Enums (A1)

Aufgabe: Steuerung eines autonomen Fahrzeugs, Programmierung der Befehle (Action). Der Befehl soll verschiedene Ausprägungen haben können: Move (Koordinate -> x, y, benannt), GoTo (Ort, Name), Stop, Wait (Dauer). Zusätzlich soll an jedem Befehl die Funktion abort aufgerufen werden können, der ein Abbruch-Code übergeben wird. Dieser Abbruch-Code ist vom Typ i8, kann aber auch mal keinen Wert haben. In dieser Funktion wird der übergebene Code auf den Basis-Code base_code = 100 vom Typ i8 addiert. Zum Abschluss soll jeder der Befehle einmal ausgeführt werden (Koordinate 49, 8; Ziel Home, Warten 5) und der Befehl Wait abgebrochen werden (Code 505). Zur Umsetzung soll das Wissen über Enums genutzt werden.

```
enum Action {
	Move { x: i32, y: i32 },
	GoTo(String),
	Stop,
	Wait(i32),
}

impl Action {
	fn abort(&self, code: Option<i16>) {
		let base_code: i16 = 100;

		if code != None {
			let result_code = base_code + code.unwrap();
			println!("{}", result_code);
		}
	}
}

fn main() {
	let a1 = Action::Move{ x: 1, y: 2 };
	let a2 = Action::GoTo("Home".to_string());
	let a3 = Action::Stop;
	let a4 = Action::Wait(5);
	a4.abort(Some(505));
}
```

Match (A2)

Erweitert Action aus Aufgabe 1 um Fly, dem ebenfalls x- und y-Koordinate übergeben werden soll und Abort. Schreibt eine Funktion, der eine Action übergeben wird. Bei Move soll die Distanz der Zielposition zur aktuellen Position (13, 42) berechnet und ausgegeben werden, bei GoTo das Ziel, bei Wait “Waiting for x seconds…” bei Stop nichts, Abort nichts und bei Fly ebenfalls die Distanz zur Zielposition. Es dürfen nur 5 Match-Arme verwendet werden.

```
enum Action {
	Move { x: i32, y: i32 },
	GoTo(String),
	Stop,
	Wait(i32),
	Fly { x: i32, y: i32 },
	Abort,
}

fn execute_action(action: Action) {
	match action {
	Action::GoTo(target) => println!("{}", target),
	Action::Wait(duration) => println!("Waiting for {} seconds…", duration),
	Action::Move {x, y} => println!("{}", calculate_distance(x, y, 13, 42)),
	Action::Fly {x, y} => println!("{}", calculate_distance(x, y, 13, 42)),
	_ => (),
	}
}

fn  calculate_distance(x1: i32, y1: i32, x2: i32, y2: i32) -> f64 {
	(((x1 - x2) as f64).powi(2) + ((y1 - y2) as f64).powi(2)).sqrt()
}

fn main() {
	execute_action(Action::GoTo("Target location".to_string()));
	execute_action(Action::Wait(123));
	execute_action(Action::Stop);
	execute_action(Action::Move{ x: 15, y: 19 });
	execute_action(Action::Fly{ x: 302, y: -127 });
	execute_action(Action::Abort);
}
```

Pattern und Matching (A3)

A 3.1:

Erstellt ein Match, das eine Zahl matcht und bei den Primzahlen bis 10 “Primzahl!”, bei Zahlen von 11-99 “Unter 100”, bei Zahlen von 107-121 “Diese Reihe von Zahlen ist willkürlich ausgewählt” und bei allen anderen Zahlen nichts ausgibt.

```
let x = 5;

match x {
        2 | 3 | 5 | 7 => println!(“Primzahl!”),
        11..=99 => println!("Unter 100"),
        107..=121 => println!("Diese Reihe von Zahlen ist willkürlich ausgewählt"),
        _ => (),
}
```

A 3.2:

Erstellt ein Match, das ein struct folgender Struktur matcht:

```
struct Word {
        a: char,
        b: char,
        c: char,
}
```

Wenn bei einer der Variablen “x” steht, soll “passt!” ausgegeben werden, sonst nichts.

```
let word = Word { a: 'z', b: 'b', c: 'x' };

match word {
        Word { a: 'x', b, c } => println!("passt!"),
        Word { a, b: 'x', c } => println!("passt!"),
        Word { a, b, c: 'x' } => println!("passt!"),
        _ => (),
}
```

A 3.3:

Erweiterung: Ist die Variable same: bool auf true gesetzt, so soll auch wenn bei a der Wert “a”, bei b der Wert “b” oder bei c der Wert “c” steht “passt!” ausgegeben werden.

```
let same: bool = true;

        match word {
        Word { a: 'x', b, c } => println!("passt!"),
        Word { a, b: 'x', c } => println!("passt!"),
```

```
        Word { a, b, c: 'x' } => println!("passt!"),
        Word { a: 'a', b, c } if same => println!("passt!"),
        Word { a, b: 'b', c } if same => println!("passt!"),
        Word { a, b, c: 'c' } if same => println!("passt!"),
        _ => (),
}
```

Advanced Types (A4)

A 4.1:

Ihr habt den Typ “Bomb” mit den Funktionen move_to(destination: String), throw(), arm(), drop(), kick(). Da ihr die Bombe ein paar Kindern als Ersatz für ihren verlorenen Ball zum Spielen geben wollt, jedoch nicht dafür verantwortlich sein wollt, dass die Kinder sich in die Luft sprengen, sollen sie die Bombe nicht scharf machen können. Nutzt deshalb das Newtype-Pattern, um eine “SaveBomb” zu erstellen. Diese kann alles, was auch Bomb kann, jedoch kann die Funktion arm() nicht aufgerufen werden.

```
struct Bomb(String);

impl Bomb {
	fn move_to(&self, destination: String) {
	//
	}

	fn throw(&self) {
	//
	}

	fn arm(&self) {
	//
	}

	fn drop(&self) {
	//
	}

	fn kick(&self) {
	//
	}
}

struct SaveBomb(Bomb);

impl SaveBomb {
	fn move_to(&self, destination: String) {
	self.0.move_to(destination);
	}

	fn throw(&self) {
	self.0.throw();
	}

	fn drop(&self) {
	self.0.drop();
```

```
        }

        fn kick(&self) {
        self.0.kick();
        }
}
```

A 4.2:

Erschafft einen Typ-Synonym “ErrorCode” vom Typ i8. Erstellt ein Match, das nach dem Parsen eines strings in den neuen Typ “ErrorCode” überprüft, ob es funktioniert hat. Konnte der Wert nicht geparst werden, so soll "Error while parsing type ‘ErrorCode’" ausgegeben werden.

```
type ErrorCode = i8;

let error: &str = “404”;

let errorCode: ErrorCode = match error.parse() {
   Ok(code) => code,
   Err(_)=> panic!("Error while parsing type ‘ErrorCode’"),
}
```