Assignments are required homework. Please complete them yourself, individually, in a timely manner.

This assignment will teach you a new coding paradigm. We will be using this technique for the **rest of this course,** so it is best to go through this assignment slowly and meticulously, grasping every learning outcome as you go.

This assignment was inspired by the book Test Driven Development By Example by Kent Beck. Beck is a legend in our craft and pioneer of TDD and eXtreme Programming (XP). TDD is the first of many principles in XP, which will be completely covered later. Unfortunately, the code examples from the book are very outdated, so this assignment aims to fill that gap while maintaining true to his teachings. That's also why this assignment is so lengthy – it is a replacement to 87 pages of a real book.

Before we dive into what TDD is, I think it's important to cover our expectations so far. We expect that you know how to code in at least 1 Object Oriented Programming (OOP) language. We expect you have built programs before. We expect that you have tested that those programs work. Most likely, that testing involved **running** your program and **supplying values** to see if the program behaved correctly. As you built up your program, you likely tested it as you went. But by the time the program is complete, how can you test that the entire program works correctly? That none of your previously built components broke functionally as you built something else? How many times do you run your **completed** program to check if **everything** works as expected? Is that even a realistic goal?

For non-trivial applications, it's not a realistic goal as there are hundreds of millions of input variations. Later in this course, you will build a complex software system. The paradigm you will learn in this assignment is a technique to building tests as you build your software. When you are done, all those tests can be run to ensure that every single little bit of code still works exactly as expected.

TDD provides a predictable way to go about software development. It results in a dramatic reduction of bugs, which results in a dramatic reduction of wasted time going backwards in development to fix bugs. TDD aims to help us conquer the fear of writing complex software that might not work, or might break later. TDD works off 2 core rules:

- Never write a single line of code unless you have a failing automated test
- Eliminate duplication (think back to this week's lab!)

We follow these rules easily by following the TDD mantra of "red green refactor". This is the set of steps you will follow in a constant repetitive loop:

1. Red – Write a small test that fails. It may fail because:
    a. It doesn't compile (the class or method doesn't exist)
    b. The code doesn't exist yet
2. Green – Write just enough code to make the test pass. Not the line of code you might "know" or think that you need. Only just enough code to make the test pass.
3. Refactor – Eliminate all duplication.

In this assignment we will build an infrastructure system for a dry bar that will charge the user based on what drinks they ordered. For now, the bar will only serve iced tea and coffee. By default, iced tea has 50mg of sugar and a price of 200 cents. By default, coffee has 100mg of sugar and a price of 300 cents. Both can be created with a specified amount of sugar instead as well. Both can have sugar added after

they are made. Any time a drink is ordered, the cost of that drink will be added to the bar's tab. At any time, we can retrieve the total tab cost to pay our bill.
We will follow TDD as religiously as possible, except for when we slip up (intentionally for educational purposes). I have tried for these slips to be as natural as possible as when you will be attempting your own TDD alone later in this course – so that you can learn how to handle those situations!

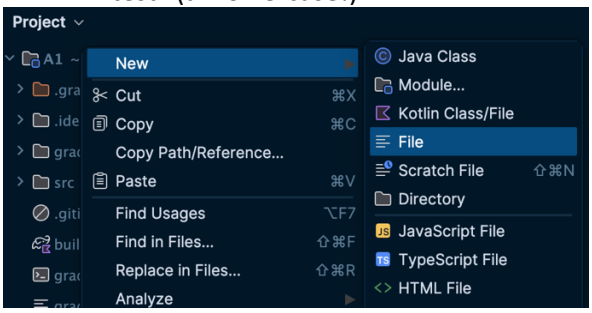
Before we start any software development, it's important to take some time to stop and **think**. During this thinking phase, it can be beneficial to create a TODO list. You can grab a piece of paper or open Notepad, whatever you prefer. What do we have to do to be **done** with our work?

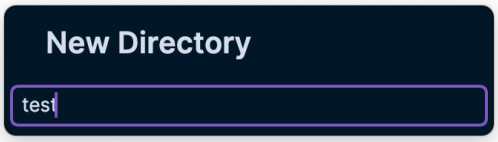
- Iced tea can be created with a default value 50
- Iced tea can be created with a specified value
- Iced tea price is 200 cents
- Coffee can be created with a default value 100
- Coffee can be created with a specified value
- Coffee price is 300 cents
- Sugar can be added to iced tea
- Sugar can be added to coffee
- A bar with a tab exists
- Drinks can be added to the bar's tab
- Price can be retrieved from the bar based on the tab

As we go, try to keep this list up to date by crossing things out as we complete them. I will do the same.

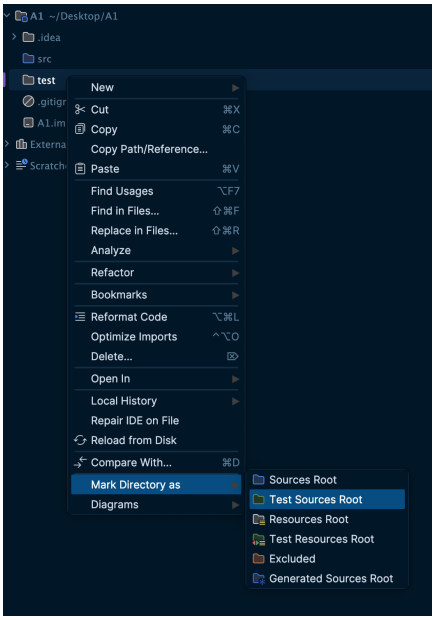
Start by creating a new IntelliJ Java Project called "TDD". **Make sure your plugins have already been set up (per the instructions in this week's lab).**

First, we need to do some setup in our IDE. Right click on your project -> New -> Directory -> name it: "test" (all lowercase!)





Right click on the new folder “test” -> Mark Directory As -> Test Sources Root



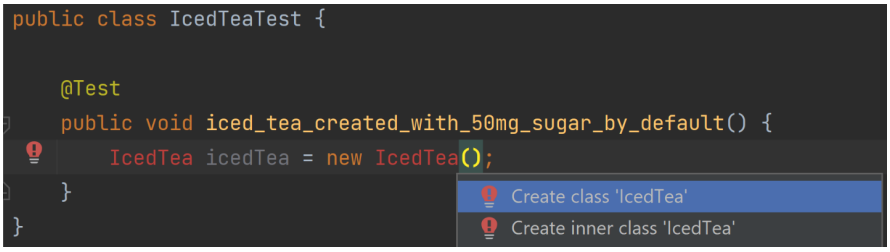
Now we can get started. But where do we start? Let’s think back to step 1 of the TDD mantra – red. What is the **simplest** test we could start with? Let’s create some iced tea. We could have started with coffee too – it doesn’t matter. What’s simpler – creating it with a specified amount of sugar or creating it with no specified sugar and making sure it has 50mg by default? I’m going to go with the latter. The test case will be: “when iced tea is created, it has 50mg of sugar in it.”

The first thing we do is create a **test class**. This will be a home to all test cases for a specific **object**. For example, if I had class called Foobar, I would first create a class called FoobarTest.
Right click on the “test” folder -> New -> Java Class -> name it: IcedTeaTest

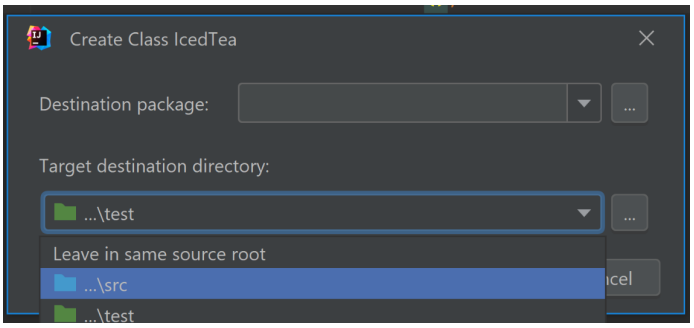
In this new file, type @Test. It will go red. The IDE will prompt you to import JUnit. Make sure you import **JUnit 5** (not 4!!). If it doesn’t prompt you, you can always hover your mouse over anything red or click on something red and hit alt+enter to bring up the prompt menu. Remember this.

Next, we will start by writing our first test:

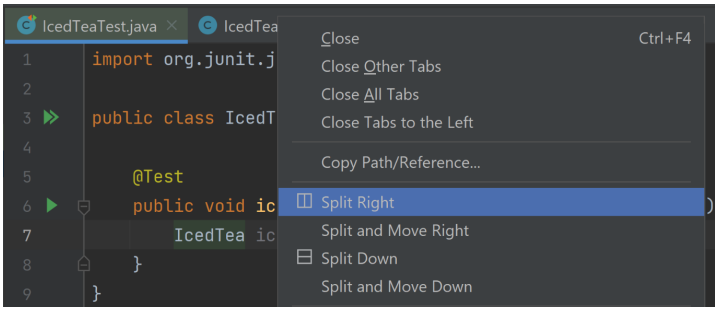
Test names should be in the naming convention lowercase_underscore_separated(). This gives them a distinction of not being real code – they are just there to test our real code. You might wonder about the test name length. Remember that test names speak to the functionality your software is delivering. The more verbose and clear, the more useful they are when presented to your client/boss in the real world. In addition, studies find that for non-technical people snake_case is read 20% faster than camelCase.
We start by creating the object we need to test – IcedTea. But we have a problem – that object doesn't exist! Good!! This is TDD done right.
Test **driven** development, right? Let the tests **drive** your code:



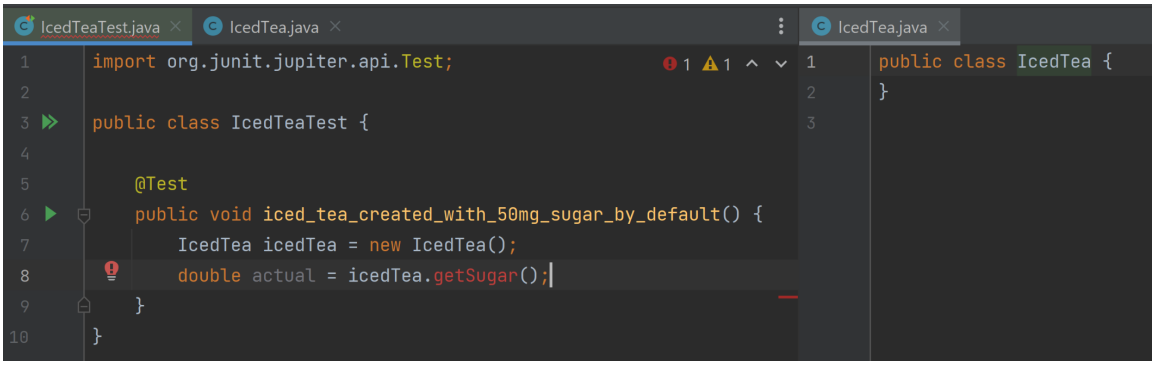
Use the dropdown menu to select "src" as the home for all your **real** code (not tests):



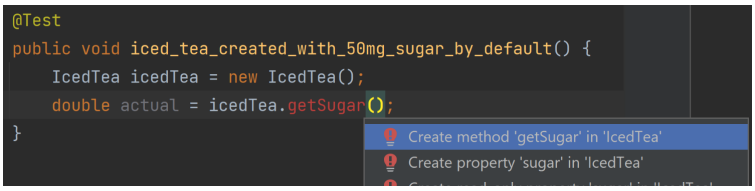
You should now have an IcedTea class. You can right click on its name at the top and select "split right" to look at both the test and the code at the same time. This is typical in the industry:



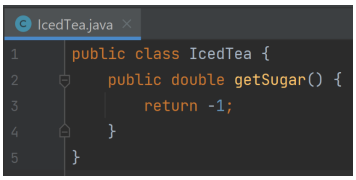
Our test case is far from complete. The next step is to get the sugar contents from our newly created drink:



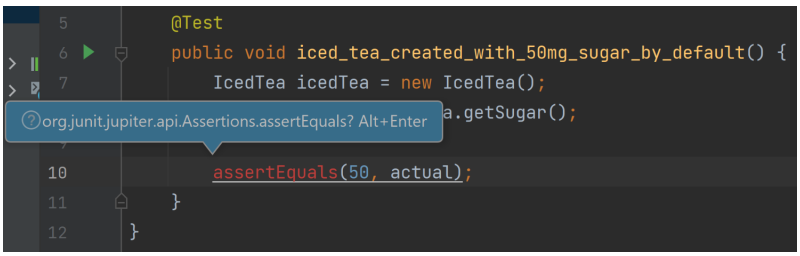
Let's first discuss "double actual". I use a "double" value here because drinks can have decimal values of sugar – for example 5.5mg of sugar would be valid. I use the name "actual" to indicate "this is the value we are going to be checking in our test case."
Now we have a new compilation problem – getSugar() doesn't exist! Good!! This is TDD done right! Let the tests **drive** your development. You will find you have to do far less work when you use TDD as a tool for your advantage:



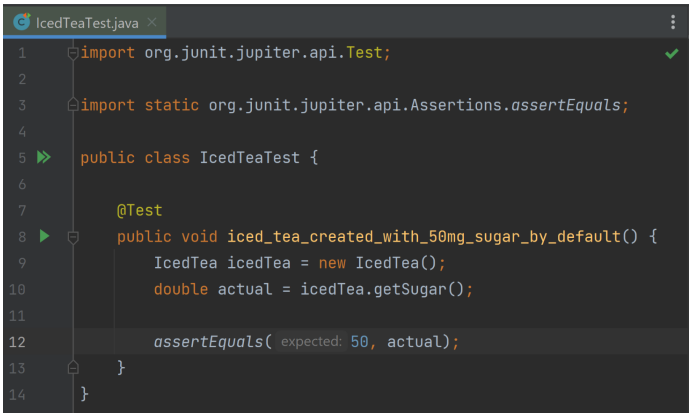
This will create a "best-guess" method in the IcedTea class. What the IDE creates is not always perfect, but it still saves us time every time.



I return -1 – why? I want to see our test **fail**. We are still on step 1 of the TDD mantra – red! While we have had compilation issues, we haven't seen a truly **failing test** yet. But we will soon:

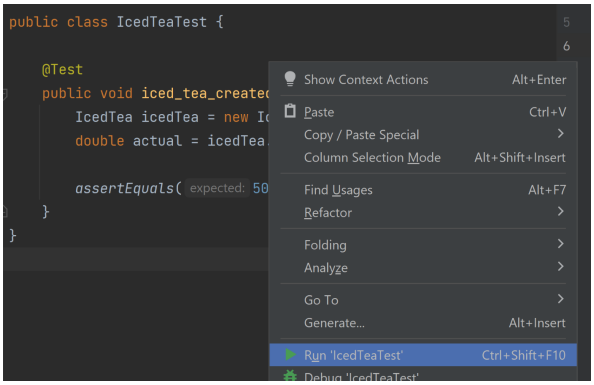


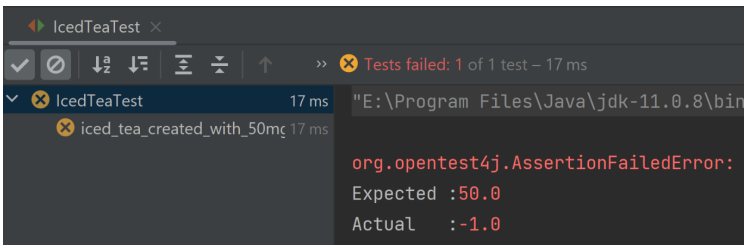
Every single test case you ever write needs to end with an **assertion**. This says “whatever actual value we got should be equal to 50.” If it’s not 50, our software does **not** work as expected! Go ahead and import from Assertions.assertEquals(). Finally, we have our first completed test case:



Let’s review this test case from top to bottom. We create an **instance** of IcedTea. We get the sugar contents from it. We **assert** that the sugar content should be equal to 50.

There are many ways to run tests. I am a keyboard user, so I use CTRL+SHIFT+F10. But you can also hit the green button next to line 5 above, and you can also right click -> Run ‘IcedTeaTest’:

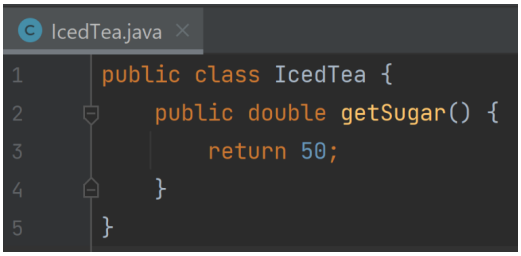




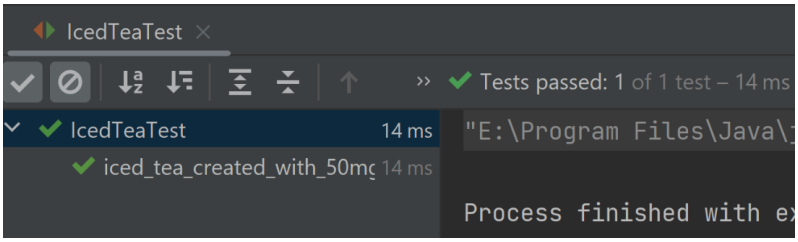
We have completed our step 1 – red. This is what a failing test case is supposed to look like. We expected 50 but we got -1. We know what the problem is – but remember that this is good. This is us following TDD correctly. This methodology enforces small steps with no overwhelming code steps.

This brings us to step 2 – green. What is the least amount of code we could possibly write to make this test pass? Not the line of code that may seem logical. You might be inclined to create a sugar value and

store it – but that's too much code. That's not the **least** amount of code. The point of TDD is to **force** any code we might think we need by creating more test cases later. For now, the solution is as simple as this:



When we rerun the test, it passes:

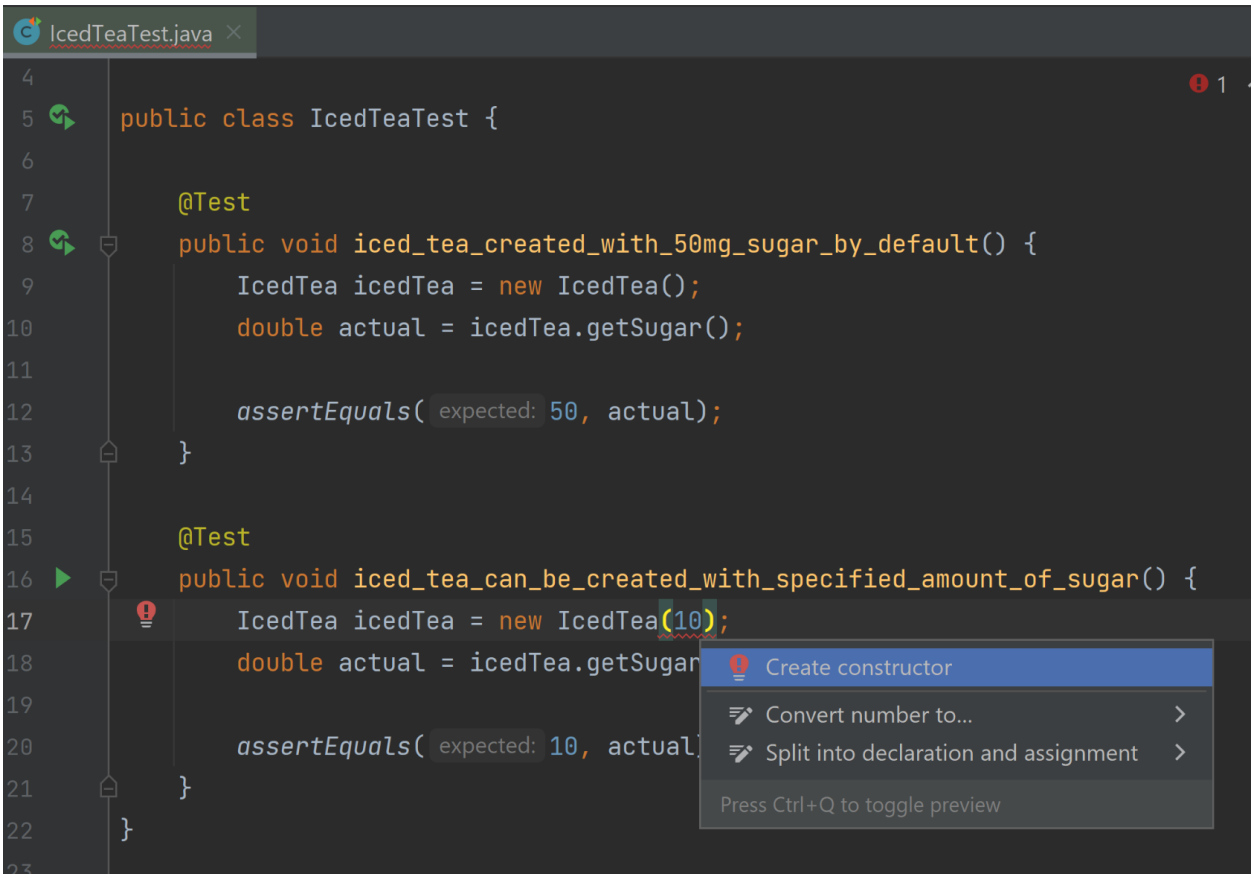


This brings us to step 3 – refactor. Is there any duplicate code on either the left-hand or right-hand side? That is, is there any duplicate code in the test or the real code? Not yet. We're safe to skip this step for now. But be careful ever skipping this step – it's dangerous if you're wrong. Throughout this journey, most of the intentional slips I make are related to skipping refactoring steps and you will feel the pain of going backwards as a result. Remember this pain. Learn from the pain. Don't repeat these mistakes after learning them.

Our TODO list:

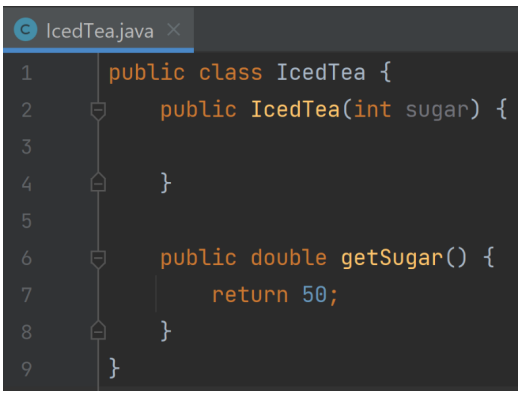
- ~~Iced tea can be created with a default value 50~~
- Iced tea can be created with a specified value
- Iced tea price is 200 cents
- Coffee can be created with a default value 100
- Coffee can be created with a specified value
- Coffee price is 300 cents
- Sugar can be added to iced tea
- Sugar can be added to coffee
- A bar with a tab exists
- Drinks can be added to the bar's tab
- Price can be retrieved from the bar based on the tab

We can move on to our next smallest test case. In this case, I think that is creating an IcedTea with a specified amount of sugar. We can add multiple test cases to a single class, so in this case we will add more tests to our IcedTeaTest class:



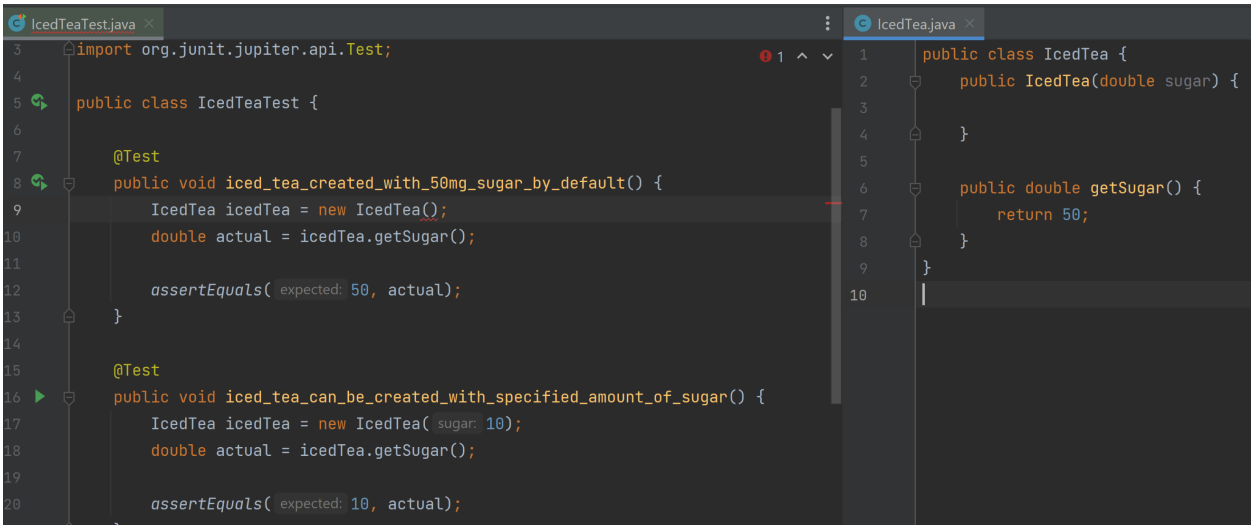
Let's review this test case from top to bottom. We create an **instance** of IcedTea with a specified value of 10. We get the sugar contents from it. We **assert** that the sugar content should be equal to 10.

This time, we have a new compilation issue. A constructor that takes a sugar value does not exist inside of IcedTea. Yet again, we let the tests **drive** our development. This leads to a new constructor in IcedTea:

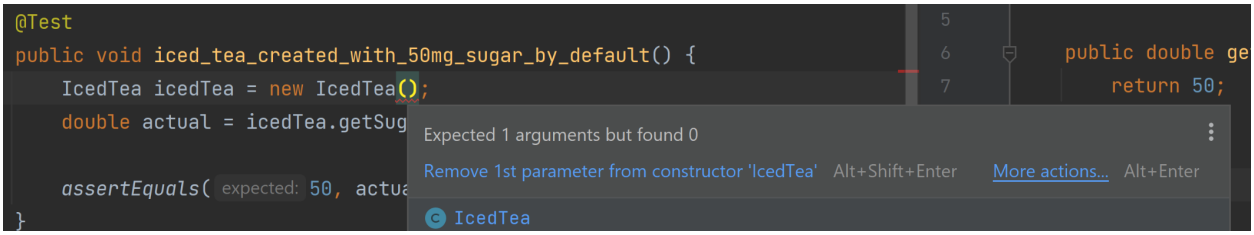


We should rename the parameter from "i" which is not a meaningful name to "sugar" which tells any other developer exactly what is expected to create an IcedTea. However, the IDE did make a mistake here… do you see it? The parameter should not be an int. We decided that sugar will be of type double

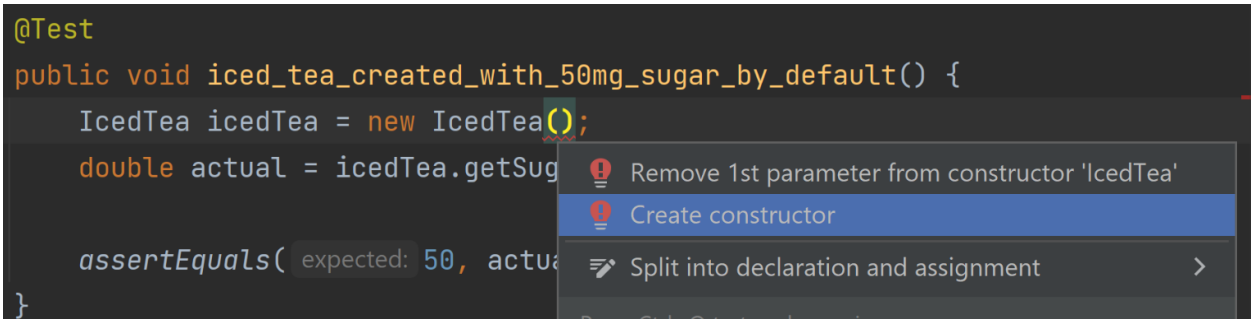
and that decimal values are allowed. So, we should change that (but still be grateful to our IDE for the help anyway):



On the right-hand side, we change “int sugar” to “double sugar”. On the left-hand side, we have yet another compilation issue on line 9! In Java, once you create a constructor with a parameter, you have to manually specify any additional constructors your class should support. In this case, a constructor with no parameters:

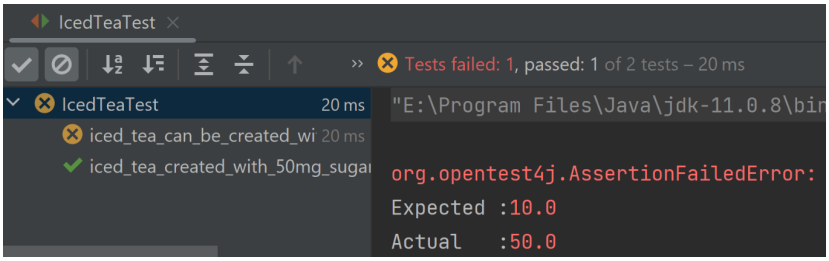


Click on “More actions…” -> Create constructor



```java
public class IcedTea {
    public IcedTea(double sugar) {

    }

    public IcedTea() {

    }

    public double getSugar() {
        return 50;
    }
}
```

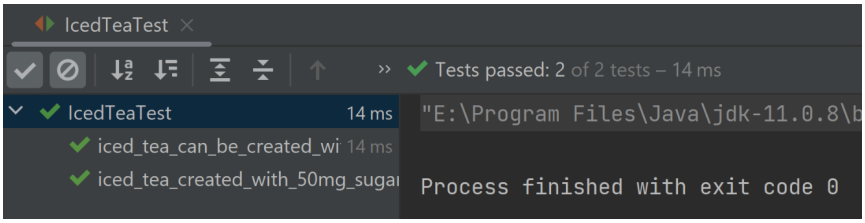
IcedTea should look like this now. We can run our tests now that all compilation issues are gone. When we do, we complete step 1 – red:



We move on to step 2 – green. What is the least amount of code we could possibly write to make this test pass? Not the line of code that might seem logical. Well, if we change “return 50” to “return 10” it would make this test pass – but then our first test would **fail** (try it out if you want). It’s important to note that would be unacceptable. Step 2 – green – means **all** tests across your **entire software system** must pass. It’s time to add a sugar **field** in IcedTea and set it accordingly:

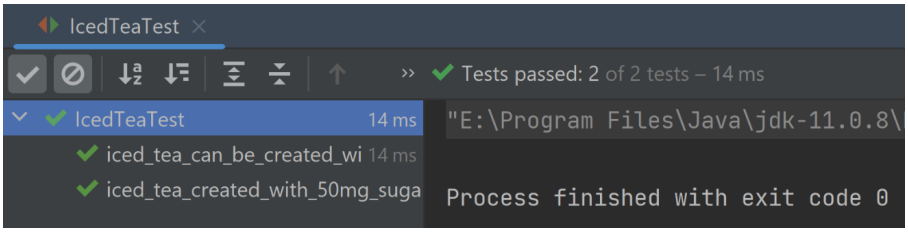
```java
public class IcedTea {
    private double sugar;

    public IcedTea(double sugar) {
        this.sugar = sugar;
    }

    public IcedTea() {
        sugar = 50;
    }

    public double getSugar() {
        return sugar;
    }
}
```

We remember to follow the principle of encapsulation – fields should be **private**. One constructor takes a specified value and sets it. Another constructor just sets sugar to be 50 as a default value. The getter returns the value of sugar. This is a straightforward Java object with a value. Our tests pass:



What's next? Think about it. After step 2 comes… step 3 – refactor! It's easy to forget this step, but this is a critical step to the success of this whole practice. On the right, there is no duplicate code. On the left, there is only a tiny bit. Both tests create a new IcedTea – but they create it differently. So while we can't eliminate the duplication of creating it, we can safely state that every single test in IcedTeaTest will need an instance of IcedTea to work with:

We define IcedTea icedTea on line 7. We **instantiate** it on lines 11 and 19 and any other test that comes next will do the same. When we are done refactoring, it's important to rerun your tests. You'll be surprised how often they **fail**, even though you thought you didn't change any functionality. The tests still pass, what a relief:



Our TODO list:

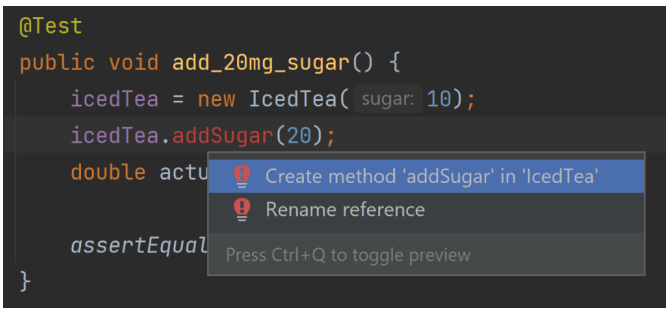
- ~~Iced tea can be created with a default value 50~~
- ~~Iced tea can be created with a specified value~~
- Iced tea price is 200 cents
- Coffee can be created with a default value 100
- Coffee can be created with a specified value
- Coffee price is 300 cents
- Sugar can be added to iced tea
- Sugar can be added to coffee
- A bar with a tab exists
- Drinks can be added to the bar's tab
- Price can be retrieved from the bar based on the tab

What is the next smallest functionality we can tackle? Switching from IcedTea to Coffee before we are done with IcedTea makes little sense. This is actually called **context switching** – a term that comes from psychology. It's our tendency to shift from one unrelated task to another. A quick search on the internet will provide you with a ton of articles explaining why context switching destroys our productivity. It's better to stay focused on one task before moving to another.

Perhaps next up is getting the price of IcedTea? Maybe. But we were just working on sugar, so let's keep going down that path. Since we have a sugar attribute already, adding sugar to our IcedTea should be the smallest next step:

```
@Test
public void add_20mg_sugar() {
    icedTea = new IcedTea( sugar: 10);
    icedTea.addSugar(20);
    double actual = icedTea.getSugar();

    assertEquals( expected: 30, actual);
}
```

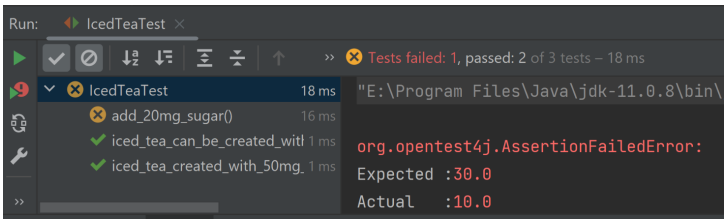
Let's review this test case from top to bottom. We create an **instance** of IcedTea with a specified value of 10. Then we add 20 sugar to it. Then we get the sugar contents from it. Finally, we **assert** that the sugar content should be equal to 30.
The only problem is that addSugar() doesn't exist.. so once again we can allow the IDE to code for us:



This adds a new method in IcedTea:

```
public void addSugar(double sugarToAdd) {

}
```

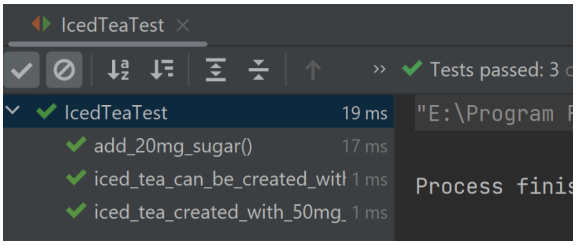
Let's rename the parameter to *sugarToAdd* as a meaningful name for our future selves to appreciate. The IDE may have made the parameter an int, but we know better and change it to a double because sugar allows for decimal values. We can leave the method empty. Our tests now compile, so we can run them to complete step 1 – red:



Step 2 – green. We can make this test pass by adding the sugar provided to our existing sugar content:

```
public void addSugar(double sugarToAdd) {
    sugar += sugarToAdd;
}
```

The tests pass:



We move on to step 3 – refactor. Upon a glance, there is duplication in the tests:

```
17      @Test
18      public void iced_tea_can_be_created_with_specified_amount_of_sugar() {
19          icedTea = new IcedTea( sugar: 10);
20          double actual = icedTea.getSugar();
21
22          assertEquals( expected: 10, actual);
23      }
24
25      @Test
26      public void add_20mg_sugar() {
27          icedTea = new IcedTea( sugar: 10);
28          icedTea.addSugar( sugarToAdd: 20);
29          double actual = icedTea.getSugar();
30
31          assertEquals( expected: 30, actual);
32      }
```

Lines 19 and 27 are exactly the same! You might notice that lines 20 and 29 are exactly the same as well, but we don't want to deal with that because "actual" will not always be a double, so we can't move that out to be a class level field. IcedTea on the other hand will always be an IcedTea:

IcedTeaTest.java

```
6   public class IcedTeaTest {
7
8       IcedTea icedTea;
9
10      @BeforeEach
11      public void setUp() {
12          icedTea = new IcedTea( sugar: 10);
13      }
```

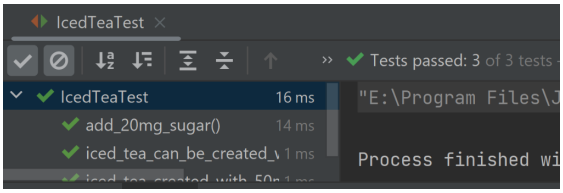
This introduces a new concept: the @BeforeEach annotation. This will run before **each and every** test. This is a fundamental concept to TDD: every test must be **independent**. That is, you might think that test_1 can create an IcedTea and then test_2 can add sugar to it. This would be **wrong**. All tests should be **independent**. That means that test_2 should create its own tea. The @BeforeEach helps with this by ensuring that every test starts off with an IcedTea with 10 sugar in it. The first test, however, can still use the default constructor:

```
@Test
public void iced_tea_created_with_50mg_sugar_by_default() {
    icedTea = new IcedTea();
    double actual = icedTea.getSugar();

    assertEquals( expected: 50, actual);
}
```

While the other tests can get rid of their first line of code:

```java
@Test
public void iced_tea_can_be_created_with_specified_amount_of_sugar() {
    double actual = icedTea.getSugar();

    assertEquals( expected: 10, actual);
}

@Test
public void add_20mg_sugar() {
    icedTea.addSugar( sugarToAdd: 20);
    double actual = icedTea.getSugar();

    assertEquals( expected: 30, actual);
}
```

As always after refactoring remember to rerun all your tests to make sure nothing broke:



Our TODO list:

- ~~Iced tea can be created with a default value 50~~
- ~~Iced tea can be created with a specified value~~
- Iced tea price is 200 cents
- Coffee can be created with a default value 100
- Coffee can be created with a specified value
- Coffee price is 300 cents
- ~~Sugar can be added to iced tea~~
- Sugar can be added to coffee
- A bar with a tab exists
- Drinks can be added to the bar's tab
- Price can be retrieved from the bar based on the tab

We're back to step 1 again – red. What is the smallest functionality we can add? Moving to Coffee or Bar makes no sense, so let's finish with IcedTea by getting its price:

```java
@Test
public void price_is_200_cents() {
    int actual = icedTea.getPrice();

    assertEquals( expected: 200, actual);
}
```

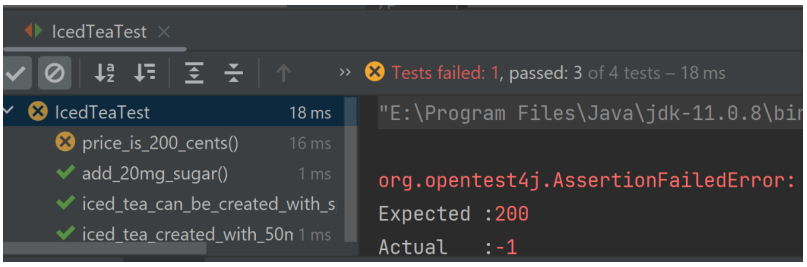
We get price from our IcedTea instance and then assert that it should be 200 cents. Remember when I said "actual" will change type? In this case, cents are integers. Why? Can you have half of a cent? Not for

the purpose of this application you can't since we are supporting only 200 and 300 cent drinks. Yet again, we let our IDE code for us:

In IcedTea:

```
public int getPrice() {
    return -1;
}
```

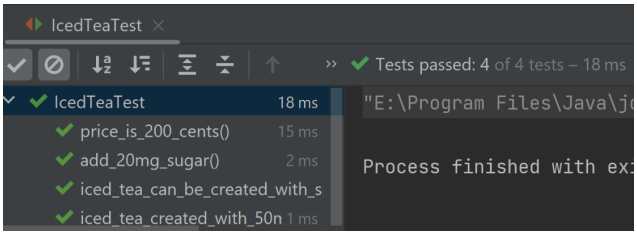
Our test fails as we complete step 1 – red:



Step 2 – green. This one is simple:

```
public int getPrice() {
    return 200;
}
```

Step 2 is complete:



Moving on to step 3: refactor. I don't see any duplicate code.

Our TODO list:

- ~~Iced tea can be created with a default value 50~~
- ~~Iced tea can be created with a specified value~~
- ~~Iced tea price is 200 cents~~
- Coffee can be created with a default value 100
- Coffee can be created with a specified value
- Coffee price is 300 cents
- ~~Sugar can be added to iced tea~~
- Sugar can be added to coffee
- A bar with a tab exists
- Drinks can be added to the bar's tab
- Price can be retrieved from the bar based on the tab

Back to step 1 – red. What is the next smallest functionality? We're done with IcedTea – it is feature complete! Woohoo! We can move on to Coffee. Coffee looks an awful lot like IcedTea functionally. If you're new to this, you might not remember to make an abstraction and do something like this instead:

Coffee.java

```java
public class Coffee {
    double sugar;

    public Coffee(double sugar) {
        this.sugar = sugar;
    }

    public Coffee() {
        sugar = 100;
    }

    public double getSugar() {
        return sugar;
    }

    public void addSugar(double sugarToAdd) {
        sugar += sugarToAdd;
    }

    public int getPrice() {
        return 300;
    }
}
```

We could repeat the actions of IcedTea all over again but with Coffee. Make a test – add a little bit of code. Four tests later we'd be done with our Coffee. But look at what we would end up with:

Coffee.java

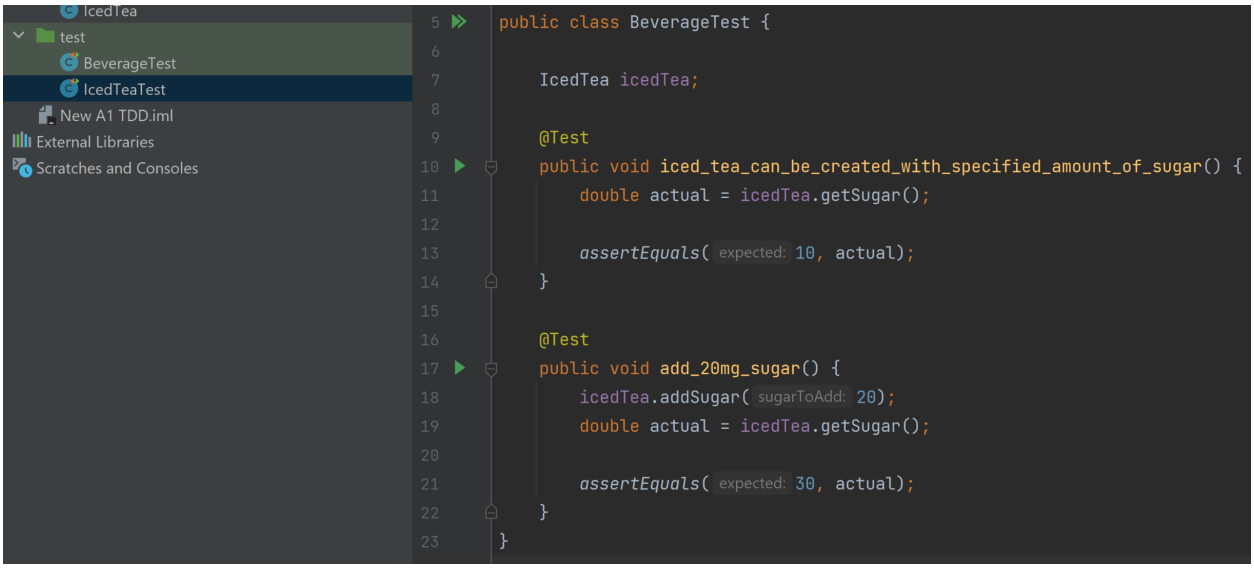
```java
public class Coffee {
    double sugar;

    public Coffee(double sugar) {
        this.sugar = sugar;
    }

    public Coffee() {
        sugar = 100;
    }

    public double getSugar() {
        return sugar;
    }

    public void addSugar(double sugarToAdd) {
        sugar += sugarToAdd;
    }

    public int getPrice() {
        return 300;
    }
}
```

IcedTea.java

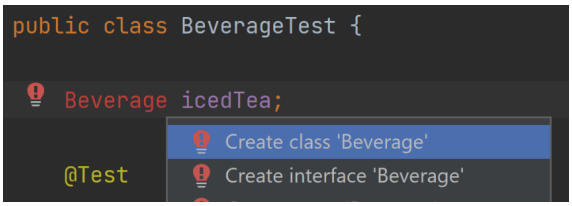
```java
public class IcedTea {
    double sugar;

    public IcedTea(double sugar) {
        this.sugar = sugar;
    }

    public IcedTea() {
        sugar = 50;
    }

    public double getSugar() { return sugar; }

    public void addSugar(double sugarToAdd) {
        sugar += sugarToAdd;
    }

    public int getPrice() {
        return 200;
    }
}
```

In the lab, I taught you to play “spot the duplicate code” and warned you that you will never be done playing this “game.” This is something we do **constantly.** When you gain experience, you can do it in your head before you even go down the path of duplication. Hopefully you can see just how much is duplicated here. We should not do this. In case you did do this, **delete CoffeeTest and Coffee now**.

Instead, we should make an **abstraction**. What is a noun that fills in the blank to both of these statements? IcedTea is a ____. Coffee is a ____. Let’s say: Beverage. We can create a BeverageTest and move all the **common** functionality tests over. We have completed 4 functionalities so far. Creating IcedTea with a default sugar value is not common – 50 vs 100 for Coffee. Creating IcedTea with a specified sugar value **is common** – works the same way for Coffee. Adding the sugar value **is common** – works the same way for Coffee. Getting price is not common – 200 vs 300 for Coffee. In total, 2/4 tests can be migrated to the parent class: Beverage. We start by creating BeverageTest and pasting them in:



Line 7 is wrong. We copied over IcedTea icedTea, but what class are we testing in a class called BeverageTest? Should be Beverage. We can assign it a type Beverage, which tells the reader that we want *icedTea* to be a Beverage. We can then let the IDE help us create the Beverage class:



Then let the IDE help you create the methods needed for the tests:

Beverage.java

```
public class Beverage {
    public double getSugar() {
        return -1;
    }

    public void addSugar(double sugarToAdd) {
    }
}
```

As we did with IcedTea, make sure getSugar returns a double, returns -1 to ensure a failing test, and that addSugar() takes a double called sugarToAdd (better name than i). Next, we need to instantiate our *icedTea* field. You might try:

```
Beverage icedTea;

@BeforeEach
public void setUp() {
    icedTea = new Beverage();
}
```

But this is fundamentally **wrong**. A beverage is not something that can be created. If you walked into a dry bar and asked for "one beverage please," you would get the response "what kind?" That means Beverage should be an **abstraction,** a way for us to prevent creating a Beverage object:

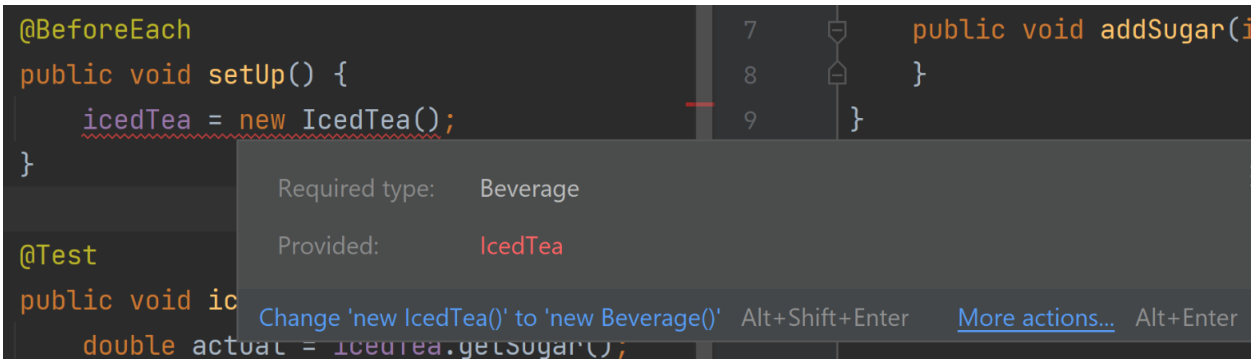
Beverage.java

```
public abstract class Beverage {
    public double getSugar() {
        return -1;
    }

    public void addSugar(double sugarToAdd) {
    }
}
```

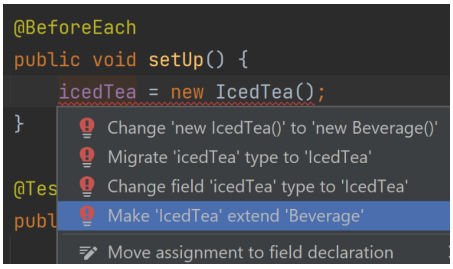
We add the word "abstract" to our class definition. Now we can't create a Beverage:

```
@BeforeEach
public void setUp() {
    icedTea = new Beverage();
}
```

But we can test a **type** of Beverage. Any type of Beverage that leverages the methods inside it will do. So, for now since we only have IcedTea, that's what we'll use:



Java will complain because IcedTea is not a Beverage yet according to our code. Click on “More actions…” and then “Make IcedTea extend Beverage”:



This adds the **inheritance** for us:

```
public class IcedTea extends Beverage {
```

For each method we put in the parent Beverage, we should delete in the child. Our child is much simpler now:

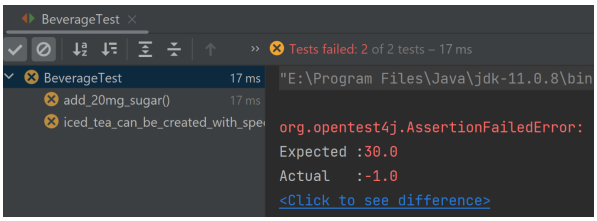
IcedTea.java

```
public class IcedTea {
    private double sugar;

    public IcedTea(double sugar) {
        this.sugar = sugar;
    }

    public IcedTea() {
        sugar = 50;
    }

    public double getSugar() {
        return 200;
    }
}
```

In BeverageTest, we need to create an IcedTea with 10 sugar:

```java
public class BeverageTest {

    Beverage icedTea;

    @BeforeEach
    public void setUp() {
        icedTea = new IcedTea( sugar: 10);
    }
```

We can finally run our BeverageTests and complete step 1 – red:



We move on to step 2 – green. In coding Beverage to pass our tests, we realize something:

Beverage.java

```java
public abstract class Beverage {
    double sugar;

    public double getSugar() {
        return sugar;
    }

    public void addSugar(double sugarToAdd) {
        sugar += sugarToAdd;
    }
}
```

Since all Beverages have sugar, the children **should not**. We remove sugar from the child class:

IcedTea.java

```java
public class IcedTea extends Beverage {

    public IcedTea(double sugar) {
        this.sugar = sugar;
    }

    public IcedTea() {
        sugar = 50;
    }

    public int getPrice() {
        return 200;
    }
}
```

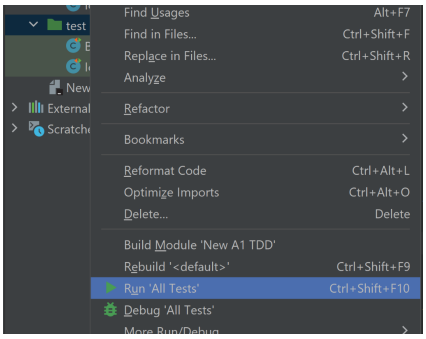
All of our tests pass, so we move on to our refactor step:

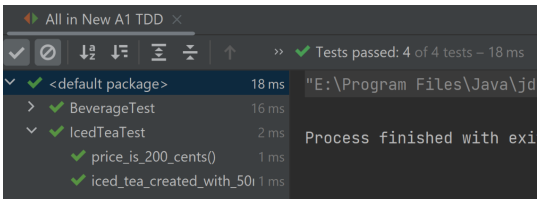
IcedTeaTest.java

```java
public class IcedTeaTest {

    IcedTea icedTea;

    @BeforeEach
    public void setUp() {
        icedTea = new IcedTea();
    }

    @Test
    public void iced_tea_created_with_50mg_sugar_by_defau
        double actual = icedTea.getSugar();

        assertEquals( expected: 50, actual);
    }

    @Test
    public void price_is_200_cents() {
        int actual = icedTea.getPrice();

        assertEquals( expected: 200, actual);
    }
}
```

We change the @BeforeEach to set up an IcedTea with the default sugar value. As a matter of fact, the only thing these tests test are default values – 50 and 200 respectively.

At this point, it's time to learn a new habit. Instead of running BeverageTests or IcedTeaTests, we should run **all the tests.** We can do this easily by right clicking on the test folder -> Run 'All Tests':





All our tests pass! Our refactor is complete.

Before we move on, this is an important time to take a learning break. Ron Jeffries, a famous software craftsman, once said: “Reduced duplication, high expressiveness, and early building of simple abstractions.”
Reduced duplication – we’re already doing very well in that aspect by following our refactoring step.
High expressiveness – this speaks to meaningful names (test names, class names, variable names, method names, etc).

Early building of simple abstractions. What does that mean? The last couple pages have been a mild **violation** of software engineering best practices. What we **should** have done is start with a BeverageTest, and work from there, **not** IcedTeaTest. **Early** building of **simple** abstractions means that even having a very simple shell of a Beverage abstraction early would have saved us a **lot of time** in cumbersome refactoring of our code later. Did you get confused or overwhelmed right around the time we finished with IcedTea and moved on to Coffee / Beverage? That’s the **pain** I warned you about at the start. Sorry for doing that to you – but that’s going to happen to you if you’re not careful and you need to be prepared with how to deal with the situation.

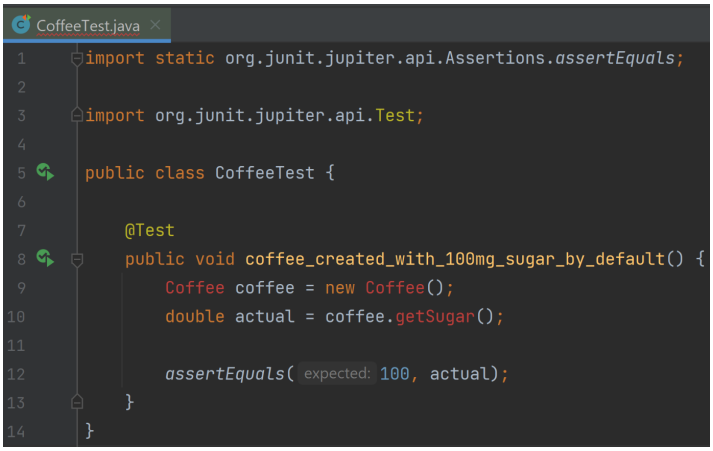
The sooner you realize you need an **abstraction**, the less **painful** transitioning will be. For the sake of this assignment, I wanted to give you the experience of what is likely to be your experience in this course – not realizing you need an abstraction until later. Now you know how to deal with that: make an abstraction, move tests that test **common** behaviors, and remember to remove any code that you put into the parent class from the child class.

Let’s move on. We’re back to step 1 – red. Our TODO list looks like this:

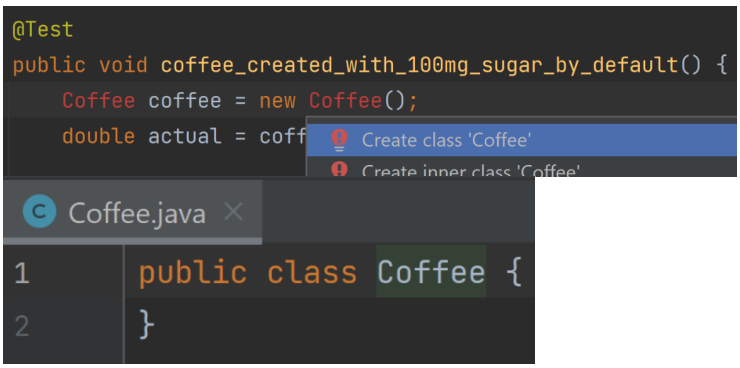
- ~~Iced tea can be created with a default value 50~~
- ~~Iced tea can be created with a specified value~~
- ~~Iced tea price is 200 cents~~
- Coffee can be created with a default value 100
- Coffee can be created with a specified value

- Coffee price is 300 cents
- ~~Sugar can be added to iced tea~~
- Sugar can be added to coffee
- A bar with a tab exists
- Drinks can be added to the bar's tab
- Price can be retrieved from the bar based on the tab

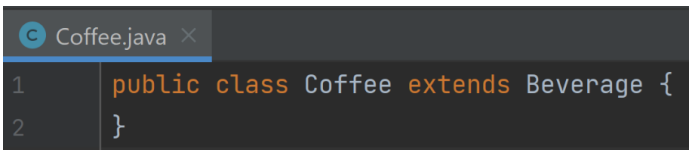
What is the next smallest functionality to work on? Let's start Coffee the same way we started IcedTea – a test to make sure we can create a Coffee with a default of 100 sugar:



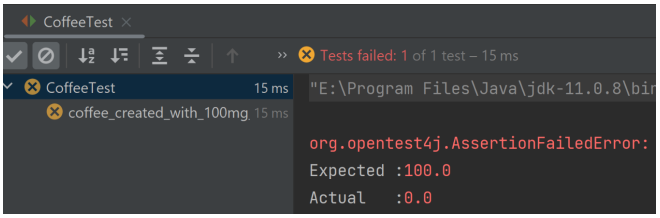
Now that we have an **abstraction** to use, our process will be a lot smoother. As we go through this Coffee part of the exercise, compare it to the IcedTea part. Reflect on how much easier IcedTea would have been if we had made a Beverage abstraction **sooner.** First, we create Coffee:



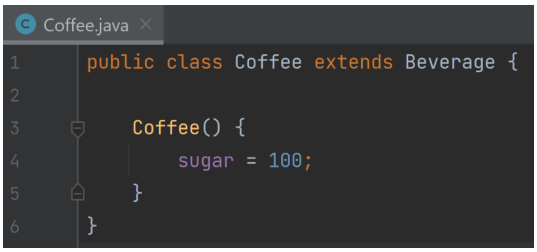
But now, when it comes to getSugar(), we know Beverage has this method. The only issue is that Coffee is not a Beverage yet. Let the tests **drive** that – the compilation issue in our test counts:



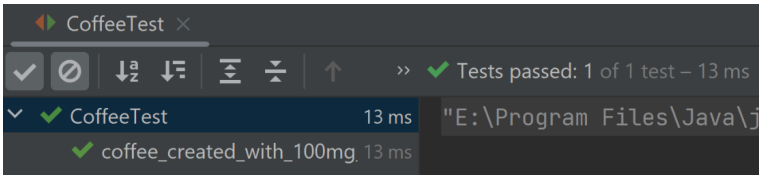
Now that we are clear of compilation issues, we can run our test:



We completed step 1 – red. We move on to step 2 – green. Since Beverage **has** a sugar field, we can just use it to make the test pass easily:



The test passes:



With step 2 – green – completed we move on to step 3 – refactor. Is there any duplicate code? I don't see any so let's move on. Our TODO list looks like this:

- ~~Iced tea can be created with a default value 50~~
- ~~Iced tea can be created with a specified value~~
- ~~Iced tea price is 200 cents~~
- ~~Coffee can be created with a default value 100~~
- Coffee can be created with a specified value
- Coffee price is 300 cents
- ~~Sugar can be added to iced tea~~
- Sugar can be added to coffee
- A bar with a tab exists
- Drinks can be added to the bar's tab
- Price can be retrieved from the bar based on the tab

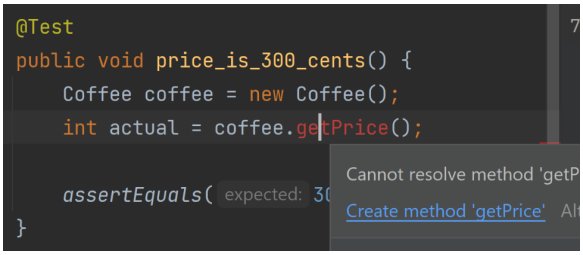
But wait! A Coffee is a Beverage now. That means that adding sugar **just works.** How do we know that? We have that functionality tested in BeverageTest. As long as that test passes, it will work for any Beverage type. Neat! So, our TODO list actually looks like this:

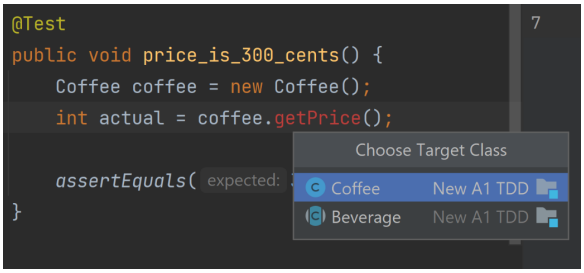
- ~~Iced tea can be created with a default value 50~~
- ~~Iced tea can be created with a specified value~~
- ~~Iced tea price is 200 cents~~

- ~~Coffee can be created with a default value 100~~
- Coffee can be created with a specified value
- Coffee price is 300 cents
- ~~Sugar can be added to iced tea~~
- ~~Sugar can be added to coffee~~
- A bar with a tab exists
- Drinks can be added to the bar's tab
- Price can be retrieved from the bar based on the tab

What's our next smallest functionality to code? If we look at IcedTea, we see only 2 tests: one for default sugar value and one for price. We already have Coffee's default sugar value, so let's move on to price:

```
@Test
public void price_is_300_cents() {
    Coffee coffee = new Coffee();
    int actual = coffee.getPrice();

    assertEquals( expected: 300, actual);
}
```

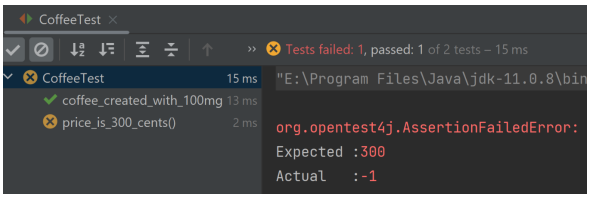




When you get to this pop-up, select Coffee. But the option here is a hint from the IDE that we made a mistake earlier. I'll get back to this later – while we are red is not the time. This adds a method to Coffee:

```
public int getPrice() {
    return -1;
}
```

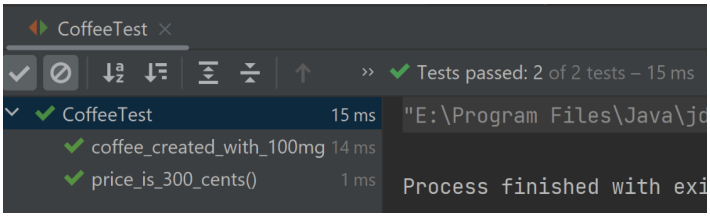
Our test fails:



Step 1 – red – is complete. Step 2 – green – should be straightforward:

```
public int getPrice() {
    return 300;
}
```

Step 2 – green – is complete:



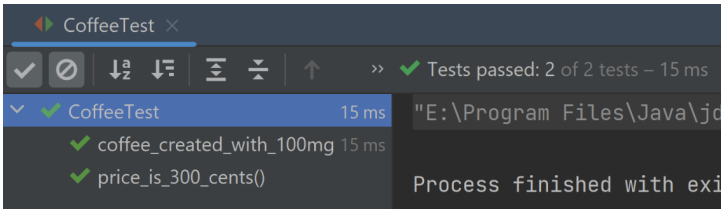
Moving on to step 3 – refactor. Spot the duplicate code:

```
7       @Test
8       public void coffee_created_with_100mg_sugar_by_default() {
9           Coffee coffee = new Coffee();
10          double actual = coffee.getSugar();
11
12          assertEquals( expected: 100, actual);
13      }
14
15      @Test
16      public void price_is_300_cents() {
17          Coffee coffee = new Coffee();
18          int actual = coffee.getPrice();
19
20          assertEquals( expected: 300, actual);
21      }
22  }
```

Lines 9 and 17 are duplicated. We know how to solve this problem:

```java
public class CoffeeTest {

    Coffee coffee;

    @BeforeEach
    public void setUp() {
        coffee = new Coffee();
    }

    @Test
    public void coffee_created_with_100mg_sugar_by_default() {
        double actual = coffee.getSugar();

        assertEquals( expected: 100, actual);
    }

    @Test
    public void price_is_300_cents() {
        int actual = coffee.getPrice();

        assertEquals( expected: 300, actual);
    }
}
```

Don’t forget to rerun tests when done refactoring, to make sure nothing broke:



Our TODO list:

- ~~Iced tea can be created with a default value 50~~
- ~~Iced tea can be created with a specified value~~
- ~~Iced tea price is 200 cents~~
- ~~Coffee can be created with a default value 100~~
- Coffee can be created with a specified value
- ~~Coffee price is 300 cents~~
- ~~Sugar can be added to iced tea~~
- ~~Sugar can be added to coffee~~
- A bar with a tab exists
- Drinks can be added to the bar’s tab
- Price can be retrieved from the bar based on the tab

What’s the next smallest functionality to work on? A Coffee can be created with a specified sugar value. This brings us to a slip up we made before – a mistake that lingered from **not starting with an abstraction** in the first place. Let’s look at our Coffee and IcedTea side by side:

Coffee.java

```java
public class Coffee extends Beverage {

    Coffee() {
        sugar = 100;
    }

    public int getPrice() {
        return 300;
    }
}
```

IcedTea.java

```java
public class IcedTea extends Beverage {

    public IcedTea(double sugar) {
        this.sugar = sugar;
    }

    public IcedTea() {
        sugar = 50;
    }

    public int getPrice() {
        return 200;
    }
}
```

If we created a test in Coffee to create it with a specified value, all we would do is end up with the same constructor IcedTea has, where we do this.sugar = sugar; But we should know better than to do that by now, so let's be better and **not create duplication in the first place**. That constructor is **common** and should have been in the parent Beverage's code. We're about to make some big changes that we should have done a long time ago. We're doing it late for the purpose of education.

Beverage.java

```java
public abstract class Beverage {
    private double sugar;

    public Beverage(double sugar) {
        this.sugar = sugar;
    }

    public double getSugar() {
        return sugar;
    }

    public void addSugar(double sugarToAdd) {
        sugar += sugarToAdd;
    }
}
```

IcedTea.java

```java
public class IcedTea extends Beverage {

    public IcedTea(double sugar) {
        super(sugar);
    }

    public IcedTea() {
        super( sugar: 50);
    }

    public int getPrice() {
        return 200;
    }
}
```

In the parent Beverage:

- We change sugar to be **private**. We couldn't do this in the past because IcedTea was using the sugar field – but it shouldn't.
- We add a **common** constructor that sets any specified sugar value.

In the child IcedTea:

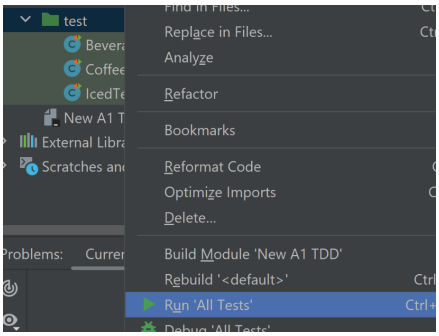
- Both of our constructors change to use our **parent's** constructor. That's what super() does. In one, we pass along a specified value. In the other, we pass along our default value of 50.

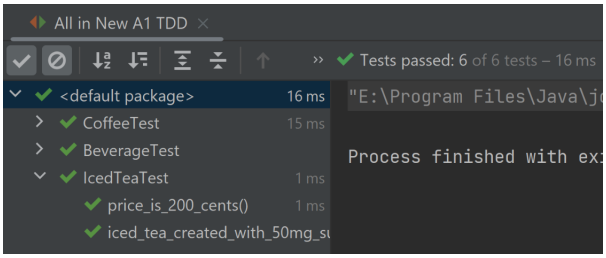
We do the same in Coffee:

Coffee.java

```java
public class Coffee extends Beverage {

    public Coffee(double sugar) {
        super(sugar);
    }

    public Coffee() {
        super( sugar: 100);
    }

    public int getPrice() {
        return 300;
    }
}
```

As always after making changes that span multiple classes, we should run **all** our tests:



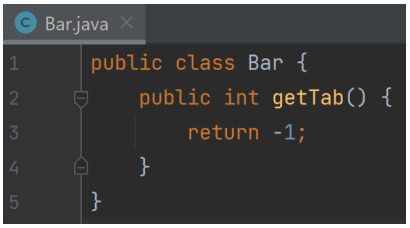


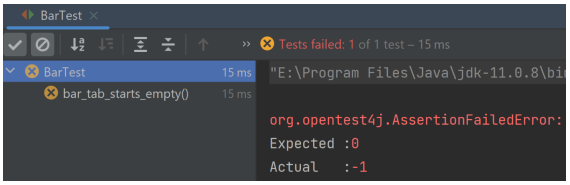
Let's move on. Our TODO list:

- ~~Iced tea can be created with a default value 50~~
- ~~Iced tea can be created with a specified value~~
- ~~Iced tea price is 200 cents~~
- ~~Coffee can be created with a default value 100~~
- ~~Coffee can be created with a specified value~~
- ~~Coffee price is 300 cents~~
- ~~Sugar can be added to iced tea~~
- ~~Sugar can be added to coffee~~
- A bar with a tab exists

- Drinks can be added to the bar's tab
- Price can be retrieved from the bar based on the tab

Back to step 1 – red. What's the smallest functionality we can add? A Bar is created with an empty tab. The following screenshots will depict the typical motions we follow when TDDing a new class:





Step 1 – red – complete. Step 2 – green:

```
Bar.java
1  public class Bar {
2      public int getTab() {
3          return 0;
4      }
5  }
```

```
BarTest
Tests passed: 1 of 1 test – 18 ms
BarTest                18 ms   "E:\Program Files\Java\
bar_tab_starts_empty()  18 ms
```

Step 3 – refactor. I don't see any duplicate code yet, so we move on. Our TODO list:

- ~~Iced tea can be created with a default value 50~~
- ~~Iced tea can be created with a specified value~~
- ~~Iced tea price is 200 cents~~
- ~~Coffee can be created with a default value 100~~
- ~~Coffee can be created with a specified value~~
- ~~Coffee price is 300 cents~~
- ~~Sugar can be added to iced tea~~
- ~~Sugar can be added to coffee~~
- ~~A bar with a tab exists~~
- Drinks can be added to the bar's tab
- Price can be retrieved from the bar based on the tab

Back to step 1. What's the next smallest functionality we could work on? Let's add a single IcedTea to the tab and make sure that we are charged 200 cents:

```
@Test
public void add_iced_tea_to_tab() {
    Bar bar = new Bar();
    bar.addBeverage(new IcedTea());
}
```

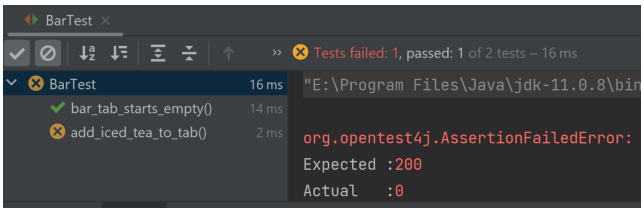
```
Create method 'addBeverage' in 'Bar'
Rename reference
```

```
public void addBeverage(Beverage beverage) {
}
```

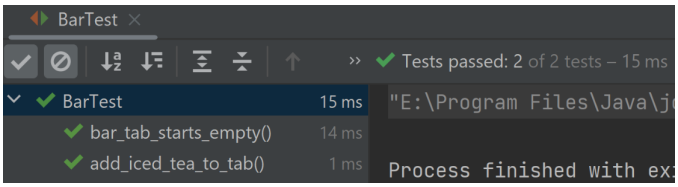
The IDE might create a method that takes an IcedTea object. But we're smart software engineers – we know that we should use the parent class so that Coffee will work as well.

```java
@Test
public void add_iced_tea_to_tab() {
    Bar bar = new Bar();
    bar.addBeverage(new IcedTea());
    int actual = bar.getTab();

    assertEquals( expected: 200, actual);
}
```

We finish our test case, asserting that the cost will be 200 when we get the tab after adding IcedTea to it:



Step 1 – red – complete. Step 2 – green. This will be much like when we added sugar to our IcedTea:

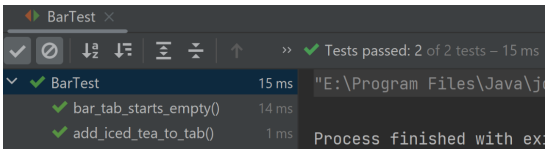


Notice that we aren't using our beverage parameter. Step 2 states to use the **least amount of code** to pass the test. The price 200 is what we needed for IcedTea. Will that work for Coffee? No. Do I need to use beverage? Yes, but I have to **prove that I need it** using another test. But let's not jump ahead of ourselves. It's easy to jump to the next test case. But that's not what we do. When we complete step 2, we go to step 3 – refactor. Hopefully by now you're very familiar with this refactor step:

BarTest.java

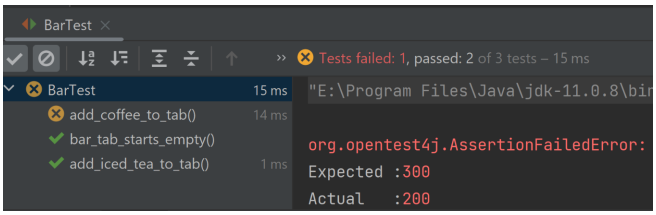
```java
public class BarTest {

    Bar bar;

    @BeforeEach
    public void setUp() {
        bar = new Bar();
    }

    @Test
    public void bar_tab_starts_empty() {
        int actual = bar.getTab();

        assertEquals( expected: 0, actual);
    }

    @Test
    public void add_iced_tea_to_tab() {
        bar.addBeverage(new IcedTea());
        int actual = bar.getTab();

        assertEquals( expected: 200, actual);
    }
```

As always after a refactor, make sure to run the tests:

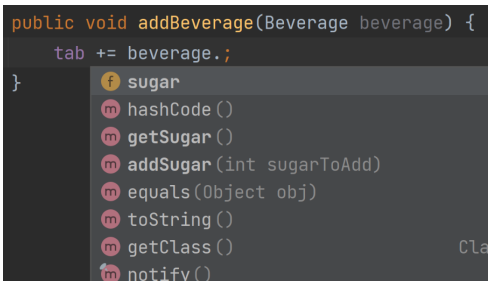


Back to step 1 – red. What's the next smallest functionality to add? We can't cross off "Drinks can be added to the bar's tab" until we prove it works for several drink types. We added an IcedTea but we haven't successfully done a second drink type. Let's do Coffee:

```java
@Test
public void add_coffee_to_tab() {
    bar.addBeverage(new Coffee());
    int actual = bar.getTab();

    assertEquals( expected: 300, actual);
}
```



Step 1 complete. Step 2 – green. Let's try to get the price from the provided Beverage:

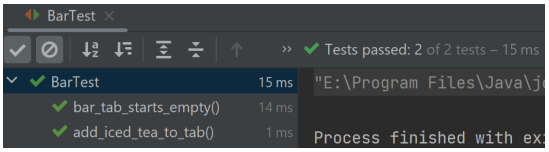


But wait! There's one more mistake that we haven't caught up to yet. Remember the hint the IDE gave us when we went to add the getPrice() method into Coffee? Coffee has a getPrice() and IcedTea has a getPrice(). But we can't use getPrice() polymorphically because we never put that **abstract method** into the parent.

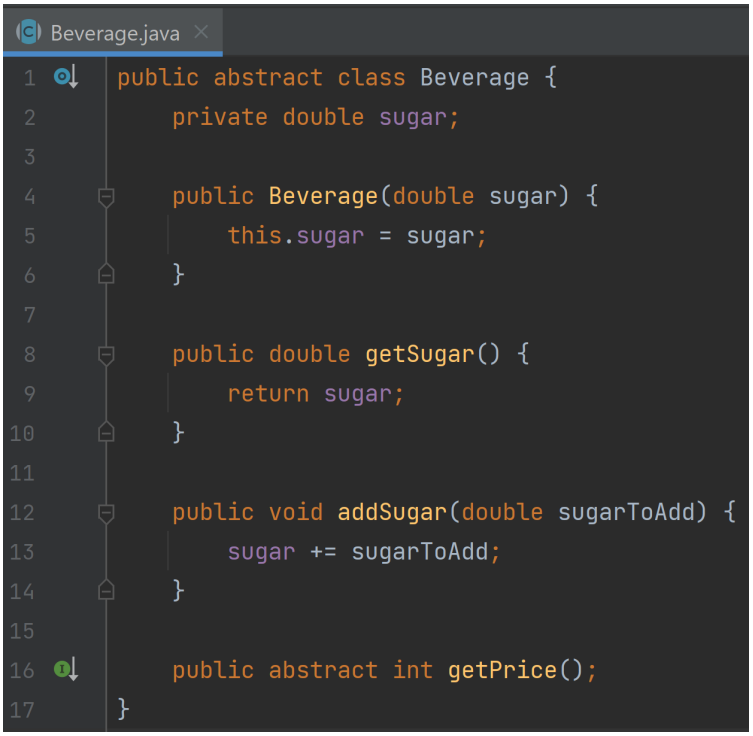
Line 11 is the same. This method should be **abstracted** out into the parent. Before we go ahead and make changes overwhelming changes, it's important **not to refactor while red**. If you ever go down the path of refactoring while red, you will not be able to run the tests at the end to find out if you successfully refactored. You might have even made things worse, as is often the case. Important principle here: always go back to green before refactoring. The easiest way to do that is to simply comment out the @Test of the test we're working on:

```
// @Test
public void add_coffee_to_tab() {
    bar.addBeverage(new Coffee());
    int actual = bar.getTab();

    assertEquals( expected: 300, actual);
}
```

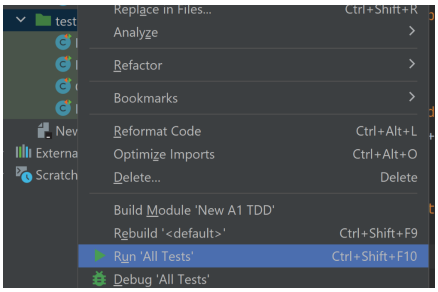
Run the tests to make sure we're green, so we can proceed with refactoring:

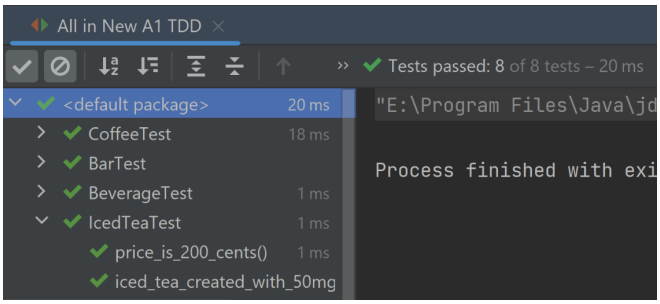


The change we need is simple:



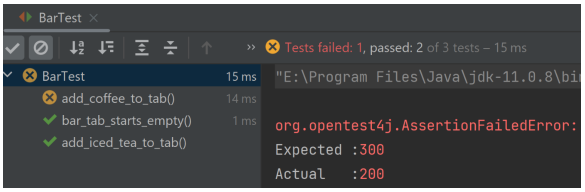
We add an abstract method definition for getPrice(). While the method has no body, it is telling Java that **every** Beverage **must** have a method called getPrice() that takes no parameters and returns an int value. This allows us to use Beverages polymorphically to call the getPrice() method.
As always when making overarching changes to classes, we should run **all** our tests to make sure everything still works correctly:





Now that we're done with our refactoring, we can go back to working on our failing test. To start, let's go back to red:

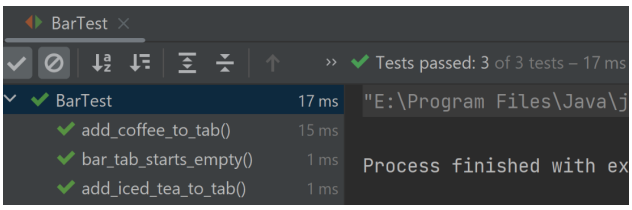
```
@Test
public void add_coffee_to_tab() {
    bar.addBeverage(new Coffee());
    int actual = bar.getTab();

    assertEquals( expected: 300, actual);
}
```



Step 1 – red – complete again! Step 2 – green – attempt 2 at success:

```
public void addBeverage(Beverage beverage) {
    tab += beverage.getPrice();
}
```

This time, we can use our beverage **polymorphically**. Voila!



Step 3 – refactor. I don't see any duplicate code yet, so we move on. Our TODO list:

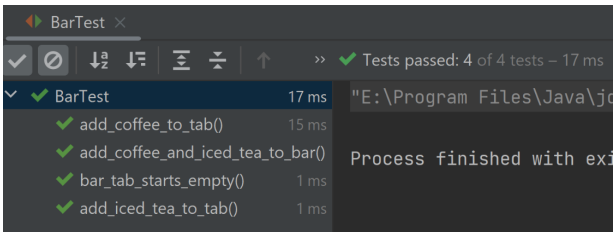
- ~~Iced tea can be created with a default value 50~~
- ~~Iced tea can be created with a specified value~~
- ~~Iced tea price is 200 cents~~
- ~~Coffee can be created with a default value 100~~
- ~~Coffee can be created with a specified value~~

- ~~Coffee price is 300 cents~~
- ~~Sugar can be added to iced tea~~
- ~~Sugar can be added to coffee~~
- ~~A bar with a tab exists~~
- Drinks can be added to the bar's tab
- ~~Price can be retrieved from the bar based on the tab~~

We have only 1 item left that we can't quite cross off yet. We have added IcedTea and made sure the cost is 200. We have added Coffee and made sure the cost is 300. But we have not added multiple drink**s** (plural is important) to our tab to make sure that functionality works:

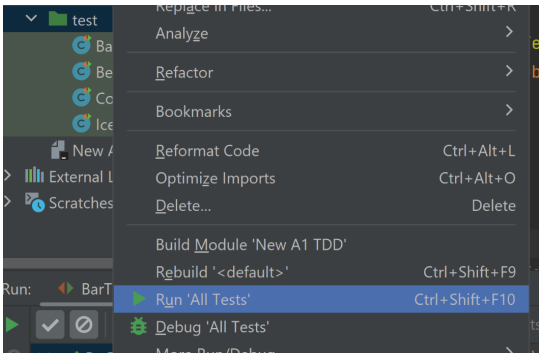
```
@Test
public void add_coffee_and_iced_tea_to_bar() {
    bar.addBeverage(new Coffee());
    bar.addBeverage(new IcedTea());
    int actual = bar.getTab();

    assertEquals( expected: 500, actual);
}
```

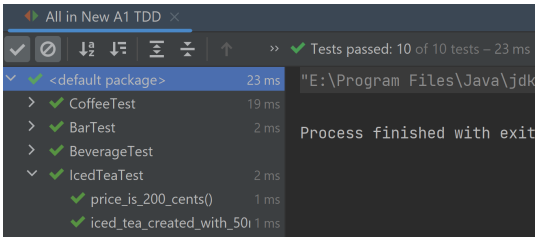
The tests all pass, with no failure this time:



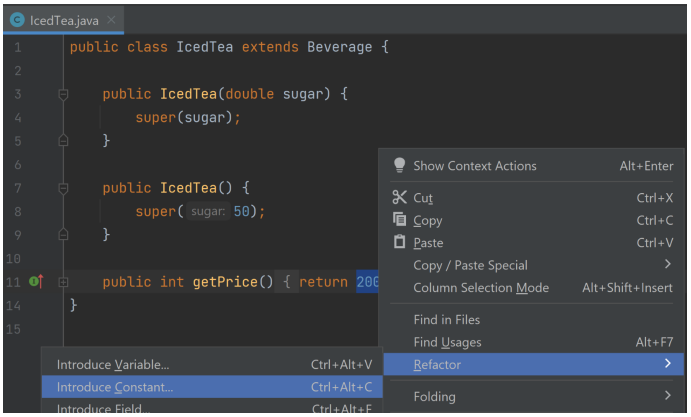
Whenever you write a test that passes immediately, a red flag should go up in your head. Did we code too much last time? If the answer to that question is yes, go backwards. Delete. Redo your work, following TDD better using smaller steps. In this case, the answer is no. We did our bare minimum amount each time. When ending a piece of software, having a test case that checks for **many** of something is a valuable final test to add. Later, any developer looking at your code will **know for sure** that functionality works exactly as expected.

Functionally, we're done. But there's still learning to be had here. Before we continue, let's make sure **all** our tests pass:





There's one piece of refactoring I have let slide this whole time. Numbers have been repeated everywhere! Think about all of them: 50, 100, 200, and 300. They are repeated in tests and code all over the place! When the price of IcedTea changes, in how many places will we have to change our code? The answer to that question **should be 1.** But it is not with our current implementation. Shame on us. Let's be better. As a matter of fact, the value 200 is used not only in IcedTea and IcedTeaTest but also in BarTest. As applications scale in complexity, that poor design decision will also scale to more and more places to change – also called **maintenance cost**. The goal of TDD and refactoring is to reduce **maintenance cost**. Let's start with the cost of IcedTea:



In IcedTea, double-click on "200" to highlight it and right click -> Refactor -> Introduce Constant. In Java, the naming convention for constants is SCREAMING_SNAKE_CASE, so we will call this value ICED_TEA_COST:

IcedTea.java

```java
public class IcedTea extends Beverage {

    public static final int ICED_TEA_COST = 200;

    public IcedTea(double sugar) {
        super(sugar);
    }

    public IcedTea() {
        super( sugar: 50);
    }

    @Override
    public int getPrice() { return ICED_TEA_COST; }
}
```

Next, we can use this value in our test in IcedTeaTest:

```java
@Test
public void price_is_200_cents() {
    int actual = icedTea.getPrice();

    assertEquals(IcedTea.ICED_TEA_COST, actual);
}
```

And in BarTest:

```java
@Test
public void add_iced_tea_to_tab() {
    bar.addBeverage(new IcedTea());
    int actual = bar.getTab();

    assertEquals(IcedTea.ICED_TEA_COST, actual);
}
```

Next up we'll do the same thing with sugar in IcedTea:

IcedTea.java

```java
public class IcedTea extends Beverage {

    public static final int ICED_TEA_COST = 200;
    public static final int ICED_TEA_DEFAULT_SUGAR = 50;

    public IcedTea(double sugar) {
        super(sugar);
    }

    public IcedTea() {
        super(ICED_TEA_DEFAULT_SUGAR);
    }

    @Override
    public int getPrice() {
        return ICED_TEA_COST;
    }
}
```

And in IcedTeaTest:

```
@Test
public void iced_tea_created_with_50mg_sugar_by_default() {
    double actual = icedTea.getSugar();

    assertEquals(IcedTea.ICED_TEA_DEFAULT_SUGAR, actual);
}
```

Next up, we do the same in Coffee:

```
public class Coffee extends Beverage {

    public static final int COFFEE_DEFAULT_SUGAR = 100;
    public static final int COFFEE_COST = 300;

    public Coffee(double sugar) {
        super(sugar);
    }

    public Coffee() {
        super(COFFEE_DEFAULT_SUGAR);
    }

    @Override
    public int getPrice() {
        return COFFEE_COST;
    }
}
```

And CoffeeTest:

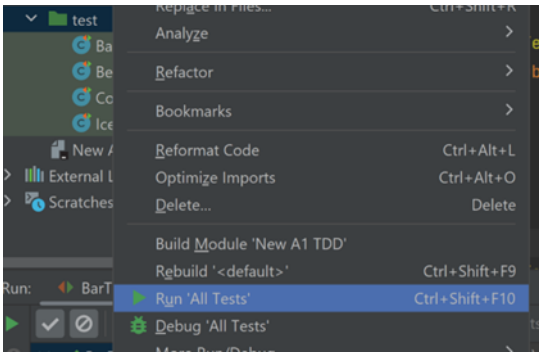
```
@Test
public void coffee_created_with_100mg_sugar_by_default() {
    double actual = coffee.getSugar();

    assertEquals(Coffee.COFFEE_DEFAULT_SUGAR, actual);
}

@Test
public void price_is_300_cents() {
    int actual = coffee.getPrice();

    assertEquals(Coffee.COFFEE_COST, actual);
}
```

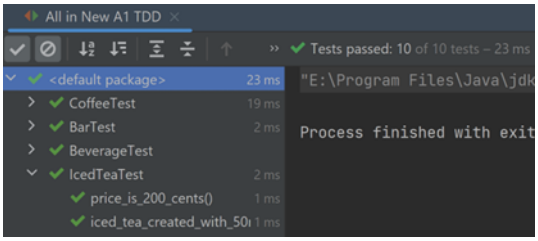
And in BarTest:

```
30      @Test
31      public void add_coffee_to_tab() {
32          bar.addBeverage(new Coffee());
33          int actual = bar.getTab();
34
35          assertEquals(Coffee.COFFEE_COST, actual);
36      }
37
38      @Test
39      public void add_coffee_and_iced_tea_to_bar() {
40          bar.addBeverage(new Coffee());
41          bar.addBeverage(new IcedTea());
42          int actual = bar.getTab();
43
44          assertEquals( expected: Coffee.COFFEE_COST + IcedTea.ICED_TEA_COST, actual);
45      }
```

On line 35, we simply use the new cost value. But on line 44 we use **both** to make sure the math adds up. With these changes, we're done our refactoring. Now when any value in the system changes, we only have to make the change in 1 place – where the constant is defined.

One last time, before we say we're done, we run **all the tests**:





It's better to create constants as you go. For example, when we first started with IcedTea, we should have immediately refactored the values 200 and 50 into constants. Don't wait. That was an intentional slip up so that I could teach you how to address this properly separately from the complexity of the TDD process.

Let's reflect on the TDD process and what we gain from it. By writing the test first, we are always making sure we take small steps – we we're not getting overwhelmed as we go. By writing just the smallest amount of code to make it pass we are granting ourselves **confidence**. As we proceed, running all our tests and watching them pass means **everything we built so far** still works. Gone are the days of running our code manually over and over to make sure we didn't break something we built long ago. By refactoring, we are ensuring that there is no duplication in our code.

A project completed using TDD is appreciated because of the **confidence** the tests provide us. Have you ever looked at someone's code and been full of fear at making even a slight change to it? Not quite sure what consequences that change might bring? Whether you might break some other functionality if you make a change here? Well, gone are those days too. With TDD, every single piece of functionality was tested before it was built, so you know for a fact that you can trust those tests after you make a change. **Confidence**.

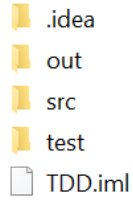
As I went through this exercise, I ran the tests at least 32 times. I did not post a screenshot after every run. When you do TDD, it's important to get in the habit of running tests **constantly.** It's a habit for me – every time I change anything at all, I run the tests. Most of the time, the result is expected. Maybe I'm on step 1 and I want to go red and I do go red. Maybe I'm on step 2 and I want to go green and I do go green. Maybe I'm on step 3 and I want to stay green during a refactor and I do it successfully. But sometimes, you will get unexpected results. You'll be **surprised**. Oh no! Wait, wait – regain composure.

Think: why is this happening? Is my assertion incorrect? Is my test case wrong? Did I code the solution incorrectly? Did I code too much last time? Did I mess up during my refactoring? Go through all the potential situations in your head, review your test and your code, and don't be afraid to **go back to green**. **Never proceed with development if you are red and don't know why.** Always go back to green, then proceed by taking smaller steps.

Submission Requirements

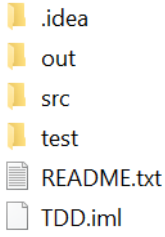
When you are done, your submission must be a **zip** (not rar or any other type) file as documented below. This will be the same process you follow for all assignment submissions.

Right click on the project -> “Show in Explorer”, or just find the directory where your project resides. You can now right click on your project folder “TDD” and create a zip using any tool of your choice. The folder structure inside should be look like this:



If you are on a Mac, the .idea folder will be hidden by default. Type “CMD + Shift + .” to show this folder in Finder.

If you have anything additional to say, please place a README.txt in that directory, parallel to TDD.iml, as such:



Zip file name should consist of your name and id (ex: A1-Boris-Valerstein-bv49.zip).

Rubric:

This assignment is testing your knowledge of TDD and we will process your submission as a ZIP file. If you do not follow TDD and/or do not submit a ZIP file, your submission will not be graded. **Straight 0, fair warning!**

Rubric Item	**Subcategory**	**Point Values**
Completion	N/A	50
	For full credit, complete everything described in the assignment. If you don't submit the tests, -30. If you don't submit the code, -10.	
Naming Conventions	N/A	10
	For full credit, all of your naming must follow all naming conventions as discussed in class. 7.5/10 if one poor naming choice is made, 5/10 if two are observed, 2.5/10 if three are observed, and 0/10 for more than 3. Examples of Java naming conventions, as taught in class: ClassName, variableName, methodName(), name_of_test(), NAME_OF_CONSTANT	
All Tests are present and pass	N/A	30
	For full credit, every valid test case from the pages above should be implemented as a JUnit test case. You will receive -5 for each test case that is missing or failing. Tests that are passing because they are missing assertions count for a -5 deduction.	
.iml file	N/A	10
	For full credit, you must remember to include your .iml file in your submission. This file must be configured to use JUnit 5, just like the instructions state. You will receive a 0/10 if you do not submit the file or if it is configured to use JUnit 4 instead of JUnit 5. This is to help the TAs spend less time grading your work, resulting in faster feedback for everyone!	