Lecture2

Recall our implementation of hailstone using array

```
int[] a = new int[100];
int i = 0;
int n = 5;
while (n != 1) {
    a[i] = n;
    i++;
    if (n % 2 == 0) {
        n = n / 2;
    }
    else {
        n = 3 * n + 1;
    }
}
a[i] = n;
```

What would happen if we tried an n that turned out to have a very long hailstone sequence?
It wouldn't fit in a length-100 array
We have a bug !
Would Java catch the bug :

a. statically
b. dynamically
c. not at all ?

答案是 b

```
public static List<Integer> hailstone(final int n) {
    final List<Integer> list = new ArrayList<Integer>();
    while (n != 1) {
        list.add(n);
        if (n % 2 == 0) {
            n = n / 2;
        }
        else {
            n = 3 * n + 1;
        }
    }
    list.add(n);
    return list;
}
```

A

B

There are two variables in our hailstone method.

Can we declare them `final`, or not?

答案是只有 B 可以用 final

Lecture 3

- You want to partition the input space of this integer square root function:

```
/**
 * @param x is a nonnegative integer
 * @return nearest integer to the square root of x
 */
public static int intSqrt(int x)
```

- Which one is a **good partition**?
  - Partition: `x < 0` and `x >= 0`
  - Partition: x is a perfect square and x is an integer > `0` but not a perfect square
  - Partition: `x = 0, x = 1, x = 5, x = 16`
  - Partition: `x even, x odd, x >= 100`

答案是第二个

In-Class Quiz 2

- Consider the following function and values:

```
/**
 * @param winsAndLosses is a string of length at most 5 consisting of 'W' or 'L' characters
 * @return the fraction of characters in winsAndLosses that are 'W'
 */
public static double winLossRatio(String winsAndLosses)
```

(i) "" (ii) "LLLLL" (iii) "WLWL" (iv) "WWWWW" (v) "xxxxx"

- Which are appropriate **boundary values** for testing this function ?
  - (i)
  - (ii), (iv), (v)
  - (i), (ii), (iv)
  - (i), (ii), (iv), (v)

答案是第三个

Factorial Quiz 1

What is factorial(4) ?

a) factorial(4) = 4
b) factorial(4) = 12
c) factorial(4) = 24

答案是 c

What is factorial(5) ?

a) factorial(5) = 5
b) factorial(5) = 24
c) factorial(5) = 120

答案是 c

Lecture4

In-class Quiz

- Which of the following are good times to re-run all your JUnit tests?
- Select one or more:
  - ❑ after rewriting a correct method to make it faster
  - ❑ when using a code coverage tool
  - ❑ after an attempt to fix a bug
  - ❑ before submitting your code to LM Autograder

答案是 abcd

Lecture5

In-Class Quiz 1

- Given the following specification :

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

select the **legal** behavior that you can then implement `find` with:

- ❑ if `arr` is empty, return `0`
- ❑ if `val` occurs twice in `arr`, set all values in `arr` to zero, then throw an exception
- ❑ if `val` does not occur in `arr`, pick random `index`, set value at `index` to `val`, return the `index`
- ❑ if `arr[0]` is `val`, continue search, if found another return the index; otherwise return `0`

答案是 abcd

In-Class Quiz 2

- Given the following specification :

```
static boolean isPalindrome(String word)
  requires: word contains only alphanumeric characters
  effects:  returns true if and only if word is a palindrome
```

which line of the Javadoc comment is ***problematic***:

```
/**
 * Check if a word is a palindrome.                                    (1)
 * A palindrome is a sequence of characters                            (2)
 * that reads the same forwards and backwards.                         (3)
 * @param String word to check, must contain only alphanumeric characters  (4)
 * @return true if and only if word is a palindrome                    (5)
 */
```

答案是 4 问题出在 string

Lecture6

In-Class Quiz 1

- We use `BirthdayBook` with the `lookup` method that `throws NotFoundException`
- Assume we have initialized the `birthdays` variable to point to a `BirthdayBook`, and that `"Makima"` is **not** in that birthday book
- What will happen with the following code:

```
try {
    LocalDate birthdate = birthdays.lookup("Makima");
}
System.out.println("done");
```

  - static error caused by incorrect syntax
  - static error caused by undeclared variable
  - dynamic error caused by `NotFoundException`
  - no errors and it prints `"done"`

答案是第一个

In-Class Quiz 2

- We use `BirthdayBook` with the `lookup` method that `throws NotFoundException`
- Assume we have initialized the `birthdays` variable to point to a `BirthdayBook`, and that `"Makima"` is **not** in that birthday book
- What will happen with the following code:

```
try {
    LocalDate birthdate = birthdays.lookup("Makima");
} catch (NotFoundException nfe) {
    birthdate = LocalDate.now();
}
System.out.println("done");
```

  - static error caused by incorrect syntax
  - static error caused by undeclared variable
  - dynamic error caused by `NotFoundException`
  - no errors and it prints `"done"`

答案是第二个

In-Class Quiz 3

- We use `BirthdayBook` with the `lookup` method that `throws NotFoundException`
- Assume we have initialized the `birthdays` variable to point to a `BirthdayBook`, and that `"Makima"` is **not** in that birthday book
- What will happen with the following code:

```
try {
    LocalDate birthdate = birthdays.lookup("Makima");
} catch (Exception NotFoundException) {
}
System.out.println("done");
```

  - static error caused by incorrect syntax
  - static error caused by undeclared variable
  - dynamic error caused by `NotFoundException`
  - no errors and it prints `"done"`

答案是第四个

Lecture8

In-Class Quiz 1

- Consider the following specification

```
static int find^CanBeMissing(int[] a, int val)
  requires: nothing
  effects:  returns index i such that a[i] == val, or -1 if no such i
```

and compare with:

```
static int find^OneOrMore,FirstIndex(int[] a, int val)
  requires: val occurs at least once in a
  effects:  returns lowest index i such that a[i] == val
```

- which is true about their ***precondition*** and ***postcondition*** ?

- ○ $\texttt{find}^{\texttt{CanBeMissing}}$ has a weaker precondition and a stronger postcondition compared to $\texttt{find}^{\texttt{OneOrMore,FirstIndex}}$
- ○ $\texttt{find}^{\texttt{CanBeMissing}}$ has a stronger precondition and a weaker postcondition compared to $\texttt{find}^{\texttt{OneOrMore,FirstIndex}}$
- ○ $\texttt{find}^{\texttt{CanBeMissing}}$ has a weaker precondition and a weaker postcondition compared to $\texttt{find}^{\texttt{OneOrMore,FirstIndex}}$
- ○ $\texttt{find}^{\texttt{CanBeMissing}}$ has a stronger precondition and a stronger postcondition compared to $\texttt{find}^{\texttt{OneOrMore,FirstIndex}}$

答案是第三个

- Which of the following is a sign of a good specification?
  - ❑ the specification is declarative
  - ❑ the specification is operational
  - ❑ the specification is as strong as possible
  - ❑ the specification is as weak as possible
  - ❑ the implementation is allowed to ignore invalid arguments in the specification
  - ❑ the implementation is allowed to use different algorithms depending on the arguments
  - ❑ the specification is making use of the reader's knowledge of the implementation

答案是第一个，第五个，第六个

Lecture9

In-Class Quiz 1

- But the resulting behavior will be very surprising to anybody who uses it!
  - Here is an example:

```java
// meanwhile, somewhere else in the code
public static void main(String[] args) {
    // ...
    List<Integer> myData = Arrays.asList(-5, -2, -10);
    System.out.println(sumOfAbsoluteValues(myData));
    System.out.println(sum(myData));
}
```

- What will this code print?

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

```java
/** @return the sum of the absolute values of the numbers in the list */
public static int sumOfAbsoluteValues(List<Integer> list) {
    // let's reuse sum(), because DRY, so first we take absolute values
    for (int i = 0; i < list.size(); ++i)
        list.set(i, Math.abs(list.get(i)));
    return sum(list);
}
```

```java
// meanwhile, somewhere else in the code...
public static void main(String[] args) {
    // ...
    List<Integer> myData = Arrays.asList(-5, -2, -10);
    System.out.println(sumOfAbsoluteValues(myData));
    System.out.println(sum(myData));
}
```

答案是 17 -17

- What is the output of `dropCourseA` on input `["a100", "a200", "a300"]` ?

```java
public static void dropCourseA(ArrayList<String> subjects) {
    MyIterator iter = new MyIterator(subjects);
    while (iter.hasNext()) {
        String subject = iter.next();
        if (subject.startsWith("a")) {
            subjects.remove(subject);
        }
    }
}
```

答案是[a200]

- Suppose you are trying to connect(5, 2) — which one is better :
  a. set 5's root to be a child of 2's root
  b. set 2's root to be a child of 5's root

答案是 a

Lecture10

In-Class Quiz 1

- `Map.keySet()` is a method from the Java library
  - below is the method's Javadoc documentation, look at its signature
- What kind of operation of an abstract data type is it?

keySet

`Set<K> keySet()`

Returns a `Set` view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own `remove` operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the `Iterator.remove`, `Set.remove`, `removeAll`, `retainAll`, and `clear` operations. It does not support the `add` or `addAll` operations.

Returns:
a set view of the keys contained in this map

  - creator
  - producer
  - observer
  - mutator

答案是第三个

In-Class Quiz 2.1

- What will be printed by those four lines?

```
public interface Animal {
  default void hello(Animal a) {
    print("hello animal"); }
  default void sniff(Animal a) {
    print("sniff animal"); }
  default void cool(Animal a) {
    print("cool animal"); }
}
```

```
public class Dog implements Animal {
  void sniff(Animal a) {
    print("sniff dog"); }
  void cool(Dog d) {
    print("cool dog"); }
}
```

```
Animal a = new Dog();
Dog d = new Dog();
a.hello(d);
a.sniff(d);
d.cool(d);
a.cool(d);
```

What is the output of this line ?

答案是 hello aninal

In-Class Quiz 2.2

- What will be printed by those four lines?

```
public interface Animal {
  default void hello(Animal a) {
    print("hello animal"); }
  default void sniff(Animal a) {
    print("sniff animal"); }
  default void cool(Animal a) {
    print("cool animal"); }
}
```

```
public class Dog implements Animal {
  void sniff(Animal a) {
    print("sniff dog"); }
  void cool(Dog d) {
    print("cool dog"); }
}
```

```
Animal a = new Dog();
Dog d = new Dog();
a.hello(d);
a.sniff(d);
d.cool(d);
a.cool(d);
```

What is the output of this line ?

答案是 sniff dog

In-Class Quiz 2.3

- What will be printed by those four lines?

```
public interface Animal {
  default void hello(Animal a) {
    print("hello animal"); }
  default void sniff(Animal a) {
    print("sniff animal"); }
  default void cool(Animal a) {
    print("cool animal"); }
}
```

```
public class Dog implements Animal {
  void sniff(Animal a) {
    print("sniff dog"); }
  void cool(Dog d) {
    print("cool dog"); }
}
```

```
Animal a = new Dog();
Dog d = new Dog();
a.hello(d);
a.sniff(d);
d.cool(d);
a.cool(d);
```

What is the output of this line ?

答案是 cool dog

In-Class Quiz 2.4

- What will be printed by those four lines?

```
public interface Animal {
  default void hello(Animal a) {
    print("hello animal"); }
  default void sniff(Animal a) {
    print("sniff animal"); }
  default void cool(Animal a) {
    print("cool animal"); }
}
```

```
public class Dog implements Animal {
  void sniff(Animal a) {
    print("sniff dog"); }
  void cool(Dog d) {
    print("cool dog"); }
}
```

```
Animal a = new Dog();
Dog d = new Dog();
a.hello(d);
a.sniff(d);
d.cool(d);
a.cool(d);
```

What is the output of this line ?

答案是 cool animal

Started on	Wednesday, 10 March 2021, 19:49
State	Finished
Completed on	Sunday, 14 March 2021, 17:14
Time taken	3 days 21 hours
Grade	**80.00** out of 150.00 (**53**%)

Question 1

Correct Mark 10.00 out of 10.00

In the buggy Java code below, is the bug caught by static checking, dynamic checking, or not at all?

```
int n = 5;
if (n) {
    System.out.println("n is " + n);
}
```

Select one:

- a. static checking
- b. dynamic checking
- c. no checking, resulting in wrong answer

Your answer is correct.

The correct answer is: static checking

Question 2

Incorrect Mark 0.00 out of 10.00

In the buggy Java code below, is the bug caught by static checking, dynamic checking, or not at all?

```
int bigNum = 200000;        // bigNum is 200,000
bigNum = bigNum * bigNum;   // bigNum should be 4 billion
```

Select one:

- a. static checking
- b. dynamic checking
- c. no checking, resulting in wrong answer

Your answer is incorrect.

The correct answer is: no checking, resulting in wrong answer

Question 3

Incorrect Mark 0.00 out of 10.00

In the buggy Java code below, is the bug caught by static checking, dynamic checking, or not at all?

```
// the probability of an event is prob = 1/5 = 0.2
double prob = 1/5;
```

Select one:

- a. static checking
- b. dynamic checking
- c. no checking, resulting in wrong answer

Your answer is incorrect.

The correct answer is: no checking, resulting in wrong answer

Question 4

Correct Mark 10.00 out of 10.00

In the buggy Java code below, is the bug caught by static checking, dynamic checking, or not at all?

```
int sum = 0;
int n = 0;
int average = sum/n;
```

Select one:

- a. static checking
- b. dynamic checking
- c. no checking, resulting in wrong answer

Your answer is correct.

The correct answer is: dynamic checking

Question 5

Incorrect Mark 0.00 out of 10.00

In the buggy Java code below, is the bug caught by static checking, dynamic checking, or not at all?

```
double sum = 7;
double n = 0;
double average = sum/n;
```

Select one:

- a. static checking
- b. dynamic checking
- c. no checking, resulting in wrong answer

Your answer is incorrect.

The correct answer is: no checking, resulting in wrong answer

Question 6

Correct | Mark 10.00 out of 10.00

Which is the correct snapshot diagram for:

final String name = "Erick";

Select one:

Your answer is correct.

The correct answer is:

Question 7

Incorrect Mark 0.00 out of 10.00

Choose the **incorrect** statement:

Select one:

- a. String is an immutable type.
- b. StringBuilder is a mutable type.
- c. final variable cannot be reassigned.
- d. object pointed by final variable cannot be mutated.
- e. List is a mutable type.

Your answer is incorrect.

The correct answer is: object pointed by final variable cannot be mutated.

Question 8

Incorrect Mark 0.00 out of 20.00

When you try to reassign a final variable, Java compiler will produce a compile error.

Therefore, final provides you immutable reference ✗ for immutable values ✗ .

Question 9

Incorrect Mark 0.00 out of 10.00

Rewrite the variable declaration below using Lists instead of arrays:

```
char[][] matrix;
```

Answer: List matrix = new ArrayList<>() ✗

The correct answer is: List<List<Character>> matrix ;

Question 10

Correct Mark 10.00 out of 10.00

Given a code:

```
List<Integer> list1 = new ArrayList<>();
list1.add(100);
list1.add(200);
final List<Integer> list2 = list1;
list1.add(300);
```

If we add a line of code below:

```
list2 = list1;
```

choose the **correct** statement:

Select one:

- a. there will be an error, detected by static checking.
- b. there will be an error, detected by dynamic checking.
- c. there is no error.

Your answer is correct.

The correct answer is: there will be an error, detected by static checking.

Question 11

Correct Mark 10.00 out of 10.00

Given a code:

```
List<Integer> list1 = new ArrayList<>();
list1.add(100);
list1.add(200);
final List<Integer> list2 = list1;
list1.add(300);
```

If we add a line of code below:

```
list1.set(1, 400);
```

choose the **correct** statement:

Select one:

- a. there will be an error, detected by static checking.
- b. there will be an error, detected by dynamic checking.
- c. there is no error.

Your answer is correct.

The correct answer is: there is no error.

Question 12

Correct Mark 10.00 out of 10.00

Given a code:

```
List<Integer> list1 = new ArrayList<>();
list1.add(100);
list1.add(200);
final List<Integer> list2 = list1;
list1.add(300);
```

If we add a line of code below:

```
list2.set(1, 400);
```

choose the **correct** statement:

Select one:

- a. there will be an error, detected by static checking.
- b. there will be an error, detected by dynamic checking.
- c. there is no error.

Your answer is correct.

The correct answer is: there is no error.

Question 13

Correct Mark 10.00 out of 10.00

Given a code:

```
List<Integer> list1 = new ArrayList<>();
list1.add(100);
list1.add(200);
final List<Integer> list2 = list1;
list1.add(300);
```

If we add a line of code below:

```
list2.set(3, 400);
```

choose the **correct** statement:

Select one:

- a. there will be an error, detected by static checking.
- b. there will be an error, detected by dynamic checking.
- c. there is no error.

Your answer is correct.

The correct answer is: there will be an error, detected by dynamic checking.

Question 14

Correct Mark 10.00 out of 10.00

Create a map named hostel with integer keys and string values, to store room number and tenant name pairs.

Then, add a key-value pair for a tenant named Alice in room number 777.

Select one:

- Map<Integer, String> hostel = new HashMap<>();
  hostel.add(777, "Alice");

- Map<Integer, String> hostel = new HashMap<>();
  hostel.put(777, "Alice");

- Map<String, Integer> hostel = new HashMap<>();
  hostel.add("Alice", 777);

- Map<String, Integer> hostel = new HashMap<>();
  hostel.put("Alice", 777);

- Map<String, int> hostel = new HashMap<>();
  hostel.add("Alice", 777);

- Map<String, int> hostel = new HashMap<>();
  hostel.put("Alice", 777);

Your answer is correct.

The correct answer is:

Map<Integer, String> hostel = new HashMap<>();

hostel.put(777, "Alice");

Finish review

Lab 2 Videos Jump to... Lab Exercise 2.1 Max Stret

Started on	Monday, 22 March 2021, 19:14
State	Finished
Completed on	Monday, 22 March 2021, 19:41
Time taken	27 mins
Grade	**90.00** out of 130.00 (**69**%)

Question 1

Correct Mark 10.00 out of 10.00

Somebody wrote a **bad** code that ***does not fail fast*** (from the Lecture 3):

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth;
}
```

Assume today is **January 3, 2019**;
which means that the correct *dayOfYear* for this date is 3,
since it's the third day of the year.

Now **another programmer** calls that method with arguments as follows:

```
dayOfYear(1, 3, 2019)
```

Choose the **correct** statement:

Select one:

- a. The programmer did not make a mistake.
  The method gave the right answer.
- b. The programmer made a mistake.
  The method gave the right answer, luckily.
- c. The programmer made a mistake.
  The method gave the wrong answer, quietly.
- d. The programmer made a mistake.

The method detected a static error.

e. The programmer made a mistake.
The method detected a dynamic error.

Your answer is correct.

The correct answer is: The programmer did not make a mistake.
The method gave the right answer.

Question 2

Incorrect Mark 0.00 out of 10.00

Now **another programmer** calls that method with arguments as follows:

```
dayOfYear(0, 3, 2019)
```

Choose the **correct** statement:

Select one:

a. The programmer did not make a mistake.
The method gave the right answer.

b. The programmer made a mistake.
The method gave the right answer, luckily.

c. The programmer made a mistake.
The method gave the wrong answer, quietly.

d. The programmer made a mistake.
The method detected a static error.

e. The programmer made a mistake.
The method detected a dynamic error.

Your answer is incorrect.

The correct answer is: The programmer made a mistake.
The method gave the right answer, luckily.

Question 3

Incorrect Mark 0.00 out of 10.00

Now **another programmer** calls that method with arguments as follows:

```
dayOfYear(3, 1, 2019)
```

Choose the **correct** statement:

Select one:

a. The programmer did not make a mistake.
The method gave the right answer.

b. The programmer made a mistake.
The method gave the right answer, luckily.

c. The programmer made a mistake.
The method gave the wrong answer, quietly.

d. The programmer made a mistake.
The method detected a static error.

e. The programmer made a mistake.

e. The programmer made a mistake.
The method detected a dynamic error.

Your answer is incorrect.

The correct answer is: The programmer made a mistake.
The method gave the wrong answer, quietly.

Question 4

Correct Mark 10.00 out of 10.00

Now **another programmer** calls that method with arguments as follows:

```
dayOfYear("January", 3, 2019)
```

Choose the **correct** statement:

Select one:

a. The programmer did not make a mistake.
The method gave the right answer.

b. The programmer made a mistake.
The method gave the right answer, luckily.

c. The programmer made a mistake.
The method gave the wrong answer, quietly.

d. The programmer made a mistake.
The method detected a static error.

e. The programmer made a mistake.
The method detected a dynamic error.

Your answer is correct.

The correct answer is: The programmer made a mistake.
The method detected a static error.

Question 5

Correct Mark 10.00 out of 10.00

Now **another programmer** calls that method with arguments as follows:

```
dayOfYear(2019, 1, 3)
```

Choose the **correct** statement:

Select one:

a. The programmer did not make a mistake.
The method gave the right answer.

b. The programmer made a mistake.
The method gave the right answer, luckily.

c. The programmer made a mistake.
The method gave the wrong answer, quietly.

d. The programmer made a mistake.
The method detected a static error.

e. The programmer made a mistake.
The method detected a dynamic error.

Your answer is correct.

The correct answer is: The programmer made a mistake.
The method gave the wrong answer, quietly.

Question 6

Incorrect Mark 0.00 out of 10.00

We should not use global variables.

Making a variable into a constant can eliminate the risk of global variables.

What keyword should be added to such global variables to make them constants ?

Answer: local ✗

The correct answer is: final

Question 7

Incorrect Mark 0.00 out of 10.00

In the 1990s, the Ariane 5 launch vehicle, designed and built for the European Space Agency, self-destructed 37 seconds after its first launch.

The reason was a control software bug that went undetected. The Ariane 5's guidance software was reused from the Ariane 4, which was a slo rocket. When the velocity calculation converted from a 64-bit floating point number (a double in Java terminology, though this software wasn't written in Java) to a 16-bit signed integer (a short), it overflowed the small integer and caused an exception to be thrown.

The exception handler had been disabled for efficiency reasons, so the guidance software crashed. Without guidance, the rocket crashed too. cost of the failure was $1 billion.

What ideas does this story demonstrate?
Choose the **correct** option.

Select one:

- a. High-quality safety-critical software cannot have residual bugs.
- b. Testing all possible inputs is the best solution to this problem.
- c. Static checking could have detected this bug.
- d. Software exhibits discontinuous behavior, unlike many physically-engineered systems.

Your answer is incorrect.

The correct answer is: Software exhibits discontinuous behavior, unlike many physically-engineered systems.

Question 8

Correct Mark 10.00 out of 10.00

Consider the following specification:

```
 * Reverses the end of a string.
 *
 *                          012345                    012345
 * For example: reverseEnd("Hello, world", 5) returns "Hellodlrow ,"
 *                                <----->                   <----->
 *
 * 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 text with the substring from start to the end of the string reversed
 */
public static String reverseEnd(String text, int start)
```

Which of the following is the **best partitions** for the **start** parameter?

Select one:

- a. start = 0, 0 < start < text.length(), start = text.length()
- b. start = 0, start = 5, start = 100
- c. start < 0, start = 0, start > 0
- d. start < text.length(), start = text.length(), start > text.length()

Your answer is correct.

The correct answer is: start = 0, 0 < start < text.length(), start = text.length()

Question 9

Correct Mark 10.00 out of 10.00

Consider the following specification:

```
/**
 * Reverses the end of a string.
 *
 *                          012345                    012345
 * For example: reverseEnd("Hello, world", 5) returns "Hellodlrow ,"
 *                                <----->                   <----->
 *
 * 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 text with the substring from start to the end of the string reversed
 */
public static String reverseEnd(String text, int start)
```

Which of the following is the **best partitions** for the **text** parameter?

Select one:

- a. text.length() = 0; text.length()-start is odd; text.length()-start is even
- b. text contains some letters; text contains no letters, but some numbers; text contains neither letters nor numbers
- c. text.length() < 0; text.length() = 0; text.length() > 0
- d. text is every possible string from length 0 to 100

Your answer is correct.

The correct answer is: text.length() = 0; text.length()-start is odd; text.length()-start is even

Question 10

Correct Mark 10.00 out of 10.00

Select the **incorrect** statement about Covering the Partitions:

Select one:

- a. For the BigInteger `multiply` example, using cover each part approach, we can choose 5 test cases.
- b. The full cartesian approach may not be the best because it could produce too many and redundant test cases.
- c. The cover each part approach may not be the best because the function may behave differently for a certain combination of inputs.
- d. For the `max` example, using full Cartesian approach, we can choose less than 75 test cases because not all combinations are possible.

Your answer is correct.

The correct answer is: For the BigInteger `multiply` example, using cover each part approach, we can choose 5 test cases.

Question 11

Correct Mark 10.00 out of 10.00

In designing the test suite for the Recursive Reverse String problem,
we include the empty string as a test case.

Which testing principle do we use?

Select one:

- a. Choose the boundaries in the partition.
- b. Divide the input space into subdomains.
- c. Choose one test case from each subdomain.
- d. Choose one test case from every legal combination of the partition.

Your answer is correct.

The correct answer is: Choose the boundaries in the partition.

Question 12

Correct Mark 10.00 out of 10.00

When you write the recursive step of your recursive method,
which part of your code that must be reached by it ?

Answer: base case

The correct answer is: the base case

Question 13

Correct Mark 10.00 out of 10.00

In solving a problem recursively, you can define a/an helper method ✔ that uses an arbitrary number of parameters.

Finish review

Lab 3 Recording Jump to... Lab Exercise 3.1 Check Substri

Started on	Thursday, 25 March 2021, 14:05
State	Finished
Completed on	Thursday, 25 March 2021, 14:46
Time taken	41 mins 24 secs
Grade	**63.33** out of 110.00 (**58**%)

Question 1

Incorrect Mark 0.00 out of 10.00

Using your favorite code coverage tool, you add test cases one-by-one, until all reachable statements in your code have been executed at least once.

Which coverage guarantee your code has now?

Select one:

- a. Statement coverage
- b. Branch coverage
- c. Path coverage
- d. Unit coverage

Your answer is incorrect.

The correct answer is: Statement coverage

Question 2

Correct Mark 10.00 out of 10.00

Consider the following method:

```
/**
 * Sort a list of integers in nondecreasing order.  Modifies the list so that
 * values.get(i) <= values.get(i+1) for all 0<=i<values.length()-1
 */
public static void sort(List<Integer> values) {
    // choose a good algorithm for the size of the list
    if (values.length() < 10) {
        radixSort(values);
    } else if (values.length() < 1000*1000*1000) {
        quickSort(values);
    } else {
        mergeSort(values);
    }
}
```

Which test case of the following test cases are likely to be **a boundary value** produced by **white box testing**?

Select one:

a. [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

b. the empty list

c. [0, 0, 1, 0, 0, 0, 0]

d. [1, 2, 3]

Your answer is correct.

The correct answer is: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

Question 3

Correct Mark 10.00 out of 10.00

After fixing a bug that caused test case X fail,

you need to rerun all your JUnit tests, not just test case X.

Select one:

True ✔

False

The correct answer is 'True'.

Question 4

Incorrect Mark 0.00 out of 10.00

Which one of these testing activities follows the principle of ***regression testing***?

Select one:

a. Changes should be tested against all inputs that induced bugs in earlier versions of the code

b. Every component in your code should have an associated set of tests that exercises all the corner cases in its specification

c. Tests should be written before you write the code as a way of checking your understanding of the specification

d. When a new test exposes a bug, you should run it on all previous versions of the code until you find the version where the bug was introduced

Your answer is incorrect.

The correct answer is: Changes should be tested against all inputs that induced bugs in earlier versions of the code

Question 5

Partially correct Mark 13.33 out of 20.00

Which of these techniques are useful for choosing test cases in test-first programming, ***before*** any code is written?

Select one or more:

- [x] Partitioning
- [x] Boundaries
- [x] Black box
- [ ] Regression
- [ ] Coverage
- [ ] White box
- [x] Integration

Your answer is partially correct.

You have selected too many options.

The correct answers are: Partitioning, Boundaries, Black box

Question 6

Correct | Mark 10.00 out of 10.00

Choose the **correct** statement about a regression test case.

Select one:

- a. A regression test case comes from the discovery of a bug
- b. A regression test case is chosen from the partitions
- c. A regression test case can come out of black-box testing
- d. A regression test case can come out of white-box testing

Your answer is correct.

The correct answer is: A regression test case comes from the discovery of a bug

Question 7

Incorrect | Mark 0.00 out of 10.00

As a temporary substitute for a method that is not yet to be developed, you write a code to simulate the method's functionality. The method can then be called by another method that you want to test.

Such method is called a/an unit tessting ✗ .

Question 8

Correct | Mark 10.00 out of 10.00

Which button to click to get the Java Visualizer run the next line of your code and show the subsequent visualization?

Select one:

a. Step Into

b. Step Over

c. Step Out

d. Step Off

e. Step On

Your answer is correct.

The correct answer is: Step Into

Question 9

Incorrect Mark 0.00 out of 10.00

Write one line of Java code that *declares* a MyList pointer named **p** and *initializes* it to the current MyList object.

Do not forget to end it with a semicolon.

Answer: MyList p = new MyList(100, null);

The correct answer is: MyList p = this;

Question 10

Correct Mark 10.00 out of 10.00

What is the println result of :

```
MyList3 list = new MyList3(100, null);
list = new MyList3(200, list);
list = new MyList3(300, list);

System.out.println(list.get(0));
```

Answer: 300

The correct answer is: 300

Finish review

Started on	Wednesday, 31 March 2021, 20:38
State	Finished
Completed on	Sunday, 4 April 2021, 15:25
Time taken	3 days 18 hours
Grade	**53.33** out of 110.00 (**48**%)

Question 1

Incorrect Mark 0.00 out of 10.00

Consider the two methods to find the value `val` in an integer array `a` below.

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

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

If clients only care about calling the find method when they know that `val` *always occurs exactly once* in `a`, `findFirst` and `findLast` are behaviorally equivalent.

Select one:

- True
- False ✗

The correct answer is 'True'.

Question 2

Incorrect Mark 0.00 out of 10.00

Consider the two methods to find the value `val` in an integer array `a` below.

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

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

If clients only care that the find method should return any index `i` such that `a[i] == val`, if `val` is in `a`;
and any integer `j` where `j` is ***not*** a valid index of array `a`, otherwise;
then `findFirst` and `findLast` are behaviorally equivalent.

Select one:

- True
- False ✗

The correct answer is 'True'.

Question 3

Correct | Mark 10.00 out of 10.00

Suppose we're working on the method below:

```
/**
 * Requires: tiles has length 7 & contains only uppercase letters.
 *           crossings contains only uppercase letters, without duplicates.
 * Effects: Returns a list of words where each word can be made by taking
 *          letters from tiles and at most 1 letter from crossings.
 */
public static List<String> scrabble(String tiles, String crossings) {
    if (tiles.length() != 7) { throw new RuntimeException(); }
    return new ArrayList<>();
}
```

Which one is a part of the *postcondition* of `scrabble`?

Select one:

- a. scrabble returns a list of strings
- b. tiles has only uppercase letters
- c. crossings has no duplicates
- d. scrabble takes two arguments

Your answer is correct.

The correct answer is: scrabble returns a list of strings

Question 4

Correct | Mark 10.00 out of 10.00

Suppose we're working on the method below:

```
/**
 * Requires: tiles has length 7 & contains only uppercase letters.
 *           crossings contains only uppercase letters, without duplicates.
 * Effects: Returns a list of words where each word can be made by taking
 *          letters from tiles and at most 1 letter from crossings.
```

```
 */
public static List<String> scrabble(String tiles, String crossings) {
    if (tiles.length() != 7) { throw new RuntimeException(); }
    return new ArrayList<>();
}
```

Which one is **not** a part of the *precondition* of `scrabble`?

Select one:

- a. scrabble returns an empty ArrayList
- b. tiles has length 7
- c. crossings is a string of uppercase letters
- d. scrabble's arguments are of type String and String

Your answer is correct.

The correct answer is: scrabble returns an empty ArrayList

Question 5

Incorrect | Mark 0.00 out of 10.00

Suppose we're working on the method below:

```
/**
 * Requires: tiles has length 7 & contains only uppercase letters.
 *           crossings contains only uppercase letters, without duplicates.
 * Effects: Returns a list of words where each word can be made by taking
 *          letters from tiles and at most 1 letter from crossings.
 */
public static List<String> scrabble(String tiles, String crossings) {
    if (tiles.length() != 7) { throw new RuntimeException(); }
    return new ArrayList<>();
}
```

Which one is the part of the spec that are ***checked statically*** by Java?

Select one:

- a. scrabble takes two arguments
- b. tiles is a string of uppercase letters
- c. crossings has no duplicates
- d. when tiles.length() != 7, scrabble throws a RuntimeException

Your answer is incorrect.

The correct answer is: scrabble takes two arguments

Question 6

Incorrect | Mark 0.00 out of 10.00

Which of the following is **not** part of a method's specification?

Select one:

- a. restrictions on used data types
- b. return type
- c. restrictions on return values

c. restrictions on return values

d. number of arguments

e. argument types

f. restrictions on argument values

Your answer is incorrect.

The correct answer is: restrictions on used data types

Question 7

Incorrect Mark 0.00 out of 10.00

Alice writes the following code:

```
public static int gcd(int a, int b) {
    if (a > b) {
        return gcd(a-b, b);
    } else if (b > a) {
        return gcd(a, b-a);
    }
    return a;
}
```

Bob writes the following test:

```
@Test public void gcdTest() {
    assertEquals(6, gcd(24, 54));
}
```

Which of the following statement is **incorrect** ?

Select one:

a. If Alice adds `a > 0` to the precondition, Bob should test negative values of a

b. If Alice does not add `a > 0` to the precondition, Bob should test negative values of a

c. Alice should write `a > 0, b > 0` in the precondition of `gcd`

d. Alice should not write `a and b are integers` in the precondition of `gcd`

Your answer is incorrect.

The correct answer is: If Alice adds `a > 0` to the precondition, Bob should test negative values of a

Question 8

Partially correct Mark 13.33 out of 20.00

Given the following specification :

```
static int find(int[] arr, int val)
    requires: arr[0] == val
    effects:  returns index i such that arr[i] == val
```

Which are the valid test cases for `find` ?

Select one or more:

- [x] `find([ 1, 2, 3 ], 1)` must return `0`
- [ ] `find([ 4, 4, 5 ], 4)` must return `0`

- [ ] `find([ 4, 4, 5 ], 4)` must return `1`
- [ ] `find([ 6, 7, 8 ], 2)` throws an exception
- [x] `find([ 3 ], 3)` must return `0`
- [x] `find([ 4 ], 5)` must not return `0`

Your answer is partially correct.
You have selected too many options.
The correct answers are: `find([ 1, 2, 3 ], 1)` must return `0`, `find([ 3 ], 3)` must return `0`

Question 9

Correct | Mark 10.00 out of 10.00

What is a condition that must be preserved and guaranteed to be true during a method's execution called ?

Answer: Invariant ✔

The correct answer is: invariant

Question 10

Correct | Mark 10.00 out of 10.00

To allow types such as Integer, String, and user-defined types to be a parameter to methods, classes, and interfaces, we use generics ✔

Using it, we can create classes that work with different data types.

Finish review

Lab 5 Recording | Jump to... | Lab Exercise 5.1 LLDeque EMP

Started on	Thursday, 8 April 2021, 15:49
State	Finished
Completed on	Thursday, 8 April 2021, 17:07
Time taken	1 hour 18 mins
Grade	**70.00** out of 130.00 (**54**%)

Question 1

Correct Mark 20.00 out of 20.00

Which of the following **cannot** be null?

Select one or more:

- [x] char c;
- [ ] static final String str;
- [ ] int[] arr;
- [ ] Double d;
- [ ] final BackAccount myBankAccount;
- [ ] String name;
- [x] double d;

Your answer is correct.

The correct answers are: char c;,
double d;

Question 2

Correct Mark 10.00 out of 10.00

Given the following code :

```
public static String nope() {
    return null;            // (1)
}

public static void main(String[] args) {
    String a = nope();      // (2)
    String b = null;        // (3)
    if (a.length() > 0) {   // (4)
        b = a;              // (5)
    }
    return b;               // (6)
}
```

Which line contains a static error ?

Which line contains a static error ?

Select one:

- (1)
- (2)
- (3)
- (4)
- (5)
- (6)

Your answer is correct.

The correct answer is: (6)

Question 3

Correct | Mark 10.00 out of 10.00

Given the same code from Question 2 above :

```
public static String nope() {
    return null;            // (1)
}

public static void main(String[] args) {
    String a = nope();      // (2)
    String b = null;        // (3)
    if (a.length() > 0) {   // (4)
        b = a;              // (5)
    }
    return b;               // (6)
}
```

Suppose you have commented out the line causing the static error in Question 2.

Now, which line contains a dynamic error ?

Select one:

- (1)
- (2)
- (3)
- (4)
- (5)
- (6)

Your answer is correct.

The correct answer is: (4)

Question 4

Incorrect | Mark 0.00 out of 10.00

Suppose we're building a robot and we want to specify the method

```
public static List<Point> findPath(Point initial, Point goal)
```

which is responsible for path-finding: determining a sequence of `Point`s that the robot should move through to navigate from `initial` to `goal`, past any obstacles that might be in the way.

In the postcondition, we say that `findPath` will search for paths only up to a bounded length (set elsewhere), and that ***it will throw an exception if it fails to find one***.

Which exception is the best exception and its type to create, according to Lecture 6?

Select one:

- a. a checked `PathNotFoundException`
- b. an unchecked `PathNotFoundException`
- c. a checked `NoPathException`
- d. an unchecked `NoPathException`

Your answer is incorrect.

The correct answer is: a checked `PathNotFoundException`

Question 5

Incorrect Mark 0.00 out of 10.00

Suppose we define a checked exception for the method `findPath`.
What will we choose as our superclass?

Select one:

- a. Exception
- b. Throwable
- c. Error
- d. RuntimeException

Your answer is incorrect.

The correct answer is: Exception

Question 6

Incorrect Mark 0.00 out of 10.00

Suppose we define an unchecked exception for the method `findPath`.
What will we choose as our superclass?

Select one:

- a. Exception

b. Throwable

c. Error

d. RuntimeException

Your answer is incorrect.

The correct answer is: RuntimeException

Question 7

Incorrect Mark 0.00 out of 20.00

Consider this code below for analyzing some `Thing` objects:

```
static List<Thing> allTheThings;

static void analyzeEverything() {
    analyzeThings();
}

static void analyzeThings() {
    try {
        for (Thing t : allTheThings) {
            analyzeOneThing(t);
        }
    } catch (AnalysisException ae) {
        return;
    }
}

static void analyzeOneThing(Thing t) throws AnalysisException {
    // ...
    // ... maybe go past the end of a list
    // ...
}
```

Note that `IndexOutOfBoundsException`, `NullPointerException`, and `OutOfMemoryError` are unchecked exceptions and `AnalysisException` is a checked exception.

Which exception could be thrown by a call to `analyzeEverything`?

Select one or more:

- [x] `AnalysisException`
- [ ] `IndexOutOfBoundsException`
- [ ] `NullPointerException`
- [ ] `OutOfMemoryError`

Your answer is incorrect.

The correct answers are: `IndexOutOfBoundsException`, `NullPointerException`, `OutOfMemoryError`

Question 8

Partially correct Mark 10.00 out of 20.00

If we want to construct a different object with the same values as the input object, we use a/an other ✗ that performs a/an

deep copy ✔ instead of a shallow copy.

Question 9

Correct | Mark 10.00 out of 10.00

Write one line of Java code that throws an `IllegalArgumentException` object with a message `"n must not be even"` to complete the if statement below :

```
if (n % 2 == 0) {
    // your code here

}
```

Do not forget to end it with a semicolon.

Answer: throw new IllegalArgumentException("n must not be even"); ✔

The correct answer is: throw new IllegalArgumentException("n must not be even");

Question 10

Correct | Mark 10.00 out of 10.00

When we throw an `IllegalArgumentException` object within a method, that method must advertise it in the method signature.

Select one:

- True
- False ✔

The correct answer is 'False'.

Finish review

Started on	Friday, 23 April 2021, 14:03
State	Finished
Completed on	Tuesday, 27 April 2021, 15:24
Time taken	4 days 1 hour
Grade	**40.00** out of 110.00 (**36**%)

Question 1

Incorrect Mark 0.00 out of 10.00

Consider the following implementation:

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

and this specification of `find`:

```
static int find(int[] arr, int val)
  requires: nothing
  effects: returns largest index i such that
           arr[i] == val, or -1 if no such i
```

Which inputs demonstrates that `findFirst` does **not** satisfy this spec?

Select one or more:

- [ ] [1, 2, 2], 2
- [ ] [1, 2, 3], 2
- [ ] [1, 2, 3], 4
- [x] none of all others, `findFirst` does satisfy this spec!

Your answer is incorrect.

The correct answers are: [1, 2, 2], 2, [1, 2, 3], 4

Question 2

Incorrect Mark 0.00 out of 10.00

Consider the following implementation:

```
static int findLast(int[] arr, int val) {
    for (int i = arr.length-1; i >= 0; i--) {
```

```
    if (arr[i] == val) return i;
  }
  return -1;
}
```

and this specification of `find`:

```
static int find(int[] arr, int val)
  requires: nothing
  effects: returns largest index i such that
           arr[i] == val, or -1 if no such i
```

Which input demonstrates that `findLast` does **not** satisfy this spec?

Select one:

- a. [1, 2, 2], 2
- b. [1, 2, 3], 2
- c. [1, 2, 3], 4
- d. none of all others, `findLast` does satisfy this spec!

Your answer is incorrect.

The correct answer is: none of all others, `findLast` does satisfy this spec!

Question 3

Incorrect | Mark 0.00 out of 10.00

For each spec below, which one is **not** deterministic (underdetermined) ?

Select one:

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

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

c.
```
static int find(int[] arr, int val)
  requires: nothing
  effects: returns largest index i such that arr[i] == val, or -1 if no such i
```

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

Your answer is incorrect.

The correct answer is:

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

Question 4

Incorrect Mark 0.00 out of 10.00

Given this specification:

```
static String join(String delimiter, String[] elements)
  effects: append together the strings in elements, but at each step,
           if there are more elements left, insert delimiter
```

Rewrite the spec so it is declarative, **not** operational.

Select one:

a.
```
effects: returns elements joined together with copies of delimiter, i.e.
         elements[0] + delimiter + elements[1] + delimiter +
         ... + delimiter + elements[elements.length-1]
```

b.
```
effects: returns the result of adding all elements to a
         new StringJoiner(delimiter)
```

c.
```
effects: returns the result of looping through elements and
         alternately appending an element and the delimiter
```

d.
```
effects: returns the result of recursive calls on the elements and
         while concatenating the delimiter
```

Your answer is incorrect.

The correct answer is:

```
effects: returns elements joined together with copies of delimiter, i.e.
         elements[0] + delimiter + elements[1] + delimiter +
         ... + delimiter + elements[elements.length-1]
```

Question 5

Incorrect Mark 0.00 out of 10.00

When a specification is strengthened :

Select one:

a. fewer implementations satisfy it, and more clients can use it

b. fewer implementations satisfy it, and fewer clients can use it

c. more implementations satisfy it, and fewer clients can use it

d. more implementations satisfy it, and more clients can use it

Your answer is incorrect.

The correct answer is: fewer implementations satisfy it, and more clients can use it

Question 6

Correct Mark 10.00 out of 10.00

Which of the following is **false** about a pair of specifications *A* and *B*?

Select one:

Select one:

- a. *A* can be stronger than *B* and have a stronger precondition
- b. *A* can be stronger than *B* and have a weaker precondition
- c. *A* can be stronger than *B* and have the same precondition
- d. *A* can be incomparable to *B*

Your answer is correct.

The correct answer is: *A* can be stronger than *B* and have a stronger precondition

Question 7

Incorrect | Mark 0.00 out of 10.00

Here are the `find` specifications from Lecture 8 :

```
static int find^ExactlyOne(int[] a, int val)
  requires: val occurs exactly once in a
  effects:  returns index i such that a[i] == val
```

```
static int find^OneOrMore,AnyIndex(int[] a, int val)
  requires: val occurs at least once in a
  effects:  returns index i such that a[i] == val
```

```
static int find^OneOrMore,FirstIndex(int[] a, int val)
  requires: val occurs at least once in a
  effects:  returns lowest index i such that a[i] == val
```

```
static int find^CanBeMissing(int[] a, int val)
  requires: nothing
  effects:  returns index i such that a[i] == val,
            or -1 if no such i
```

We already know that $\texttt{find}^{\texttt{OneOrMore,FirstIndex}}$ is stronger than $\texttt{find}^{\texttt{OneOrMore,AnyIndex}}$, which is stronger than $\texttt{find}^{\texttt{ExactlyOne}}$.

Where is $\texttt{find}^{\texttt{ExactlyOne}}$ on the diagram ?

Select one:

a.

b.

c.

d.

Your answer is incorrect.

The correct answer is:

Question 8

Incorrect Mark 0.00 out of 10.00

Here are the `find` specifications from Lecture 8 :

```
static int find^ExactlyOne(int[] a, int val)
  requires: val occurs exactly once in a
  effects:  returns index i such that a[i] == val
```

```
static int find^OneOrMore,AnyIndex(int[] a, int val)
  requires: val occurs at least once in a
  effects:  returns index i such that a[i] == val
```

```
static int find^OneOrMore,FirstIndex(int[] a, int val)
  requires: val occurs at least once in a
  effects:  returns lowest index i such that a[i] == val
```

```
static int find^CanBeMissing(int[] a, int val)
  requires: nothing
  effects:  returns index i such that a[i] == val,
            or -1 if no such i
```

We already know that $\texttt{find}^{\texttt{OneOrMore,FirstIndex}}$ is stronger than $\texttt{find}^{\texttt{OneOrMore,AnyIndex}}$, which is stronger than $\texttt{find}^{\texttt{ExactlyOne}}$.

How does $\texttt{find}^{\texttt{CanBeMissing}}$ compare to $\texttt{find}^{\texttt{ExactlyOne}}$?

Select one:

- a. $\texttt{find}^{\texttt{CanBeMissing}}$ is stronger than $\texttt{find}^{\texttt{ExactlyOne}}$
- b. $\texttt{find}^{\texttt{CanBeMissing}}$ is weaker than $\texttt{find}^{\texttt{ExactlyOne}}$
- c. $\texttt{find}^{\texttt{CanBeMissing}}$ and $\texttt{find}^{\texttt{ExactlyOne}}$ are incomparable
- d. none of the options is correct

Your answer is incorrect.

The correct answer is: $\texttt{find}^{\texttt{CanBeMissing}}$ is stronger than $\texttt{find}^{\texttt{ExactlyOne}}$

Question 9

Correct Mark 10.00 out of 10.00

Here are the find specifications from Lecture 8 :

```
static int find^ExactlyOne(int[] a, int val)
  requires: val occurs exactly once in a
  effects:  returns index i such that a[i] == val
```

```
static int find^OneOrMore,AnyIndex(int[] a, int val)
  requires: val occurs at least once in a
  effects:  returns index i such that a[i] == val
```

```
static int find^OneOrMore,FirstIndex(int[] a, int val)
  requires: val occurs at least once in a
  effects:  returns lowest index i such that a[i] == val
```

```
static int find^CanBeMissing(int[] a, int val)
  requires: nothing
  effects:  returns index i such that a[i] == val,
            or -1 if no such i
```

We already know that $\texttt{find}^{\texttt{OneOrMore,FirstIndex}}$ is stronger than $\texttt{find}^{\texttt{OneOrMore,AnyIndex}}$, which is stronger than $\texttt{find}^{\texttt{ExactlyOne}}$.

Where is $\texttt{find}^{\texttt{CanBeMissing}}$ on the diagram ?

Select one:

a.

b.

c.

d.

e.

Your answer is correct.

The correct answer is: find OneOrMore,AnyIndex

Question 10

Correct Mark 10.00 out of 10.00

In our ARList implementation, we use a technique called resizing array that doubles the size of the array whenever it is full.

Question 11

Correct Mark 10.00 out of 10.00

You want to use a generic array using casting in your implementation of a data structure.
For example, you write the following line in your constructor or your method:

```
T[] elements = (T[]) new Object[numOfElements];
```

Write the annotation that you need to write before the constructor or the method:

Answer: @SuppressWarnings("unchecked")

The correct answer is: @SuppressWarnings("unchecked")

Finish review

Lab 8 Recording Jump to... Lab Exercise 8.1 ARDeque EMP

Started on	Tuesday, 4 May 2021, 13:42
State	Finished
Completed on	Tuesday, 4 May 2021, 13:44
Time taken	1 min 53 secs
Marks	60.00/150.00
Grade	**64.00** out of 160.00 (**40**%)

Question 1

Incorrect Mark 0.00 out of 10.00

Consider the following code, executed in order:

```
char text0 = 'a';
final char text1 = vowel0;

String text2 = text1 + "eiou";
final String text3 = text2;

char[] text4 = new char[] { text0, 'e', 'i', 'o', 'u' };
final char[] text5 = text4;
```

Which of the following statements are legal Java,
that is, produce **no** compiler error if placed ***after*** the code above?

Select one:

- a. text2 = text3;
- b. text1 = text0;
- c. text5 = text4;
- d. text3 = text2;

Your answer is incorrect.

The correct answer is: text2 = text3;

Question 2

Correct Mark 10.00 out of 10.00

Consider the following code, executed in order:

```
char text0 = 'a';
final char text1 = vowel0;

String text2 = text1 + "eiou";
final String text3 = text2;
```

```
char[] text4 = new char[] { text0, 'e', 'i', 'o', 'u' };
final char[] text5 = text4;
```

Which of the following statements are legal Java,
that is, produce **no** compiler error if placed ***after*** the code above?

Select one:

- a. text5[0] = 'x';
- b. text2[0] = 'x';
- c. text3[0] = 'x';
- d. text0[0] = 'x';

Your answer is correct.

The correct answer is: text5[0] = 'x';

Question 3

Incorrect Mark 0.00 out of 10.00

Consider this (incomplete) method:

```
/**
 * Solves quadratic equation ax^2 + bx + c = 0.
 *
 * @param a quadratic coefficient, requires a != 0
 * @param b linear coefficient
 * @param c constant term
 * @return a list of the real roots of the equation
 */
public static List<Double> quadraticRoots(final int a, final int b, final int c) {
    List<Double> roots = new ArrayList<Double>();
    // A
    ... // compute roots
    // B
    return roots;
}
```

What assertion would be reasonable to write at position **A** (*before* computing the roots) ?

Select one:

- a. assert a != 0;
- b. assert b != 0;
- c. assert c != 0;
- d. assert roots.size() >= 0;
- e. assert roots.size() <= 2;

Your answer is incorrect.

The correct answer is: assert a != 0;

Question 4

Incorrect Mark 0.00 out of 10.00

Consider this (incomplete) method:

Consider this (incomplete) method:

```
/**
 * Solves quadratic equation ax^2 + bx + c = 0.
 *
 * @param a quadratic coefficient, requires a != 0
 * @param b linear coefficient
 * @param c constant term
 * @return a list of the real roots of the equation
 */
public static List<Double> quadraticRoots(final int a, final int b, final int c) {
    List<Double> roots = new ArrayList<Double>();
    // A
    ... // compute roots
    // B
    return roots;
}
```

What assertion would be reasonable to write at position **B** (*after* computing the roots) ?

Select one:

- a. assert a != 0;
- b. assert b != 0;
- c. assert c != 0;
- d. assert roots.size() >= 0;
- e. assert roots.size() <= 2;

Your answer is incorrect.

The correct answer is: assert roots.size() <= 2;

Question 5

Correct | Mark 10.00 out of 10.00

Consider the following code, which is *missing* some variable declarations :

```
class Apartment {

    Apartment(String newAddress) {
        this.address = newAddress;
        this.roommates = new HashSet<Person>();
    }

    String getAddress() {
        return address;
    }

    void addRoommate(Person newRoommate) {
        roommates.add(newRoommate);
        if (roommates.size() > MAXIMUM_OCCUPANCY) {
            roommates.remove(newRoommate);
            throw new TooManyPeopleException();
        }
    }

    int getMaximumOccupancy() {
        return MAXIMUM_OCCUPANCY;
    }
}
```

Which one is the best declaration for the `roommates` variable?

Select one:

- a. final Set<Person> roommates;
- b. List<Person> roommates;
- c. Set<Person> roommates;

d. HashSet<Person> roommates;

Your answer is correct.

The correct answer is: final Set<Person> roommates;

Question 6

Correct Mark 10.00 out of 10.00

Consider the following code, which is *missing* some variable declarations :

```
class Apartment {

    Apartment(String newAddress) {
        this.address = newAddress;
        this.roommates = new HashSet<Person>();
    }

    String getAddress() {
        return address;
    }

    void addRoommate(Person newRoommate) {
        roommates.add(newRoommate);
        if (roommates.size() > MAXIMUM_OCCUPANCY) {
            roommates.remove(newRoommate);
            throw new TooManyPeopleException();
        }
    }

    int getMaximumOccupancy() {
        return MAXIMUM_OCCUPANCY;
    }
}
```

Which one is the best declaration for the `MAXIMUM_OCCUPANCY` variable?

Select one:

- a. static final int MAXIMUM_OCCUPANCY = 8;
- b. final int MAXIMUM_OCCUPANCY = 8;
- c. static int MAXIMUM_OCCUPANCY = 8;
- d. int MAXIMUM_OCCUPANCY = 8;
- e. public int MAXIMUM_OCCUPANCY = 8;
- f. public static int MAXIMUM_OCCUPANCY = 8;

Your answer is correct.

The correct answer is: static final int MAXIMUM_OCCUPANCY = 8;

Question 7

Incorrect Mark 0.00 out of 10.00

Consider the following snapshot diagram:

Is it possible that a client with the variable terrarium could modify the Turtle in red?

Select one:

- a. No, because the "Turtle" is immutable
- b. Yes, because all the references between "terrarium" and the "Turtle" are mutable
- c. Yes, because of some reference between "terrarium" and the "Turtle" that is mutable
- d. Yes, because the "Turtle" is mutable
- e. No, because of some reference between "terrarium" and the "Turtle" that is immutable
- f. No, because all the references between "terrarium" and the "Turtle" are immutable

Your answer is incorrect.

The correct answer is: No, because the "Turtle" is immutable

Question 8

Incorrect | Mark 0.00 out of 10.00

Consider the following snapshot diagram:

Is it possible that a client with the variable george could modify the Gecko in blue?

Select one:

- a. No, because the "Gecko" is immutable
- b. Yes, because all the references between "george" and the "Gecko" are mutable
- c. Yes, because of some reference between "george" and the "Gecko" that is mutable

d. Yes, because the "Gecko" is mutable

e. No, because of some reference between "george" and the "Gecko" that is immutable

f. No, because all the references between "george" and the "Gecko" are immutable

Your answer is incorrect.

The correct answer is: No, because the "Gecko" is immutable

Question 9

Incorrect Mark 0.00 out of 10.00

Consider the following snapshot diagram:

Is it possible that a client with the variable petStore could do something such that a client with the variable terrarium could no longer access th
Gecko in blue?

Select one:

a. No, because the "Gecko" is immutable

b. Yes, because all the references between "petStore" and the "Gecko" are mutable

c. Yes, because of some reference between "petStore" and the "Gecko" that is mutable

d. Yes, because the "Gecko" is mutable

e. No, because of some reference between "petStore" and the "Gecko" that is immutable

f. No, because all the references between "petStore" and the "Gecko" are immutable

Your answer is incorrect.

The correct answer is: Yes, because of some reference between "petStore" and the "Gecko" that is mutable

Question 10

Incorrect Mark 0.00 out of 10.00

Consider `MyIterator`'s `next` method:

```
public class MyIterator {
```

```
    private final ArrayList<String> list;
    private int index;

    ...

    /**
     * Get the next element of the list.
     * Requires: hasNext() returns true.
     * Modifies: this iterator to advance it to the element
     *           following the returned element.
     * @return next element of the list
     */
    public String next() {
        final String element = list.get(index);
        index++;
        return element;
    }
}
```

What is the type of the input to `next`?

Select one:

- a. MyIterator
- b. void
- c. ArrayList
- d. String
- e. boolean
- f. int

Your answer is incorrect.

The correct answer is: MyIterator

Question 11

Correct | Mark 10.00 out of 10.00

Consider `MyIterator`'s `next` method:

```
public class MyIterator {

    private final ArrayList<String> list;
    private int index;

    ...

    /**
     * Get the next element of the list.
     * Requires: hasNext() returns true.
     * Modifies: this iterator to advance it to the element
     *           following the returned element.
     * @return next element of the list
     */
    public String next() {
        final String element = list.get(index);
        index++;
        return element;
    }
}
```

What is the type of the output to `next`?

Select one:

- a. MyIterator
- b. void
- c. ArrayList

d. String

e. boolean

f. int

Your answer is correct.

The correct answer is: String

Question 12

Incorrect Mark 0.00 out of 10.00

Consider `MyIterator`'s `next` method:

```
public class MyIterator {

    private final ArrayList<String> list;
    private int index;

    ...

    /**
     * Get the next element of the list.
     * Requires: hasNext() returns true.
     * Modifies: this iterator to advance it to the element
     *           following the returned element.
     * @return next element of the list
     */
    public String next() {
        final String element = list.get(index);
        ++index;
        return element;
    }
}
```

`next` has the precondition `requires: hasNext() returns true`.

Which input to `next` is constrained by the precondition?

Select one:

a. this

b. list

c. index

d. element

e. hasNext

Your answer is incorrect.

The correct answer is: this

Question 13

Correct Mark 10.00 out of 10.00

Consider `MyIterator`'s `next` method:

```
public class MyIterator {

    private final ArrayList<String> list;
    private int index;
```

```
    ...

    /**
     * Get the next element of the list.
     * Requires: hasNext() returns true.
     * Modifies: this iterator to advance it to the element
     *           following the returned element.
     * @return next element of the list
     */
    public String next() {
        final String element = list.get(index);
        ++index;
        return element;
    }
}
```

When the precondition is **not** satisfied, the implementation is free to do anything.

What does *this particular implementation* do when the precondition is **not** satisfied?

Select one:

- a. throw an unchecked exception
- b. throw a checked exception
- c. return null
- d. return some other element of the list

Your answer is correct.

The correct answer is: throw an unchecked exception

Question 14

Incorrect Mark 0.00 out of 10.00

Consider `MyIterator`'s `next` method:

```
public class MyIterator {

    private final ArrayList<String> list;
    private int index;

    ...

    /**
     * Get the next element of the list.
     * Requires: hasNext() returns true.
     * Modifies: this iterator to advance it to the element
     *           following the returned element.
     * @return next element of the list
     */
    public String next() {
        final String element = list.get(index);
        ++index;
        return element;
    }
}
```

Part of the postcondition of `next` is: `@return next element of the list.`

Which output from `next` are constrained by that postcondition?

Select one:

- a. the return value
- b. this
- c. hasNext

d. list

Your answer is incorrect.

The correct answer is: the return value

Question 15

Correct Mark 10.00 out of 10.00

Consider `MyIterator`'s `next` method:

```
public class MyIterator {

    private final ArrayList<String> list;
    private int index;

    ...

    /**
     * Get the next element of the list.
     * Requires: hasNext() returns true.
     * Modifies: this iterator to advance it to the element
     *           following the returned element.
     * @return next element of the list
     */
    public String next() {
        final String element = list.get(index);
        ++index;
        return element;
    }
}
```

Another part of the postcondition of `next` is `modifies: this iterator to advance it to the element following the returned element.`

What is constrained by that postcondition?

Select one:

a. the return value

b. this

c. hasNext

d. list

Your answer is correct.

The correct answer is: this

Finish review

Reading 7: Designing Specifications
Introduction
Deterministic vs. underdetermined specs
Declarative vs. operational specs
Stronger vs. weaker specs
Diagramming specifications
Designing good specifications
Preconditions or postconditions
About access control
About static vs. instance methods
Summary

Reading 7: Designing Specifications

Software in 6.005

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

Objectives

- Understand underdetermined specs, and be able to identify and assess nondeterminism
- Understand declarative vs. operational specs, and be able to write declarative specs
- Understand strength of preconditions, postconditions, and specs, and be able to compare specs: strength
- Be able to write coherent, useful specifications of appropriate strength

Introduction

In this reading we'll look at different specs for similar behaviors, and talk about the tradeoffs between them. We'll look at three dimensions for comparing specs:

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

Not all specifications are right choice for a module are equally useful, and we'll explore what makes some specifications better than others.

Deterministic vs. underdetermined specs

Recall the two example implementations of `find`, we began with in the previous reading:

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

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

Note that the subscripts `i` and `arr` are not actual Java syntax. We're using them here to distinguish the two implementations for the sake of discussion. In the actual code, both implementations should be Java methods called `find`.

Here is one possible specification of `find`:

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

This specification is **deterministic**: when presented with a state satisfying the precondition, the outcome is completely determined. Only one return value and one final state is possible. There are no valid inputs for which there is more than one valid output.

Both `find_alt` and `find` satisfy the specification, so if this is the specification on which the clients relied, the two implementations are equivalent and substitutable for one another.

Here is a slightly different specification:

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

This specification is not deterministic. It doesn't say which index is returned if `val` occurs more than once. It simply says that if you look up the entry at the index given by the returned value, you'll find `val`. This specification allows multiple valid outputs for the same input.

Note that this is different from nondeterminism in the usual sense of that word. Nondeterministic code sometimes behaves one way and sometimes another, even if called in the same program with the same inputs. Multiprocessor timing, hardware failures, and network latency are common sources of this kind of behavior, as are the timing of concurrent processes. But a specification which is not deterministic doesn't have to have a nondeterministic implementation. It can be satisfied by a fully deterministic implementation.

To avoid the confusion, we'll refer to specifications that are not deterministic as **underdetermined**.

This underdetermined `find` spec is again satisfied by both `find_alt` and `find`. `find` satisfies the underdeterminedness in its own (fully deterministic) way. A client of `find` or `find_alt` cannot tell why an index will be returned if `val` appears more than once. The spec would be satisfied by a nondeterministic implementation, too — for example, one that tosses a coin to decide whether to start searching from the beginning or the end of the array. But in almost all cases we'll encounter, underdetermined specifications are satisfied by code that is made by the implementer of implementation time. An underdetermined spec is typically implemented by a fully deterministic implementation.

READING EXERCISES

Distinguished

Overlunder

Declarative vs. operational specs

Roughly speaking, there are two kinds of specifications. Operational specifications give a series of steps that the method performs. Declarative descriptions are operational. Declarative specifications 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 specifications are preferable. They're usually shorter, easier to understand, and most importantly, they don't inadvertently expose implementation details that a client may rely on (and then find no longer hold when the implementation is changed). For example, if we want to allow either implementation of `find` to be used, we can't specify that it will return the first index where `val` occurs until it finds `val`, since aside from being rather vague, this spec suggests that the search proceeds from lower to higher indices and that the lowest will be returned, which perhaps the specifier did not intend.

One reason programmers sometimes lapse into operational specifications is because they're using the spec comment to explain the implementation for a maintainer. Don't. Do that using comments within the body of the method, not in the spec comment.

For a given specification, there may be many ways to express it declaratively:

```
static boolean startsWith(String str, String prefix)
  requires: str and prefix are non-null Unicode String suffix
    such that prefix + suffix == str

static boolean startsWith(String str, String prefix)
  effects: returns true if and only if there exists integer i
    such that str.substring(0, i) == prefix

static boolean startsWith(String str, String prefix)
  effects: returns true if the first prefix.length() characters of str
    are the characters of prefix, false otherwise
```

It's up to us to choose the clearest specification for clients and maintainers of the code.

READING EXERCISES

Joint declaration

Stronger vs. weaker specs

Suppose you want to change a method – either how its implementation behaves, or the specification itself. There are always clients that depend on the method's current specification. How do you compare the behaviors of two specifications to decide whether it's safe to replace the old spec with the new spec?

A specification S1 is stronger than or equal to a specification S2 if

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

If this is the case, then an implementation that satisfies S2 can be used to satisfy S1 as well, and it's safe to replace S1 with S2 in your program.

These two rules similarly sound dense. They tell you that you can always weaken the precondition, placing fewer demands on a client will never upset them. And you can always strengthen the post-condition, which means making more promises.

For example, this spec for `find`:

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

can be replaced with:

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

which has a weaker precondition. The old `find` can be replaced with:

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

which has a stronger postcondition.

What about this specification:

```
static int find(int[] a, int val)
  requires: nothing
  effects: returns index i such that a[i] = val,
    or -1 if no such i
```

We'll come back to `find`'s specification in the exercises.

Diagramming specifications

Imagine (very abstractly) the space of all possible Java methods.

Each point in this space represents a method implementation.

Find we'll diagram `find_alt` and `find`. defined above. Look back at the code and see that `find_alt` and `find` are not specs. They are implementations, with method bodies that implement their actual behavior. So we denote them as points in the space.

A specification defines a region in the space of all possible implementations. A given implementation either behaves according to the spec, satisfying the precondition-implies-postcondition contract (it is inside the region), or it does not (outside the region).

Both `find_alt` and `find` satisfy `find`'s specification, so they are inside the region defined by that spec:

We can imagine clients looking in on this space: the specification acts as a firewall.

- Implementers have the freedom to move around inside the spec, changing their code without fear of upsetting a client. This is crucial in order for the implementer to be able to improve the performance of their algorithm, fix bugs, clarify their code, or to change their approach when they discover a bug, etc.
- Clients don't know which implementation they will get. They must respect the spec, but also have the freedom to change how they're using the implementation without worries that it will suddenly break.

How will similar specifications relate to one another? Suppose we start with specification S1 and use it to create a new specification S2.

If S2 is stronger than S1, how will these spaces appear in our diagram?

- Let's start by **strengthening the postcondition**. If S2's postcondition is now stronger than S1's postcondition, then S2 is the stronger specification.

Think about what strengthening the postcondition means for implementers: it means they have less freedom, the requirements on their output are stronger. Perhaps they previously satisfied `find`'s spec by returning the first index, but now they must return the lowest index i. So there are now implementations inside `find`'s specification, but outside `find_alt`'s specification.

Could there be implementations inside `find_alt`'s specification, but outside `find`'s specification? No. All of these implementations satisfy a stronger postcondition than what `find`'s specification demands.

- Think through what happens if we **weaken the precondition**, which will again make S2 a stronger specification. Implementations will have to handle new inputs that were previously excluded by the spec. If they behaved badly on those inputs before, we wouldn't have noticed, but now their bad behavior is exposed.

We see that when S2 is stronger than S1, it defines a smaller region in this diagram, a weaker specification defines a larger region.

In our figure, since `find_alt` iterates from the end of the array `arr`, it does not satisfy `find`'s specification, and is outside that region.

Another specification S3 that is neither stronger nor weaker than S1 might overlap both that there and implementations that satisfy only S1, only S3, and both S1 and S3 (it might be disjoint.

READING EXERCISES

Bulking up

Strength

Finding findExactlyOne

Finding findOneOrMissing

Found

Designing good specifications

What makes a good method? Designing a method means primarily writing a specification.

About the form of the specification: it should obviously be succinct, clear, and well-structured, so that it's easy to read.

The content of the specification, however, is harder to prescribe. There are no infallible rules, but there are some useful guidelines.

The specification should be coherent

The spec shouldn't have lots of different cases. Long argument lists, deeply nested if-statements, and boolean flags are all signs of trouble. Consider this specification:

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

Is this a well-designed procedure? Probably not. It's incoherent, since it does several things (finding in two arrays and summing the indexes) that are not really related. It would be better to use two separate procedures, one that finds the indexes, and the other that sums them.

Here's another example, the `countLongWords` method from Code Review:

```
public static int LONG_WORD_LENGTH = 5;
public static String longerWord;

/**
 * Update longerWord to be the longest element of words, and print
 * the number of elements with length > LONG_WORD_LENGTH to the console.
 * Ignore words list to search for long words
 */
public static void countLongWords(List<String> words)
```

In addition to terrible use of global variables and printing instead of returning, the specification is not coherent — it does two different things, counting words and finding the longest word.

Separating these two responsibilities into two different methods will make them simpler (easy to understand) and more useful in other contexts (ready for change).

The results of a call should be informative

Consider the specification of a method that pulls a value in a map:

```
static V pw(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,
    and returns the previous mapping for key, and
    returns old value for key, or null none,
    in which case it returns null
```

Note that the precondition does not rule out `null` values as the map can store `null`s. But the postcondition doesn't let the client know what happened. If the return value is `null`, it could mean that there was no mapping or that whether the key was not found previously, or whether it was in fact bound to `null`. This is not a very good design, because the return value is useless unless you know for sure that you didn't insert `null`s.

The specification should be strong enough

Of course the spec should give clients a strong enough guarantee in the general case — it needs to satisfy their basic requirements. We must use extra care when specifying the special cases, to make sure they don't undermine what would otherwise be a useful method.

For example, there's no point throwing a checked exception for a bad argument but allowing arbitrary mutations, because a client won't be able to determine what mutations have actually been made. Here's a specification illustrating this flaw (and also written in an inappropriately operational style):

```
static void addAll(List<T> l, List<T> list)
  effects: adds the elements of list to list,
    in the order they appear in list,
    at which point it throws a NullPointerException

If a NullPointerException is thrown, the client is left to figure out on their own which elements of list2
actually made it to list1.
```

The specification should also be weak enough

Consider this specification for a method that opens a file:

```
static File openString(String filename)
  effects: opens a file named filename
```

This is a bad specification. It lacks important details: is the file opened for reading or writing? Does it already exist or is it created? And it's too strong, since there's no way it can guarantee to open a file. The process in which it runs may lack permission to open a file, or there might be some problem with the file system beyond the control of the programmer. Instead, the specification should say something much weaker that it attempts to open a file, and if it succeeds, the file has certain properties.

The specification should use abstract types where possible

We saw early on in the Java Collections section of Basic Java that we can distinguish between more abstract notions like a `List` or `Set`, and particular implementations like `ArrayList` or `HashSet`.

Writing our specification with abstract types gives more freedom to both the client and the implementer. In Java, this often means using an interface type, like `Map`, `List`, `Reader`, instead of specific implementation types like `HashMap` or `FileReader`. Consider this specification:

```
static ArrayList<T> reverse(ArrayList<T> list)
  effects: returns a new list that is the reversal of list, i.e.,
    reverse(list) == list[n-1]
      for all 0 <= i < n, where n = list.size()
```

This forces the client to pass in an `ArrayList`, and forces the implementer to return an `ArrayList`, even if other kinds of lists would be more convenient. Suppose the client has a `LinkedList` on hand, and the specification doesn't depend on anything specific about `ArrayList`s, it would be better to write this spec in terms of the more abstract `List`.

Precondition or postcondition?

Another design issue is whether to use a precondition, and if so, whether the method code should attempt to make sure the precondition has been met before proceeding. In fact, the most common use of preconditions is to demand a property (sorted) because it would be hard or expensive for the method to check it.

As mentioned above, a non-trivial precondition inconvenience clients, because they have to ensure that they don't call the method in a bad state (that violates the precondition). If they do, this is predictable way to recover from the error. So cases of which don't fit preconditions. That's why the Java API classes, for example, don't require that their collections be sorted, and instead throw exceptions when they are inappropriate. The exception makes it easier to find the bug or incorrect assumption in the caller code that led to passing bad arguments. In general, it's better to **fail fast**, as close as possible to the site of the bug, rather than let bad values propagate through a program far from their original cause.

Sometimes, it's not feasible to check a condition without making a method unacceptably slow, and a precondition is almost necessary in this case. If we wanted to implement the `find` method using binary search, we would have to require that the array be sorted. Forcing the method to actually check that the array is sorted would defeat the entire purpose of the binary search. In cases where it is feasible, it's better to do it.

The decision of whether to use a precondition is an engineering judgment. The key factors are the cost of the checks, and how inconvenient it is for clients to satisfy the precondition. Remember that if the cost of a precondition can be mitigated by carefully checking all the cites that call the method, but the method is public, and used by other developers, it would be less wise to use a precondition. Instead, the Java API classes, you should throw an exception.

About access control

Read **Packages** (7 short pages) in the Java Tutorials.

Read **Controlling Access** (7 pages) in the Java Tutorials.

We have been using public for almost all of our methods, without really thinking about it. The decision to make a method public or private is actually a decision about the contract of the class. Public methods are freely accessible to other parts of the program. Making a method public advertises it as a service that your class is willing to provide. If you make all your methods public — including helper methods that are really meant only for your class to use — then you're making your class responsible for supporting all of those methods. It's harder to change the internal implementation of the class in the future. Your code won't be as **ready for change**.

Making internal helper methods public will also add clutter to the visible interface your class offers. Keeping internal things private makes your class's public interface smaller and more coherent (meaning that it does one thing and does it well). Your code will be **easier to understand**.

We will see even stronger reasons to use private in the next two classes, when we start to write classes with persistent internal state. Protecting this state will help keep the program **safe from bugs**.

About static vs. instance methods

Read the **static** keyword on CodeGuru.

We have also been using static for almost all of our methods, again without much discussion. Static methods are not associated with any particular instance of a class, while instance methods (declared without the `static` keyword) must be called on a particular object.

Specifications for instance methods are written just the same way as specifications for static methods, but they will often refer to properties of the instance on which they were called.

For example, by now we're very familiar with this specification:

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

Instead of using an `int[]`, what if we had a class `IntArray` designed for storing arrays of integers? The `IntArray` class might provide an instance method with the specification:

```
int find(int val)
  requires: this occurs in ethis.array
  effects: returns index i such that ethis.array[i] = val
    in val
```

We'll have much more to say about specifications for instance methods in future classes!

READING EXERCISES

Show me a sign

That's an odd way of looking at it

Behavioral oddities

Odd dad

Summary

A specification acts as a crucial firewall between implementor and client — both between people (or the same person at different times) and between code. As we saw last time, it makes separate development possible: the client is free to write code that uses a module without seeing its source code, and the implementor is free to write the implementation code without knowing how it will be used.

Declarative specifications are the most useful in practice. Preconditions (which weaken the specification) make life harder for the client, but applied judiciously they are a vital tool in the software designer's repertoire, allowing the implementor to make necessary assumptions.

As always, our goal is to design specifications that make our software

- **Safe from bugs**. Without specifications, even the trivial change to any part of our program could be the tipped dominos that knocks the whole thing over. Well-structured, coherent specifications minimize unexpected interactions between components. Write correct code with the help of static checking, careful reasoning, testing, and code review.
- **Easy to understand**. A well-written declarative specification means the client doesn't have to read or understand the code. You've probably never read the code for, say Python dict.update(), and doing so isn't nearly as useful to the Python programmer as reading the declarative spec.
- **Ready for change**. An appropriate-strength specification gives freedom to the implementer, and an appropriately strong specification gives freedom to the client. We can even change the specs themselves, without having to change every piece of code that uses them, as long as we're only strengthening their weakening preconditions and strengthening postconditions.

Collaboratively authored by Christopher Terman, Karen Anderson, Adam Hansen, Jonathan Hoffman, Stephen Ruth, Mike Phillips, Daniel Jackson, Rob Miller, Martin Rinard, Saman Amarasinghe, Joseph Bonn. This work is licensed under CC BY-NC-SA.

Reading 08: Avoiding Debugging
First Defense: Make Bugs Impossible
Second Defense: Localize Bugs
Assertions
What to Assert
What Not to Assert
Incremental Development
Modularity & Encapsulation

You are not logged in.

Reading 08: Avoiding Debugging

6.005 Prime Objective

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

Objectives

The topic of today's class is debugging – or rather, how to avoid debugging entirely, or keep it easy when you have to do it.

First Defense: Make Bugs Impossible

The best defense against bugs is to make them impossible by design.

One way that we've already talked about is **static checking**. Static checking eliminates many bugs by catching them at compile time.

We also saw some examples of **dynamic checking** in earlier class meetings. For example, Java makes array overflow bugs impossible by catching them dynamically. If you try to use an index outside the bounds of an array or a List, then Java automatically produces an error. Older languages like C and C++ silently allow the bad access, which leads to bugs and security vulnerabilities.

Immutability (immunity from change) is another design principle that prevents bugs. An immutable type is a type whose values can never change once they have been created.

String is an immutable type. There are no methods that you can call on a String that will change the sequence of characters that it represents. Strings can be passed around and shared without fear that they will be modified by other code.

Java also gives us immutable references: variables declared with the keyword `final`, which can be assigned once but never reassigned. It's good practice to use `final` for declaring the parameters of a method and as many local variables as possible. Like the type of the variable, these declarations are important documentation, useful to the reader of the code and statically checked by the compiler.

Consider this example:

```
final char[] vowels = new char[] {'a','e','i','o','u'};
```

The `vowels` variable is declared final, but is it really unchanging? Which of the following statements will be illegal (caught statically by the compiler), and which will be allowed?

```
vowels = new char[] {'x','y','z'};
vowels[0] = 'z';
```

You'll find the answers in the exercise below. Be careful about what `final` means! It only makes the reference immutable, not necessarily the object that the reference points to.

READING EXERCISES

Final references, immutable objects

Afterwards

Second Defense: Localize Bugs

If we can't prevent bugs, we can try to localize them to a small part of the program, so that we don't have to look too hard to find the cause of a bug. When localized to a single method or small module, bugs may be found simply by studying the program text.

We already talked about **fail fast**: the earlier a problem is observed (the closer to its cause), the easier it is to fix.

Let's begin with a simple example:

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

Now suppose somebody calls `sqrt` with a negative argument. What's the best behavior for `sqrt`? Since the caller has failed to satisfy the requirement that `x` should be nonnegative, `sqrt` is no longer bound by the terms of its contract, so it is technically free to do whatever it wants: return an arbitrary value, or enter an infinite loop, or melt down the CPU. Since the bad call indicates a bug in the caller, however, the most useful behavior would point out the bug as early as possible. We do this by inserting a runtime assertion that tests the precondition. Here is one way we might write the assertion:

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

When the precondition is not satisfied, this code terminates the program by throwing an AssertionError exception. The effects of the caller's bug are prevented from propagating.

Checking preconditions is an example of **defensive programming**. Real programs are rarely bug-free. Defensive programming offers a way to mitigate the effects of bugs even if you don't know where they are.

Assertions

It is common practice to define a procedure for these kinds of defensive checks, usually called `assert`:

```
assert (x >= 0);
```

This approach abstracts away from what exactly happens when the assertion fails. The failed assert might exit; it might record an event in a log file; it might email a report to a maintainer.

Assertions have the added benefit of documenting an assumption about the state of the program at that point. To somebody reading your code, `assert(x>=0)` says "at this point, it should always be true that `x >= 0`." Unlike a comment, however, an assertion is executable code that enforces the assumption at runtime.

In Java, runtime assertions are a built-in feature of the language. The simplest form of the assert statement takes a boolean expression, exactly as shown above, and throws AssertionError if the boolean expression evaluates to false:

```
assert x >= 0;
```

An assert statement may also include a description expression, which is usually a string, but may also be a primitive type or a reference to an object. The description is printed in an error message when the assertion fails, so it can be used to provide additional details to the programmer about the cause of the failure. The description follows the asserted expression, separated by a colon. For example:

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

If `x == -1`, then this assertion fails with the error message

```
x is -1
```

along with a stack trace that tells you where the assert statement was found in your code and the sequence of calls that brought the program to that point. This information is often enough to get started in finding the bug.

A serious problem with Java assertions is that assertions are off by default.

If you just run your program as usual, none of your assertions will be checked! Java's designers did this because checking assertions can sometimes be costly to performance. For most applications, however, assertions are not expensive compared to the rest of the code, and the benefit they provide in bug checking is worth that small cost in performance.

So you have to enable assertions explicitly by passing `-ea` (which stands for enable assertions) to the Java virtual machine. In Eclipse, you enable assertions by going to Run → Run Configurations → Arguments, and putting `-ea` in the VM arguments box. It's best, in fact, to enable them by default by going to Preferences → Java → Installed JREs → Edit → Default VM Arguments, as you hopefully did in the Getting Started instructions.

It's always a good idea to have assertions turned on when you're running JUnit tests. You can ensure that assertions are enabled using the following test case:

```
@Test(expected=AssertionError.class)
public void testAssertionsEnabled() {
  assert false;
}
```

If assertions are turned on as desired, then `assert false` throws an `AssertionError`. The annotation `(expected=AssertionError.class)` on the test expects and requires this error to be thrown, so the test passes. If assertions are turned off, however, then the body of the test will do nothing, failing to throw the expected exception, and JUnit will mark the test as failing.

Note that the Java `assert` statement is a different mechanism from the JUnit methods `assertTrue()`, `assertEquals()`, etc. They are about a predicate about your code, but are designed for use in different contexts. The `assert` statement should be used in implementation code, for defensive checks inside the implementation. JUnit `assert...()` methods should be used in JUnit tests, to check the result of a test. The `assert` statements don't run without `-ea`, but the JUnit `assert...()` methods always run.

What to Assert

Here are some things you should assert:

Method argument requirements, like we saw for `sqrt`.

Method return value requirements. This kind of assertion is sometimes called a self check. For example, the sqrt method might square its result to check whether it is reasonably close to x:

```
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 the 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': return "A";
  default: Assert.fail();
}
```

The assertion in the default clause has the effect of asserting that `vowel` must be one of the five vowel letters.

When should you write runtime assertions? As you write the code, not after the fact. When you're writing the code, you have the invariants in mind. If you postpone writing assertions, you're less likely to do it, and you're liable to omit some important invariants.

What Not to Assert

Runtime assertions are not free. They can clutter the code, so they must be used judiciously. Avoid trivial assertions, just as you would avoid uninformative comments. For example:

```
// don't do this:
x = y + 1;
assert x == y+1;
```

This assertion doesn't find bugs in your code. It finds bugs in the compiler or Java virtual machine, which are components that you should trust until you have good reason to doubt them. If an assertion is obvious from its local context, leave it out.

Never use assertions to test conditions that are external to your program, such as the existence of files, the availability of the network, or the correctness of input typed by a human user. Assertions test the internal state of your program to ensure that it is within the bounds of its specification. When an assertion fails, it indicates that the program has run off the rails in some sense, into a state in which it was not designed to function properly. Assertion failures therefore indicate bugs. External failures are not bugs, and there is no change you can make to your program in advance that will prevent them from happening. External failures should be handled using exceptions instead.

Many assertion mechanisms are designed so that assertions are executed only during testing and debugging, and turned off when the program is released to users. Java's assert statement behaves this way. The advantage of this approach is that you can write very expensive assertions that would otherwise seriously degrade the performance of your program. For example, a procedure that searches an array using binary search has a requirement that the array be sorted. Asserting this requirement requires scanning through the entire array, however, turning an operation that should run in logarithmic time into one that takes linear time. You should be willing (eager?) to pay this cost during testing, since it makes debugging much easier, but not after the program is released to users.

However, disabling assertions in release has a serious disadvantage. With assertions disabled, a program has far less error checking when it needs it most. Novice programmers are usually much more concerned about the performance impact of assertions than they should be. Most assertions are cheap, so they should not be disabled in the official release.

Since assertions may be disabled, the correctness of your program should never depend on whether or not the assertion expressions are executed. In particular, asserted expressions should not have side-effects. For example, if you want to assert that an element removed from a list was actually found in the list, don't write it like this:

```
// don't do this:
assert list.remove(x);
```

If assertions are disabled, the entire expression is skipped, and `x` is never removed from the list. Write it like this instead:

```
boolean found = list.remove(x);
assert found;
```

For 6.005, you are **required** to have assertions turned on, all the time. Make sure you did this in Eclipse, following the instructions in the **Getting Started handout**. If you don't have assertions turned on, you will be sad, and the staff won't have much sympathy.

READING EXERCISES

Assertions

Incremental Development

A great way to localize bugs to a tiny part of the 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 more likely to be in the part that you just wrote, rather than anywhere in a huge pile of code.

Our class on testing talked about two techniques that help with this:

- Unit testing: when you test a module in isolation, you can be confident that any bug you find is in that unit – or maybe in the test cases themselves.
- Regression testing: when you're adding a new feature to a big system, run the regression test suite as often as possible. If a test fails, the bug is probably in the code you just changed.

Modularity & Encapsulation

You can also localize bugs by better software design.

Modularity. Modularity means dividing up a system into components, or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system. The opposite of a modular system is a monolithic system – big and with all of its pieces tangled up and dependent on each other.

A program consisting of a single, very long main() function is monolithic – harder to understand, and harder to isolate bugs in. By contrast, a program broken up into small functions and classes is more modular.

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

One kind of encapsulation is access control, using `public` and `private` to control the visibility and accessibility of your variables and methods. A public variable or method can be accessed by any code (assuming the class containing that variable or method is also public). A private variable or method can only be accessed by code in the same class. Keeping things private as much as possible, especially for variables, provides encapsulation, since it limits the code that could inadvertently cause bugs.

Another kind of encapsulation comes from **variable scope**. The scope of a variable is the portion of the program text over which that variable is defined, in the sense that expressions and statements can refer to the variable. A method parameter's scope is the body of the method. A local variable's scope extends from its declaration to the next closing curly brace. Keeping variable scopes as small as possible makes it much easier to reason about where a bug might be in the program. For example, suppose you have a loop like this:

```
for (i = 0; i < 100; ++i) {
  ...
  doSomeThings();
  ...
}
```

...and you've discovered that this loop keeps running forever – `i` never reaches 100. Somewhere, somebody is changing `i`. But where? If `i` is declared as a global variable like this:

```
public static int i;
...
for (i =0; i < 100; ++i) {
  ...
  doSomeThings();
  ...
}
```

...then its scope is the entire program. It might be changed anywhere in your program: by `doSomeThings()`, by some other method that `doSomeThings()` calls, by a concurrent thread running some completely different code. But if `i` is instead declared as a local variable with a narrow scope, like this:

```
for (int i = 0; i < 100; ++i) {
  ...
  doSomeThings();
  ...
}
```

...then the only place where `i` can be changed is within the for statement – in fact, only in the ... parts that we've omitted. You don't even have to consider `doSomeThings()`, because `doSomeThings()` doesn't have access to this local variable.

Minimizing the scope of variables is a powerful practice for bug localization. Here are a few rules that are good for Java:

- **Always declare a loop variable in the for-loop initializer.** So rather than declaring it before the loop:

```
int i;
for (i = 0; i < 100; ++i) {
```

which makes the scope of the variable the entire rest of the outer curly-brace block containing this code, you should do this:

```
for (int i = 0; i < 100; ++i) {
```

which makes the scope of `i` limited just to the for loop.

- **Declare a variable only when you first need it, and in the innermost curly-brace block that you can.** Variable scopes in Java are curly-brace blocks, so put your variable declaration in the innermost one that contains all the expressions that need to use the variable. Don't declare all your variables at the start of the function – it makes their scopes unnecessarily large. But note that in languages without static type declarations, like Python and Javascript, the scope of a variable is normally the entire function anyway, so you can't restrict the scope of a variable with curly braces, alas.
- **Avoid global variables.** Very bad idea, especially as programs get large. Global variables are often used as a shortcut to provide a parameter to several parts of your program. It's better to just pass the parameter into the code that needs it, rather than putting it in global space where it can inadvertently reassigned.

READING EXERCISES

Variable scope

Summary

In this reading, we looked at some ways to minimize the cost of debugging:

- avoid debugging
  - make bugs impossible with techniques like static typing, automatic dynamic checking, and immutable types and references
- keep bugs confined
  - failing fast with assertions keeps a bug's effects from spreading
  - incremental development and unit testing confine bugs to your recent code
  - scope minimization reduces the amount of the program you have to search

Thinking about our three main measures of code quality:

- **Safe from bugs.** We're trying to prevent them and get rid of them.
- **Easy to understand.** Techniques like static typing, final declarations, and assertions are additional documentation of the assumptions in your code. Variable scope minimization makes it easier for a reader to understand how the variable is used, because there's less code to look at.
- **Ready for change.** Assertions and static typing document the assumptions in an automatically-checkable way, so that when a future programmer changes the code, accidental violations of those assumptions are detected.

Collaboratively authored with contributions from: Saman Amarasinghe, Adam Chlipala, Michael Ernst, Michael Frind, Max Goldman, John Guttag, Daniel Jackson, Joseph Kiniry, Martin Rinard, and Armando Solar-Lezama. This work is licensed under CC BY 3.4-5.0

Reading 9: Mutability & Immutability

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and tomorrow	Communicating clearly with future programmers, including future you	Ready to meet new requirements; change without destroying

Objectives:

- Understand mutability and immutable objects
- Use immutability and understand its pros and cons
- Use immutability to avoid bugs, understand, and ease changes

Mutability

Recall that Maps (and other set datatypes) can't handle keys that are mutable. Consider this example:

```
String s = new String("1");
m.put(s, "foo");
m.get(s);
s = s.concat("2");
m.put(s, "bar");
m.get(s);
```

Suppose that Map.put() and Map.get() are implemented using a hash table. Then the first put() puts the key into bucket 49, and the get() looks in bucket 49 and finds the key there. But the second put() puts the key into bucket 50, and the second get() looks in bucket 50 and finds the key there.

Mutability means that the rep of an object can change. Mutable objects can be very convenient, because they can be used to store information that changes over time. But mutability can also be very dangerous, because it can cause bugs in other parts of the program that are using the same object, if they don't expect that the object will change.

In this reading, we'll see how to use immutable types to avoid bugs, and how to design and implement mutable types safely.

Risks of mutation

Mutable types seem much more powerful than immutable types. If you were shopping in the Datatype Experiment, and you had to choose between a boxing immutable String and a boxing mutable StringBuilder, which would you choose? Immutable types seem much more limiting. But there are many good reasons why we want to use immutable types, which can be just as useful and powerful as mutable types.

The process in the **immutable** type are safer from bugs, easier to understand, and more ready for change. Immutable types are also more convenient to use in a program in design, and much faster in particular contexts. There are tradeoffs that Reading 9 explores.

Risky example #1: passing mutable values

Let's start with a simple method that turns the integers 0 to n into

```
[0, 1, 2, ..., n]
```

```
/** requires n >= 0
public static int[] sum(int n) {
    int[] result = new int[n+1];
    int sum = 0;
    for (int i = 0; i <= n; i++) {
        sum += i;
        result[i] = sum;
    }
    return result;
}
```

Suppose we also need a method that sums the absolute values. Following good DRY practice (Don't Repeat Yourself), the programmer writes a method that uses sum():

```
/** requires true
public static int[] sumAbs(int[] ints) {
    int[] result = new int[ints.length];
    for (int i = 0; i < ints.length; i++) {
        int sum = sum(ints[i]);
        result[i] = sum;
    }
    return result;
}
```

Notice that the method does its job by modifying the array. It reserves space for the result, and then it fills in the result array, one element at a time, by calling sum() on each element of the array.

Suppose one of the clients of sum() and sumAbs() needs to get the absolute values of the integers 0 to n. The client can do it by calling sum(n) and then calling sumAbs() on the result array:

```
int[] ints = sum(n);
int[] result = sumAbs(ints);
for (int i = 0; i < ints.length; i++) {
    System.out.print(result[i] + " ");
}
```

This code prints the sum of the absolute values of the numbers in the range 0 to n:

```
0 1 3 6 10 15 21 28 36 45 55
```

Suppose we also need a method that sums the absolute values, following good DRY practice (Don't Repeat Yourself). The programmer writes a method that uses sum():

```
/** requires true
public static int[] sumAbs(int[] ints) {
    int[] result = new int[ints.length];
    for (int i = 0; i < ints.length; i++) {
        int sum = sum(ints[i]);
        result[i] = sum;
    }
    return result;
}
```

Notice that this method does its job by **modifying the array**. It reserves space for the result, and then it fills in the result array, one element at a time, by calling sum() on each element of the array.

Suppose one of the clients of sum() and sumAbs() needs to get the absolute values of the integers 0 to n. The client can do it by calling sum(n) and then calling sumAbs() on the result array:

```
int[] ints = sum(n);
int[] result = sumAbs(ints);
for (int i = 0; i < ints.length; i++) {
    System.out.print(result[i] + " ");
}
```

This code prints the sum of the absolute values of the numbers in the range 0 to n:

```
0 1 3 6 10 15 21 28 36 45 55
```

Wait, what happened? The client expected to get the sum of the absolute values of the numbers in the range 0 to n, but instead it got the sum of the numbers in the range 0 to n. The client's code didn't even call sum(n) on the array returned by sumAbs(), so how could the client's code be affected by the call to sum(n)?

The problem is that the **implementation** of sum() and sumAbs() is shared between the client and the implementer. The implementer of sum() and sumAbs() thinks that it is allocating a new array for its result, but because of a programmer error, the array is not actually allocated. Instead, the array is shared between the implementer of sum() and sumAbs() and the client. The implementer of sum() and sumAbs() thinks that it is initializing the array for its own use, but because the array is shared, the client's code can see the array too, and the client's code can also modify the array. The implementer of sum() and sumAbs() thinks that it is initializing the array for its own use, but because the array is shared, the client's code can also modify the array. The implementer of sum() and sumAbs() thinks that it is initializing the array for its own use, but because the array is shared, the client's code can also modify the array.

Because the implementer of sum() and sumAbs() and the client share the same array, the client's code can see the array that the implementer of sum() and sumAbs() is initializing, and the client's code can also modify the array that the implementer of sum() and sumAbs() is initializing. This is a bug. The implementer of sum() and sumAbs() and the client share the same array, so

Reading 10: Recursion
Recursion
Choosing the Right Decomposition for a Problem
Structure of Recursive Implementations
Helper Methods
Choosing the Right Recursive Subproblem
Recursive Problems vs. Recursive Data
Reentrant Code
When to Use Recursion Rather Than Iteration
Common Mistakes in Recursive Implementations
Summary

You are not logged in.

Reading 10: Recursion

Software in 6.005

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

Objectives

After today's class, you should:

- be able to decomposse a recursive problem into recursive steps and base cases
- know when and how to use helper methods in recursion
- understand the advantages and disadvantages of recursion vs. iteration

Recursion

In today's class, we're going to talk about how to implement a method, once you already have a specification. We'll focus on one particular technique, recursion. Recursion is not appropriate for every problem, but it's an important tool in your software development toolbox, and one that many people scratch their heads over. We want you to be comfortable and competent with recursion, because you will encounter it over and over. (That's a pun, but it's also true.)

Since you've taken 6.01, recursion is not completely new to you, and you have seen and written recursive functions like factorial and fibonacci before. Today's class will delve more deeply into recursion than you may have gone before. Comfort with recursive implementations will be needed in upcoming classes.

A recursive function is defined in terms of base cases and recursive steps:

- In a base case, we compute the result immediately given the inputs to the function call.
- In a recursive step, we compute the result with the help of one or more recursive calls to this same function, but with the inputs somehow reduced in size or complexity, closer to a base case.

Consider writing a function to compute factorial. We can define factorial in two different ways:

Product

$$n! = n \times (n-1) \times \cdots \times 2 \times 1 = \prod_{k=1}^{n} k$$

(where the empty product equals multiplicative identity 1)

which leads to two different implementations:

Recursive step reduces n by 1

```
public static long factorial(int n) {
  long product = 1;
  for (int i = 1; i <= n; i++) {
    fact = fact * i;
  }
  return fact;
}
```

Recursive step reduces n by half

```
public static long factorial(int n) {
  if (n == 0 || n == 1) {
    return 1;
  }
  return n * factorial(n-1);
}
```

In the recursive implementation on the right, the base case is $n = 0$, where we compute and return the result immediately (it is defined to be 1. The recursive step is $n = 0$, where we compute the result with the help of a recursive call to obtain $(n-1)!$, then compute the computation by multiplying by n.

To visualize the execution of a recursive function, it is helpful to diagram the call stack of currently-executing functions as the computation proceeds.

Let's run the recursive implementation of factorial in a main method:

```
public static void main(String[] args) {
  long x = factorial(3);
}
```

At each step, with time moving left to right:

In the diagram, we can see how the stack grows as main calls factorial and factorial then calls itself, until factorial(0) does not make a recursive call. Then the call stack unwinds, each call to factorial returning its answer to the caller, until factorial(3) returns to main.

Here's an interactive visualization of factorial. You can step through the computation to see the recursion in action. New stack frames grow down instead of up in this visualization.

You've probably seen factorial before, because it's a common example for recursive functions. Another common example is the Fibonacci series:

```
/**
 * @param n >= 0
 * @return the nth Fibonacci number
 */
public static int fibonacci(int n) {
  if (n == 0 || n == 1) {
    return 1; // base cases
  } else {
    return fibonacci(n-1) + fibonacci(n-2); // recursive step
  }
}
```

Fibonacci is interesting because it has multiple base cases: n=0 and n=1. You can look at an interactive visualization of Fibonacci. Notice that where factorial's stack steadily grows to a maximum depth and then shrinks back to the answer, Fibonacci's stack grows and shrinks repeatedly over the course of the computation.

READING EXERCISES

- Recursive factorial
- Recursive Fibonacci

Choosing the Right Decomposition for a Problem

Finding the right way to decompose a problem, such as a method implementation, is important. Good decompositions are simple, short, easy to understand, safe from bugs, and ready for change.

Recursion is an elegant and simple decomposition for some problems. Suppose we want to implement this specification:

```
/**
 * @param word consisting only of letters A-Z or a-z
 * @return all subsequences of word, in order by commas,
 * where a subsequence is a string of letters found in word
 * in the same order that they appear in word.
 */
public static String subsequences(String word);
```

For example, subsequences("abc") might return "abc,ab,bc,ac,a,b,c,". Note the trailing comma preceding the empty subsequence, which is also a valid subsequence.

This problem lends itself to an elegant recursive decomposition. Take the first letter of the word. We can form one set of subsequences that include that letter, and another set of subsequences that exclude that letter, and those two sets completely cover the set of possible subsequences.

```
1 public static String subsequences(String word) {
2   if (word.isEmpty()) {
3     return "," // base case
4   } else {
5     char firstLetter = word.charAt(0);
6     String restOfWord = word.substring(1);
7
8     String subsequencesOfRest = subsequences(restOfWord);
9
10    String result = "";
11    for (String subsequence : subsequencesOfRest.split(",", -1)) {
12      result += ",," + subsequence;
13      result += ",," + firstLetter + subsequence;
14    }
15    return result.substring(1); // remove extra leading comma
16    return result;
17  }
18 }
```

READING EXERCISES

- subsequences("c")
- subsequences("gu")

Structure of Recursive Implementations

A recursive implementation always has two parts:

- **base case**, which is the simplest, smallest instance of the problem, that can't be decomposed any further. Base cases often correspond to empotiness – the empty string, the empty list, the empty set, the empty tree, zero, etc.
- **recursive step**, which **decomposes** a larger instance of the problem into one or more simpler or smaller instances that can be solved by recursive calls, and then **recombines** the results of those subproblems to produce the solution to the original problem.

It's important for the recursive step to transform the problem instance into something smaller, otherwise the recursion may never end. If every recursive step shrinks the problem, and the base case lies at the bottom, then the recursion is guaranteed to be finite.

A recursive implementation may have more than one base case, or more than one recursive step. For example, the Fibonacci function has two base cases, n=0 and n=1.

READING EXERCISES

- Recursive structure

Helper Methods

The recursive implementation we just saw for subsequences() is one possible recursive decomposition of the problem. We took a solution to a subproblem – the subsequences of the remainder of the string after removing the first character – and used it to construct solutions to the original problem, by taking each subsequence and adding the first character or omitting it. This is in a sense, a direct-minded, naive implementation, where we are using the existing specification of the recursive method to solve the subproblems.

In some cases, it's useful to require a stronger (or different) specification for the recursive steps, to make the recursive decomposition simpler or more elegant. In this case, what if we built up a partial subsequence using the initial letters of the word, and used the recursive calls to complete the subsequence using the remaining letters of the word? For example, suppose the original word is "orange". At first, we would select "o" to be in the partial subsequence, and recursively extend it with all subsequences of "range". Then, we would use "" as the partial subsequence, and again recursively extend it with all subsequences of "range".

Using this approach, our code now looks much simpler:

```
/**
 * Return all subsequences of word (as defined above) separated by commas,
 * with partialSubsequence prepended to each one.
 */
private static String subsequencesAfter(String partialSubsequence, String word) {
  if (word.isEmpty()) {
    // base case
    return partialSubsequence;
  } else {
    // recursive step
    return subsequencesAfter(partialSubsequence, word.substring(1))
        + ",,"
        + subsequencesAfter(partialSubsequence + word.charAt(0), word.substring(1)
);
  }
}
```

This subsequencesAfter method is called a **helper method**. It satisfies a different spec from the original subsequences, because it has a new parameter partialSubsequence. This parameter lifts a similar role that a local variable would in an iterative implementation. It holds temporary state during the evolution of the computation. The recursive calls steadily extend the partial subsequence, selecting or ignoring each letter in the word, until finally reaching the end of the word (the base case), at which point the partial subsequence is returned as the only result. Then the recursion backtracks and lifts in other possible subsequences.

To finish the implementation, we need to implement the original subsequences spec, which gets the ball rolling by calling the helper method with an initial value for the partial subsequence parameter:

```
public static String subsequences(String word) {
  return subsequencesAfter("", word);
}
```

Don't expose the helper method to your clients. Your decision to decompose the recursion this way instead of another way is entirely implementation-specific. In particular, if you discover that you need temporary variables like partialSubsequence in your recursion, don't change the original spec of your method, and don't force your clients to correctly initialize those parameters. That exposes your implementation to the client and reduces your ability to change it in the future. Use a private helper function for the recursion, and have your public method call it with the correct initializations, as shown above.

READING EXERCISES

- Unhelpful 1
- Unhelpful 2
- Unhelpful 3

Choosing the Right Recursive Subproblem

Let's look at another example. Suppose we want to convert an integer to a string representation with a given base, following this spec:

```
/**
 * @param n integer to convert to string
 * @param base base for the representation, Requires 2<=base<=36.
 * @return a represented as a string of digits in the specified base, with
 *         a minus sign if n<0.
 */
public static String stringValue(int n, int base)
```

For example, stringValue(16, 10) should return "10", and stringValue(16, 2) should return "10000".

Let's develop a recursive implementation of this method. One recursive step here is straightforward: we can handle negative integers simply by recursively calling the representation of the corresponding positive integer:

```
if (n < 0) return "-" + stringValue(-n, base);
```

This shows that the recursive subproblem can be smaller or simpler in more subtle ways than just the value of a numeric parameter or the size of a string or list parameter. We have still effectively reduced the problem by reducing it to positive integers.

The next question is, given that we have a positive n, say n=829 in base 10, how should we decomposse it into a recursive subproblem? Thinking about the number as we would write it down on paper, we could either start with 8 (the leftmost or highest-order digit), or 9 (the rightmost, lower-order digit). Starting at the left and seems natural, because that's the direction we write, but it's harder in this case, because we would need to find first the number of digits in the number to figure out how to extract the leftmost digit. Instead, a better way to decomposse it is to take the remainder modulo-base (which gives the rightmost digit) and also divide by base (which gives the subproblem, the remaining higher-order digits):

```
return stringValue(n/base, base) + "0123456789".charAt(n%base);
```

Think about several ways to break down the problem, and try to write the recursive steps. You want to find the one that produces the simplest, most natural recursive step.

It remains to figure out what the base case is, and include an if statement that distinguishes the base case from this recursive step.

READING EXERCISES

- Implementing stringValue
- Calling stringValue

Recursive Problems vs. Recursive Data

The examples we've seen so far have been cases where the problem structure lends itself naturally to a recursive definition. Factorial is easy to define in terms of smaller subproblems. Naming a recursive problem like this is one cue that you should pull a recursive solution out of your toolbox.

Another cue is when the data you are operating on is inherently recursive in structure. We'll see many examples of recursive data, a few classes from now, but for now let's look at the recursive data found in every laptop computer: its filesystem. A filesystem consists of named files. Some files are folders, which can contain other files. So a filesystem is recursive: folders contain other folders contain other folders, until finally at the bottom of the recursion are plain (non-folder) files.

The Java library represents the file system using java.io.File. This is a recursive data type, in the sense that f.getParentFile() returns the parent folder of a file f, which is a file object as well, and f.listFiles() returns the files contained by f, which is an array of other File objects.

For recursive data, it's natural to write recursive implementations:

```
/**
 * @param f a file in the filesystem
 * @return the full pathname of f from the root of the filesystem
 */
public static String fullPathname(File f) {
  if (f.getParentFile() == null) {
    // base case: f is at the root of the filesystem
    return f.getName();
  } else {
    // recursive step
    return fullPathname(f.getParentFile()) + "/" + f.getName();
  }
}
```

Recent versions of Java have added a new API, java.nio.files and java.nio.Path, which offer a cleaner separation between the filesystem and the pathnames used to name files in it. But the data structure is still fundamentally recursive.

Reentrant Code

Recursion – a method calling itself – is a special case of a general phenomenon in programming called **reentrancy**. Reentrant code can be safely re-entered, meaning that it can be called again even while a call to it is underway. Reentrant code keeps its state entirely in local parameters and local variables, and doesn't use static variables or global variables, and doesn't share objects in mutable ways with other parts of the program, or other calls to itself.

Direct recursion is one way that reentrancy can happen. We've seen many examples of that during this reading. The factorial() method is designed so that factorial(3) can be called even though factorial() hasn't yet finished working.

Mutual recursion between two or more functions is another way this can happen – A calls B, which calls A again. Direct mutual recursion is virtually always intentional and designed by the programmer. But unexpected mutual recursion can lead to bugs.

When we talk about concurrency later in the course, reentrancy will come up again, since in a concurrent program, a method may be called at the same time by different parts of the program that are running concurrently.

It's good to design your code to be reentrant as much as possible. Reentrant code is safer from bugs and can be used in more situations, like concurrency, callbacks, or mutual recursion.

When to Use Recursion Rather Than Iteration

We've seen two common reasons for using recursion:

- The problem is naturally recursive (e.g. Fibonacci)
- The data is naturally recursive (e.g. filesystem)

Another reason to use recursion is to take advantage of immutability. It is ok to recurse without worrying about mutating state, because the parameters and the local variables are not at all pure functions in the sense that they do not mutate anything. The behavior of a method can be understood simply as a relationship between its parameters and its return value, with no side effects on any other part of the program. This kind of paradigm is called functional programming, and it is far easier to reason about than imperative programming with loops and variables.

In iterative implementations, by contrast, you inevitably have non-final variables or mutable objects that are modified during the course of the iteration. Reasoning about the program then requires thinking about snapshots of the program state at various points in time, rather than thinking about pure input/output behavior.

One downside of recursion is that it may take more space than an iterative solution. Building up a stack of recursive calls consumes memory temporarily, and the stack is limited in size, which may become a limit on the size of the problem that your recursive implementation can solve.

Common Mistakes in Recursive Implementations

Here are two common ways that a recursive implementation can go wrong:

- The base case is missing entirely, or the problem needs more than one base case but not all the base cases are covered.
- The recursive step doesn't reduce to a smaller subproblem, so the recursion doesn't converge.

Look for these when you're debugging.

On the bright side, what would be an infinite loop in an iterative implementation usually becomes a StackOverflowError in a recursive implementation. A buggy recursive program fails faster.

READING EXERCISES

- subsequences("12345678")

Summary

We saw these ideas:

- recursive problems and recursive data
- comparing alternative decompositions of a recursive problem
- using helper methods to strengthen a recursive step
- recursion vs. iteration

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

- **Safe from bugs.** Recursive code is simpler and often uses immutable variables and immutable objects.
- **Easy to understand.** Recursive implementations for naturally recursive problems and recursive data are often shorter and easier to understand than iterative solutions.
- **Ready for change.** Recursive code is also naturally reentrant, which makes it safer from bugs and ready to use in more situations.

Reading 11: Debugging
Reproduce the Bug
Understand the Location and Cause of the Bug
Fix the Bug

You are not logged in.

Reading 11: Debugging

6.005 Prime Objective

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

Objectives

The topic of today's class is systematic debugging.

Sometimes you have no choice but to debug – particularly when the bug is found only when you plug the whole system together, or reported by a user after the system is deployed, in which case it may be hard to localize it to a particular module. For those situations, we can suggest a systematic strategy for more effective debugging.

Reproduce the Bug

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

Nevertheless, any effort you put into making the 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 develop a fix for it. Furthermore, after you've successfully fixed the bug, you'll want to add the test case to your regression test suite, so that the bug never crops up again. Once you have a test case for the bug, making this test work becomes your goal.

Here's an example. Suppose you have written this function:

```
/**
 * Find the most common word in a string.
 * @param text string containing zero or more words, where a word
 *     is a string of alphanumeric characters bounded by nonalphanumerics.
 * @return a word that occurs maximally often in text, ignoring alphabetic case.
 */
public static String mostCommonWord(String text) {
    ...
}
```

A user passes the whole text of Shakespeare's plays into your method, something like `mostCommonWord(allShakespearesPlaysConcatenated)`, and discovers that instead of returning a predictably common English word like `"the"` or `"a"`, the method returns something unexpected, perhaps `"e"`.

Shakespeare's plays have 100,000 lines containing over 800,000 words, so this input would be very painful to debug by normal methods, like print-debugging and breakpoint-debugging. Debugging will be easier if you first work on reducing the size of the buggy input to something manageable that still exhibits the same (or very similar) bug:

- does the first half of Shakespeare show the same bug? (Binary search! Always a good technique. More about this below.)
- 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 smaller test case, and then go back to the original buggy input and confirm that you fixed the same bug.

READING EXERCISES

Reducing a bug to a test case

Regression testing

Understand the Location and Cause of the Bug

To localize the bug and its cause, you can use the scientific method:

1. **Study the data.** Look at the test input that causes the bug, and the incorrect results, failed assertions, and stack traces that result from it.
2. **Hypothesize.** Propose a hypothesis, consistent with all the data, about where the bug might be, or where it *cannot* be. It's good to make this hypothesis general at first.
3. **Experiment.** Devise an experiment that tests your hypothesis. It's good to make the experiment an *observation* at first – a probe that collects information but disturbs the system as little as possible.
4. **Repeat.** Add the data you collected from your experiment to what you knew before, and make a fresh hypothesis. Hopefully you have ruled out some possibilities and narrowed the set of possible locations and reasons for the bug.

Let's look at these steps in the context of the `mostCommonWord()` example, fleshed out a little more with three helper methods:

```
/**
 * Find the most common word in a string.
 * @param text string containing zero or more words,
 *     where a word is a string of alphanumeric
 *     characters bounded by nonalphanumerics.
 * @return a word that occurs maximally often in text,
 *         ignoring alphabetic case.
 */
public static String mostCommonWord(String text) {
    ... words = splitIntoWords(text); ...
    ... frequencies = countOccurrences(words); ...
    ... winner = findMostCommon(frequencies); ...
    ... return winner;
}

/** Split a string into words ... */
private static List<String> splitIntoWords(String text) {
    ...
}

/** Count how many times each word appears ... */
private static Map<String,Integer> countOccurrences(List<String> words) {
    ...
}

/** Find the word with the highest frequency count ... */
private static String findMostCommon(Map<String,Integer> frequencies) {
    ...
}
```

1. Study the Data

One important form of data is the stack trace from an exception. Practice reading the stack traces that you get, because they will give you enormous amounts of information about where and what the bug might be.

The process of isolating a small test case may also give you data that you didn't have before. You may even have two related test cases that *bracket* the bug in the sense that one succeeds and one fails. For example, maybe `mostCommonWords("c c, b")` is broken, but `mostCommonWords("c c b")` is fine.

2. Hypothesize

It helps to think about your program as modules, or steps in an algorithm, and try to rule out whole sections of the program at once.

The flow of data in `mostCommonWord()` is shown at right. If the symptom of the bug is an exception in `countOccurrences()`, then you can rule out everything downstream, specifically `findMostFrequent()`.

Then you would choose a hypothesis that tries to localize the bug even further. You might hypothesize that the bug is in `splitIntoWords()`, corrupting its results, which then cause the exception in `countOccurrences()`. You would then use an experiment to test that hypothesis. If the hypothesis is true, then you would have ruled out `countOccurrences()` as the source of the problem. If it's false, then you would rule out `splitIntoWords()`.

3. Experiment

A good experiment is a gentle observation of the system without disturbing it much. It might be:

- Run a **different test case.** The test case reduction process discussed above used test cases as experiments.
- Insert a **print statement** or **assertion** in the running program, to check something about its internal state.
- Set a **breakpoint** using a debugger, 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. This is almost always the wrong thing to do. First, it leads to a kind of ad-hoc guess-and-test programming, which produces awful, complex, hard-to-understand code. Second, your fixes may just mask the true bug without actually removing it.

For example, if you're getting an `ArrayOutOfBoundsException`, try to understand what's going on first. Don't just add code that avoids or catches the exception, without fixing the real problem.

Other tips

Bug localization by binary search. Debugging is a search process, and you can sometimes use binary search to speed up the process. For example, in `mostCommonWords`, the data flows through three helper methods. To do a binary search, you would divide this workflow in half, perhaps guessing that the bug is found somewhere between the first helper method call and the second, and insert probes (like breakpoints, print statements, or assertions) there to check the results. From the answer to that experiment, you would further divide in half.

Prioritize your hypotheses. When making your hypothesis, you may want to keep in mind that different parts of the system have different likelihoods of failure. For example, old, well-tested code is probably more trustworthy than recently-added code. Java library code is probably more trustworthy than yours. The Java compiler and runtime, operating system platform, and hardware are increasingly more trustworthy, because they are more tried and tested. You should trust these lower levels until you've found good reason not to.

Swap components. If you have another implementation of a module that satisfies the same interface, and you suspect the module, then one experiment you can do is to try swapping in the alternative. For example, if you suspect your binarySearch() implementation, then substitute a simpler linearSearch() instead. If you suspect java.util.ArrayList, you could swap in java.util.LinkedList instead. If you suspect the Java runtime, run with a different version of Java. If you suspect the operating system, run your program on a different OS. If you suspect the hardware, run on a different machine. You can waste a lot of time swapping unfailing components, however, so don't do this unless you have good reason to suspect a component.

Make sure your source code and object code are up to date. Pull the latest version from the repository, and delete all your binary files and recompile everything (in Eclipse, this is done by Project → Clean).

Get help. It often helps to explain your problem to someone else, even if the person you're talking to has no idea what you're talking about. Lab assistants and fellow 6.005 students usually do know what you're talking about, so they're even better.

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

Fix the Bug

Once you've found the bug and understand its cause, the third step is to 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, like a misspelled variable or interchanged method parameters, or a design error, like an underspecified or insufficient interface. Design errors may suggest that you step back and revisit your design, or at the very least consider all the other clients of the failing interface to see if they suffer from the bug too.

Think also whether the bug has any relatives. If I just found a divide-by-zero error here, did I do that anywhere else in the 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?

Finally, after you have applied your fix, add the bug's test case to your regression test suite, and run all the tests to assure yourself that (a) the bug is fixed, and (b) no new bugs have been introduced.

READING EXERCISES

Debugging strategy

Summary

In this reading, we looked at how to debug systematically:

- reproduce the bug as a test case, and put it in your regression suite
- find the bug using the scientific method
- fix the bug thoughtfully, not slapdash

Thinking about our three main measures of code quality:

- **Safe from bugs.** We're trying to prevent them and get rid of them.
- **Easy to understand.** Techniques like static typing, final declarations, and assertions are additional documentation of the assumptions in your code. Variable scope minimization makes it easier for a reader to understand how the variable is used, because there's less code to look at.
- **Ready for change.** Assertions and static typing document the assumptions in an automatically-checkable way, so that when a future programmer changes the code, accidental violations of those assumptions are detected.

Collaboratively authored with contributions from: Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama.
This work is licensed under CC BY-SA 4.0.

Reading 12: Abstract Data Types
What Abstraction Means
Classifying Types and Operations
Designing an Abstract Type
Representation Independence
Realizing AOT Concepts in Java
Testing an Abstract Data Type
Summary

You are not logged in.

Reading 12: Abstract Data Types

Software in 6.005

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

Objectives

This reading's three introduces two ideas:

- Abstract data types
- Representation independence

In this reading, we look at a powerful idea, abstract data types, which enable us to separate how we use a data structure in a program from the particular way that we implement it in the program.

Abstract data types achieve a particularly powerful form of this separation. Clients making assumptions about the type's operations can be separated from the type's implementation, even if the type is mutable. We'll also discuss the classification of operations, and some principles of good design for abstract data types.

Access Control in Java

You should already have read **Controlling Access to Members of a Class** in the Java Tutorials.

READING EXERCISES

The following questions use the code below. Study it first, then answer the questions.

```
class Wallet {
    private int amount;

    public void loanTo(Wallet that) {
        // put all of this wallet's money into that wallet
/*1a*/  that.amount += this.amount;
/*1b*/  amount = 0;
    }

    public static void main(String[] args) {
        Wallet a = new Wallet();
/*2a*/  a.amount = 100;
/*2b*/  a.loanTo(a);
    }
}

class Person {
    private Wallet w;

    public int getMoneyHeld() {
/*3a*/  return w.amount;
    }

    public boolean isBroke() {
/*3b*/  return Wallet.amount == 0;
    }
}
```

Access control A

Access control B

Access control C

Access control D

Access control E

Access control F

Access control G

What Abstraction Means

Abstract data types are an instance of a general principle in software engineering, which goes by many names with slightly different shades of meaning. Here are some of the names that are used for this idea:

- **Abstraction.** Omitting or hiding low-level details with a simpler, higher-level idea.
- **Modularity.** Breaking a program into components that can be developed, tested, debugged, improved, tested, reimproved, tested, redebugged, and reused separately from the rest of the system.
- **Encapsulation.** Building walls around a module (a hard shell or capsule) so that the module is self-contained and independent, protecting its internal parts from being damaged by other modules.
- **Information hiding.** Hiding details of a module's 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 across multiple modules.

As a software engineer, you should know these terms, because you will run into them frequently. The fundamental purpose of all of these ideas is to help achieve the three important properties that we care about in 6.005: safety from bugs, ease of understanding, and readiness for change.

User-Defined Types

In the early days of computing, a programming language came with built-in types (such as integers, booleans, and floating-point numbers), sometimes also arrays, or lists, or tuples. Advanced users could define their own procedures, that's all. Programs were built from procedures.

A major advance in software development was the idea of abstract types. One can could design a programming language to allow user-defined types, too. This idea came out of the work of many researchers, notably Dahl the inventor of the Simula language, many (who developed many of the techniques we now use to reason about abstract types, Parnas who invented the ideas of information hiding and first articulated the idea of representation independence, Liskov who invented CLU and Argus, Guttag who invented the CLU language and wrote Guttag, who did seminal work on the specification of abstract types, and in programming language support for them – and developed the original 6.170, the predecessor to 6.005. Barbara Liskov earned the Turing Award, computer science's equivalent of the Nobel Prize, for her work on abstract types.

The key idea of data abstraction is that a type is characterized by the operations you can perform on it. A number is something you can add and subtract; a list is something you can append to and iterate through; a window is something you can move and resize; and so on. In a sense, users could already define their own types in early programming languages: you could create a record type date, for example, with integer fields for day, month, and year. But you can't add or subtract dates, you can't append to dates, and you can't iterate through them. You could also define a record type with fields for values every actually stored. In the same way that a programmer can ignore the complex actually stored images. All that matters is the operations.

In Java, as in most modern programming languages, the separation between type-in-use and user-defined types a lot thinly. The classes in java.lang, such as Integer and Boolean are built-in, whether you regard or not, and the primitive types int and boolean are not very different from Integer and Boolean. Java blurs the line by having primitive types that are not objects. The set of these types, such as int and boolean, cannot be extended by the user.

Classifying Types and Operations

Types, whether built-in or user-defined, can be classified as **mutable** or **immutable**. The objects of a mutable type can be changed; that is, they provide operations which when executed cause the results of other operations on the object to change. Immutable types don't provide any such operations. Strings are immutable and observe change with the .getPoint() operation. But String is immutable, because its operations create new String objects rather than changing existing ones. Sometimes a type will be provided in two forms, a mutable and an immutable form. StringBuilder, for example, is a mutable version of String (although the two are certainly not the same Java type, and are not interchangeable).

The operations of an abstract type are classified as follows:

- **Creators** create new objects of the type. A creator may take an object as an argument, but not an object of the type being created.
- **Producers** create new objects from old objects of the type. The concat method of String, for example, a producer. It takes two strings and produces a new one representing their concatenation. Creators and producers often call other creators and return objects of a different type. The .split() method of String, for example, returns an array.
- **Mutators** change objects. The add method of List, for example, mutates a list by adding an element to the end.

We can summarize these distinctions schematically like this (explanation to follow):

```
creator T ->T
producer T,* ->T
mutator T,* ->void
observer T,* ->anoOT
```

These show informally the shape of the signatures of operations in the various classes. Each T is the abstract type itself and is some other type. The * marker indicates that the type may occur one or more times in that part of the signature, and the | marker indicates that it occurs zero or more times. For example, a producer that takes two strings and produces a new one, concatenating them, has the signature String,String->String. We will stay also be confined, since some observers take no non-abstract arguments, and some take several.

A creator operation is often implemented as a constructor, like new ArrayList(). But 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**. The sample String.valueOf method in Java are other examples of creators implemented as factory methods.

Mutators are often signaled by a void return type. A method that returns void must be called for some kind of side effect, and that side effect is typically a mutation. But not always: a method that returns void may return a boolean indicate whether the set was actually changed. In Java's graphical user interface toolkit, Component.add() returns the object itself, so that multiple add() calls can be chained together.

Abstract Data Type Examples

Here are some examples of abstract data types, along with some of their operations, grouped by kind.

List is Java's primitive integer type. int is immutable, so it has no mutators.

- creators: the numeric literals 0, 1, 2, ...
- producers: arithmetic operators +, -, *, /, %
- observers: comparison operators <, >, <=, >=, ==, !=
- mutators: none (it's immutable)

List is Java's list type. List is mutable. List is also an interface, which means that other classes provide other implementations of the list type. These classes include ArrayList and LinkedList.

- creators: ArrayList and LinkedList constructors, Collections.singleton(int element), Collections.emptyList()
- observers: size, get
- mutators: add, remove, addAll, Collections.sort

String is Java's string type. String is immutable.

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

This classification gives some useful terminology, but it's not perfect. In complicated data types, there may be operations that are both producers and a mutator, for example. Some people reserve the term producer only for operations that do not mutation.

READING EXERCISES

Operations

Designing an Abstract Type

Designing an abstract type involves choosing good operations and determining how they should behave. Here are a few rules of thumb.

It's better to have **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 should have a **coherent** behavior rather than a panoply of special cases. We probably shouldn't add a sin operation to List, for example. It might help someone who is doing lots of trigonometry on lists of numbers, but it's strange! Or needed lots! All these special cases would make List a hard operation to understand and use.

The set of operations should be **adequate**: it the sense that there must be enough to do the kinds of computations clients are likely to want to do. A good test is to ask what kind of loop would be required if the type can be extended. For example, if there were no get operation, we would not be able to find out what the elements of a list are, and we would be unable to write a loop to iterate through the elements of a list. The add method is not strictly necessary for a list, because we could apply get on increasing indices until we get a failure, but this is inefficient and inconvenient.

The type may be general, a list of a set, or a graph, for example. Or it may be domain-specific: a street map, an employee database, a phone book, etc. But **it should not mix generic and domain-specific features**. A List of Strings should be a List of Strings. Clients shouldn't have a general .add() method that accepts arbitrary objects like integers or strings. Conversely, it wouldn't make sense to put a domain-specific method like deal(int) into the generic type List.

Representation Independence

Ultimately, a good abstract data type should be **representation independent**. This means that the use of an abstract type is independent of its representation (the actual data structure or data fields used to implement it). Clients of an abstract type should be able to use it without knowing how it is implemented. The type's operations offered by List are independent of whether the list is represented as a linked list or an array.

You won't be able to change the representation of an AOT at all unless its operations are fully specified with preconditions and postconditions, so that clients know what to depend on, and you know what you can change.

Example: Different Representations for Strings

Let's look at a simple abstract data type to see what representation independence means and why it's useful. The MyString type below has four fields and four operations that the real Java String, and their specs are a little different, but it's still illustrative. Here are the specs for the AOT:

```
// MyString represents an immutable sequence of characters, or
public class MyString {

  ///////////////// Example of a creator operation ///////////////
  // Overview: A boolean value
  //           Returns true if the sequence of 5, prior "True" or "False" or
  public static MyString valueOf(boolean b) { ... }

  ///////////////// Examples of observer operations ///////////////
  // Overview: number of characters in this string or
  public int length() { ... }

  // Overview: 1-character position (requires 0 <= i < string length)
  //           character at position i or
  public char charAt(int i) { ... }

  ///////////////// Example of a producer operation ///////////////
  // Overview: substring of this, starting at start (inclusive) and ending
  //           at, quote start starting index,
  //           quote and ending index, .Requires 0 <= start <= end <= string length,
  //           Returns a new string that contains the characters from start
  public MyString substring(int start, int end) { ... }
}
```

These public operations and their specifications are the only information that a client of this data type is allowed to know. Following the best rep-independence practice, in fact, the first client we should create is a test suite that exercises these operations according to their specs. At the moment, however, nothing stops a client that uses accessFields directly on MyString objects wouldn't work, because we don't have an equality operation yet, and the default one is the same as the operation on the underlying fields. But, for the moment, the only operations we can perform with MyString are the ones we've defined above: valueOf, length, charAt and substring. Our tests have to limit themselves to those operations. For example, here's one test for the valueOf operation:

```
MyString s = MyString.valueOf(true);
assertEquals(4, s.length());
assertEquals('t', s.charAt(0));
assertEquals('r', s.charAt(1));
assertEquals('u', s.charAt(2));
assertEquals('e', s.charAt(3));
```

We'll come back to the question of testing AOTs at the end of this reading.

Now, let's look at a simple representation for MyString: just an array of characters, exactly the length of the string, with the first character at the end. There's how that internal representation would be declared, as an instance variable within the class:

```
private char[] a;
```

With that choice of representation, the operations would be implemented in a straightforward way:

```
public static MyString valueOf(boolean b) {
  int len = (b ? 4 : 5);
  a = e = 0 // new char[0 + len]
  char[] ca = (b ? "t", "r", "u", "e" : "f", "a", "l", "s");
  for (int i = 0; i < len; i++)
    a[i] = ca[i];
  return s;
}

public int length() {
  return a.length;
}

public char charAt(int i) {
  return a[i];
}

public MyString substring(int start, int end) {
  MyString that = new MyString();
  that.a = this.a;
  that.start = this.start + start;
  that.end = this.start + end;
  return that;
}
```

Because MyString is working directly with the guts of its public methods, not on its private fields, it's easy to make the representation change to request and change all that client code. That's the power of representation independence.

READING EXERCISES

Representation 1

Representation 2

Representation 3

Representation 4

Realizing AOT Concepts in Java

Let's summarize some of the general ideas we've discussed in this reading, which are applicable in general to programming in any language, and their specific realization using Java language features. The point is that Java provides a lot of support for us to be able to specify and understand the big ideas, like a creator operation, and different ways to achieve that idea in practice.

The only item in this table that hasn't yet been discussed in this reading is the use of a constant object as a creator operation. This pattern is commonly seen in immutable types, where the empty or default value of the type is simply a public constant, and producers are used to build up more complex values from it.

AOT concept	Ways to do it in Java	Examples
Creator operation	Constructor Static factory method Constant	Array.copyOf() Collections.singleton(int), Arrays.asList() BigInteger.ZERO
Observer operation	Instance method Static method	List.get() Collections.max()
Producer operation	Instance method Static method	String.trim() Collections.concat(Iterable,int)
Mutator operation	Instance method Static method	List.add() Collections.copy()
Representation	private fields	

Testing an Abstract Data Type

We built a test suite for an abstract data type by creating tests for each of its operations. These tests inevitably interact with each other. The only way to test creators, producers, and mutators is by calling observers on the results, and the only way to test observers is by calling creators to create interesting objects for them to observe. Here's how we might combine the test spec of the four operations in our MyString type:

```
// testing strategy for each operation of MyString:
//
// valueOf():
//   boolean value
//   no inputs
//
//   string len = 4, 5, 6
//   string contents produced by valueOf(), produced by substring()
//
//   string len = 1, n
//   string contents produced by
//   string = produced by valueOf(), produced by substring()
// substring():
//   start = 0, n, 1, n
//   start = 0, middle, len
//   end = 0, middle, len
//   string length 1, n
//   string = produced by valueOf(), produced by substring()
```

Then a compact test suite that covers all these partitions might look like:

```
@Test public void testValueOfTrue() {
  MyString s = MyString.valueOf(true);
  assertEquals(4, s.length());
  assertEquals('t', s.charAt(0));
  assertEquals('r', s.charAt(1));
  assertEquals('u', s.charAt(2));
  assertEquals('e', s.charAt(3));
}

@Test public void testValueOfFalse() {
  MyString s = MyString.valueOf(false);
  assertEquals(5, s.length());
  assertEquals('f', s.charAt(0));
  assertEquals('a', s.charAt(1));
  assertEquals('l', s.charAt(2));
  assertEquals('s', s.charAt(3));
  assertEquals('e', s.charAt(4));
}

@Test public void testSubstringMiddle() {
  MyString s = MyString.valueOf(false);
  MyString t = s.substring(1, 3);
  assertEquals(2, t.length());
  assertEquals('a', t.charAt(0));
  assertEquals('l', t.charAt(1));
}

@Test public void testSubstringWholeString() {
  MyString s = MyString.valueOf(false);
  MyString t = s.substring(0, 5);
  assertEquals(5, t.length());
  assertEquals('f', t.charAt(0));
  assertEquals('a', t.charAt(1));
  assertEquals('l', t.charAt(2));
  assertEquals('s', t.charAt(3));
  assertEquals('e', t.charAt(4));
}
```

Notice that each test case typically calls a few operations that make or modify objects of the type (creators, producers, mutators) and some operations that inspect objects of the type (observers). As a result, each test case covers parts of several operations.

READING EXERCISES

These questions use the following datatype:

```
// Immutable datatype representing a student's progress through school, or
class Student {

  // new make a freshman rc
  public Student[] [ ... ]

  // pre-condition: a student promoted to the next year, i.e.
  //                freshman returns a sophomore,
  //                sophomore returns a junior,
  //                junior returns a senior,
  //                senior returns an alum
  //                post-condition: this can't be promoted further, or
  public Student promote() { ... }

  // pre-condition: number of years of school completed, i.e.
  //                0 for a freshman, 4 for an alum or
  public int getYear() { ... }

}
```

Partitioning AOT operations

Choosing AOT test cases

Summary

- Abstract data types are characterized by their operations.
- Operations can be classified into creators, producers, observers, and mutators.
- Good AOTs are few, simple, coherent, adequate, and representation-independent.
- A good AOT is simple, coherent, adequate, and representation-independent.
- An AOT is tested by generating tests for each of its operations, but using the creators, producers, mutators, and observers together in these tests.

These ideas connect to our three top properties of good software as follows:

- **Safe from bugs.** A good AOT offers a well-defined contract for its clients. Clients can trust its clients to expect from the data type, and implementers have well-defined freedom to vary.
- **Easy to understand.** A good AOT makes its representation behind a set of simple operations, so that programmers using the AOT only need to understand the operations, not the details of the representation.
- **Ready for change.** Representation independence allows the implementation of an abstract data type to change without requiring changes from its clients.

Creative commons: Attribution-Noncommercial-Share Alike 3.0 Unported License. Feedback: Report an error on this page. Contact us at 6005-staff@mit.edu. Privacy and Terms of Use. 6.005 Staff. Last updated: Wed Sep 9 12:48 EDT 2015.

You are not logged in.

Reading 13: Abstraction Functions & Rep Invariants

Software in 6.005

Safe from bugs | Easy to understand | Ready for change

Correct today and correct in the unknown future.

Understandable by maintainers who know programming, including future you.

Ready for change: Adapt to unknown future changes without rewriting.

Objectives

Today's reading introduces several ideas:

- invariants
- representation exposure
- representation rep
- representation rep invariants

In this reading, our study is more formal mathematical ideas of what it means for a class to implement an ADT, and what it means for a client to use an ADT correctly. We'll see how representation invariants and abstraction functions manage. The abstraction function will give us a way to relate, within the equality operation on an abstract data type (which will focus in more depth in a future class), the rep invariant will make it easier to verify that equality function is correct.

Invariants

Invariants are properties of a value that make a good abstract data type. Rather, and perhaps most important, whether or not a given abstract data type is that **consistent for each invariants**. An invariant is a property of a program that is always true, for some possible runtime state of the program. Invariants is one crucial property that we'll use to reason about the correctness of our program. Invariants are important because they make same days, for a reasonable lifetime. Reading that the ADT expressly do not invariants means that the ADT is responsible for ensuring that its own invariants hold. It doesn't depend on good behavior from its clients.

Invariants are also important because they make debugging easier. Invariants narrow the search for bugs. Instead of on the hard-to-find bugs never change, you can rule out that possibility when you're debugging code that uses Strings – or when you're trying to establish an invariant for another ADT that uses Strings. Contract that with a mutable ADT, which could be changed by some other code, and the invariant could be in danger. Then you have to check all the places that could alter the string right to see.

Immutability

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

```
// ...
// This immutable date type represents a human from Tech:
// ...
public class Tweet {
  // ...
  public String author;
  public String text;
  public Date tweetDate;

  // ...
  // Make a Tweet:
  // post-cond: Twitter user who wrote the tweet,
  // post-cond: text of the tweet,
  // post-cond: tweetDate is when the tweet was sent,
  // ...
  public Tweet(String author, String text, Date tweetDate) {
    this.author = author;
    this.text = text;
    this.tweetDate = tweetDate;
  }

  // ...
  // post-cond: System uses pre-write the tweet w/
  // tweetDate as timestamp:
  // ...
  public boolean post(String text) {
    return true;
  }

  // ...
  // post-cond: System logs the tweet was sent w/
  // tweetDate as timestamp:
  // ...
  public void logTweet(String text) {
    return (nothing);
  }

  // ...
  // Provides and public keywords indicate which fields and methods are accessible only within the class
  // and which can be accessed from outside the class. The final keyword also helps by guaranteeing that the
  // fields of this immutable type can't be reassigned after the object is constructed.
}
```

But that's not the end of the story. We've lied to exposed! Consider this perfectly reasonable client of the Tweet ADT:

```
// Consider a client that uses Tweets to log tweets:
public static Tweet[] tweetLogs(Tweet[] log) {
  int n = log.length;
  Tweet[] copy = new Tweet[n];
  for (int i = 0; i < n; i++) {
    copy[i] = log[i];
  }
  return copy;
}

// Client code:
Tweet[] log = new Tweet[100];
// ...
Tweet[] copy = tweetLogs(log);
log[0].setAuthor("Mr. Bean");
log[0].setText("Hello");
log[0].setDate(new Date());
// ...
log[0].post("Hello");
log[0].logTweet("Hello");
// ...
\n
```

This code intends to advance a single Date object through the 24 hours in a day, creating a tweet for every hour. But notice that the construction of Tweet saves the reference that was passed in, so all 24 Tweet objects share the same Date object, which ends up at midnight.

Again, the immutability of Tweet has been violated. We can fix the problem by using defensive defensive copying. We'll fine in the construction:

```
// Consider a client using Author, String, Text, Date (immutable):
// ...
public class Tweet {
  // ...
  // Make a Tweet:
  // post-cond: Twitter user who wrote the tweet,
  // post-cond: text of the tweet,
  // post-cond: tweetDate is when the tweet was sent, latter must never
  // ...
  public Tweet(String author, String text, Date tweetDate) {
    this.author = author;
    this.text = text;
    this.tweetDate = new Date(tweetDate.getTime());
  }
  // ...
}
```

In general, you should carefully inspect the arguments type and return types of your ADT operations. If any of the types are mutable, make sure your implementation doesn't return or store references to representation objects.

Is this the end of the story for immutability? Why make all these copies of dates? Why can't we just solve the problem by a carefully written specification, like this?

```
// ...
// Make a Tweet:
// post-cond: Twitter user who wrote the tweet,
// post-cond: text of the tweet,
// post-cond: tweetDate is when the tweet was sent, latter must never
// ...
public Tweet(String author, String text, Date tweetDate) {
  // ...
  this.tweetDate = tweetDate;
  // ...
}
```

The code intends to advance a single Date object through the 24 hours in a day, creating a tweet for every hour. But notice that the construction of Tweet saves the reference that was passed in, so all 24 Tweet objects share the same Date object, which ends up at midnight.

Again, the immutability of Tweet has been violated. We can fix the problem by the using defensive defensive copying. We'll fine in the construction:

```
// Consider a client using Author, String, Text, Date (immutable):
// ...
public class Tweet {
  // ...
  // Make a Tweet:
  // post-cond: Twitter user who wrote the tweet,
  // post-cond: text of the tweet,
  // post-cond: tweetDate is when the tweet was sent, latter must never
  // ...
  public Tweet(String author, String text, Date tweetDate) {
    this.author = author;
    this.text = text;
    this.tweetDate = new Date(tweetDate.getTime());
  }
  // ...
}
```

In general, you should carefully inspect the arguments type and return types of your ADT operations. If any of the types are mutable, make sure your implementation doesn't return or store references to representation objects.

Is this the end of the story for immutability? Why make all these copies of dates? Why can't we just solve the problem by a carefully written specification, like this?

```
// ...
// Make a Tweet:
// post-cond: Twitter user who wrote the tweet,
// post-cond: text of the tweet,
// post-cond: tweetDate is when the tweet was sent, latter must never
// ...
public Tweet(String author, String text, Date tweetDate) {
  // ...
  this.tweetDate = tweetDate;
  // ...
}
```

The code intends to advance a single Date object through the 24 hours in day, creating a tweet for every hour. But notice that the construction of Tweet saves the reference that was passed in, so all 24 Tweet objects share the same Date object, which ends up at midnight.

Again, the immutability of Tweet has been violated. We can fix the problem by the using defensive defensive copying. We'll fine in the construction:

```
// Consider a client using Author, String, Text, Date (immutable):
// ...
public class Tweet {
  // ...
  // Make a Tweet:
  // post-cond: Twitter user who wrote the tweet,
  // post-cond: text of the tweet,
  // post-cond: tweetDate is when the tweet was sent, latter must never
  // ...
  public Tweet(String author, String text, Date tweetDate) {
    this.author = author;
    this.text = text;
    this.tweetDate = new Date(tweetDate.getTime());
  }
  // ...
}
```

This approach is sometimes easier when there's only one other immutable alternative – for example, when the mutable object is too large to copy efficiently. But the cost in poor ability to reason about the program, and the risk of bugs, is very high. Immutable types are easier to reason about, because the values they represent always stand for an abstract value that has to guarantee its own invariants, and preserving its equivalence is essential to that.

Immutable types are also easier to prefer immutable types. If we had used an immutable date object, like `java.time.ZonedDateTime`, instead of the mutable `Date`, `java.util.Date`, then we would avoid the entire issue. Instead of using `Date`, we could use `ZonedDateTime`, which is supported by some types but not all of the standard library. `ZonedDateTime` is an immutable type, which means that its methods, such as `plusHours()`, return a new `ZonedDateTime` object.

So we've done some defensive copying in the secure value of `getTimeStamp()`. But we've forgotten to make the `setTweetTo()` field accessible. Consider this client code, which uses `Tweet` objects:

```
// Consider a Tweet and an inspecting Tweet[] and get Tweet today:
public static List<Tweet> Tweet[] = {};
public static void main(String[] args) {
  Date date = new Date();
  // ...
  Tweet[] log = new Tweet[];
  log.add(new Tweet("Mr. Bean", "Hello", date);
  // ...
  return log;
}

// ...
\n
```


This code intends to advance a single Date object through the 24 hours in day, creating a tweet for every hour. But notice that the construction of Tweet saves the reference that was passed in, so all 24 Tweet objects share the same Date object, which ends up at midnight.

Again, the immutability of Tweet has been violated. We can fix the problem by the using defensive defensive copying. We'll fine in the construction:

```
// Consider a Tweet and an inspecting Tweet[] and get Tweet today:
public static List<Tweet> Tweet[] = {};
public static void main(String[] args) {
  Date date = new Date();
  // ...
  Tweet[] log = new Tweet[];
  log.add(new Tweet("Mr. Bean", "Hello", date);
  // ...
  return log;
}

// ...
\n
```


This code intends to advance a single Date object through the 24 hours in day, creating a tweet for every hour. But notice that the construction of Tweet saves the reference that was passed in, so all 24 Tweet objects share the same Date object, which ends up at midnight.

Again, the immutability of Tweet has been violated. We can fix the problem by the using defensive defensive copying. We'll fine in the construction:

```
// Consider a Tweet and an inspecting Tweet[] and get Tweet today:
public static List<Tweet> Tweet[] = {};
public static void main(String[] args) {
  Date date = new Date();
  // ...
  Tweet[] log = new Tweet[];
  log.add(new Tweet("Mr. Bean", "Hello", date);
  // ...
  return log;
}

// ...
\n
```


Rep Invariant and Abstraction Function

We now have a deeper look at the Rational underlying abstract data type. The rational rep is not only elegant and interesting in its own right, it also has immediate practical application to the design and implementation of Rational. Rational is a good example of a library clients can't fix the design or internal bugs in, and so the data rep is their only public hope.

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

The first space of values is the space of values that the abstract type is supposed to represent. Mathematicians define this space of values as an abstract value space. Mathematicians don't care about how the abstract value is represented in a concrete value, but it helps to think of a concrete value as a particular concrete value. Mathematicians don't care about how the abstract value is represented in a concrete value, but it helps to think of a concrete value as a particular concrete value.

The space of abstract values consists of the values that the type is designed to represent. These are a feature of our imaginations. They're patterns written that don't exist on computers. But they are the way we want to think about the data we're representing. For example, the abstract value space for Rational is the set of rational numbers. Rational might have the mathematical integers as its abstract value space, but it might be interesting to have an array of primitive Doubleprimitive integers, as is not relevant to the user of the type.

The second space of values is the space of values that the type actually represents. In the implementation stage, which is the implementation's job to achieve the illusion of the abstract value space that the client sees.

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

```
public class Character {
  private String s;
  // ...
}
```

Then the rep space for a character values, the abstract space is the mathematical sets of characters. We can then ask how abstract value and concrete value are related. We can ask how abstract value and concrete value are related. We can ask how abstract value and concrete value are related.

- **Every abstract value is mapped to by some rep value.** The purpose of implementing the abstract type is to support operations on abstract values. Presumably, there will need to be data to create and manipulate all possible abstract values. Otherwise, the type wouldn't be useful.
- **Some abstract values are mapped to by more than one rep value.** The rep space may have more values than the abstract space, so it may be necessary for rep to represent an unordered set of characters as a string.
- **Not every rep value is mapped to by an abstract value.** If a rep mapped to this case, we have decided that the string should not contain duplicates. This will allow us to manipulate the various strings when we do the implementation of a particular character, since we know there can be at most one.

In practice, we can only illustrate a few elements of the rep spaces and their relationships. The graph on a whole is infinite. We can illustrate this by giving two things:

- An abstraction function that maps rep values to abstract values this represent.

```
AF: R -> A
```

- A rep invariant that tells us which rep values are legal.

```
RI: R -> boolean
```

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

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

```
public class Character {
  private String s;
  // ...
  // Constructor:
  // post-cond: s contains no repeated characters,
  // ...
  // AF: represents the set of characters found in s
  // ...
}
```

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

In fact, the abstraction function and rep invariant are a function of the implementation chosen for the same abstract type. A rep invariant could equally be appropriate for any string, as long as it's well-formed. It could be for each possible character. Clearly we need two different abstraction functions to map these two different rep invariants.

Its deep strikes why the choice of rep space doesn't determine AF and RI. There is not a single rep space for the rep, and thus choosing the values for the space of rep values, does not determine which of the rep values are legal, or which abstract value each rep value represents. The choice of rep space is part of the decision, so we can't have the rep space determine the choice. In addition, we could instead allow duplicates, but of the same time require that the characters be sorted, appearing in nondecreasing order. This would allow us to represent the set of characters `{a, b, c}` with the rep invariant `r.length() == 3`. Rather than the rep. Same rep value space – different rep invariant.

Even with the same type for the rep value space and the same rep invariant RI, we might still interpret the rep differently, with different abstraction functions AF. Suppose RI requires that the rep be a string of length 3, with no duplicates, but that the rep value is also supposed to be sorted, as a representation of the set that Rational is a given reason to let the rep decide the representation. Perhaps we'd interpret consecutive pairs of characters as quotangles, so that the string `"abc"` represents the set `{"ab", "bc"}`, and therefore represents the set `{(a,b), (b,c)}`. Here's what the AF and RI would look like for this representation:

```
public class Character {
  private String s;
  // ...
  // Rep invariant:
  // s.length == 3,
  // s contains no duplicates, s == s.toCharacter()
  // ...
  // Abstraction function: AF(s) = the set
  // {(s[i],s[i+1]) for each adjacent pair of characters
  // ...
}
```


The essential point is that designing an abstract type means not only **deciding how the rep works**: the abstract value space for the underlying rep and the rep value space for the implementation – **but also deciding which rep values to use and how to interpret them**.

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

READING EXERCISES

Who knows what?

Rep space problem

Trying to implement without an AF/RI

Example: Rational Numbers

Here's an example of an abstract data type for rational numbers. Look closely at its rep invariant and abstraction function.

```
public class Rational {
  private int numer, denom;
  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 Rational from n/d:
  // post-cond: n/d != 0,
  // ...
  public Rational(int n, int d) {
    if (n == 0) {
      numer = 0;
      denom = 1;
      return;
    }
    // Make a new Rational in lowest terms:
    // post-cond: n != 0,
    // ...
    int g = Math.gcd(Math.abs(n), Math.abs(d));
    int numer = n / g;
    int denom = d / g;
    // ...
    this.numer = numer;
    this.denom = denom;
  }

  // ...
  // Return a new Rational that is -r:
  // post-cond: r != null,
  // ...
  public Rational negate(Rational r) {
    return new Rational(-r.numer, r.denom);
  }

  public Rational(int n, int d) throws ArithmeticException {
    if (denom == 0) {
      // ...
      throw new ArithmeticException();
    }
    // ...
    if (d < 0) {
      n = -n;
      d = -d;
    }
    // ...
    int g = Math.gcd(Math.abs(n), Math.abs(d));
    int numer = n / g;
    int denom = d / g;
    // ...
    this.numer = numer;
    this.denom = denom;
  }
}
```

Here's a picture of the abstraction function and rep invariant for this code. The RI requires the rep to represent a rational number in lowest form (i.e., lowest terms), so pairs like (2,4) and (10,15) above should be shown as outside the RI.

It would be completely reasonable to design another implementation of this same ADT with a more permissive RI. With such a choice, some operations might become more expensive to perform, and others cheaper.

Checking the Rep Invariant

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

```
// Check that the invariant is true
// post-cond: throws an AssertionFailedException if invariant checking
// to warning: recommendations to fixes
private boolean checkRep() {
  assert denom > 0;
  assert gcd(numer, denom) == 1;
}
```

You should certainly add this method to assert the invariant at the end of every operation that creates or modifies the rep. In other words, constructors, producers, and mutators. Look back at the Rational code above, and see how this would work. Why is it useful to do this? Two reasons:

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

Why is `checkRep()` private? It's should be responsible for checking and enforcing a rep invariant – clients, or subclasses, shouldn't care.

No Null Values in the Rep

Recall that the `equals()` method that uses null values are troublesome and unsafe, so much so that we try to remove them from our programming entirely. In 6.005, this precondition prevents clients from using null values. Suppose we require that objects and rep values can't be null.

Then we can write a `checkRep()` method that checks all declared data types. By default, A `checkRep()` the rep invariant implicitly includes `r != null`. But we know differently. It's little that that object type (including references inside arrays) or lists. So if your rep is:

```
class Character {
  private Character[] s;
  // ...
}
```

then the rep invariant automatically includes `s != null`, and you don't need to state it in a rep invariant comment.

What's left? One is to ensure that rep values don't contain null. However, you can't ensure that `s != null`. Instead, you can check that your `checkRep()` correctly fail when `s == null`. Other ways to ensure for free from Java, because checking other parts of your rep invariant will throw an exception if `s` is null. For example, if your `checkRep()` looks like this:

```
private void checkRep() {
  assert s.length() > 0;
  // ...
}
```

then you don't need assert `s != null`, because the call to `s.length()` will fail just as effectively as a null reference. But if `s` is not otherwise checked by your invariant, then assert `s != null` explicitly.

READING EXERCISES

Checking the rep invariant

Documenting the AF, RI, and Safety from Rep Exposure

It's good practice to document the abstraction function and rep invariant in the code, using comments right beside the declaration of the rep. We've done this in the Rational code above.

Another piece of documentation that's 6.005 and you to write is the **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 the constructor and methods that return values), and argues that the rep is safe from exposure. Let's see how to prepare a rep exposure safety argument.

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

```
// Immutable type representing a tweet:
public class Tweet {
  // ...
  private final String author;
  private final String text;
  private final Date tweetDate;

  // Rep invariant:
  // author is a valid Twitter username to nonempty string of letters, digits, underscor
  // ...
  // Abstraction function:
  // represents a tweet posted by author, with content text, at time tweetDate
  // ...
  // Safety from rep exposure:
  // all fields are private and final,
  // author and text are strings, so are guaranteed immutable,
  // tweetDate is a Date, which is mutable, so we must be careful:
  // - make defensive copies to avoid sharing the rep with clients' data
  // ...

  // Operations (types and method bodies omitted to save space):
  public Tweet(String author, String text, Date tweetDate) {
    this.author = author;
    this.text = text;
    this.tweetDate = new Date(tweetDate.getTime());
  }
  // ...
}
```

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

READING EXERCISES

Proving rep exposure safety

How to Establish Invariants

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

To make an invariant hold, we need to:

- ensure that it holds when the object first exists of the object, and
- ensure that an changes to the object keep the invariant true.

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

- creators and producers must establish the invariant for new object instances, and
- mutators and observers must preserve the invariant.

Of course, we can't just write down an invariant and hope it holds. Instead, if the rep is violated, then the object might be changed unpredictably or dangerous, and you'd be ADT's guarantees, and we can't guarantee that the invariant still holds after those arbitrary changes. So the hot rule for proving invariants is:

Structural induction: If an invariant of an abstract data type is

- established by creators and producers,
- preserved by mutators and observers,
- its representation exposure-free,

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

READING EXERCISES

Structural induction

ADT invariants replace preconditions

Now let's look at a deeper subject: the eventual advantage of well-designed abstract data types that it enapsulates and enforces properties that we would otherwise have to stipulate in a precondition. For example, suppose we have a type like this, with a particular invariant:

```
// Entry<K,V> is a sorted set of characters with no repeats:
// ...
// Abstract Character Set:
// a finite sequence of characters
// ...
class SortedCharSet<K,V> {
  // ...
  // Rep invariant:
  // s.length() == 3,
  // s contains no duplicates,
  // s == s.toCharacter()
  // ...
  // Abstraction function:
  // AF(s) = the set {s[0], s[1], s[2]}
  // ...
  // All fields are private, and all types in the rep are immutable.
  // ...
  // Operations (types and method bodies omitted to save space):
  public SortedCharSet(String s) {
    this.s = s;
    // ...
  }
  // ...
}
```

Notice that an invariant has a particularly easy to argue for safety from rep exposure.

READING EXERCISES

Enforcing preconditions in ADTs

Summary

- An invariant is a property that is always true of an ADT object instance, for the lifetime of the object.
- Invariants are important because they make reasoning about correctness and debugging easier, and are preserved by observers and mutators.
- Immutability is a powerful invariant, it makes representation, and should be chosen and written with checkRep().
- Representation exposure threatens both representation independence and invariant preservation.

The basics of history is reading correct to our three properties of good software: is safe from bugs.

- **Safe from bugs.** A good ADT preserves its own invariants, so that those invariants are safe vulnerable to bugs in the ADT's clients, and violations of the invariants can be more easily isolated within the implementation of the ADT. Rep invariants, abstraction functions, and rep exposure safety arguments are all tools that help us identify and describe interrelationships and logic, rather than constructing an with a strong type system.
- **Easy to understand.** Rep invariants and abstraction functions explicitly describe the meaning of a data type's representation, and how it relates to its abstraction.
- **Ready for change.** Abstract data types separate the abstraction from the concrete representation, which makes it possible to change the representation without having to change client code.

Contents
Objectives
Invariants
Immutability
Rep Invariant and Abstraction Function
Example: Rational Numbers
Checking the Rep Invariant
No Null Values in the Rep
Documenting the AF, RI, and Safety from Rep Exposure
How to Establish Invariants
ADT invariants replace preconditions
Summary
Feedback
Report a bug
Show keyboard shortcuts
Fullscreen
Print view
Sections
Glossary
Discussion
Ask a question
Log out
Settings
Help
Exit
Prev
Next
6.005 Home
Collaboration
Schedule
Ask a question
Staff
Resources
Lectures
Reading
Exercises
Tools
Quiz
Checkoff
Assignments
Handout
Tips
Tools
Eclipse
Git
Gradle
Style guide
Testing guide
Debugging guide
Performance guide
Resources
Piazza
Stellar
Discussion
Ask a question
Staff

You are not logged in.

Reading 14: Interfaces
Interfaces
Example: MyString
Example: Set
Why Interfaces?
Realizing ADT Concepts in Java
Summary

Reading 14: Interfaces

Software in 6.005

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

Objectives

The topic of today's class is interfaces: separating the interface of an abstract data type from its implementation, and using Java `interface` types to enforce that separation.

After today's class, you should be able to define ADTs with interfaces, and write classes that implement interfaces.

Interfaces

Java's `interface` is a useful language mechanism for expressing an abstract data type. An interface in Java is a list of method signatures, but no method bodies. A class implements an interface if it declares the interface in its `implements` clause, and provides method bodies for all of the interface's methods. So one way to define an abstract data type in Java is as an interface, with its implementation as a class implementing that 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 programmer 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 an 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 abstract data type can co-exist in the same program, as different classes implementing the interface. When an abstract data type is represented just as a single class, without an interface, it's harder to have multiple representations. In the `MyString` example from *Abstract Data Types*, `MyString` was a single class. We explored two different representations for `MyString`, but we couldn't have both representations for the ADT in the same program.

Java's static type checking allows the compiler to catch many mistakes in implementing an ADT's contract. For instance, it is a compile-time error to omit one of the required methods, or to give a method the wrong return type. Unfortunately, the compiler doesn't check for us that the code adheres to the specs of those methods that are written in documentation comments.

In the Java Tutorials, read these pages:

- **Defining an Interface**
- **Implementing an Interface**
- **Using an Interface as a Type**

READING EXERCISES

Java interfaces

Example: `MyString`

Let's revisit `MyString`. Using an interface instead of a class for the ADT, we can support multiple implementations:

```
/** MyString represents an immutable sequence of characters. */
public interface MyString {

  // We'll skip this creator operation for now
  // /** @param b a boolean value
  // ∗ @return string representation of b, either "true" or "false" */
  // public static MyString valueOf(boolean b) { ...}

  /** @return number of characters in this string */
  public int length();

  /** @param i character position (requires 0 <= i < string length)
   ∗ @return character at position i */
  public char charAt(int i);

  /** Get the substring between start (inclusive) and end (exclusive).
   ∗ @param start starting index
   ∗ @param end ending index, Requires 0 <= start <= end <= string length.
   ∗ @return string consisting of charAt(start),...,charAt(end-1) */
  public MyString substring(int start, int end);
}
```

We'll skip the static `valueOf` method and come back to it in a minute. Instead, let's go ahead using a different technique from our toolbox of ADT concepts in Java: constructors.

Here's our first implementation:

```
public class SimpleMyString implements MyString {

  private char[] a;

  /* Create an uninitialized SimpleMyString. */
  private SimpleMyString() {}

  /** Create a string representation of b, either "true" or "false".
   ∗ @param b a boolean value */
  public SimpleMyString(boolean b) {
    a = b ? new char[] { 't', 'r', 'u', 'e' }
          : new char[] { 'f', 'a', 'l', 's', 'e' };
  }

  @Override public int length() { return a.length; }

  @Override public char charAt(int i) { return a[i]; }

  @Override public MyString substring(int start, int end) {
    SimpleMyString that = new SimpleMyString();
    that.a = new char[end - start];
    System.arraycopy(this.a, start, that.a, 0, end - start);
    return that;
  }
}
```

And here's the optimized implementation:

```
public class FastMyString implements MyString {

  private char[] a;
  private int start;
  private int end;

  /* Create an uninitialized FastMyString. */
  private FastMyString() {}

  /** Create a string representation of b, either "true" or "false".
   ∗ @param b a boolean value */
  public FastMyString(boolean b) {
    a = b ? new char[] { 't', 'r', 'u', 'e' }
          : new char[] { 'f', 'a', 'l', 's', 'e' };
    start = 0;
    end = a.length;
  }

  @Override public int length() { return end - start; }

  @Override public char charAt(int i) { return a[start + i]; }

  @Override public MyString substring(int start, int end) {
    FastMyString that = new FastMyString();
    that.a = this.a;
    that.start = this.start + start;
    that.end = this.start + end;
    return that;
  }
}
```

- Compare these classes to the implementations of `MyString` in *Abstract Data Types*. Notice how the code that previously appeared in static `valueOf`. methods now appears in the constructors, slightly changed to refer to the rep of `this`.
- Also notice the use of `@Override`. This annotation informs the compiler that the method must have the same signature as one of the methods in the interface we're implementing. But since the compiler already checks that we've implemented all of the interface methods, the primary value of `@Override` here is for readers of the code: it tells us to look for the spec of that method in the interface. Repeating the spec wouldn't be DRY, but saying nothing at all makes the code harder to understand.
- And notice the private empty constructors we use to make new instances in `substring(..)` before we fill in their reps with data. We didn't have to write these empty constructors before because Java provided them by default; adding the constructors that take `boolean b` means we have to declare the other constructors explicitly.

  Now that we know good ADTs scrupulously preserve their own invariants, these do-nothing constructors are a **bad** pattern: they don't assign any values to the rep, and they certainly don't establish any invariants. We should strongly consider revising the implementation. Since `MyString` is immutable, a starting point would be making all the fields `final`.

How will clients use this ADT? Here's an example:

```
MyString s = new FastMyString(true);
System.out.println("The first character is: " + s.charAt(0));
```

This code looks very similar to the code we write to use the Java collections classes:

```
List<String> s = new ArrayList<String>();
...
```

Unfortunately, this pattern **breaks the abstraction barrier** we've worked so hard to build between the abstract type and its concrete representations. Clients must know the name of the concrete representation class. Because interfaces in Java cannot contain constructors, they must directly call one of the concrete class' constructors. The spec of that constructor won't appear anywhere in the interface, so there's no static guarantee that different implementations will even provide the same constructors.

Fortunately, (as of Java 8) interfaces are allowed to contain static methods, so we can implement the creator operation `valueOf` as a static factory method in the interface `MyString`:

```
public interface MyString {

  /** @param b a boolean value
   ∗ @return string representation of b, either "true" or "false" */
  public static MyString valueOf(boolean b) {
    return new FastMyString(true);
  }

  // ...
```

Now a client can use the ADT without breaking the abstraction barrier:

```
MyString s = MyString.valueOf(true);
System.out.println("The first character is: " + s.charAt(0));
```

READING EXERCISES

Code review

Example: `Set`

Java's collection classes provide a good example of the idea of separating interface and implementation.

Let's consider as an example one of the ADTs from the Java collections library. `Set`. `Set` is the ADT of finite sets of elements of some other type `E`. Here is a simplified version of the `Set` interface:

```
/** A mutable set.
 ∗ @param <E> type of elements in the set */
public interface Set<E> {

  // ...
```

`Set` is an example of a generic type: a type whose specification is in terms of a placeholder type to be filled in later. Instead of writing separate specifications and implementations for `Set<String>`, `Set<Integer>`, and so on, we design and implement one `Set<E>`.

We can match Java interfaces with our classification of ADT operations, starting with a creator:

```
  // example creator operation
  /** Make an empty set.
   ∗ @param <E> type of elements in the set
   ∗ @return a new set instance, initially empty */
  public static <E> Set<E> make() { ... }
```

The `make` operation is implemented as a static factory method. Clients will write code like:
`Set<String> strings = Set.make();`
and the compiler will understand that the new `Set` is a set of `String` objects.

```
  // example observer operations

  /** Get size of the set.
   ∗ @return the number of elements in this set */
  public int size();

  /** Test for membership.
   ∗ @param e an element
   ∗ @return true iff this set contains e */
  public boolean contains(E e);
```

Next we have two observer methods. Notice how the specs are in terms of our abstract notion of a set; it would be malformed to mention the details of any particular implementation of sets with particular private fields. These specs should apply to any valid implementation of the set ADT.

```
  // example mutator operations

  /** Modifies this set by adding e to the set.
   ∗ @param e element to add */
  public void add(E e);

  /** Modifies this set by removing e, if found.
   ∗ If e is not found in the set, has no effect.
   ∗ @param e element to remove */
  public void remove(E e);
```

The story for these mutators is basically the same as for the observers. We still write specs at the level of our abstract model of sets.

In the Java Tutorials, read these pages:

- **Lesson: Interfaces**
- **The Set Interface**
- **Set Implementations**
- **The List Interface**
- **List Implementations**

READING EXERCISES

Collection interfaces & implementations

Why Interfaces?

Interfaces are used pervasively in real Java code. Not every class is associated with an interface, but there are a few good reasons to bring an interface into the picture.

- **Documentation for both the compiler and for humans**. Not only does an interface help the compiler catch ADT implementation bugs, but it is also much more useful for a human to read than the code for a concrete implementation. Such an implementation interspersed ADT-level types and specs with implementation details.
- **Allowing performance trade-offs**. Different implementations of the ADT can provide methods with very different performance characteristics. Different applications may work better with different choices, but we would like to code these applications in a way that is representation-independent. From a correctness standpoint, it should be possible to drop in any new implementation of a key ADT with simple, localized code changes.
- **Optional methods**. `List` from the Java standard library marks all mutator methods as optional. By building an implementation that does not support these methods, we can provide immutable lists. Some operations are hard to implement with good enough performance on immutable lists, so we want mutable implementations, too. Code that doesn't call mutators can be written to work automatically with either kind of list.
- **Methods with intentionally underdetermined specifications**. An ADT for finite sets could leave unspecified the element order one gets when converting to a list. Some implementations might use slower method implementations that manage to keep the set representation in some sorted order, allowing quick conversion to a sorted list. Other implementations might make many methods faster by not bothering to support conversion to sorted lists.
- **Multiple views of one class**. A Java class may implement multiple methods. For instance, a user interface widget displaying a drop-down list is natural to view as both a widget and a list. The class for this widget could implement both interfaces. In other words, we don't implement an ADT multiple times just because we are choosing different data structures; we may make multiple implementations because many different sorts of objects may also 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 is correct, while you can work harder to build a fancier version that is more likely to contain bugs. You can choose implementations for applications based on how bad it would be to get bitten by a bug.

Realizing ADT Concepts in Java

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.toList()`
		`BigInteger.ZERO`

Observer operation	Instance method Static method	`List.get()` `Collections.max()`
Producer operation	Instance method Static method	`String.trim()` `Collections.unmodifiableList()`
Mutator operation	Instance method Static method	`List.add()` `Collections.copy()`
Representation	`private` fields	

Summary

Java interfaces help us formalize the idea of an abstract data type as a set of operations that must be supported by a type.

This helps make our code...

- **Safe from bugs.** An ADT is defined by its operations, and interfaces do just that. When clients use an interface type, static checking ensures that they only use methods defined by the interface. If the implementation class exposes other methods — or worse, has visible representation — the client can't accidentally see or depend on them. When we have multiple implementations of a data type, interfaces provide static checking of the method signatures.
- **Easy to understand.** Clients and maintainers know exactly where to look for the specification of the ADT. Since the interface doesn't contain instance fields or implementations of instance methods, it's easier to keep details of the implementation out of the specifications.
- **Ready for change.** We can easily add new implementations of a type by adding classes that implement interface. If we avoid constructors in favor of static factory methods, clients will only see the interface. That means we can switch which implementation class clients are using without changing their code at all.

Collaboratively authored with contributions from: Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. Originally based on a thread under
CC BY-SA 4.0.

Reading 15: Equality
Introduction
Three Ways to Regard Equality (== vs. equals()
Equality of Immutable Types
The Object Contract
Equality of Mutable Types
The Final Rule for equals and hashCode()
Summary

You are not logged in.

Reading 15: Equality

Software in 6.005

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

Objectives

- Understand equality defined in terms of the abstraction function, an equivalence relation, and representation invariance.
- Differentiate between reference equality and object equality.
- Differentiate between shallow observational and behavioral equality for mutable types.
- Understand the Object contract and be able to implement equality correctly for mutable and immutable types.

Introduction

In the previous readings we've developed a rigorous notion of data abstraction by creating types that are characterized by their operations, not by their representation. For an abstract data type, the abstraction function explains how to interpret a concrete representation value as a value of the abstract type, and we saw how the choice of abstraction function determines how to write the code implementing each of the ADT's operations.

In this reading we turn to how we define the notion of equality of values in a data type. the abstraction function provides a key tool for understanding and reasoning about equality.

In the physical world, every object is distinct—at some level, even two snowflakes are different, even if the distinction is just the position they occupy in space. We can't literally look at all substance particles, but true enough at large objects like snowflakes and baseballs and people, the two physical objects are never truly "equal" to each other; they only have degrees of similarity.

In the world of abstract data types, which are defined by a set of mathematical concepts, you can have multiple names for the same thing. So it's natural to ask when two expressions represent the same thing: 1+2, 2+1 and 3 are alternative expressions for the same ideal mathematical value.

Three Ways to Regard Equality

Formally, we can regard equality in several ways.

Using an abstraction function. Recall that an abstraction function $\mathbb{R} \to A$ maps concrete instances of a data type to their corresponding abstract values. To use f as a definition for equality, we would say that a equals b if and only if $f(a) = f(b)$.

Using a relation. An equivalence is a relation $\sim \subseteq T \times T$ that is:

- reflexive: $\forall a \in T: a \sim a$
- symmetric: $\forall a, b \in T: a \sim b \Rightarrow b \sim a$
- transitive: $\forall a, b, c \in T: a \sim b \wedge b \sim c \Rightarrow a \sim c$

To use $\sim$ as a definition for equality, we would say that a equals b if and only if $a \sim b$.

These two notions are equivalent. An equivalence relation induces an abstraction function (the relation partitions the set into equivalence classes, and the abstraction function maps each element to an abstraction function is an equivalence relation (check for yourself that the three properties hold).

A third way we can talk about the equality between abstract values is in terms of what an outsider to (client) can observe about them:

Using observation. We can say that two objects are equal when they cannot be distinguished by the observation tests — every operation can apply produces the same result for both objects. Consider the set expressions {1,2} and {2,1}. Using the observer operations available for sets, contains() $\{\ldots\}$ and membership $\in$, these expressions are equivalent because:

- $\forall x \in \{1,2\} \in \{2,1\}$
- $1 \in \{1,2\}$ is true, and $1 \in \{2,1\}$ is true
- $2 \in \{1,2\}$ is true, and $2 \in \{2,1\}$ is true
- $3 \in \{1,2\}$ is false, and $3 \in \{2,1\}$ is false
- $|\{1,2\}| = |\{2,1\}| = 2$

In terms of abstract data types, "observation" means calling operations on the objects. So two objects are equal if and only if they cannot be distinguished by calling any operations of the abstract data type.

Example: Duration

Here's a simple example of an immutable ADT.

```
public class Duration {
  private final int mins;
  private final int secs;
  // AF(this) = this.mins * 60 + this.secs
  // abstraction function:
  //   represents a span of time of mins minutes and secs seconds

  /*m Make a duration lasting for m minutes and s seconds. */
  public Duration(int m, int s) {
    mins = m; secs = s;
  }
  /* pretty length of this duration in seconds */
  public long getLength() {
    return mins*60 + secs;
  }
}
```

Now which of the following values should be considered equal?

```
Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 2);
Duration d3 = new Duration (1, 61);
Duration d4 = new Duration (2, 2);
```

Think in terms of both the abstraction function definition of equality, and the observational equality definition.

READING EXERCISES

Any second now

Eye on the clock

== vs. equals()

Like many languages, Java has two different operations for testing equality, with different meanings.

- The `==` operator compares references. More precisely, it tests referential equality. Two references are `==` if they point to the same storage in memory. In terms of the conceptual diagrams we've been drawing, two references are `==` if their arrows point to the same object bubble.
- The `equals()` operation compares object contents—in other words, object equality. In the sense that we've been talking about it in this reading. The equals operation has to be defined appropriately for every object type.

For comparison, here are the equality operators in several languages:

	referential equality	*object equality*
Java	`==`	`equals()`
Objective C	`==`	`isEqual:`
C#	`==`	`equals()`
Python	`is`	`==`
Javascript	`===`	`==`

Note that `==` unfortunately flips its meaning between Java and Python. Don't let that confuse you: `==` in Java just tests reference identity, it doesn't compare object contents.

As programmers in any of these languages, we can't change the meaning of the referential equality operator. In Java, we can define an appropriate equals() operation for our data types, but we can't tell the compiler to decide what object equality means for values of the data type, and implement the `equals()` operation appropriately.

Equality of Immutable Types

The `equals()` method is defined by `Object`, and its default implementation looks like this:

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

In other words, the default meaning of `equals()` is the same as referential equality. For immutable data types, this is almost always wrong. So you have to **override** the `equals()` method, replacing it with your own implementation:

Here's our first try for `Duration`:

```
public class Duration {
  ...
  /* Problematic definition of equals()
  public boolean equals(Duration that) {
    return this.getLength() == that.getLength();
  }
}
```

There's a subtle problem here. Why doesn't this work? Let's try this code:

```
Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 2);
Object o1 = d1;
d1.equals(d2) // true
d1.equals(o1) // false
```

You can see this code in action. You'll see that even though d1 and d2 end up referring to the very same object in memory, you still get different results for them from `equals()`.

What's going on? It turns out that `Duration` has **overloaded** the `equals()` method, because the method signature differs from that in `Object`. `Duration` has `equals(Duration)` methods in `Duration`, an implicit `equals(Object)` inherited from `Object`, and the new `equals(Duration)`.

```
public class Duration extends Object {
  // implicitly inherited from Object:
  public boolean equals (Object that) {
    return this == that;
  }
  // implicitly inherited from Object:
  public boolean equals (Object that) {
    return this == that;
  }
}
```

`Duration` has inherited two ways that it knows will be spoken evenly over the indices. But occasionally a conflict occurs, and two keys are placed at the same index. So rather than holding a single value at an index, a hash table actually holds a list of key/value pairs, usually called a hash bucket. A key/value pair is implemented in Java simply as an object with two fields. On insertion, you add a pair to the list in the array slot determined by the hash code. For lookup you hash the key, find the right slot, and then examine each of the pairs until one is found that matches the key you're looking for.

Now it should be clear why the `Object` contract requires equal objects to have the same hashcode. If two equal objects had distinct hashcodes, they might be placed in different slots. So if you attempt to lookup a value using a key equal to the one with which it was originally stored, the lookup may fail.

`Object`'s default `hashCode()` implementation is consistent with the default `equals()`:

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

For references `a` and `b`, if `a == b`, then the address of `a` is the address of `b`. So the `Object` contract is satisfied.

But immutable objects need a different implementation of `hashCode()`. For `Duration`, since we haven't overridden the default `hashCode()` yet, we're currently breaking the `Object` contract:

```
Duration d1 = new Duration(1, 2);
Duration d2 = new Duration(1, 2);
d1.equals(d2) // true
d1.hashCode() // 2992
d2.hashCode() // 4823
```

`d1` and `d2` are `equal()`, but they have different hash codes. So we need to fix that.

A simple and drastic way to ensure that the contract is met is for `hashCode()` to always return some constant value, so every object's hash code is the same. This satisfies the `Object` contract, but it would have a terrible performance consequence—all the objects will be stored in the same slot, and every lookup will degenerate to a linear search along a long list.

The standard way to construct a more reasonable hash code that still satisfies the contract is to compute a hash code for each component of the object that is used in the observational equality, and then combine them. Usually by calling the `hashCode()` methods of each component, and then combining these. Knowing in a few arithmetic operations. For `Duration`, this is easy, because the abstract value of the class is already an integer value:

```
/Override
public int hashCode() {
  return (int) getLength();
}
```

Joshua Bloch's book, Effective Java, explains this issue in more detail, and gives some strategies for writing decent hash code functions. The advice is summarized in a good StackOverflow post. Recent versions of Eclipse have a utility method `Objects.hash()` that makes it easier to implement a hash code combining multiple fields.

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

Most crucially, note that it is your job to override `hashCode()` at all, you'll get the one from `Object`, which is based on the address of the object in memory. Unless you override `equals()`, this will mean that you will have almost certainly violated the contract. So as a general rule:

Always override hashCode when you override equals.

Many years ago it is proverbial to 6.005 (unfortunately hallmarked, it's 6.170), a student spent hours tracking down a bug in a project that amounted to nothing more than misspelling `hashCode` as `hashCode`. This created a new method that didn't override the `hashCode` method of `Object` at all, and strange things happened. Use `@Override`!

READING EXERCISES

Does the File code

Equality of Mutable Types

We've been focusing on equality of immutable objects so far in this reading. What about mutable objects?

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

- **Shallow observational equality:** two objects are equal if the changes made to the state of objects, `o1`, `o2` cannot be distinguished by observation. Shallow observational equality is **reference equality**, because it tests whether the two objects "look" the same, in the current state of the program.
- **Deep observational equality:** two objects are equal if the changes made to the state of objects, `o1`, `o2` cannot be distinguished by observation. Deep observational equality is **object equality**, because it tests whether the two objects look "behave" the same, in the rest of the future.

For immutable objects, observational and behavioral equality are identical, because there aren't any mutator methods.

For mutable objects, it's tempting to implement strict observational equality. Java uses observational equality for most of its mutable data types. In fact, if two distinct List objects contain the same sequence of elements, then `equals()` reports that they are equal.

But using observational equality leads to subtle bugs, and in fact allows us to easily break the key invariants of other collection data structures. Suppose we make a List, and then drop it into a Set:

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

Set<List<String>> set = new HashSet<>(Arrays.asSet(list));
set.add(list);
```

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

```
set.contains(list) // true
```

But now we mutate the list:

```
list.add("goodbye");
```

And it no longer appears in the set:

```
set.contains(list) // false
```

It's worse than that. In fact, when we iterate over the members of the set, we still find the list in them, but contains() says it's not there:

```
for (List<String> l : set) {
  set.contains(l) // false
}
```

If the set's core iterator and its contains() method disagree about whether an element is in the set, then the set clearly is broken. You can see this code in action on Online Python Tutor.

Worse, the set's `add()` method, which should be a simple operation, has a linear implementation of collection. Assume the List implementation offers the result of `equals()` and `hashCode()`. When the Set 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. So it can never be found again.

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

Here's a telling quote from the specification of `java.util.Set`:

> Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set.

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 behavioral equality.

The lesson we should draw from this example is that **`equals()` should implement behavioral equality**. In particular, it should be a function of the rep, and it should not be affected by mutation of the rep of the object. So mutable objects should just inherit `equals()` and `hashCode()` from `Object`. For objects that need a notion of observational equality (whether two mutable objects "look" the same in the current state), it's better to define a new method, e.g., `isSelfEqual()`.

The Final Rule for equals and hashCode()

For immutable types:

- Override `equals()` and compute abstract values. This is the same as saying `equals()` should provide behavioral equality.
- `hashCode()` should map the abstract value to an integer.

So immutable types must override both `equals()` and `hashCode()`.

For mutable types:

- `equals()` should compare references, just like `==`. Again, this is the same as saying `equals()` should provide behavioral equality.
- `hashCode()` should map the reference into an integer.

So mutable types should not override `equals()` and `hashCode()` at all, and should simply use the default implementation provided by `Object`. Java doesn't follow this rule for its collections, unfortunately, leading to the pitfalls that we saw above.

READING EXERCISES

Bag

Bag behavior

Bean bag

Autoboxing and Equality

Over more instructive pitfall in Java. We've talked about primitive types and their object type equivalents—for example, `int` and `Integer`. The object type implements `equals()` in the correct way, so that if you create two `Integer` objects with the same value, they'll be `equals()` to each other:

```
Integer a = new Integer(3);
Integer b = new Integer(3);
a.equals(b) // true
```

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

```
a == b // returns false
```

But for primitive types like `int`, `==` implements behavioral equality:

```
int3a = 3;
int3b = 3;
3a == 3b // returns true
```

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

```
Map<String, Integer> m = new HashMap<>();
m.put("a", 3000); // put some items into the map
m.put("b", 128);
m.put("c", 3000); // is "c" equal to "a"? it should be, but it isn't!
```

You can see this code in action on Online Python Tutor.

Summary

- Equality should be an equivalence relation (reflexive, symmetric, transitive).
- Equality and hashcode should be consistent (with each other, so that data structures that use hash tables work correctly, and with mutability).
- The abstraction function is the basis for equality in immutable data types.
- Reference equality is the basis for equality in mutable data types. this is the only way to ensure consistency over time and avoid breaking rep invariants of hash tables.

Equality is one part of implementing an abstract data type, and we've always been important ADTs so we can reason about our programs. Equality is important for three reasons:

- **Safe from bugs.** Correct implementation of equality and hash codes is necessary for use with collection data types like sets and maps. It's also highly desirable for writing tests. Since every object in Java inherits the `Object` implementations, immutable types must override them.
- **Easy to understand.** Clients and other programmers rely read our specs will expect our types to implement an appropriate equality operation, and will be surprised and confused if we do not.
- **Ready for change.** Correctly implemented equality for immutable types separates equality of reference from equality of value, so that we can use immutable values in a mutable implementation without trouble. Choosing behavioral rather than observational equality for mutable types helps avoid unexpected aliasing bugs.

Collaborative, activity-based learning with course-based forums. Powered by the open-source edX platform. Developed at Harvard, Stanford, and Texas A&M. Contact us at info@edx.org. Terms of Service and Honor Code. Privacy Policy. © 2015 edX Inc.

Reading 16: Recursive Data Types
Introduction
Part 1: Recursive Data Types
Part 2: Writing a Program with ADTs
Summary

Reading 16: Recursive Data Types

Software in 6.005

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

Objectives

- Understand recursive datatypes
- Read and write datatype definitions
- Understand and implement functions over recursive datatypes
- Understand immutable lists and know the standard operations on immutable lists
- Know and follow a recipe for writing programs with ADTs

Introduction

In this reading we'll look at recursively-defined types, how to specify operations on such types, and how to implement them. Our main example will be *immutable lists*.

Then we'll use another recursive datatype example, *matrix multiplications*, to walk through our process for programming with ADTs.

Part 1: Recursive Data Types

Part 2: Writing a Program with ADTs

Summary

Let's review how recursive datatypes fit in with the main goals of this course:

- **Safe from bugs**. Recursive datatypes allow us to tackle problems with a recursive or unbounded structure. Implementing appropriate data structures that encapsulate important operations and maintain their own invariants is crucial for correctness.
- **Easy to understand**. Functions over recursive datatypes, specified in the abstract type and implemented in each concrete variant, organize the different behavior of the type.
- **Ready for change**. A recursive ADT, like any ADT, separates abstract values from concrete representations, making it possible to change low-level code and high-level structure of the implementation without changing clients.

Collaboratively authored with contributions from: Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. This work is licensed under CC BY-SA 4.0.

Reading 17: Regular Expressions & Grammars
Introduction
Grammars
Regular Expressions
Summary

You are not logged in.

Reading 17: Regular Expressions & Grammars

Software in 6.005

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

Objectives

After today's class, you should:

- Understand the ideas of grammar productions and regular expression operators
- Be able to read a grammar or regular expression and determine whether it matches a sequence of characters
- Be able to write a grammar or regular expression to match a set of character sequences and parse them into a data structure

Introduction

Today's reading introduces several ideas:

- grammars, with productions, nonterminals, terminals, and operators
- regular expressions
- parser generators

Some program modules take input or produce output in the form of a sequence of bytes or a sequence of characters, which is called a string when it's simply stored in memory, or a stream when it flows into or out of a module. In today's reading, we talk about how to write a specification for such a sequence. Concretely, a sequence of bytes or characters might be:

- A file on disk, in which case the specification is called the file format
- Messages sent over a network, in which case the specification is a wire protocol
- A command typed by the user on the console, in which case the specification is a *command line interface*
- A string stored in memory

For these kinds of sequences, we introduce the notion of a grammar, which allows us not only to distinguish between legal and illegal sequences, but also to parse a sequence into a data structure that a program can work with. The data structure produced from a grammar will often be a recursive data type like we talked about in the recursive data type reading.

We also talk about a specialized form of a grammar called a *regular expression*. In addition to being used for specification and parsing, regular expressions are a widely-used tool for many string processing tasks that need to disassemble a string, extract information from it, or transform it.

The next reading will talk about parser generators, a kind of tool that translate a grammar automatically into a parser for that grammar.

Grammars

To describe a sequence of symbols, whether they are bytes, characters, or some other kind of symbol drawn from a fixed set, we use a compact representation called a grammar.

A grammar defines a set of sentences, where each sentence is a sequence of symbols. For example, our grammar for URLs will specify the set of sentences that are legal URLs in the HTTP protocol.

The symbols in a sentence are called terminals (or tokens).
They're called terminals because they are the leaves of a tree that represents the structure of the sentence. They don't have any children, and can't be expanded any further. We generally write terminals in quotes, like `'http'` or `':'`.

A grammar is described by a set of *productions*, where each production defines a *nonterminal*. You can think of a nonterminal like a variable that stands for a set of sentences, and the production as the definition of that variable in terms of other variables (nonterminals), operators, and constants (terminals). Nonterminals are internal nodes of the tree representing a sentence.

A production in a grammar has the form:

```
nonterminal ::= expression of terminals, nonterminals, and operators
```

In 6.005, we will name nonterminals using lowercase identifiers, like `x` or `y` or `url`.

One of the nonterminals of the grammar is designated as the root. The set of sentences that the grammar recognizes are the ones that match the root nonterminal. This nonterminal is often called `root` or `start`, but in the grammars below we will typically choose more memorable names like `url`, `html`, and `markdown`.

Grammar Operators

The three most important operators in a production expression are:

- concatenation

```
x ::= y z     an x is a y followed by a z
```

- repetition

```
x ::= y*      an x is zero or more y
```

- union

```
x ::= y | z   an x is a y or a z
```

You can also use additional operators which are just syntactic sugar (i.e., they're equivalent to combinations of the big three operators):

- option (0 or 1 occurrence)

```
x ::= y?      an x is a y or is the empty sentence
```

- 1+ repetition (1 or more occurrences)

```
x ::= y+      an x is one or more y
              (equivalent to  x ::= y y* )
```

- character classes

```
x ::= [abc]   is equivalent to  x ::= 'a' | 'b' | 'c'

x ::= [^b]    is equivalent to  x ::= 'a' | 'c' | 'd' | 'e' | 'f'
                                | ... (all other characters)
```

By convention, the operators `+`, `?`, and `*` have highest precedence, which means they are applied first. Alternation `|` has lowest precedence, which means it is applied last. Parentheses can be used to override this precedence, so that a sequence or alternation can be repeated:

- grouping using parentheses

```
x ::= (y z | a b)+   an x is zero or more y-z or a-b pairs
```

READING EXERCISES

Reading a Grammar 1

Reading a Grammar 2

Reading a Grammar 3

Reading a Grammar 4

Example: URL

Suppose we want to write a grammar that represents URLs. Let's build up a grammar gradually by starting with simple examples and extending the grammar as we go.

Here's a simple URL:

```
http://mit.edu/
```

A grammar that represents the set of sentences containing only this URL would look like:

```
url ::= 'http://mit.edu/'
```

But we'll generalize it to capture other domains, as well:

```
http://stanford.edu/
http://google.com/
```

We can write this as one line, like this:

```
url ::= 'http://' [a-z]+ '.' [a-z]+ '/'
```


This grammar represents the set of all URLs that consist of just a two-part hostname, where each part of the hostname consists of 1 or more letters. So `http://mit.edu/` and `http://yahoo.com/` would match, but not `http://cs012.com/`. Since it has only one nonterminal, a parse tree for this URL grammar would look like the picture on the right.

In this one-line form, with a single nonterminal whose production uses only operators and terminals, a grammar is called a *regular expression* (more about that later). But it will be easier to understand if we name the parts using new nonterminals:

```
url ::= 'http://' hostname '/'
hostname ::= word '.' word
word ::= [a-z]+
```


The parse tree for this grammar is now shown at right. The tree has more structure now. The leaves of the tree are the parts of the string that have been parsed. If we concatenated the leaves together, we would recover the original string. The `hostname` and `word` nonterminals are labeling nodes of the tree whose subtrees match those rules in the grammar. Notice that the immediate children of a nonterminal node like `hostname`: follow the pattern of the `hostname` rule, `word '.' word`.

How else do we need to generalize? Hostnames can have more than two components, and there can be an optional port number:

```
http://didit.csail.mit.edu:8949/
```

To handle this kind of string, the grammar is now:

```
url ::= 'http://' hostname (':' port?) '/'
hostname ::= word '.' hostname | word '.' word
port ::= [0-9]+
word ::= [a-z]+
```

Notice how *hostname* is now defined recursively in terms of itself. Which part of the hostname definition is the base case, and which part is the recursive step? What kinds of hostnames are allowed?

Using the repetition operator, we could also write hostname like this:

```
hostname ::= (word '.')+ word
```

Another thing to observe is that this grammar allows port numbers that are not technically legal, since port numbers can only range from 0 to 65535. We could write a more complex definition of port that would allow only these integers, but that's not typically done in a grammar. Instead, the constraint 0 <= port <= 65535 would be specified alongside the grammar.

There are more things we should do to go further:

- generalizing `http` to support the additional protocols that URLs can have
- generalizing the `/` at the end to a slash-separated path
- allowing hostnames with the full set of legal characters instead of just a-z

READING EXERCISES

Writing a Grammar

Example: Markdown and HTML

Now let's look at grammars for some file formats. We'll be using two different markup languages that represent typographic style in text. Here they are:

Markdown

```
This is _italic_.
```

HTML

```
Here is an <i>italic</i> word.
```

For simplicity, our example HTML and Markdown grammars will only specify italics, but other text styles are of course possible.

Here's the grammar for our simplified version of Markdown:

```
markdown ::= ( normal | italic ) *
italic ::= '_' normal '_'
normal ::= text
text ::= [^_]+
```


Here's the grammar for our simplified version of HTML:

```
html ::= ( normal | italic ) *
italic ::= '<i>' html '</i>'
normal ::= text
text ::= [^<]+
```


READING EXERCISES

Recursive Grammars

Regular Expressions

A regular grammar has a special property: by substituting every nonterminal (except the root one) with its righthand side, you can reduce it down to a single production for the root, with only terminals and operators on the right-hand side.

Our URL grammar was regular. By replacing nonterminals with their productions, it can be reduced to a single expression:

```
url ::= 'http://' ([a-z]+ '.')+ [a-z]+ ('/' [0-9]+)? '/'
```

The Markdown grammar is also regular:

```
markdown ::= ([^_]* | '_' [^_]+ '_' )*
```

But our HTML grammar can't be reduced completely. By substituting righthand sides for nonterminals, you can eventually reduce it to something like this:

```
html ::= ( [^<]+ | '<i>' html '</i>' ) *
```

... but the recursive use of `html` on the righthand side can't be eliminated, and can't be simply replaced by a repetition operator either. So the HTML grammar is not regular.

The reduced expression of terminals and operators can be written in an even more compact form, called a *regular expression*. A regular expression does away with the quotes around the terminals, and the spaces between terminals and operators, so that it consists just of terminal characters, parentheses for grouping, and operator characters. For example, the regular expression for our `markdown` format is just

```
([^_]*|_[^_]+_)*
```

Regular expressions are also called regexes for short. A regex is far less readable than the original grammar, because it lacks the nonterminal names that documented the meaning of each subexpression. But a regex is fast to implement, and there are libraries in many programming languages that support regular expressions.

The regex syntax commonly implemented in programming language libraries has a few more special operators, in addition to the ones we used above in grammars. Here's the are some common useful ones:

```
.      any single character
\d     any digit, same as [0-9]
\s     any whitespace character, including space, tab, newline
\w     any word character, including letters and digits

\., \/, \), \*, \+, \?, ...
       quotes an operator or special character so that it matches literally
```

Using backslashes is important whenever there are terminal characters that would be confused with special characters. Because our `url` regular expression has `.` in it as a terminal, we need to use a backslash to escape it:

```
http:(([a-z]+\.)+[a-z]+(:[0-9]+)/
```

READING EXERCISES

Regular Expressions

Using regular expressions in Java

Regular expressions ("regexes") are widely used in programming, and you should have them in your toolbox.

In Java, you can use regexes for manipulating strings (see String.split, String.matches, java.util.regex.Pattern). They're built in as a first-class feature of modern scripting languages like Python, Ruby, and Javascript, and you can use them in many text editors for find and replace. Regular expressions are your friend. Most of the time. Here are some examples.

Replace all runs of spaces with a single space:

```
String trimmedString = string.replaceAll(" +", " ");
```

Match a URL:

```
Pattern regex = Pattern.compile("http://([a-z]+\.)+[a-z]+(:[0-9]+)?/");
Matcher m = regex.matcher(string);
if (m.matches()) {
  // then string is a url
}
```

Extract part of an HTML tag:

```
Pattern regex = Pattern.compile("<a href=(\"[^\"]*\")+>([^<]*)</a>");
Matcher m = regex.matcher(string);
if (m.matches()) {
  String url = m.group(1);
  // Matcher.group(n) returns the nth parenthesized part of the regex
}
```

Notice the backslashes in the URL and HTML tag examples. In the URL example, we want to match a literal period `.`, so we have to first escape it as `\.`. To protect it from being interpreted as the regex match-any-character operator, and then we have to further escape it as `\\.`. To protect the backslash from being interpreted as a Java string escape character, in the HTML example, we have to escape the quote mark `"` as `\"` to keep it from ending the string. The frequency of backslash escapes makes regexes still less readable.

READING EXERCISES

Using regexes in Java

Context-Free Grammars

In general, a language that can be expressed with our system of grammars is called context-free. Not all context-free languages are also regular; that is, some grammars can't be reduced to single nonterminal productions. Our HTML grammar is context-free but not regular.

The grammars for most programming languages are also context-free. In general, any language with nested structure (like nesting parentheses or braces) is context-free but not regular. That description applies to the Java grammar, shown here in part:

```
statement ::=
  '{' statements '}'
| 'if' '(' expression ')' statement ('else' statement)?
| 'for' '(' forinit ';' expression? ';' forupdate? ')' statement
| 'while' '(' expression ')' statement
| 'do' statement 'while' '(' expression ')' ';'
| 'try' '{' statements '}' ( catches | catches? 'finally' '{' statements* '}' )
| 'switch' '(' expression ')' '{' switchgroups '}'
| 'synchronized' '(' expression ')' statement '}'
| 'return' expression? ';'
| 'throw' expression ';'
| 'break' identifier? ';'
| 'continue' identifier? ';'
| expression ';'
| identifier ':' statement
| ';'
```

Summary

Machine-processed textual languages are ubiquitous in computer science. Grammars are the most popular formalism for describing such languages, and regular expressions are an important subclass of grammars that can be expressed without recursion.

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

- **Safe from bugs.** Grammars and regular expressions are declarative specifications for strings and streams, which can be used directly by libraries and tools. These specifications are often simpler, more direct, and less likely to be buggy than parsing code written by hand.
- **Easy to understand.** A grammar captures the shape of a sequence in a form that is easier to understand than hand-written parsing code. Regular expressions, also, are often not easy to understand, because they are a one-line reduced form of what might have been a more understandable regular grammar.
- **Ready for change.** A grammar can be easily edited, but regular expressions, unfortunately, are much harder to change, because a complex regular expression is cryptic and hard to understand.

Collaboratively authored with contributions from: Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, Joe Gibbs Politz, Daniel Jackson, Bob Miller, Martin Rinard, and Martin Rinard and others. This work is licensed under CC BY-SA 4.0.

Reading 18: Parser Generators
Parser Generators
An Antlr Grammar
Generating the parser
Calling the parser
Traversing the parse tree
Constructing an abstract syntax tree
Handling errors
Summary

You are not logged in.

Reading 18: Parser Generators

Software in 6.005

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

Objectives

After today's class, you should:

- Be able to use a grammar in combination with a parser generator, to parse a character sequence into a parse tree
- Be able to convert a parse tree into a useful data type

Parser Generators

A parser generator is a good tool that you should make part of your toolbox. A parser generator takes a grammar as input and automatically generates source code that can parse streams of characters using the grammar.

The generated code is a parser, which takes a sequence of characters and tries to match the sequence against the grammar. The parser typically produces a parse tree, which shows how grammar productions are expanded into a sentence that matches the character sequence. The root of the parse tree is the starting nonterminal of the grammar. Each node of the parse tree expands into one production of the grammar. We'll see how a parse tree actually looks in the next section.

The final step of parsing is to do something useful with this parse tree. We're going to translate it into a value of a recursive data type. Recursive abstract data types are often used to represent an expression in a language, like HTML, or Markdown, or Java, or algebraic expressions. A recursive abstract data type that represents a language expression is called an *abstract syntax tree* (AST).

Antlr is a mature and widely-used parser generator for Java, and other languages as well. The remainder of this reading will get you started with Antlr. If you run into trouble and need a deeper reference, you can look at:

- Definitive Antlr 4 Reference. A book about Antlr, both tutorial and reference.
- Antlr 4 Documentation Wiki. Concise documentation for grammar syntax.
- Antlr 4 Runtime API. Reference documentation for Antlr's Java classes and interfaces.

An Antlr Grammar

The code for the examples that follow can be found on Github as **fa15-ex18-parser-generators**.

Here is what our HTML grammar looks like as an Antlr source file:

```
grammar Html;

root : html EOF;
html : (italic | normal) * ;
italic : '<i>' html '</i>';
normal : TEXT;
TEXT : ~[<>]+;  /* represents a string of one or more characters that are not < or > */
```

Let's break it down.

Each Antlr rule consists of a name, followed by a colon, followed by its definition, terminated by a semicolon.

Nonterminals in Antlr have to be lowercase. `root`, `html`, `normal`, `italic`. Terminals are either quoted strings, like `'<i>'`, or capitalized names, like `EOF` and `TEXT`.

```
root : html EOF;
```

`root` is the entry point of the grammar. This is the nonterminal that the whole input needs to match. We don't have to call it `root`. The entry point can be any nonterminal.

`EOF` is a special terminal, defined by Antlr, that means the end of the input. It stands for *end of file*, though your input may also come from a string or a network connection rather than just a file.

```
html : (normal | italic) * ;
```

This rule shows that Antlr rules can have the alternation operator `|`, the repetition operators `*` and `+`, and parentheses for grouping, in the same way we've been using in the grammars reading. Optional parts can be marked with `?`, just like we did earlier, but this particular grammar doesn't use `?`.

```
italic : '<i>' html '</i>';
normal : TEXT;
TEXT : ~[<>]+;
```

`TEXT` is a terminal matching sequences of characters that are neither `<` nor `>`. In the more conventional regular expression syntax used earlier in this reading, we would write `[^<>]` to represent characters except `<` and `>`. Antlr uses a slightly different syntax `~` means not, and it is put in front of the square brackets instead of inside them, so `~[<>]` matches any character except `<` and `>`.

In Antlr, terminals can be defined using regular expressions, not just fixed strings. For example, here are some other terminal patterns we used in the URL grammar earlier in the reading, now written in Antlr syntax and with Antlr's required naming convention:

```
IDENTSTEM : [a-z]+;
INTEGER : [0-9]+;
```

More about Antlr's grammar the syntax can be found in Chapter 5 of the Definitive ANTLR 4 Reference.

READING EXERCISES

Make an Antlr grammar

Generating the parser

The rest of this reading will focus on the Sum grammar used in the exercise above, which we'll store in a file called `Sum.g4`. Antlr 4 grammar files end with `.g4` by convention.

The Antlr parser generator tool converts a grammar source file like `Sum.g4` into Java classes that implement a parser. To do that, you need to go to a command prompt (Terminal or Command Prompt) and run a command like this:

```
java -jar antlr.jar Sum.g4
```

You'll need to make sure you're in the right folder (where `Sum.g4` is) and that `antlr.jar` is in there too (or refer to it where it is in your project folder structure – it may be a relative path like `../../libs/antlr.jar`).

Assuming you don't have any syntax errors in your grammar file, the parser generator will produce new Java source files in the current folder.The generated code is divided into several cooperating classes:

- the **lexer** takes a stream of characters as input, and produces a stream of terminals as output, like `NUMBER`, `+`, and `(`. For `Sum.g4`, the generated lexer is called `SumLexer.java`.
- the **parser** takes the stream of terminals produced by the lexer and produces a parse tree. The generated parser is called `SumParser.java`.
- the **tree walker** lets you write code that walks over the parse tree produced by the parser, as explained below. The generated tree walker files are the interface `SumListener.java`, and an empty implementation of the interface, `SumBaseListener.java`.

Antlr also generates two text files, `Sum.tokens` and `SumLexer.tokens`, that list the terminals that Antlr found in your grammar. These aren't needed for a simple parser, but they're needed when you include grammars inside other grammars.

Make sure that you:

- **Never edit the files generated by Antlr.** The right way to change your parser is to edit the grammar source file, `Sum.g4`, and then regenerate the Java classes.
- **Regenerate the files whenever you edit the grammar file.** This is easy to forget when Eclipse is compiling all your Java source files automatically. Eclipse does not regenerate your parser automatically. Make sure you rerun the `java -jar ...` command whenever you change your `.g4` file.
- **Refresh your project in Eclipse each time you regenerate the files.** You can do this by clicking on your project in Eclipse and pressing F5, or right-clicking and choosing Refresh. The reason is that Eclipse sometimes uses older versions of files already part of the source, even if they have been modified on your filesystem.

More about using the Antlr parser generator to produce a parser can be found in Chapter 3 of the Definitive ANTLR 4 Reference.

READING EXERCISES

Parsing

Calling the parser

Now that you've generated the Java classes for your parser, you'll want to use them from your own code.

First we need to make a stream of characters to feed to the lexer. Antlr has a class `ANTLRInputStream` that makes this easy. It can take a String, or a Reader, or an InputStream as input. Here we are using a string:

```
CharStream stream = new ANTLRInputStream("54+(2+89*7);");
```

Next, we create an instance of the lexer class that our grammar file generated, and pass it the character stream:

```
SumLexer lexer = new SumLexer(stream);
TokenStream tokens = new CommonTokenStream(lexer);
```

The result is a stream of terminals, which we can then feed to the parser:

```
SumParser parser = new SumParser(tokens);
```

To actually do the parsing, we call a particular nonterminal on the parser. The generated parser has one method for every nonterminal in our grammar, including `root()`, `sum()`, and `addend()`. We want to call the nonterminal that represents the set of strings that we want to match – in this case, `root()`.

Calling it produces a parse tree:

```
ParseTree tree = parser.root();
```

For debugging, we can then print this tree out:

```
System.err.println(tree.toStringTree(parser));
```

Or we can display it in a handy graphical form:

```
Trees.inspect(tree, parser);
```

which pops up a window with the parse tree shown on the right.

READING EXERCISES

Parse trees

Traversing the parse tree

So we've used the parser to turn a stream of characters into a parse tree, which shows how the grammar matches the stream. Now we need to do something with this parse tree. We're going to translate it into a value of a recursive abstract data type.

The first step is to learn how to traverse the parse tree. To do this, we use a `ParseTreeWalker`, which is an Antlr class that walks over a parse tree, visiting every node in order, top-to-bottom, left-to-right. As it visits each node in the tree, the walker calls methods on a *listener* object that we provide, which implements `SumListener` interface.

Just to warm up, here's a simple implementation of `SumListener` that just prints a message every time the walker calls us, so we can see how it gets used:

```
class PrintEverything implements SumListener {
    @Override public void enterRoot(SumParser.RootContext context) {
        System.err.println("entering root");
    }
    @Override public void exitRoot(SumParser.RootContext context) {
        System.err.println("exiting root");
    }

    @Override public void enterSum(SumParser.SumContext context) {
        System.err.println("entering sum");
    }
    @Override public void exitSum(SumParser.SumContext context) {
        System.err.println("exiting sum");
    }

    @Override public void enterAddend(SumParser.AddendContext context) {
        System.err.println("entering addend");
    }
    @Override public void exitAddend(SumParser.AddendContext context) {
        System.err.println("exiting addend");
    }

    @Override public void visitTerminal(TerminalNode terminal) {
        System.err.println("terminal " + terminal.getSymbol());
    }

    // don't need these here, so just use the default implementations
    @Override public void enterEveryRule(ParserRuleContext context) { }
    @Override public void exitEveryRule(ParserRuleContext context) { }
    @Override public void visitErrorNode(ErrorNode node) { }
}
```

Notice that every nonterminal `N` in the grammar has corresponding `enterN()` and `exitN()` methods in the listener interface, which are called when the tree walk enters and exits a parse tree node for nonterminal `N`, respectively. There's also a `visitTerminal()` that is called when the walk reaches a leaf of the parse tree. Each of these methods has a parameter that provides information about the nonterminal or terminal node that the walk is currently visiting.

The listener interface also has some methods that we don't need. The methods `enterEveryRule()` and `exitEveryRule()` are for adding logging or entering and exiting any nonterminal node, in case we want some generic behavior. The method `visitErrorNode()` is called if the input contained a syntax error that produced an error node in the parse tree. In the parser we're writing, however, a syntax error causes an exception to be thrown, so we won't see any parse trees with error nodes in them. The interface requires us to implement these methods, but we can just leave their method bodies empty.

```
ParseTreeWalker walker = new ParseTreeWalker();
SumListener listener = new PrintEverything();
walker.walk(listener, tree);
```

If we walk over the parse tree with this listener, then we see the following output:


```
entering root
entering sum
entering addend
terminal 54
exiting addend
terminal +
entering addend
terminal (
entering sum
entering addend
terminal 2
exiting addend
terminal +
entering addend
terminal 89
exiting addend
exiting sum
terminal )
entering addend
terminal *
exiting addend
terminal 7
terminal <EOF>
exiting root
```

Compare this printout with the parse tree shown at the right, and you'll see that the `ParseTreeWalker` is stepping through the nodes of the tree in order, from parents to children, and from left to right through the siblings.

Constructing an abstract syntax tree

We need to convert the parse tree into a recursive data type. Here's the definition of the recursive data type that we're going to use to represent sums of positive integers:

```
Sum = Plus(left:Sum, right:Sum) + Number(n:int)
```

If this syntax is mysterious, review recursive data type definitions.

When a recursive data type represents a language this way, it is often called an abstract syntax tree. A *Sum* value captures the important features of the expression – its grouping and the integers in it – while omitting unnecessary details of the sequence of characters that created it.

By contrast, the parse tree that we just generated with the Sum parser is a concrete syntax tree. It's called concrete, rather than abstract, because it contains more details about how the expression is represented in actual characters. For example, the strings `2+2`, `(2)+(2)` and `0002+0002` would each produce a different concrete syntax tree, but these trees would all correspond to the same abstract *Sum* value `Plus(Number(2), Number(2))`.

Now we create a listener that constructs a *Sum*. We write it's walking over the parse tree. Each parse tree node will correspond to a *Sum* variant: *sum* nodes will create `Plus` objects, and *addend* nodes will create `Number` objects.

Whenever the walker exits each node of the parse tree, we have walked over the entire subtree under that node, so we create the next *Sum* object at exit time. But we have to keep track of all the children that were created during the walk over that subtree. We use a stack to store them.

Here's the code:

```
/** Make a Sum value from a parse tree. */
class MakeSum implements SumListener {
    private StackSum stack = new StackImpl();
    // Invariant: stack contains the Sum value of each parse subtree
    // that has been completely walked over, in order.
    // has not yet been exited by the walk. The stack is ordered
    // by recency of visit, so that the top of the stack is the
    // Sum for the most recently walked subtree.
    //
    // At the start of the walk, the stack is empty, because
    // no subtrees have been fully walked.
    //
    // Whenever a node is exited by the walk, the Sum values of
    // its children are on top of the stack, in order with the last
    // child visited on top. To construct the Sum for the node,
    // those child Sum values from the stack, combine them with the
    // appropriate Sum producer, and push back a Sum value
    // representing the entire subtree under the node.
    //
    // At the end of the walk, after all subtrees have been walked
    // and the root has been exited, only the entire tree satisfies
    // the invariant's "fully walked but parent not yet exited" property,
    // so the top of the stack is the Sum of the entire parse tree.

    /**
     * Returns the Sum constructed by this object.
     * Requires that this object has completely walked over a Sum parse tree
     * using ParseTreeWalker.
     * @return Sum for the Sum parse tree that was walked
     */
    public Sum getSum() {
        return stack.getOb();
    }

    @Override public void exitRoot(SumParser.RootContext context) {
        // do nothing, root has only one child so its value is
        // already on top of the stack
    }

    @Override public void exitSum(SumParser.SumContext context) {
        // matched the addend ('+' addend)* rule
        List<SumParser.AddendContext> addends = context.addend();
        assert stack.size() == addends.size();

        // the pattern above always has at least 1 child;
        // pop the last child
        assert addends.size() > 0;
        Sum sum = stack.pop();

        // pop the older children, one by one, and add them on
        for (int i = 1; i < addends.size(); ++i) {
            sum = new Plus(stack.pop(), sum);
        }

        // the result is this subtree's Sum
        stack.push(sum);
    }

    @Override public void exitAddend(SumParser.AddendContext context) {
        if (context.NUMBER() != null) {
            // matched the NUMBER alternative
            int n = Integer.parseInt(context.NUMBER().getText());
            Sum sum = new Number(n);
            stack.push(sum);
        } else {
            // matched the '(' sum ')' alternative
            // do nothing, because sum's value is already on the stack
        }
    }

    // don't need these here, so just use the default implementations
    @Override public void enterRoot(SumParser.RootContext context) { }
    @Override public void enterSum(SumParser.SumContext context) { }
    @Override public void enterAddend(SumParser.AddendContext context) { }

    @Override public void visitTerminal(TerminalNode terminal) { }
    @Override public void enterEveryRule(ParserRuleContext context) { }
    @Override public void exitEveryRule(ParserRuleContext context) { }
    @Override public void visitErrorNode(ErrorNode node) { }
}
```

More about Antlr's parse tree listeners can be found in Section 7.2 of the Definitive ANTLR 4 Reference.

READING EXERCISES

String to AST 1

String to AST 2

Handling errors

By default, Antlr parsers print errors to the console. In order to make the parser modular, however, we need to handle those errors differently. You can attach an ErrorListener to the lexer and parser in order to throw an exception when an error is encountered during parsing. The Sum.g4 file defines a method `reportErrorsAsExceptions()` which does this. So if you copy the technique used in this grammar file, you can call:

```
lexer.reportErrorsAsExceptions();
parser.reportErrorsAsExceptions();
```

right after you create the lexer and parser. Then when you call parser.root(), it will throw an exception as soon as it encounters something that it can't match.

This is a simplistic approach to handling errors. Antlr offers more sophisticated forms of error recovery as well. To learn more, see Chapter 9 in the Definitive Antlr 4 Reference.

Summary

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

- **Safe from bugs.** A grammar is a declarative specification for strings and streams. Strings such as sums can be implemented automatically by a parser generator. These specifications are often simpler, more direct, and less likely to be buggy than parsing code written by hand.
- **Easy to understand.** A grammar captures the shape of a sequence in a form that is compact and easier to understand than hand-written parsing code.
- **Ready for change.** A grammar can be easily edited, then run through a parser generator to regenerate the parsing code.

Collaboratively authored with contributions from: Max Goldman, Joe Cottrill, Chris Terman, Erik Demaine, Michael Ernst, Max Hailperin, Nancy Lynch, Daniel Jackson, Bob Miller, Martin Rinard, and Rob Miller, Massachusetts Institute of Technology
CC BY SA 4.0