");
Set<List<String>> set = new HashSet<List<String>>();
```

We can check that the set contains the list we put in, and it does

```
set.add(list);
set.contains(list); → true
```

But now we mutate the list

```
list.add("goodbye");
set.contains(list) → false
```

List<String> is a mutable object and mutation affects the result of equals() and hashCode(). When the list is first put into the HashSet, it is stored in the hash bucket corresponding to hashCode result at that time. When the list is subsequently mutated, its hashcode changes, but HashSet doesn't realize it should be moved to a different bucket.

If we iterate through the set, we still find the list there as it goes through buckets one at time without using its hashcode to look up

When equals() and hashCode() are affected by mutation we can break the rep invariant of a hash table that uses that object as a key

The Java library is unfortunately inconsistent about its interpretation of equals() for mutable classes. Collections use observational equality, but other mutable classes (like StringBuilder) use behavioural equality

To be safe from bugs, equals() should implement behavioural equality ie two references should be equal iff they are aliases for the same object

So mutable objects should just inherit equals() and hashCode() from Object. For clients that need observational equality (& check whether two objects "look" the same in the current state) it's better to define a new method eg. similar()

The Find table for equals() and hashCode()

For immutable types

- equals() should compare abstract values this is the same as saying equals() should provide behavioral equality
- hashCode() should map the abstract value to an integer

Immutable types must override both!

For mutable types

- equals() should compare references, like == does. This is the same as saying it should provide behavioral, but also referential equality.
- hashCode() should map the reference into an integer

Autoboxing and Equality

We talked about primitive type and their object type equivalents. The object type implements equals() in the correct way

```
Integer x = new Integer(3);
Integer y = new Integer(3);
x.equals(y) → true
```

But there's a subtle problem here; == is overloaded. For reference types like Integer it implements referential equality:

```
x == y
→ False
```

But for primitive types, == implements behavioural equality

```
(int) x == (int) y
```

so you can't really use Integer interchangeably with int. The fact that Java automatically converts int and Integer (this is called autoboxing and autounboxing) can lead to subtle bugs! You have to be aware what the compile-time types of your expressions are.

Consider this:

```
Map<String, Integer> a = new HashMap(), b = new HashMap();
a.put("c", 130); // autoboxed to Integer
b.put("c", 130); // different Integer
a.get("c") == b.get("c") → False!
```