CS 420 Advanced Programming Languages
Fall Semester, 2022
Doc 11 Assignment 2
Oct 4, 2022

```
fn main() {
    let me = String::from("Lucian");
    let vec1 = vec![50.3,50.4,50.5];
    let s1 = StudentGrades {
        name: me,
        grades: vec1,
    };

    let s2 = StudentGrades {
        name: String::from("Dylan"),
        grades: vec![95.0,96.0,97.0],
    };

    println!("Name is: {}", s1.name);
    println!("Average is: {}", s1.average());
    println!("Grade is : {}", s1.grades());

    println!("Name2 is: {}", s2.name);
    println!("Average2 is: {}", s2.average());
    println!("Grade2 is : {}", s2.grades());
}
```

```
let name = String::from("Lucian");
let grades = vec![50.3,50.4,50.5];
let lucian = StudentGrades {
    name: name,
    grades: grades,
};

let lucian = StudentGrades {
    name: String::from("Lucian"),
    grades: vec![50.3,50.4,50.5],
};

let dylan = StudentGrades {
    name: String::from("Dylan"),
    grades: vec![95.0,96.0,97.0],
};
```

Are you going to remember what Average 2 is?

What happens when you have 10,000 lines of code

PrintIn does not scale

```
mod tests {
    use super::*;
    fn setup_student() -> StudentGrades {
        let stud1 = StudentGrades{
            name: String::from("John"),
            grades: vec![70.,70.,75.,80.,80.],
        };
        return stud1;
    }
    pub fn test_student_average(){
        assert_eq!(setup_student().average(), 75.);
    }
    pub fn test_student_grade(){
        assert_eq!(setup_student().grade(), "C");
    }
    fn setup_course() -> CourseGrades {
        let cg: CourseGrades = CourseGrades::from_file("./test.csv").unwrap();
        return cg;
    }
    pub fn test_course_average(){
        assert_eq!(setup_course().average(3).unwrap(), 51.0);
    }
}
```

```
fn main() {
   tests::test_student_average();
    tests::test_student_grade();
    tests::test_course_average();
    tests::test_course_student();
    tests::test_courses_at();
}
```

No need for the main calling the test

Cargo will call tests for you

```
mod tests {
    use super::*;
    fn setup_student() -> StudentGrades {
        let stud1 = StudentGrades{
            name: String::from("John"),
            grades: vec![70.,70.,75.,80.,80.],
        };
        return stud1;
    }
```

stud1 = only one student so why the 1?
Use complete names
The last version does the same thing - shorter

```
    fn setup_student() -> StudentGrades {
        let student = StudentGrades{
            name: String::from("John"),
            grades: vec![70.,70.,75.,80.,80.],
        };
        return student;
    }
```

```
    fn setup_student() -> StudentGrades {
        return StudentGrades{
            name: String::from("John"),
            grades: vec![70.,70.,75.,80.,80.],
        };
    }
```

```
}
```

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_student_grades() {
        let student_grades = StudentGrades {
            name: String::from("John"),
            grades: vec![100.0, 90.0, 80.0, 70.0, 60.0],
        };

        assert_eq!(student_grades.name, "John");
        assert_eq!(student_grades.average(), 80.0);
        assert_eq!(student_grades.grade(), 'B');
        assert_ne!(student_grades.name, "not_a_student");
        assert_ne!(student_grades.grade(), 'A');
    }
```

The command:

cargo test

will run all the tests

```
//file path reads the file given and stores grades
fn file_path(file_path:String)->Self
{
    let mut contents = fs::read_to_string(String::from(file_path))
    .expect("Unable to Read File");

let string_vector = contents.split("\n").collect::<Vec<&str>>();
let mut all_courses: Vec<Course> = vec![];
for line in string_vector.iter()
{
 let mut new_catalog_num = "".to_string();
 let mut new_title = "".to_string();
 let mut new_instructor = "".to_string();
 let mut new_time = "".to_string();
 let mut new_day = "".to_string();
 let mut counter:i64 = 0;

 for word in line.split("\t")
 {
     if counter == 4{
```

Indentation

```
#[derive(PartialEq)]
struct StudentGrades {

    name: String,

    grades: Vec<f32>,

}

impl StudentGrades {

    fn average(&self) -> f32 {

    let mut sum = 0.0;

    for grade in &self.grades {

        sum += grade;

    }

    sum / (self.grades.len() as f32)

}
```

7

```
#[derive(PartialEq)]
struct StudentGrades {
   name: String,
   grades: Vec<f32>,
}

impl StudentGrades {

    fn average(&self) -> f32 {
        let mut sum = 0.0;
        for grade in &self.grades {
            sum += grade;
        }
        sum / (self.grades.len() as f32)
}
```

Which is easier to read

```
pub struct CourseGrades {
}
```

What is the point of CourseGrades?

```
impl CourseGrades {
    // Read txt file line by line
    fn from_file<FilePath>(filename: FilePath) ->
                    io::Result<io::Lines<io::BufReader<File>>> where FilePath: AsRef<Path>, {
        let file = File::open(filename)?;
        Ok(io::BufReader::new(file).lines())
    }

    fn average(N: i32) {
        let option = true;
    }

    fn student(name: &str) {
        let student_name = "Peter";
    }
}
```

```
pub struct CourseGrades {
}

impl CourseGrades {
    // Read txt file line by line
    fn from_file<FilePath>(filename: FilePath) ->
                    io::Result<io::Lines<io::BufReader<File>>> where FilePath: AsRef<Path>, {
       // not sure what to do
    }

fn from_string(&self, student_grades: &str) {

}

fn add_student(&self, student: StudentGrades) {

}
```

```
fn grade(&self) -> String {
    let avg = self.average();
    if avg >= 0.0 && avg <= 59. {
        return String::from("F");
    } else if avg >= 60. && avg <= 69. {
        return String::from("D");
    } else if avg >= 70. && avg <= 79. {
        return String::from("C");
    } else if avg >= 80. && avg <= 89. {
        return String::from("B");
    } else if avg >= 90. && avg <= 100. {
        return String::from("A");
    } else {
        return String::from("Wrong grade scale");
    }
}
```

```
fn grade(&self) -> char {
    let result: f64 = self.average();
    match result {
        x if (60.0..70.0).contains(&x) => 'D',
        x if (70.0..80.0).contains(&x) => 'C',
        x if (80.0..90.0).contains(&x) => 'B',
        x if (90.0..100.1).contains(&x) => 'A',
        _ => ' F',
    }
}
```

```
#[derive(Clone, Default, Debug)]

struct StudentGrades{studentname: String, gradevector: Vec<f64>}


 trait StuGraFunctions {
     fn average (&self) -> f64;
     fn grade (&self) -> char;

     //fn builder(name: String, grades: Vec<f64>) -> StudentGrades;
 }
```

```
| |
```

Question 1

```
#[derive(Debug, PartialEq)]
pub struct StudentGrades {
    name: String,
    grades: Vec<f32>,
}

impl StudentGrades {
    pub fn average(&self) -> f32 {
        self.grades.iter().sum::<f32>() / self.grades.len() as f32
    }

    pub fn grade(&self) -> char {
        match self.average() {
            x if x >= 90.0 => 'A',
            x if x >= 80.0 => 'B',
            x if x >= 70.0 => 'C',
            x if x >= 60.0 => 'D',
            _ => 'F',
        }
    }
}
```

Question 2

```
pub struct CourseGrades {
    course: Vec<StudentGrades>,
}
```

impl CourseGrades

```
impl CourseGrades {
    pub fn from_file(file_path: String) -> Self {
        let file = File::open(file_path).unwrap();
        let reader = BufReader::new(file);
        let mut course = Vec::new();
        for line in reader.lines() {
            let line = line.unwrap();
            let mut parts = line.split(',');
            // first part in parts is the name and the rest are grades
            let name = parts.next().unwrap().to_string();
            let mut grades: Vec<f32> = Vec::new();
            for part in parts {
               let part = part.trim();
                grades.push(part.parse::<f32>().unwrap());
            }
            course.push(StudentGrades { name, grades });
        }
        CourseGrades { course }
    }
```

Note CourseGrades needs to know about StudentGrades structure

impl CourseGrades

```
pub fn average(&self, grade_event: i32) -> Option<f32> {
    let mut sum = 0.0;
    let mut count = 0;
    for student in &self.course {
        if student.grades.len() >= grade_event as usize {
            sum += student.grades[(grade_event - 1) as usize];
            count += 1;
        } else {
            // when grade event is out of range, None is returned
            return None;
        }
    }
    Some(sum / count as f32)
}
```

impl CourseGrades

```
pub fn student(&self, student_name: String) -> Option<&StudentGrades> {
    for student in &self.course {
        if student.name == student_name {
            return Some(student);
        }
    }
    None
}
```

Question 3

```
#[derive(Debug)]
pub struct Course {
    catalog_number: String,
    title: String,
    time: String,
    day: String,
    instructor: String,
}

pub struct CourseSchedule {
    courses: Vec<Course>,
}
```

impl CourseSchedule

```
pub fn courses_at(&self, time: String, day: String) -> Vec<&Course> {
    let mut courses = Vec::new();
    for course in &self.courses {
        if course.time == time && course.day == day {
            courses.push(course);
        }
    }
    courses
  }
}
```

```
impl CourseSchedule {
    pub fn from_file(file_path: String) -> Self {
        let file = File::open(file_path).unwrap();
        let reader = BufReader::new(file);
        let mut courses = Vec::new();
        for (l_index, line) in reader.lines().enumerate() {
            if l_index < 2 { continue; }
             let line = line.unwrap();
             let parts = line.split('\t');
             let mut catalog_number = String::new();
             let mut title = String::new();
             let mut time = String::new();
             let mut day = String::new();
             let mut instructor = String::new();
                                                    for (p_index, part) in parts.enumerate() {
                                                        let part = part.trim();
                                                       if p_index == 4 { // 4th index holds course catalog nu
                                                            catalog_number = part.to_string();
                                                        } else if p_index == 5 { // 5th index holds course title
                                                            title = part.to_string();
                                                       ...
                                                       } else if p_index == 25 { // 25th index holds course ins
                                                            instructor = part.to_string();
                                                        }
                                                    }
                                                    courses.push(Course {  catalog_number, title, time,
                                                                                        day, instructor });
                                                }
                                                CourseSchedule { courses }
                                            }
```

```
impl CourseSchedule {
    fn from_file(file_path: String) -> CourseSchedule {
        // open up the file we want to work with and loop through it line by line
        let file = File::open(file_path).unwrap();
        let reader = BufReader::new(file);
        let mut courses = Vec::new(); // vector to hold all the courses in the file
        for (_index, line) in reader.lines().enumerate() {
            let focus: Vec<&str> = line.unwrap().split("\t").collect();
            if focus.len() >= 26 {
                let course = Course {
                    title: focus[5].to_string(),       // index 5 represents title of course
                    instructor: focus[25].to_string(), // index 25 represents instructor of course
                    start_time: focus[22].to_string(), // i22  = start time of course
                    end_time: focus[23].to_string(),   // 23  = end time
                    days: focus[24].to_string(),       // 24 = days
                };
                courses.push(course);
            }
        }
        return CourseSchedule { courses: courses };
    }
```

Question 2 - Revisited

```
#[derive(Debug, PartialEq)]
pub struct StudentGrades {
    name: String,
    grades: Vec<f32>,
}

impl StudentGrades {
    pub fn new(name: String, grades: Vec<f32>) -> StudentGrades {
        StudentGrades { name, grades }
    }

    pub fn from_string(line: &str) -> StudentGrades {
        let mut parts = line.split(',');
        let name = parts.next().unwrap().to_string();
        let grades = parts.map(|s| s.trim().parse().unwrap()).collect();
        StudentGrades::new(name, grades)
    }
```

let StudentGrades handle the parsing of its data

```
pub fn average(&self) -> f32 {
    self.grades.iter().sum::<f32>() / self.grades.len() as f32
}

pub fn grade(&self) -> char {
    match self.average() {
        x if x >= 90.0 => 'A',
        x if x >= 80.0 => 'B',
        x if x >= 70.0 => 'C',
        x if x >= 60.0 => 'D',
        _ => 'F',
    }
}
```

```
#[test]
fn test_student() {
    let student = StudentGrades::new("John".to_string(), vec![90.0, 80.0, 70.0]);
    assert_eq!(student.name, "John");
    let student = StudentGrades::from_string("John,90 ,80.0 ,70");
    assert_eq!(student.name, "John");
    assert_eq!(student.grades, vec![90.0, 80.0, 70.0]);
}
```

Now we can test the parsing by it self

```
impl CourseGrades {
    pub fn from_file(file_path: String) -> Self {
        let file = File::open(file_path).unwrap();
        let reader = BufReader::new(file);
        let mut course = Vec::new();
        for line in reader.lines() {
            let line = line.unwrap();
            let student = StudentGrades::from_string(&line);
            course.push(student);
        }
        CourseGrades { course }
    }
```

Now CourseGrades from_file is simpler

```
impl CourseGrades {
    pub fn from_file(file_path: String) -> Self {
        let file = File::open(file_path).unwrap();
        let  course = BufReader::new(file)
            .lines()
            .map(|line| StudentGrades::from_string(&line.unwrap()))
            .collect();
        CourseGrades { course }
    }
```