```
[[questions]]
type = "Tracing"
prompt.program = """
pub trait MakeNoise<T> {
    fn make_noise(self: @T) {
        println!("noise");
    }
}
pub mod dog {
    use super::MakeNoise;
    #[derive(Drop)]
    pub struct Dog {}
    impl Barking of MakeNoise<Dog> {
        fn make_noise(self: @Dog) {
            println!("bark");
        }
    }
}
pub mod cat {
    use super::MakeNoise;
    #[derive(Drop)]
    pub struct Cat {}
}
fn main() {
    let dog = dog::Dog {};
    dog.make_noise();
    let cat = cat::Cat {};
    cat.make_noise();
}
"""
answer.doesCompile = false
context = """
It is not possible for a trait function to have a body in Cairo.
The `make_noise` method should be declared without a body. Only the
implementations can have a body.
"""
id = "4120c3cf-a86a-4d30-b307-4a50b2e9d627"
[[questions]]
type = "MultipleChoice"
prompt.prompt = """
What line of code is correct to draw the shape `my_shape` in the `main` function ?
```
#[derive(Drop)]
struct Shape {}
trait Drawable<T> {
    fn draw(self: @T);
```
```

```
}
impl DrawableShape of Drawable<Shape> {
    fn draw(self: @Shape){
        println!("Drawing a shape!");
    }
}
fn main() {
    let my_shape = Shape{};
    // the line goes here
}
```
"""
prompt.distractors = ["`draw(my_shape);`", "`draw(@my_shape);`"]
answer.answer = "`my_shape.draw();`"
context = """
The `draw` method is a method implemented for the `Shape` type, so it should be called on the `my_shape` object.
It is also possible to call the method on the snapshot of the object by using the `@` operator, for example `Drawable::draw(@my_shape)`.
"""
id = "af789385-e4e4-4405-8fb7-049b9a51bc3b"
[[questions]]
type = "Tracing"
prompt.program = """
#[derive(Drop)]
struct ProducerType {}
#[derive(Drop, Debug)]
struct AnotherType {}
#[derive(Drop, Debug)]
struct AThirdType {}
trait Producer<T> {
    fn produce(self: T) -> u32;
}
trait Consumer<T> {
    fn consume(self: T, input: u32);
}
impl ProducerImpl of Producer<ProducerType> {
    fn produce(self: ProducerType) -> u32 {
        42
    }
}
impl TConsumerImpl<T, +core::fmt::Debug<T>, +Drop<T>, -Producer<T>> of Consumer<T> {
    fn consume(self: T, input: u32) {
        println!("{:?} consumed value: {}", self, input);
    }
```
```

```
}
impl ThirdConsumerImpl of Consumer<AThirdType> {
    fn consume(self: AThirdType, input: u32) {
        println!("{:?} consumed value: {}", self, input);
    }
}
fn main() {
    let producer = ProducerType {};
    let another_type = AnotherType {};
    let third_type = AThirdType {};
    let production = producer.produce();
    another_type.consume(production);
    third_type.consume(production);
}
"""
answer.doesCompile = false
context = """
The code doesn't compile because there are two implementations of the trait
`Consumer<T>` for the type `AThirdType`.
The implementation `TConsumerImpl` is a negative implementation of the trait
`Producer<T>`, implementing the trait `Consumer<T>` for all types `T` that don't
implement the trait `Producer<T>`.
The implementation `ThirdConsumerImpl` is thus redundant.
"""
id = "26b22cf1-9a7f-4a2d-98f3-34dab7a6dfcd"
```