

Code-Level Design

Martin Kellogg

Reading quiz: code-level design

Q1: **TRUE** or **FALSE**: The Prettier team believes that the biggest reason users adopt Prettier is to stop all ongoing debate over style.

Q2: Which example does the “Law of Triviality” article use to illustrate the “law”?

- A. the color of a button on a webpage
- B. the color of the bikeshed at a nuclear power plant
- C. the color of the seats in a car
- D. the color of the signs in a train station

Reading quiz: code-level design

Q1: **TRUE** or **FALSE**: The Prettier team believes that the biggest reason users adopt Prettier is to stop all ongoing debate over style.

Q2: Which example does the “Law of Triviality” article use to illustrate the “law”?

- A. the color of a button on a webpage
- B. the color of the bikeshed at a nuclear power plant
- C. the color of the seats in a car
- D. the color of the signs in a train station

Reading quiz: code-level design

Q1: **TRUE** or **FALSE**: The Prettier team believes that the biggest reason users adopt Prettier is to stop all ongoing debate over style.

Q2: Which example does the “Law of Triviality” article use to illustrate the “law”?

- A. the color of a button on a webpage
- B.** the color of the bikeshed at a nuclear power plant
- C. the color of the seats in a car
- D. the color of the signs in a train station

Code-level Design

Today's agenda:

- Why does code-level design matter?
- Some general principles, with examples
- In-class exercise
- Automation and linting
- Our course style guide

Code-level Design

Today's agenda:

- **Why does code-level design matter?**
- Some general principles, with examples
- In-class exercise
- Automation and linting
- Our course style guide

Why does code-level design matter?

Why does code-level design matter?

- Software systems need to be understandable to humans

Why does code-level design matter?

- Software systems need to be understandable to humans
 - Maintenance is the largest part of the software lifecycle - estimated to be **50-80%** of total development cost
 - **Reading** code is one of the most time-consuming tasks that software engineers engage in regularly

Coupling makes code hard to understand

Definition: Two pieces of code are *coupled* if a change to one requires a change to the other. (Alternative term: *connascence*)

Coupling makes code hard to understand

Definition: Two pieces of code are *coupled* if a change to one requires a change to the other. (Alternative term: *connascence*)

Two pieces of code might be coupled for many reasons:

Coupling makes code hard to understand

Definition: Two pieces of code are *coupled* if a change to one requires a change to the other. (Alternative term: *connascence*)

Two pieces of code might be coupled for many reasons:

- names
- order of arguments
- algorithms
- meaning of data
- types

Coupling makes code hard to understand

Definition: Two pieces of code are **coupled** if a change to one requires a change to the other. (Alternative term: **connascence**)

Two pieces of code might be coupled for many reasons:

- names
- order of arguments
- algorithms
- meaning of data
- types

If two pieces of code are coupled, one must understand both to modify either. Therefore, **more coupling = harder to understand.**

Surprises make code hard to understand

Surprises make code hard to understand

- follow **established conventions**, especially for naming
 - varies by language and by codebase
 - do as others do
 - this includes **bad** conventions that otherwise violate the rules I'm about to show you!

Surprises make code hard to understand

- follow **established conventions**, especially for naming
 - varies by language and by codebase
 - do as others do
 - this includes **bad** conventions that otherwise violate the rules I'm about to show you!
- avoid “clever” implementations unless you really need them
 - also avoid **premature optimization**

Surprises make code hard to understand

- follow **established conventions**, especially for naming
 - varies by language and by codebase
 - do as others do
 - this includes **bad** conventions that otherwise violate the rules I'm about to show you!
- avoid “clever” implementations unless you really need them
 - also avoid **premature optimization**
- try to make code as “**greppable**” as you can
 - ask yourself if someone using this code in the future will be able to find the location you're editing

Greppability

Suppose you have two database tables named `shipping_addresses` and `billing_addresses`. You could get them this way:

```
const getTableName = (addressType: 'shipping' | 'billing') => {  
  return `${addressType}_addresses`  
}
```

Greppability

Suppose you have two database tables named `shipping_addresses` and `billing_addresses`. You could get them this way:

```
const getTableName = (addressType: 'shipping' | 'billing') => {  
  return `${addressType}_addresses`  
}
```

Now suppose further that I'm **debugging** something to do with the `shipping_addresses` table, so I search the codebase for “shipping_addresses”. Will I find this code?

Greppability

Suppose you have two database tables named `shipping_addresses` and `billing_addresses`. You could get them this way:

```
const getTableName = (addressType: 'shipping' | 'billing') => {  
  return `${addressType}_addresses`  
}
```

Now suppose further that I'm **debugging** something to do with the `shipping_addresses` table, so I search the codebase for “shipping_addresses”.
Will I find this code? **NO**

Code-level Design

Today's agenda:

- Why does code-level design matter?
- **Some general principles, with examples**
- In-class exercise
- Automation and linting
- Our course style guide

Some general code-level design principles

- use good names
- make your data meaningful
- one job per method
- don't repeat yourself (DRY)
- avoid magic numbers/strings (don't hardcode)

Some general code-level design principles

- **use good names**
- make your data meaningful
- one job per method
- don't repeat yourself (DRY)
- avoid magic numbers/strings (don't hardcode)

Use good names

- Names are the only part of the documentation that's **actually required** :)

Use good names

- Names are the only part of the documentation that's **actually required** :)
- Follow naming **conventions** (to avoid surprises)
 - conventions vary by language, project, etc: know yours!

Use good names

- Names are the only part of the documentation that's **actually required** :)
- Follow naming **conventions** (to avoid surprises)
 - conventions vary by language, project, etc: know yours!
- Applies to **everything** that you name, including:
 - methods
 - variables
 - types/classes
 - files
 - constants
 - etc.

A man with a mustache and wide eyes is looking at a quiz interface. The interface consists of a blue banner with the question 'parseDBMXML means:' and four options below it, each in a blue box with a yellow border. The options are: A: parse DBM XML, B: parse DB MXML, C: parse DB Mx Markup Language, and D: parse DB Mx Machine Learning.

parseDBMXML means:

A: parse DBM XML

B: parse DB MXML

C: parse DB Mx Markup Language

D: parse DB Mx Machine Learning

Use good names: example 1

```
var t : number
```

```
var l : number
```

Use good names: example 1

```
var temp : number
```

```
var loc : number
```

Use good names: example 1

```
var temp : Temperature
```

```
var loc : SensorLocation
```

Use good names: example 1

```
var temperature : Temperature
```

```
var location : SensorLocation
```

Use good names: example 2

```
function checkLine (line : string) : boolean
```


Use good names: example 2

```
function lineIsTooLong (line : string) : boolean
```

Naming principles

- use noun-like names for functions/methods that return a value

Naming principles

- use noun-like names for functions/methods that return a value

```
function diameter (c : Circle) : number
```

vs.

```
function calculateDiameter (c : Circle) : number
```

Naming principles

- use noun-like names for functions/methods that return a value

```
function diameter (c : Circle) : number
```

vs.

```
function calculateDiameter (c : Circle) : number
```

- use verb-like names only for methods that have side-effects

Naming principles

- use noun-like names for functions/methods that return a value

```
function diameter (c : Circle) : number
```

VS.

```
function calculateDiameter (c : Circle) : number
```

- use verb-like names only for methods that have side-effects

```
function printDiameter (c : Circle) : void
```

Some general code-level design principles

- use good names
- **make your data meaningful**
- one job per method
- don't repeat yourself (DRY)
- avoid magic numbers/strings (don't hardcode)

Make your data meaningful

Three decisions:

Make your data meaningful

Three decisions:

- Decide **what part** of the information in the "real world" needs to be represented as data

Make your data meaningful

Three decisions:

- Decide **what part** of the information in the "real world" needs to be represented as data
- Decide **how** that information needs to be represented as data

Make your data meaningful

Three decisions:

- Decide **what part** of the information in the "real world" needs to be represented as data
- Decide **how** that information needs to be represented as data
- Document how to **interpret** the data in your computer as information about the real world

Make your data meaningful: shirt example

- Suppose that I am wearing a red shirt, and I've decided I need to represent that fact in my program.
- How should I represent that in my program?
- We need to decide:

Make your data meaningful: shirt example

- Suppose that I am wearing a red shirt, and I've decided I need to represent that fact in my program.
- How should I represent that in my program?
- We need to decide:
 - how to represent shirts (including their color)
 - how to represent colors
 - how to represent my shirt

Make your data meaningful: shirt example

```
type Shirt = {  
  /** the color of the shirt */  
  color: Color  
}
```

```
type Color = { ... }
```

```
/** My shirt */  
const myShirt: Shirt
```

```
myShirt.color = red
```

Make your data meaningful: shirt example

my shirt is red

representation

interpretation

```
type Shirt = {  
  /** the color of the  
  shirt */  
  color: Color  
}  
type Color = { ... }  
  
/** My shirt */  
const myShirt: Shirt  
myShirt.color = red
```

Make your data meaningful: shirt example

my shirt is red

representation

interpretation

How do we **know** these are connected?

```
type Shirt = {  
  /** the color of the  
  shirt */  
  color: Color  
}  
  
type Color = { ... }  
  
/** My shirt */  
const myShirt: Shirt  
myShirt.color = red
```

Make your data meaningful: shirt example

my shirt is red

representation

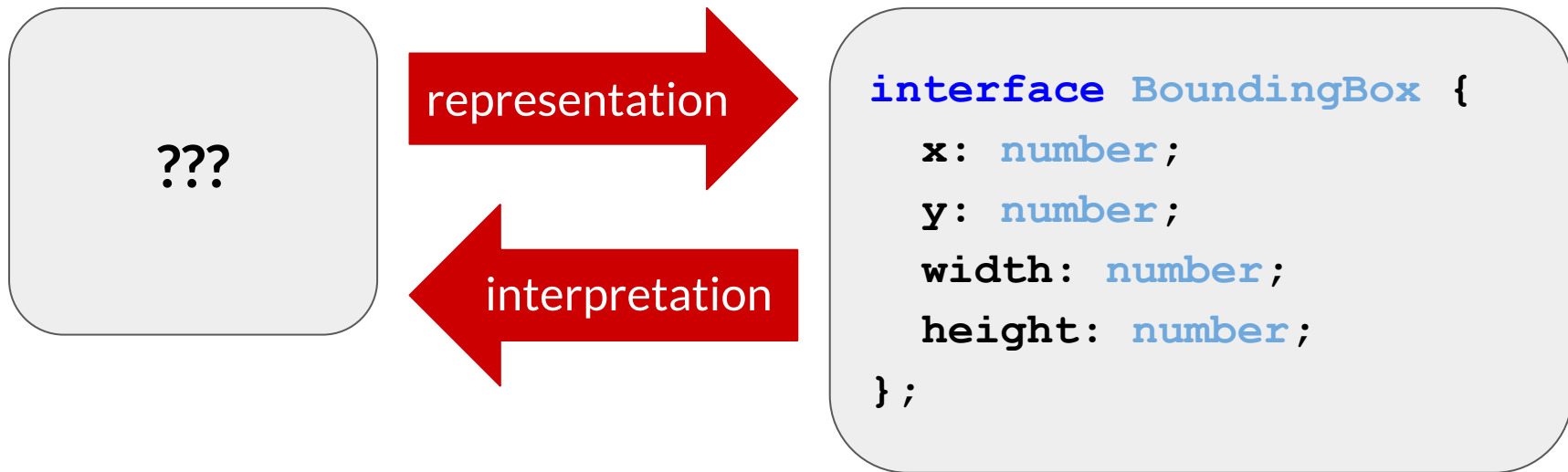
interpretation

```
type Shirt = {  
  /** the color of the  
  shirt */  
  color: Color  
}  
  
type Color = { ... }  
  
/** My shirt */  
const myShirt: Shirt  
myShirt.color = red
```

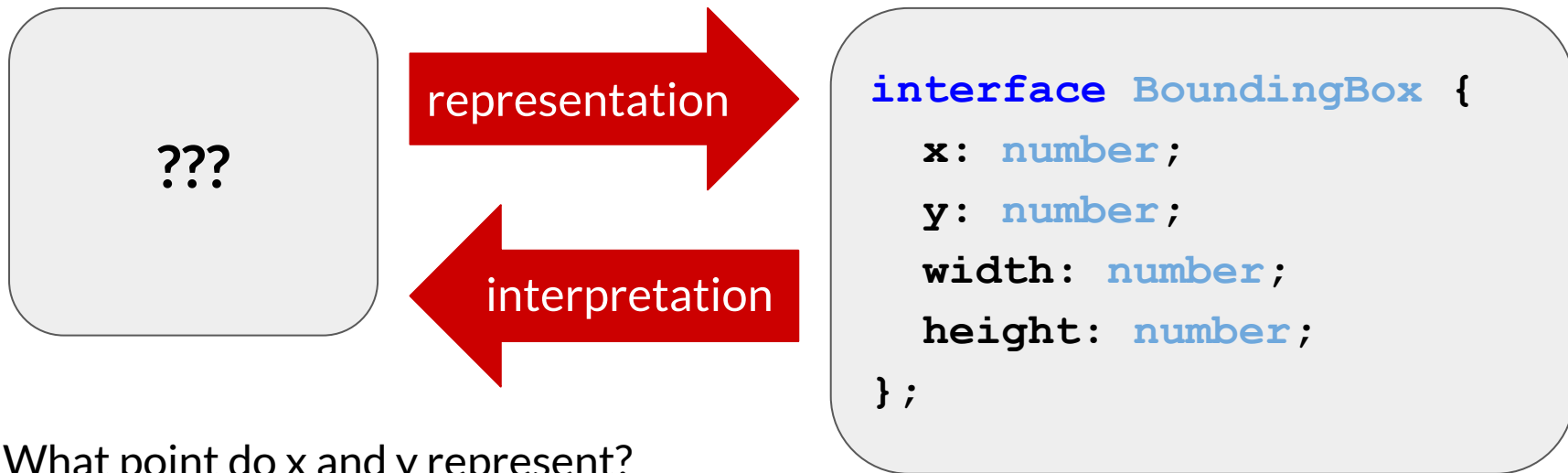
How do we **know** these are connected?

We have to **write it down!**

Make your data meaningful: xy example



Make your data meaningful: xy example



- What point do x and y represent?
- What units are these values in (pixels? feet?)
- Does y grow moving up or down?
- What is this “bounding”? How close is the box to the “bound” thing?

Make your data meaningful

Three decisions:

- Decide **what part** of the information in the "real world" needs to be represented as data
- Decide **how** that information needs to be represented as data
- Document how to **interpret** the data in your computer as information about the real world

Make your data meaningful

Three decisions:

- Decide **what part** of the information in the "real world" needs to be represented as data
- Decide **how** that information needs to be represented as data
- Document how to **interpret** the data in your computer as information about the real world

Make sure you write all of this down!
This is what **comments** are for.

Some general code-level design principles

- use good names
- make your data meaningful
- **one job per method**
- don't repeat yourself (DRY)
- avoid magic numbers/strings (don't hardcode)

One job per method

- Each class, and each method of that class, should have one job, and only one job

One job per method

- Each class, and each method of that class, should have one job, and only one job
- If your method has more than one job, split it into 2 methods.
Why?

One job per method

- Each class, and each method of that class, should have one job, and only one job
- If your method has more than one job, split it into 2 methods.
Why?
 - You might want one part but not the other
 - It's easier to test a method that has only one job
- You call both of them if you need to (or write a method that does)

One job per method

- Each class, and each method of that class, should have one job, and only one job
- If your method has more than one job, split it into 2 methods.
Why?
 - You might want one part but not the other
 - It's easier to test a method that has only one job
- You call both of them if you need to (or write a method that does)
- Same principle applies for classes

Some general code-level design principles

- use good names
- make your data meaningful
- one job per method
- **don't repeat yourself (DRY)**
- avoid magic numbers/strings (don't hardcode)

Don't repeat yourself (DRY)

- If you need something more than once, give it a name and use that name everywhere

Don't repeat yourself (DRY)

- If you need something more than once, give it a name and use that name everywhere
- Applies to:
 - constants/variables
 - methods (turn any differences between almost-clones into parameters!)
 - code blocks (turn them into methods)
 - classes (use a superclass)

My project's codebase when I paste another copy of the same lines I already have in few other files



My project's codebase when I paste another copy
of the same lines I already have in few other files



Don't be this person!

Don't repeat yourself: example

```
function testequal (testname: string, actualVal: T, correctVal: T) {  
    test(testname, function () {  
        expect(actualVal).toBe(correctVal) })  
    }  
}
```

```
describe('tests for countOfLocalMorks', function () {  
    testequal('empty crew', countOfLocalMorks(ship1), 0)  
    testequal('just Mork', countOfLocalMorks(ship2), 1)  
    testequal('just Mindy', countOfLocalMorks(ship3), 0)  
    testequal('two Morks', countOfLocalMorks(ship4), 2)  
    testequal('drone has no Morks', countOfLocalMorks(drone1), 0)  
})
```

Don't repeat yourself: example

```
function testequal (testname: string, actualVal: T, correctVal: T) {  
  test(testname, function () {  
    expect(actualVal).toBe(correctVal) })  
}
```

```
describe('tests for countOfLocalMorks', function () {  
  testequal('empty crew', countOfLocalMorks(ship1), 0)
```

```
function testship (testname: string, ship : Ship, correctVal: number) {  
  testequal(testname, countOfLocalMorks(ship), correctVal);  
}
```

```
})
```


Some general code-level design principles

- use good names
- make your data meaningful
- one job per method
- don't repeat yourself (DRY)
- **avoid magic numbers/strings (don't hardcode)**

Avoid magic numbers

- integer and float literals should usually not appear in complex expressions (exception: $x = x + 1$ is always okay)
- same applies to string literals

Avoid magic numbers

- integer and float literals should usually not appear in complex expressions (exception: `x = x + 1` is always okay)
- same applies to string literals


Give them names!

Avoid magic numbers: examples

```
let salesprice = netPrice * 1.06
```

Avoid magic numbers: examples

```
let salesprice = netPrice * 1.06
```



this is a magic number:

Avoid magic numbers: examples

```
let salesprice = netPrice * 1.06
```

this is a magic number:

- no documentation of what it is
- if it needs to change, is this the only place it's used?

Avoid magic numbers: examples

```
let salesprice = netPrice * 1.06
```

this is a magic number:

- no documentation of what it is
- if it needs to change, is this the only place it's used?

```
const salesTaxRate = 1.06
```

```
let salesprice = netPrice * salesTaxRate
```

Avoid magic numbers: another example

- Suppose we are computing income tax in a state with four rates:
 - No tax on incomes less than \$10,000
 - 10% on incomes between \$10,000 and \$20,000
 - 20% on incomes between \$20,000 and \$50,000
 - 25% on incomes greater than \$50,000

Avoid magic numbers: another example

```
function grossTax(income : number): number {  
  if ((0 <= income) && (income <= 10000)) {  
    return 0  
  } else if ((10000 < income) && (income <= 20000)) {  
    return 0.10 * (income - 10000)  
  } else if ((20000 < income) && (income <= 50000)) {  
    return 1000 + 0.20 * (income - 20000)  
  } else {  
    return 7000 + 0.25 * (income - 50000)  
  }  
}
```

Avoid magic numbers: another example

```
function grossTax(income : number): number {  
  if ((0 <= income) && (income <= 10000)) {  
    return 0  
  } else if ((10000 < income) && (income <= 20000)) {  
    return 0.10 * (income - 10000)  
  } else if ((20000 < income) && (income <= 50000)) {  
    return 1000 + 0.20 * (income - 20000)  
  } else {  
    return 7000 + 0.25 * (income - 50000)  
  }  
}
```

What might change?

- boundaries of the tax brackets
- number of brackets

In-class exercise: rewrite to avoid magic numbers

```
function grossTax(income : number): number {  
  if ((0 <= income) && (income <= 10000)) {  
    return 0  
  } else if ((10000 < income) && (income <= 20000)) {  
    return 0.10 * (income - 10000)  
  } else if ((20000 < income) && (income <= 50000)) {  
    return 1000 + 0.20 * (income - 20000)  
  } else {  
    return 7000 + 0.25 * (income - 50000)  
  }  
}
```

In-class exercise: my solution, part 1

```
// defines the tax bracket for income lower < income <= upper.  
// if upper is null, then lower < income (no upper bound)  
type TaxBracket = {  
    lower: number,  
    upper: number | null,  
    base : number,  
    rate : number  
}  
let brackets : TaxBracket[] = [  
    {lower:0, upper:10000, base:0, rate:0},  
    {lower:10000, upper:20000, base:0, rate:0.10},  
    {lower:20000, upper:50000, base:1000, rate:0.20},  
    {lower:50000, upper: null, base:7000, rate:0.25} ]
```

In-class exercise: my solution, part 2

```
// defines the incomes covered by a bracket function
function isInBracket(income : number, bracket : TaxBracket) : boolean {
  return (bracket.upper == null) ?
    (bracket.lower <= income) :
    ((bracket.lower <= income) && (income < bracket.upper))
}

function income2bracket(income : number,
                        brackets : TaxBracket[]) : TaxBracket {
  return brackets.find(b0 => isInBracket(income, b0))
}

function taxByBracket(income : number, bracket : TaxBracket) : number {
  return bracket.base + bracket.rate * (income - bracket.lower)
}

function grossTax(income: number, brackets: TaxBracket[]) : number {
  return taxByBracket(income, income2bracket(income, brackets))
}
```

Avoid magic numbers: another example

- Which of the two is simpler?

Avoid magic numbers: another example

- Which of the two is simpler?
- Answer depends on who you ask:
 - **code writer**: magic numbers version is simpler

Avoid magic numbers: another example

- Which of the two is simpler?
- Answer depends on who you ask:
 - **code writer**: magic numbers version is simpler
 - **code reader**: magic numbers version is shorter, but no magic numbers version is better documented. Toss up.

Avoid magic numbers: another example

- Which of the two is simpler?
- Answer depends on who you ask:
 - **code writer**: magic numbers version is simpler
 - **code reader**: magic numbers version is shorter, but no magic numbers version is better documented. Toss up.
 - **code maintainer who needs to make a change**: magic number version is difficult to deal with, no magic numbers makes the change trivial

Avoid magic numbers: another example

- Which of the two is simpler?
- Answer depends on who you ask:
 - **code writer**: magic numbers version is simpler
 - **code reader**: magic numbers version is shorter, but no magic numbers version is better documented. Toss up.
 - **code maintainer who needs to make a change**: magic number version is difficult to deal with, no magic numbers makes the change trivial

Who to optimize for?

Who to optimize for?

- The code **writer**: only if you expect to throw the code away after you use it once.

Who to optimize for?

- The code **writer**: only if you expect to throw the code away after you use it once.

Example: simple bash script to accomplish a specific, one-off task

Who to optimize for?

- The code **writer**: only if you expect to throw the code away after you use it once.
- The code **reader**: any code you expect to keep. A good heuristic that I use: am I going to check this into source control?

Who to optimize for?

- The code **writer**: only if you expect to throw the code away after you use it once.
- The code **reader**: any code you expect to keep. A good heuristic that I use: am I going to check this into source control?
- The code **maintainer**: any code that is likely to change. This is most code that you're writing in the real world!

Who to optimize for?

- The code **writer**: only if you expect to throw the code away after you use it once.
- The code **reader**: any code you expect to keep. A good heuristic that I use: am I going to check this into source control?
- The code **maintainer**: any code that is likely to change. This is most code that you're writing in the real world!

DANGER: premature optimization via over-engineering
don't sacrifice readability or usability for maintainability!

Code-level Design

Today's agenda:

- Why does code-level design matter?
- Some general principles, with examples
- In-class exercise
- **Automation and linting**
- Our course style guide

A surprise: non-standard formatting

What's wrong with the following (Java) code?

```
public abstract class racecar {  
  
    private final int Number_of_gears = 6;  
  
        public abstract void DRIVE();  
  
    public int GetNumberOfGears() {return Number_of_gears;}  
  
}
```

A surprise: non-standard formatting

What's wrong with the following (Java) code?

```
public abstract class RaceCar {  
  
    private final int NUMBER_OF_GEARs = 6;  
  
    public abstract void drive();  
  
    public int getNumberOfGears() {  
        return NUMBER_OF_GEARs;  
    }  
}
```

A surprise: non-standard formatting

- Doing this ourselves is time-consuming and error-prone
- How do we decide which format is best?

A surprise: non-standard formatting

- Doing this ourselves is time-consuming and error-prone
- How do we decide which format is best?

Solution to both problems: use an **automatic** formatting tool

A surprise: non-standard formatting

- Doing this ourselves is time-consuming and error-prone
- How do we decide which format is best?

Solution to both problems: use an **automatic** formatting tool

- avoids flamewars about e.g., tabs vs spaces
- automatically enforced = we don't have to think about it
- reduces surprises when reading code

Automated formatters

- There's at least one for every language you are likely to be using

Automated formatters

- There's at least one for every language you are likely to be using
- E.g.,:
 - Java has Spotless, GoogleJavaFormat, Checkstyle
 - Python has black, autopep8, yapf
 - Go has gofmt
 - JavaScript has prettier (which we'll use in this class)

Automated formatters

- There's at least one for every language you are likely to be using
- E.g.,:
 - Java has Spotless, GoogleJavaFormat, Checkstyle
 - Python has black, autopep8, yapf
 - Go has gofmt
 - JavaScript has prettier (which we'll use in this class)
- **Lesson:** always use an automated formatter

Aside: “opinionated”

Definition: a tool is *opinionated* if it builds in assumptions about how its target (e.g., your code for an automated formatter) should be

Aside: “opinionated”

Definition: a tool is *opinionated* if it builds in assumptions about how its target (e.g., your code for an automated formatter) should be

A good automated formatter is opinionated: reduces intra-team arguments about formatting.

Automated formatters vs linters

Definition: a *linter* is a static code style checker

Automated formatters vs linters

Definition: a *linter* is a static code style checker

- Linters **find** style problems.
- Automated formatters **fix** style problems.

Automated formatters vs linters

Definition: a *linter* is a static code style checker

- Linters **find** style problems.
- Automated formatters **fix** style problems.

You'll see both terms, and some linters also look for other mistakes.

We'll use both `prettier` (an automated formatter) and `ESLint` (a linter) in this course.

Code-level Design

Today's agenda:

- Why does code-level design matter?
- Some general principles, with examples
- In-class exercise
- Automation and linting
- **Our course style guide**

Course style guide

<https://web.njit.edu/~mjk76/teaching/cs490-au24/policies/style/>

Course style guide

<https://web.njit.edu/~mjk76/teaching/cs490-au24/policies/style/>

I expect you to follow this style guide for all assignments in this course.

Action items before next class

- Finish Individual Project 0 by today AoE
- Mandatory readings for next class (see course calendar)
- Remaining OH for IP0 questions:
 - ~~Nathan: 1:30-2:30pm~~
 - ~~Tiffany: 5:30-6:30pm~~
 - all TA OH are in GITC 4403
 - or ask your questions on Discord

Action items before next class

- Finish Individual Project 0 by today AoE
- Mandatory readings for next class (see course calendar)
- Remaining OH for IP0 questions:
 - ~~Nathan: 1:30-2:30pm~~
 - ~~Tiffany: 5:30-6:30pm~~
 - all TA OH are in GITC 4403
 - or ask your questions on Discord

TAs unfortunately aren't on the payroll officially yet, so unless they make an announcement on Discord these OH won't happen.

