Readability

When you’re working in groups

When you’re looking at something you did last month
Documentation

Clear method & variable names
Read the code and it’s obvious what it does

Detailed comments when things get tricky
Example: when implementing an algorithm and explaining how it works
Consistency of coding style

Variable naming, indentation, whitespace, comment style, ...
You want your new code to feel at home with pre-existing code
Maintainability / Extensibility

When you need to add a new feature into an old program
Old-school programmers reimplemented trees, lists, etc. Often the same bugs repeated by cut-and-paste

New-school programmers write it once The importance of generics and understanding covariance/contravariance
Conciseness

Huge amounts of boilerplate suck

Why we teach Python in Comp140: the “code” looks like “pseudocode”
Full-featured

It’s sometimes nice to add features that you might need later on
Example: tree deletion or tree range-queries
Consistency of design

Once you learn your way around a library, the concepts are reused. Example: `List.Empty.create()` then `Treap.Empty.create()`
Easy to use correctly

Or, conversely, hard to use incorrectly
This is where type systems are fantastically helpful to us.
Also @annotations.
Modularity

As codebases grow, they become fragile
Example: If you change edu.rice.list.IList, tons of code depends on it!

Limited public interfaces → more freedom to evolve the internals
Testability

Unit tests are immensely valuable
Example: Google has a whole series of “testing on the toilet” notes

All tests must pass before you can commit your code
Testability

Unit tests are immensely valuable.
Example: Google has a whole series of "testing on the toilet" notes.
All tests must pass before you can commit your code.
“Whistle while you work”
Logs give insight into what’s going on inside a class.

On many platforms (smartphones, big clusters) it’s all you get.
Google publishes a detailed coding style guide
https://google.github.io/styleguide/javaguide.html

There’s even a “CheckStyle” plugin for IntelliJ that can enforce it
We’ll start requiring you to conform to this after the midterm.
4.1.2 Nonempty blocks: K & R style

Braces follow the Kernighan and Ritchie style ("Egyptian brackets") for nonempty blocks and block-like constructs:

- No line break before the opening brace.
- Line break after the opening brace.
- Line break before the closing brace.
- Line break after the closing brace if that brace terminates a statement or the body of a method, constructor or named class. For example, there is no line break after the brace if it is followed by `else` or a comma.

Example:

```java
return new MyClass() {
    @Override public void method() {
        if (condition()) {
            try {
                something();
            } catch (ProblemException e) {
                recover();
            }
        }
    }
};
```

A few exceptions for enum classes are given in Section 4.8.1, [Enum classes](#).
Are the edu.rice classes “Google compliant”?

Mostly, yes.
I’ve been using four-spaces for indentation rather than two, but otherwise yes.

Most of you can improve your code with more line breaks.

4.5.1 Where to break

The prime directive of line-wrapping is: prefer to break at a higher syntactic level. Also:

1. When a line is broken at a non-assignment operator the break comes before the symbol. (Note that this is not the same practice used in Google style for other languages, such as C++ and JavaScript.)
   - This also applies to the following "operator-like" symbols: the dot separator (.), the ampersand in type bounds (<T extends Foo & Bar>), and the pipe in catch blocks (catch (FooException | BarException e)).
2. When a line is broken at an assignment operator the break typically comes after the symbol, but either way is acceptable.
   - This also applies to the "assignment-operator-like" colon in an enhanced for ("foreach") statement.
3. A method or constructor name stays attached to the open parenthesis (()) that follows it.
4. A comma (,) stays attached to the token that precedes it.
Logging

We gave you a great logging library. Few of you use it. In the real world, this is often all that you get. It’s powerful stuff. (And will be immensely valuable to you later in the semester.)

```java
import edu.rice.Log;

class Foo {
    private static final String TAG = "Foo";
    public Foo() {
        Log.i(TAG, "running constructor");
    }
}
```
Logging in Android

Example 1: The NFC (near field communications) system is always on

Maybe this “unknown event” is nothing. Maybe it’s a bug.
If/when these logs go back to Google, they can discover and fix systemic issues.

```
    status=0x0
    status=0x0
```
Logging in Android

Example 2: The phone got hot and the thermal engine reacted!
Do you suppose there's a connection between it and the garbage collector?

10-01 12:55:23.389 212-292/? I/ThermalEngine: TM Id 'SKIN_THERMAL_management_1' Sensor 'xo_thermPu2' - alarm raised at 40.0 degC
10-01 12:55:25.593 747-1767/system_process I/art: Explicit concurrent mark sweep GC freed 26293(864KB) AllocSpace objects, 0(0B) LOS objects, 27% free, 41MB/57MB, paused 2.203ms total 114.856ms
10-01 12:55:28.744 747-1763/system_process I/art: Explicit concurrent mark sweep GC freed 26119(861KB) AllocSpace objects, 0(0B) LOS objects, 27% free, 41MB/57MB, paused 2.388ms total 117.267ms
10-01 12:55:31.446 747-1767/system_process I/art: Explicit concurrent mark sweep GC freed 29901(1044KB) AllocSpace objects, 0(0B) LOS objects, 27% free, 41MB/57MB, paused 2.146ms total 90.970ms
10-01 12:55:49.226 747-775/system_process I/PowerManagerService: Going to sleep due to screen timeout (uid 1000)...
10-01 12:55:49.228 747-775/system_process I/PowerManagerService: Sleeping (uid 1000)...
Android “Maps” was crashing

I emailed the “logcat” to buddies at Google

Two days later, there was an updated Maps app

Dan: “Hey, don’t worry about it. Looks like the Maps app works again.”
Buddies: “Yeah, we fixed your bug.”
Dan: “Uhh, thanks!”
Literate programming

Coined by Don Knuth. Roughly, you write a book with your program inside. Automatic tools extract your program from the book.
Specially formatted comments that document your classes & methods
Automatically extracted by IntelliJ; Javadoc produces beautiful web pages

/**
 * general-purpose parsing tokens, used by NamedMatcher
 * @see NamedMatcher
 */

public class Token<TokenType extends Enum<TokenType> & IEnumVal<String>> { ...
Javadoc: kinda literate programming

Specially formatted comments that document your classes & methods
Automatically extracted by IntelliJ; Javadoc produces beautiful web pages

/**
 * general-purpose parsing tokens, used by NamedMatcher
 * @see NamedMatcher
 */
public class Token<TokenType extends Enum<TokenType> & IEnumVal<String>> { ...
/**
 * fold right: combines the head with the folded accumulation of the tail. Example the list is \{a,b,c,d,e\} and the folding
 * function is plus, then this returns \(a + (b + (c + (d + e)))\).
 *
 * Note: when the result type is different from the list element type (e.g., if you're folding a list of strings into a tree
 * of strings) then the accumulatorVal should be of the result type, and the function \(f\) will take two arguments: the
 * list element type and the result type.
 * @param f folding function (e.g., for strings, this might be string concatenation)
 * @param accumulatorVal value when there's nothing else to fold (e.g., for strings, this might be the empty string)
 */

default <U> U foldr(BiFunction<T,U,U> f, U accumulatorVal)
/**
 * fold right: combines the head with the folded accumulation of the tail. Example the list is {a,b,c,d,e} and the folding
 * function is plus, then this returns (a + (b + (c + (d + e)))�).
 * 
 * Note: when the result type is different from the list element type (e.g., if you're folding a list of strings into a tree
 * of strings) then the accumulatorVal should be of the result type, and the function f will take two arguments: the
 * list element type and the result type.
 * 
 * @param f folding function (e.g., for strings, this might be string concatenation)
 * @param accumulatorVal value when there's nothing else to fold (e.g., for strings, this might be the empty string)
 */

default <U> U foldr(BiFunction<T,U,U> f, U accumulatorVal)
Given a Java enum which defines regexes for scanning tokens, this class builds a regex with "named-capturing groups" and then lets you use it to tokenize your input. The names of the tokens will come from the Enum, and the regular expressions for the tokens will come by calling <code>toString()</code> on each of the enums. If you have a token type that doesn't ever occur in your input, such as "FAIL" in the example below, make sure its value is the empty string, and it will be ignored while constructing the regular expressions.

To make this work, make sure your enum implements the <code>IEnumVal&lt;String&gt;</code> type, which means it will have an extra method, <code>value()</code> which returns the regex pattern.

Example:

```java
public enum TokenType implements IEnumVal&lt;String&gt; {
    OPENCURLY("\{"),
    CLOSECURLY("\}"),
    WHITESPACE("\s"),
    FAIL(""),
    
    public final String pattern;
    
    TokenType(String pattern) {
        this.pattern = pattern;
    }
    
    public String value() { return pattern; }
}
```

Typical usage:
Given a Java enum which defines regexes for scanning tokens, this class builds a regex with “named-capturing groups” and then lets you use it to tokenize your input. The names of the tokens will come from the Enum, and the regular expressions for the tokens will come by calling <code>toString()</code> on each of the enums. If you have a token type that doesn’t ever occur in your input, such as "FAIL" in the example below, make sure its value is the empty string, and it will be ignored while constructing the regular expressions.

To make this work, make sure your enum implements the <code>IEnumVal&lt;String&gt;</code> type, which means it will have an extra method, <code>value()</code> which returns the regex pattern.

Example:

```java
public enum TokenType implements IEnumVal&lt;String&gt; {
    OPENCURLY("\{"),
    CLOSECURLY("\}"),
    WHITESPACE("\s"),
    FAIL(""),
    
    public final String pattern;
    
    TokenType(String pattern) {
        this.pattern = pattern;
    }
    
    public String value() { return pattern; }
}
```

Typical usage:

```java
```
Given a Java enum which defines regexes for scanning tokens, this class builds a regex with “named-capturing groups” and then lets you use it to tokenize your input. The names of the tokens will come from the Enum, and the regular expressions for the tokens will come by calling <code>toString()</code> on each of the enums. If you have a token type that doesn’t ever occur in your input, such as "FAIL" in the example below, make sure its value is the empty string, and it will be ignored while constructing the regular expressions.

To make this work, make sure your enum implements the <code>IEnumVal&lt;String&gt;</code> type, which means it will have an extra method, <code>value()</code> which returns the regex pattern.

Example:

```java
public enum TokenType implements IEnumVal&lt;String&gt; {
    OPENCURLY("\{"),
    CLOSECURLY("\}"),
    WHITESPACE("\s"),
    FAIL(""),
    
    public final String pattern;

    TokenType(String pattern) {
        this.pattern = pattern;
    }

    public String value() { return pattern; }
}
```

Typical usage:

```java
```

HTML escapes (!) for < and >
Javadoc: the payoff

Gee, how does that work again? Ctrl-J in IntelliJ then...

```java
public static IList<Token<TokenType>> scan(@NotNull String input) {
    NamedMatcher<TokenType> nm = new NamedMatcher<>(TokenType.class);
    IList<Token<IToken>> tokens = nm.scan(input);
    return tokens;
}
```

```java
public class NamedMatcher<T extends Enum<?<T>>> extends Object {

    Given a Java enum which defines regexes for scanning tokens, this class builds a regex with "named-capturing groups" and then lets you use it to tokenize your input. The names of the tokens will come from the Enum, and the regular expressions for the tokens will come by calling `tostring()` on each of the enums. If you have a token type that doesn't even occur in your input, such as "FAIL" in the example below, make sure its value is the empty string, and it will be ignored while constructing the regular expressions.

    To make this work, make sure your enum implements the `IEnumVal<String>` type, which means it will have an extra method, `value()` which returns the regex pattern.

Example:

```java
public enum TokenType implements IEnumVal<String> {
    OPENCURLY("\{\}"),
    CLOSECURLY("\}\"),
    WHITESPACE("\s+"),
    FAIL(""),
    ...

    public final String pattern;
}
```
Finding bugs early is beautiful

“Software Development Cost Pyramid”
Finding bugs earlier requires less effort.
   Finish faster for the same effort if you’re smart.
   Or, get more done in the same time.

Techniques to find bugs early, therefore, are hugely valuable!

(And therefore, are beautiful.)
Contracts / Annotations

We made a “no-null” rule early in Comp215
Beautiful code will enforce that rule if a programmer forgets

Really-old-school (won’t fail until much later):

```java
public IList<T> add(T t) {
    return new List<>(t, this);
}
```
We made a “no-null” rule early in Comp215
Beautiful code will enforce that rule if a programmer forgets

Old-school (fails at list-insertion time):

```java
public IList<T> add(T t) {
    if(t == null)
        throw new RuntimeException("nulls not allowed in List.add");
    return new List<>(t, this);
}
```
Contracts / Annotations

We made a “no-null” rule early in Comp215
Beautiful code will enforce that rule if a programmer forgets

New-school (compile-time error):

```java
import org.jetbrains.annotations.NotNull;

public IList<T> add(@NotNull T t) {
    return new List<>(t, this);
}
```
Annotations: the payoff

IntelliJ gives you an immediate warning

```java
@javax.annotation.ParametersAreNonnullByDefault
public class TestAnnotation {
    @Test
    public void testAddNull() throws Exception {
        IList<String> emptyList = List.Empty.create();
        IList<String> badNewsList = emptyList.add(null);
    }
}
```

And if you run it anyway

```
java.lang.IllegalArgumentException: Argument for @NotNull parameter 't' of edu/rice/list/List.add must not be null
    at edu.rice.list.List.add(List.java)
    at edu.rice.list.ListTest.testAddNull(ListTest.java:162)
    ...
```
Annotations to catch common errors

Example: say you forgot to save the result of folding a list into a tree (Because you forgot that you were doing functional programming.) Wouldn't it be nice to get a warning?

inputList.foldl((map, string) -> map.merge(string, 1, (a,b)->a+b), emptyMap);

Result of 'List.foldl()' is ignored more... (按F1)
Annotations to catch common errors

Example: say you forgot to save the result of folding a list into a tree (Because you forgot that you were doing functional programming.) Wouldn't it be nice to get a warning?

```plaintext
inputList.foldl((map, string) -> map, get(1, (a,b)->a+b), emptyMap);
```

Result of 'List.foldl()' is ignored more... (Ctrl-F1)
"Pure" functional contracts

This is sooooo cool.

```java
import org.jetbrains.annotations.Contract;
import org.jetbrains.annotations.NotNull;

@NotNull
@Contract(pure = true)
default <U> U foldl(
    @NotNull BiFunction<U, T, U> f,
    @NotNull U accumulatorVal)
{

...

}

Contracts can also say things about null behavior.
```
Annotations are compiler plugins!

Example: JUnit4’s @Test annotations
When you use those, JUnit runs inside the Java compiler and emits more code

Writing your own annotations: beyond the scope of Comp215
You’re basically looking at Java code and rewriting it

Using libraries that provide crazy annotations
Quite common. Some devs love this stuff to make code cleaner.
More static checking for bugs: “lint”

Claims Wikipedia:
“The term was derived from the name of the undesirable bits of fiber and fluff found in sheep's wool.”

Java has a very sophisticated lint tool built right into the compiler.
More static checking for bugs: "lint"

Claims Wikipedia: "The term was derived from the name of the undesirable bits of fiber and fluff found in sheep's wool."

Java has a very sophisticated lint tool built right into the compiler.

![Java Compiler Preferences](image)
Enable "all" checks, except the "serial" check. Complete list:
https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javac.html#BHCJCABJ
Too many warnings!

Suppress them in cases when you know it’s okay
Otherwise, “beautiful” code should generate no warnings at all.

```java
public static <T> IList<T> create() {
    // The same instance can be used for any T since it will always be empty.
    // Details:
    // http://stackoverflow.com/questions/14313528/returning-a-generic-empty-list
    @SuppressWarnings("unchecked")
    IList<T> typedEmptyList = (IList<T>) singleton;
    return typedEmptyList;
}
```
Further static checking

ErrorProne: a recent tool from Google
We tried installing this plugin for IntelliJ and it caused problems.
(Sigh.)

FindBugs: a more mature tool from U. Maryland
Somebody try it out and let’s see how well it works.
Further annotations

UW Checker Framework
http://types.cs.washington.edu/checker-framework/

Among other cool things, you can annotate variables with *units*
Distance (meters), time (seconds), etc.

Static verification that your units match up
e.g., velocity = distance / time.
Further annotations

UW Checker Framework
http://types.cs.washington.edu/checker-framework/

Among other cool things, you can annotate variables with *units*

Distance (meters), time (seconds), etc.

Static verification that your units match up

*e.g.,* velocity = distance / time.
Last trick for today: declarative data

Here's a thing we wanted to parse:

```java
public String bigJson = "{\n    "itemCount": 2,\n    "subtotal": "$15.50",\n    "items":[\n        {\n            "title": "The Big Book of Foo",\n            "description": "Bestselling book of Foo by A.N. Other",\n            "imageUrl": "\"/images\books\12345.gif\",\n            "price": "$10.00",\n            "qty": 1\n        },\n        {\n            "title": "Javascript Pocket Reference",\n            "description": "Handy pocket-sized reference for the Javascript language",\n            "imageUrl": "\"/images\books\56789.gif\",\n            "price": "$5.50",\n            "qty": 2\n        }\n    ]\n}";
```
Last trick for today: declarative data

Here's the unit test for the whole parser:

```java
Parser.JSONObject bigObject = Parser.parse(bigJson).get();
Parser.JSONObject bigComparison =
    object(
        jpair("itemCount", jnumber(2)),
        jpair("subtotal", javaString("$15.50")),
        jpair("items",
            jarray(
                object(
                    jpair("title", "The Big Book of Foo"),
                    jpair("description", "Bestselling book of Foo by A.N. Other"),
                    jpair("imageUrl", "/images/books/12345.gif"),
                    jpair("price", "$10.00"),
                    jpair("qty", jnumber(1)))
            ))
    );

assertEquals(bigComparison, bigObject);
```
Java nerds call this “fluent programming”

The code exactly follows the structure of what you want; no fluff!

Parser JObject bigObject = Parser.parse(bigJson).get();
Parser JObject bigComparison =
    object(
        jpair("itemCount", jnumber(2)),
        jpair("subtotal", javaString("$15.50")),
        jpair("items",
            jarray(
                object(
                    jpair("title", "The Big Book of Foo"),
                    jpair("description", "Bestselling book of Foo by A.N. Other"),
                    jpair("imageUrl", "/images/books/12345.gif"),
                    jpair("price", "$10.00"),
                    jpair("qty", jnumber(1)))))
            )

assertEquals(bigComparison, bigObject);

Just one line of “magic” code:

import static edu.rice.json.Builder.*;
Other libraries take mutating approaches

org.json:
String myString =
    new JSONObject().put("JSON", "Hello, World!").toString();

Lots of extraneous fluff as things get larger.
Why is declarative data beautiful?

Which is more likely to be bug-free?

String myString =
    jobject(jpair("JSON", "Hello, World!")).toString();

String myString2 =
    String.format("{ \"%s\" : \"%s\"}", "JSON", "Hello, World!");

They do the same thing, right?
Why is declarative data beautiful?

Which is more likely to be bug-free?

```java
public static String jsonObjectThing(String key, String value) {
    // option 1: declarative data
    return jobject(jpair(key, value)).toString();

    // option 2: string concatenation
    return String.format("{ "%s" : "%s"}", key, value);
}
```

What if there was a quotation mark inside one of those strings?
The declarative style is **safe**, no matter how ugly the strings are.

Therefore, fewer weird bugs.
(Also, fewer weird security vulnerabilities!)
Hi, this is your son's school. We're having some computer trouble.

Oh, dear - did he break something?

In a way -

DID you really name your son Robert'); DROP TABLE Students;--?

Oh, yes. Little Bobby Tables, we call him.

Well, we've lost this year's student records. I hope you're happy.

And I hope you've learned to sanitize your database inputs.

https://xkcd.com/327/