Exercise 1

```rust
pub trait Printable {
    fn println(&self);
}

impl Printable for i32 {
    fn println(&self) {
        println!("{}", self);
    }
}

impl Printable for String {
    fn println(&self) {
        println!("{}", self);
    }
}

impl<T> Printable for Vec<T>
where
    T: Printable,
{
    fn println(&self) {
        for el in self {
            el.println();
        }
    }
}

pub fn print(t: impl Printable) {
    t.println();
}

pub fn print_dyn(t: &dyn Printable) {
    t.println();
}
```

Exercise 2

```rust
use rand::RngCore;

//                         vvvvv this is for testing purposes
#[derive(Debug, Default, PartialEq)]
enum Category {
    Fantasy,
    Horror,
    SciFi,
    Romance,
    Thriller,
    Historical,
    Comedy,
    Drama,
    #[default]
    Poetry,
    Other,
}

#[derive(Debug)]
struct Book {
    title: String,
    cat: Category,
}

impl Default for Book {
    fn default() -> Self {
        let mut title = String::new();
        for _ in 0..10 {
            title.push((rand::thread_rng().next_u32() % 26 +
('a' as u32)) as u8 as char);
        }

        Book {
            title,
            cat: Category::default(),
        }
    }
}

#[derive(Debug, Default)]
struct Library {
    bookcases: [Vec<Book>; 10],
}

impl Book {
    fn default_with_cat(cat: Category) -> Self {
        Book {
            cat,
            ..Self::default()
        }
    }
}

trait Populatable {
    fn populate(&mut self);
}

impl Populatable for Library {
    fn populate(&mut self) {
        for bookcase in self.bookcases.iter_mut() {
            for _ in 0..3 {
                bookcase.push(Book::default());
            }
        }
    }
}
```

Exercise 3

```rust
use std::fmt::{Debug, Display};

pub fn restricted<T, U>(t1: T, t2: T, u: U) -> impl Debug +
PartialOrd + Ord
where
    T: Debug + PartialOrd + Ord,
    U: Display,
{
    let minor = if t1 < t2 { t1 } else { t2 };

    println!("minor: {:?}", minor);
    println!("u: {}", u);
    minor
}
```

Exercise 4

```
#[derive(Debug, Default)]
struct Tasks {
    tasks: Vec<Task>,
}

impl Tasks {
    pub fn new(v: &[Task]) -> Self {
        Self { tasks: v.to_vec() }
    }
}

#[derive(Clone, Debug, Default)]
struct Task {
    name: String,
    priority: i32,
    done: bool,
}

impl Task {
    pub fn new(name: &str, priority: i32, done: bool) -> Self {
        Self {
            name: name.to_string(),
            priority,
            done,
        }
    }
}

impl Iterator for Tasks {
    type Item = Task;

    fn next(&mut self) -> Option<Self::Item> {
        self.tasks
            .iter()
            .position(|t| !t.done)
            .map(|i| self.tasks.remove(i))
    }
}
```

Exercise 5

```
use std::ops::{Add,Sub,Mul};

#[derive(Clone, PartialEq, Debug)]
struct Pair(i32, String);

impl Add<i32> for Pair {
    type Output = Pair;

    fn add(self, rhs: i32) -> Self::Output {
        Pair(self.0 + rhs, self.1)
    }
}

impl Add<&str> for Pair {
    type Output = Pair;

    fn add(self, rhs: &str) -> Self::Output {
        Pair(self.0, self.1 + rhs)
    }
}

impl Add<Pair> for Pair {
    type Output = Pair;

    fn add(self, rhs: Pair) -> Self::Output {
        self + rhs.0 + rhs.1.as_str()
    }
}

impl Sub<i32> for Pair {
    type Output = Pair;

    fn sub(self, rhs: i32) -> Self::Output {
        Pair(self.0 - rhs, self.1)

    }
```

```rust
}

impl Sub<&str> for Pair {
    type Output = Pair;

    fn sub(self, rhs: &str) -> Self::Output {
        Pair(self.0, self.1.replace(rhs, ""))

    }
}

impl Sub<Pair> for Pair {
    type Output = Pair;

    fn sub(self, rhs: Pair) -> Self::Output {
        self - rhs.0 - rhs.1.as_str()
    }
}

impl Mul<i32> for Pair {
    type Output = Pair;

    fn mul(self, rhs: i32) -> Self::Output {
        Pair(self.0.pow(rhs as u32), self.1.repeat(rhs as
usize))
    }
}
```

Exercise 6

```rust
use rand::{self, Rng};

#[derive(Debug)]
struct Open;
#[derive(Debug)]
struct Closed;
#[derive(Debug)]
struct Stopped {
    _reason: String,
}
```

```rust
#[derive(Debug)]
struct Gate<S> {
    _state: S,
}

impl Gate<Open> {
    pub fn new() -> Gate<Open> {
        Gate { _state: Open }
    }

    pub fn close(self) -> Result<Gate<Closed>, Gate<Stopped>> {
        let r = rand::thread_rng().gen_range(0..20);
        match r {
            0..=12 => Ok(Gate { _state: Closed }),
            13..=15 => Err(Gate {
                _state: Stopped {
                    _reason: "Motor error".to_string(),
                },
            }),
            _ => Err(Gate {
                _state: Stopped {
                    _reason: "Photocell detected an
object".to_string(),
                },
            }),
        }
    }
}

impl Gate<Stopped> {
    pub fn new(reason: &str) -> Gate<Stopped> {
        Gate {
            _state: Stopped {
                _reason: reason.to_string(),
            },
        }
    }

    pub fn open(self) -> Gate<Open> {
        Gate { _state: Open }
    }
    pub fn close(self) -> Gate<Closed> {
        Gate { _state: Closed }
    }
}

impl Gate<Closed> {
    pub fn new() -> Gate<Closed> {
        Gate { _state: Closed }
    }

    pub fn open(self) -> Result<Gate<Open>, Gate<Stopped>> {
        let r = rand::thread_rng().gen_range(0..10);
        match r {
            0..=12 => Ok(Gate { _state: Open }),
            13..=15 => Err(Gate::<Stopped>::new("Motor error")),
            _ => Err(Gate::<Stopped>::new("Photocell detected an object")),
        }
    }
}
```

Exercise 7

```
trait Heater {
    fn heat(&self, t: &mut dyn Heatable);
}

trait Frier {
    fn fry(&self, t: &mut dyn Friable);
}

trait Friable {
    fn cook(&mut self);
}

trait Heatable {
    fn cook(&mut self);
}
```

```rust
struct Oven;
impl Heater for Oven {
    fn heat(&self, t: &mut dyn Heatable) {
        t.cook()
    }
}


struct Pan;
impl Heater for Pan {
    fn heat(&self, t: &mut dyn Heatable) {
        t.cook()
    }
}
impl Frier for Pan {
    fn fry(&self, t: &mut dyn Friable) {
        t.cook()
    }
}


trait Edible {
    fn eat(&self);
}

struct Pie {
    ready: bool
}
impl Heatable for Pie {
    fn cook(&mut self) {
        if self.ready {
            println!("you burnt the pie!")
        } else {
            self.ready = true;
        }
    }
}
impl Edible for Pie {
    fn eat(&self) {
        if self.ready {
            println!("yummy!");
        } else {
            println!("you got stomach ache")
        }
    }
}

#[derive(PartialEq)]
enum CarrotState {
    Raw,
    Cooked,
    Fried,
    Burnt
}
struct Carrot {
    state: CarrotState,
}
impl Heatable for Carrot {
    fn cook(&mut self) {
        if self.state != CarrotState::Raw {
            self.state = CarrotState::Burnt;
        } else {
            self.state = CarrotState::Cooked;
        }
    }
}
impl Friable for Carrot {
    fn cook(&mut self) {
        if self.state == CarrotState::Fried {
            self.state = CarrotState::Burnt;
        } else {
            self.state = CarrotState::Fried
        }
    }
}
impl Edible for Carrot {
    fn eat(&self) {
        match self.state {
            CarrotState::Raw => println!("mmh, crunchy"),
            CarrotState::Cooked => println!("mmh, yummy"),
            CarrotState::Fried => println!("mmh, crispy"),
            CarrotState::Burnt => println!("mmh, burnt")
```

```
                }
            }
    }
```