DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 21

0. ANY QUESTIONS ABOUT THE FINAL PROJECT PROPOSAL?

1. USEFUL PREDEFINED GENERIC DATA TYPES

2. TRAITS

THE FINAL PROJECT PROPOSAL

ANY QUESTIONS?

1. USEFUL PREDEFINED GENERIC DATA TYPES

2. TRAITS

LAST TIME: GENERICS AND GENERIC DATA TYPES

- Generic code
- Method for avoiding copying code
- No runtime penalty: different versions created during compilation

LAST TIME: GENERICS AND GENERIC DATA TYPES

- Generic code
- Method for avoiding copying code
- No runtime penalty: different versions created during compilation

Generic data types:

- Data types (struct/enum) parameterized by types

Two useful predifined types: `Option<T>` and `Result<T, E>`

ENUM `Option<T>`

`Some(T)` or `None`

- Useful for when there may be no output
- Compared to `None` or `null` in other programming languages:
    - Rust forces handling of this case

Presentation: "Null References: The Billion Dollar Mistake"

Track: Historically bad ideas

Time: Friday 13:00 - 14:00

Location: Abbey Room

Abstract: I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. In recent years, a number of program analysers like PREfix and PREfast in Microsoft have been used to check references, and give warnings if there is a risk they may be non-null. More recent programming languages like Spec# have introduced declarations for non-null references. This is the solution, which I rejected in 1965.

Tony Hoare, Inventor of QuickSort, Turing Award Winner

Sir Charles Antony Richard Hoare (Tony Hoare or C.A.R. Hoare, born January 11, 1934) is a British computer scientist, probably best known for the development in 1960 of Quicksort (or Hoaresort), one of the world's most widely used sorting algorithms.

He also developed Hoare logic for verifying program correctness, and the formal language Communicating Sequential Processes (CSP) used to specify the interactions of concurrent processes (including the Dining philosophers problem) and the inspiration for the Occam programming language.

ENUM `Option<T>`

`Some(T)` or `None`

- Useful for when there may be no output
- Compared to `None` or `null` in other programming languages:
    - Rust forces handling of this case

Presentation: "Null References: The Billion Dollar Mistake"

Track: Historically bad ideas

Time: Friday 13:00 - 14:00

Location: Abbey Room

Abstract: I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. In recent years, a number of program analysers like PREfix and PREfast in Microsoft have been used to check references, and give warnings if there is a risk they may be non-null. More recent programming languages like Spec# have introduced declarations for non-null references. This is the solution, which I rejected in 1965.

Tony Hoare, Inventor of QuickSort, Turing Award Winner

Sir Charles Antony Richard Hoare (Tony Hoare or C.A.R. Hoare, born January 11, 1934) is a British computer scientist, probably best known for the development in 1960 of Quicksort (or Hoaresort), one of the world's most widely used sorting algorithms.

He also developed Hoare logic for verifying program correctness, and the formal language Communicating Sequential Processes (CSP) used to specify the interactions of concurrent processes (including the Dining philosophers problem) and the inspiration for the Occam programming language.

In [2]:
```rust
fn prime(x:u32) -> bool {
    for i in 2..x {
        if x % i == 0 {
            return false;
        }
    }
    if x <= 1 {false} else {true}
}

fn prime_in_range(a:u32,b:u32) -> Option<u32> {
    for i in a..=b {
        if prime(i) {return Some(i);}
    }
    None
}
```

ENUM `Option<T>`

`Some(T)` or `None`

- Useful for when there may be no output
- Compared to `None` or `null` in other programming languages:
  - Rust forces handling of this case

Presentation: "Null References: The Billion Dollar Mistake"

Track: Historically bad ideas

Time: Friday 13:00 - 14:00

Location: Abbey Room

Abstract: I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. In recent years, a number of program analysers like PREfix and PREfast in Microsoft have been used to check references, and give warnings if there is a risk they may be non-null. More recent programming languages like Spec# have introduced declarations for non-null references. This is the solution, which I rejected in 1965.

Tony Hoare, Inventor of QuickSort, Turing Award Winner

Sir Charles Antony Richard Hoare (Tony Hoare or C.A.R. Hoare, born January 11, 1934) is a British computer scientist, probably best known for the development in 1960 of Quicksort (or Hoaresort), one of the world's most widely used sorting algorithms.

He also developed Hoare logic for verifying program correctness, and the formal language Communicating Sequential Processes (CSP) used to specify the interactions of concurrent processes (including the Dining philosophers problem) and the inspiration for the Occam programming language.

```
In [2]: fn prime(x:u32) -> bool {
            for i in 2..x {
                if x % i == 0 {
                    return false;
                }
            }
            if x <= 1 {false} else {true}
        }

        fn prime_in_range(a:u32,b:u32) -> Option<u32> {
            for i in a..=b {
                if prime(i) {return Some(i);}
            }
            None
        }
```

```
In [3]: prime_in_range(888,906)

Out[3]: None
```

ENUM `Option<T>`

`Some(T)` or `None`

- Useful for when there may be no output
- Compared to `None` or `null` in other programming languages:
  - Rust forces handling of this case

Presentation: "Null References: The Billion Dollar Mistake"

Track: Historically bad ideas

Time: Friday 13:00 - 14:00

Location: Abbey Room

Abstract: I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. In recent years, a number of program analysers like PREfix and PREfast in Microsoft have been used to check references, and give warnings if there is a risk they may be non-null. More recent programming languages like Spec# have introduced declarations for non-null references. This is the solution, which I rejected in 1965.

Tony Hoare, Inventor of QuickSort, Turing Award Winner

Sir Charles Antony Richard Hoare (Tony Hoare or C.A.R. Hoare, born January 11, 1934) is a British computer scientist, probably best known for the development in 1960 of Quicksort (or Hoaresort), one of the world's most widely used sorting algorithms.

He also developed Hoare logic for verifying program correctness, and the formal language Communicating Sequential Processes (CSP) used to specify the interactions of concurrent processes (including the Dining philosophers problem) and the inspiration for the Occam programming language.

In [2]:
```
fn prime(x:u32) -> bool {
    for i in 2..x {
        if x % i == 0 {
            return false;
        }
    }
    if x <= 1 {false} else {true}
}

fn prime_in_range(a:u32,b:u32) -> Option<u32> {
    for i in a..=b {
        if prime(i) {return Some(i);}
    }
    None
}
```

In [3]:
```
prime_in_range(888,906)
```

Out[3]: None

In [4]:
```
let tmp : Option<u32> = prime_in_range(830,856);
tmp
```

Out[4]: Some(839)

ENUM `Option<T>`

`Some(T)` or `None`

- Useful for when there may be no output
- Compared to `None` or `null` in other programming languages:
  - Rust forces handling of this case

Presentation: "Null References: The Billion Dollar Mistake"

Track: Historically bad ideas

Time: Friday 13:00 - 14:00

Location: Abbey Room

Abstract: I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. In recent years, a number of program analysers like PREfix and PREfast in Microsoft have been used to check references, and give warnings if there is a risk they may be non-null. More recent programming languages like Spec# have introduced declarations for non-null references. This is the solution, which I rejected in 1965.

Tony Hoare, Inventor of QuickSort, Turing Award Winner

Sir Charles Antony Richard Hoare (Tony Hoare or C.A.R. Hoare, born January 11, 1934) is a British computer scientist, probably best known for the development in 1960 of Quicksort (or Hoaresort), one of the world's most widely used sorting algorithms.

He also developed Hoare logic for verifying program correctness, and the formal language Communicating Sequential Processes (CSP) used to specify the interactions of concurrent processes (including the Dining philosophers problem) and the inspiration for the Occam programming language.

In [2]:
```
fn prime(x:u32) -> bool {
    for i in 2..x {
        if x % i == 0 {
            return false;
        }
    }
    if x <= 1 {false} else {true}
}

fn prime_in_range(a:u32,b:u32) -> Option<u32> {
    for i in a..=b {
        if prime(i) {return Some(i);}
    }
    None
}
```

In [3]:
```
prime_in_range(888,906)
```

Out[3]: None

In [4]:
```
let tmp : Option<u32> = prime_in_range(830,856);
tmp
```

Out[4]: Some(839)

In [5]:
```
// extracting the content of Some(...)
if let Some(x) = tmp {
    println!("Some({})",x);
}
match tmp {
    Some(x) => println!("Some({})",x),
    None => println!("None"),
};
```

```
Some(839)
Some(839)
```

INTERESTING RELATED FACT: BERTRAND'S POSTULATE

THERE IS ALWAYS A PRIME NUMBER IN $[k, 2k]$.

ENUM `Option<T>`: USEFUL METHODS

Check the variant

- `.is_some() -> bool`
- `.is_none() -> bool`

Get the value in `Some` or terminate with an error

- `.unwrap() -> T`
- `.expect(message) -> T`

Get the value in `Some` or a default value

- `.unwrap_or(default_value:T) -> T`

ENUM `Option<T>`: USEFUL METHODS

Check the variant

- `.is_some() -> bool`
- `.is_none() -> bool`

```
In [6]: let x = Some(3);
        x.is_none()

Out[6]: false
```

Get the value in `Some` or terminate with an error

- `.unwrap() -> T`
- `.expect(message) -> T`

Get the value in `Some` or a default value

- `.unwrap_or(default_value:T) -> T`

ENUM `Option<T>`: USEFUL METHODS

Check the variant

- `.is_some() -> bool`
- `.is_none() -> bool`

Get the value in `Some` or terminate with an error

- `.unwrap() -> T`
- `.expect(message) -> T`

Get the value in `Some` or a default value

- `.unwrap_or(default_value:T) -> T`

```
In [7]: let x = Some(3);
        x.is_some()

Out[7]: true
```

ENUM `Option<T>`: USEFUL METHODS

Check the variant

- `.is_some() -> bool`
- `.is_none() -> bool`

Get the value in `Some` or terminate with an error

- `.unwrap() -> T`
- `.expect(message) -> T`

Get the value in `Some` or a default value

- `.unwrap_or(default_value:T) -> T`

In [7]:
```
let x = Some(3);
x.is_some()
```

Out[7]: true

In [8]:
```
//let x = Some(3);
let x = None;
let y = x.expect("This should have been an integer");
y
```

```
thread '<unnamed>' panicked at 'This should have been a
n integer', src/lib.rs:126:11
stack backtrace:
   0: rust_begin_unwind
             at /rustc/9d1b2106e23b1abd32fce1f17267604a
5102f57a/library/std/src/panicking.rs:498:5
   1: core::panicking::panic_fmt
             at /rustc/9d1b2106e23b1abd32fce1f17267604a
5102f57a/library/core/src/panicking.rs:116:14
   2: core::panicking::panic_display
             at /rustc/9d1b2106e23b1abd32fce1f17267604a
5102f57a/library/core/src/panicking.rs:72:5
   3: core::panicking::panic_str
             at /rustc/9d1b2106e23b1abd32fce1f17267604a
5102f57a/library/core/src/panicking.rs:56:5
   4: core::option::expect_failed
             at /rustc/9d1b2106e23b1abd32fce1f17267604a
5102f57a/library/core/src/option.rs:1817:5
   5: run_user_code_7
   6: evcxr::runtime::Runtime::run_loop
   7: evcxr::runtime::runtime_hook
   8: evcxr_jupyter::main
note: Some details are omitted, run with `RUST_BACKTRAC
E=full` for a verbose backtrace.
Segmentation fault.
```

ENUM `Option<T>`: USEFUL METHODS

Check the variant

- `.is_some() -> bool`
- `.is_none() -> bool`

Get the value in `Some` or terminate with an error

- `.unwrap() -> T`
- `.expect(message) -> T`

Get the value in `Some` or a default value

- `.unwrap_or(default_value:T) -> T`

```
In [7]: let x = Some(3);
        x.is_some()

Out[7]: true
```

```
In [9]: let x = Some(3);
        //let x = None;
        let y = x.expect("This should have been an integer");
        y

Out[9]: 3
```

ENUM `Option<T>`: USEFUL METHODS

Check the variant

- `.is_some() -> bool`
- `.is_none() -> bool`

Get the value in `Some` or terminate with an error

- `.unwrap() -> T`
- `.expect(message) -> T`

Get the value in `Some` or a default value

- `.unwrap_or(default_value:T) -> T`

```
In [7]:
let x = Some(3);
x.is_some()

Out[7]: true
```

```
In [9]:
let x = Some(3);
//let x = None;
let y = x.expect("This should have been an integer");
y

Out[9]: 3
```

```
In [13]:
let x = Some(3);
//let x = None;
x.unwrap_or(0)

Out[13]: 3
```

ENUM `Option<T>`: USEFUL METHODS

Check the variant

- `.is_some() -> bool`
- `.is_none() -> bool`

Get the value in `Some` or terminate with an error

- `.unwrap() -> T`
- `.expect(message) -> T`

Get the value in `Some` or a default value

- `.unwrap_or(default_value:T) -> T`

```
In [7]:
let x = Some(3);
x.is_some()

Out[7]: true
```

```
In [9]:
let x = Some(3);
//let x = None;
let y = x.expect("This should have been an integer");
y

Out[9]: 3
```

```
In [14]:
//let x = Some(3);
let x = None;
x.unwrap_or(0)

Out[14]: 0
```

ENUM `Option<T>`: USEFUL METHODS

Check the variant

- `.is_some() -> bool`
- `.is_none() -> bool`

Get the value in `Some` or terminate with an error

- `.unwrap() -> T`
- `.expect(message) -> T`

Get the value in `Some` or a default value

- `.unwrap_or(default_value:T) -> T`

```
In [7]:
let x = Some(3);
x.is_some()

Out[7]: true
```

```
In [9]:
let x = Some(3);
//let x = None;
let y = x.expect("This should have been an integer");
y

Out[9]: 3
```

```
In [14]:
//let x = Some(3);
let x = None;
x.unwrap_or(0)

Out[14]: 0
```

More details:

- https://doc.rust-lang.org/std/option/
- https://doc.rust-lang.org/std/option/enum.Option.html

ENUM `Result<T, E>`

`Ok(T)` or `Err(E)`

- Useful when you want to pass a solution or information about an error

ENUM `Result<T, E>`

`Ok(T)` or `Err(E)`

- Useful when you want to pass a solution or information about an error

```
In [15]: fn divide(a:u32,b:u32) -> Result<u32,String> {
             match b {
                 0 => Err(String::from("Division by zero")),
                 _ => Ok(a / b)
             }
         }
```

ENUM `Result<T, E>`

`Ok(T)` or `Err(E)`

- Useful when you want to pass a solution or information about an error

```
In [15]: fn divide(a:u32,b:u32) -> Result<u32,String> {
             match b {
                 0 => Err(String::from("Division by zero")),
                 _ => Ok(a / b)
             }
         }
```

```
In [16]: divide(3,0)

Out[16]: Err("Division by zero")
```

ENUM `Result<T, E>`

`Ok(T)` or `Err(E)`

- Useful when you want to pass a solution or information about an error

```
In [15]: fn divide(a:u32,b:u32) -> Result<u32,String> {
             match b {
                 0 => Err(String::from("Division by zero")),
                 _ => Ok(a / b)
             }
         }

In [16]: divide(3,0)

Out[16]: Err("Division by zero")

In [17]: divide(2022,3)

Out[17]: Ok(674)
```

ENUM `Result<T, E>`: USEFUL METHODS

Check the variant

- `.is_ok() -> bool`
- `.is_err() -> bool`

Get the value in `Ok` or terminate with an error

- `.unwrap() -> T`
- `.expect(message) -> T`

Get the value in `Ok` or a default value

- `.unwrap_or(default_value:T) -> T`

```
In [18]: let r1 : Result<i32,()> = Ok(3);
         //r1.is_ok()
         r1.is_err()

Out[18]: false
```

ENUM `Result<T, E>`: USEFUL METHODS

Check the variant

- `.is_ok() -> bool`
- `.is_err() -> bool`

Get the value in `Ok` or terminate with an error

- `.unwrap() -> T`
- `.expect(message) -> T`

Get the value in `Ok` or a default value

- `.unwrap_or(default_value:T) -> T`

```
In [20]: let r1 : Result<i32,()> = Ok(3);
         //r1.is_err()
         r1.is_ok()

Out[20]: true
```

ENUM `Result<T, E>`: USEFUL METHODS

Check the variant

- `.is_ok() -> bool`
- `.is_err() -> bool`

Get the value in `Ok` or terminate with an error

- `.unwrap() -> T`
- `.expect(message) -> T`

Get the value in `Ok` or a default value

- `.unwrap_or(default_value:T) -> T`

```
In [20]: let r1 : Result<i32,()> = Ok(3);
         //r1.is_err()
         r1.is_ok()

Out[20]: true

In [21]: r1.unwrap()

Out[21]: 3
```

ENUM `Result<T, E>`: USEFUL METHODS

Check the variant

- `.is_ok() -> bool`
- `.is_err() -> bool`

Get the value in `Ok` or terminate with an error

- `.unwrap() -> T`
- `.expect(message) -> T`

Get the value in `Ok` or a default value

- `.unwrap_or(default_value:T) -> T`

```
In [20]: let r1 : Result<i32,()> = Ok(3);
         //r1.is_err()
         r1.is_ok()

Out[20]: true

In [21]: r1.unwrap()

Out[21]: 3

In [22]: let r2 : Result<u32,()> = Err(());
         let r3 : Result<u32,()> = Ok(123);
         println!("r2: {}\nr3: {}",
             r2.unwrap_or(0),
             r3.unwrap_or(0));

         r2: 0
         r3: 123
```

ENUM `Result<T, E>`: USEFUL METHODS

Check the variant

- `.is_ok() -> bool`
- `.is_err() -> bool`

Get the value in `Ok` or terminate with an error

- `.unwrap() -> T`
- `.expect(message) -> T`

Get the value in `Ok` or a default value

- `.unwrap_or(default_value:T) -> T`

```
In [20]: let r1 : Result<i32,()> = Ok(3);
         //r1.is_err()
         r1.is_ok()

Out[20]: true

In [21]: r1.unwrap()

Out[21]: 3

In [22]: let r2 : Result<u32,()> = Err(());
         let r3 : Result<u32,()> = Ok(123);
         println!("r2: {}\nr3: {}",
             r2.unwrap_or(0),
             r3.unwrap_or(0));

         r2: 0
         r3: 123
```

More details:

- https://doc.rust-lang.org/std/result/
- https://doc.rust-lang.org/std/result/enum.Result.html

1. USEFUL PREDEFINED GENERIC DATA TYPES

2. TRAITS

TRAITS

- Common behavior for a set of types
- Some other programming languages: interface

TRAITS

- Common behavior for a set of types
- Some other programming languages: interface

SAMPLE TRAIT DEFINITION

In [23]:
```
trait Person {
    // method header specifications
    fn get_name(&self) -> String;
    fn get_age(&self) -> u32;

    // default implementation of a method
    fn description(&self) -> String {
        format!("{} ({})",self.get_name(),self.get_age())
    }
}
```

SAMPLE TRAIT IMPLEMENTATION 1

In [24]:

```
struct SoccerPlayer {
    name: String,
    age: u32,
    team: String,
}

impl Person for SoccerPlayer {
    fn get_age(&self) -> u32 {
        self.age
    }

    fn get_name(&self) -> String {
        self.name.clone()
    }
}

impl SoccerPlayer {
    fn create(name:String,age:u32,team:String) -> SoccerPlayer{
        SoccerPlayer{name,age,team}
    }
}
```

SAMPLE TRAIT IMPLEMENTATION 2

In [25]:

```
#[derive(Debug)]
struct RegularPerson {
    year_born: u32,
    first_name: String,
    middle_name: String,
    last_name: String,
}

impl Person for RegularPerson {
    fn get_age(&self) -> u32 {
        2022 - self.year_born
    }

    fn get_name(&self) -> String {
        if self.middle_name == "" {
            format!("{} {}",self.first_name,self.last_name)
        } else {
            format!("{} {} {}",self.first_name,self.middle_name,self.last_name)
        }
    }
}

impl RegularPerson {
    fn create(first_name:String,middle_name:String,last_name:String,year_born:u32) -> RegularPerson {
        RegularPerson{first_name,middle_name,last_name,year_born}
    }
}
```

USING TRAITS IN FUNCTIONS

In [26]:
```rust
// sample function accepting object implementing trait
fn long_description(person: &impl Person) {
    println!("{}, who is {} old", person.get_name(), person.get_age());
}
```

USING TRAITS IN FUNCTIONS

In [26]:
```rust
// sample function accepting object implementing trait
fn long_description(person: &impl Person) {
    println!("{}, who is {} old", person.get_name(), person.get_age());
}
```

EXAMPLES

In [27]:
```rust
let mlk = RegularPerson::create(
    String::from("Martin"),
    String::from("Luther"),
    String::from("King"),
    1929
);

let zlatan = SoccerPlayer::create(String::from("Zlatan Ibrahimovic"), 40, String::from("AC Milan"));
```

USING TRAITS IN FUNCTIONS

In [26]:
```
// sample function accepting object implementing trait
fn long_description(person: &impl Person) {
    println!("{}, who is {} old", person.get_name(), person.get_age());
}
```

EXAMPLES

In [27]:
```
let mlk = RegularPerson::create(
    String::from("Martin"),
    String::from("Luther"),
    String::from("King"),
    1929
);

let zlatan = SoccerPlayer::create(String::from("Zlatan Ibrahimovic"), 40, String::from("AC Milan"));
```

In [28]:
```
println!("{}",mlk.description());
long_description(&zlatan);
```

```
Martin Luther King (93)
Zlatan Ibrahimovic, who is 40 old
```

USING TRAITS IN FUNCTIONS: LONG VS. SHORT FORM

In [29]:
```rust
// short version
fn long_description(person: &impl Person) {
    println!("{}, who is {} old", person.get_name(), person.get_age());
}

// longer version
fn long_description_2<T: Person>(person: &T) {
    println!("{}, who is {} old", person.get_name(), person.get_age());
}
```

USING TRAITS IN FUNCTIONS: LONG VS. SHORT FORM

In [29]:
```
// short version
fn long_description(person: &impl Person) {
    println!("{}, who is {} old", person.get_name(), person.get_age());
}

// longer version
fn long_description_2<T: Person>(person: &T) {
    println!("{}, who is {} old", person.get_name(), person.get_age());
}
```

In [30]:
```
long_description(&zlatan);
long_description_2(&zlatan);
```

```
Zlatan Ibrahimovic, who is 40 old
Zlatan Ibrahimovic, who is 40 old
```

USING TRAITS IN FUNCTIONS: MULTIPLE TRAITS

In [31]:
```rust
use core::fmt::Debug;

fn multiple_1(person: &(impl Person + Debug)) {
    println!("{:?}",person);
    println!("Age: {}",person.get_age());
}
```

USING TRAITS IN FUNCTIONS: MULTIPLE TRAITS

In [31]:
```
use core::fmt::Debug;

fn multiple_1(person: &(impl Person + Debug)) {
    println!("{:?}",person);
    println!("Age: {}",person.get_age());
}
```

In [32]:
```
multiple_1(&zlatan);
```

```
multiple_1(&zlatan);
           ^^^^^^^ `SoccerPlayer` cannot be formatted using `{:?}`
multiple_1(&zlatan);
^^^^^^^^^^ required by a bound introduced by this call
`SoccerPlayer` doesn't implement `Debug`
help: the trait `Debug` is not implemented for `SoccerPlayer`
```

USING TRAITS IN FUNCTIONS: MULTIPLE TRAITS

In [31]:
```
use core::fmt::Debug;

fn multiple_1(person: &(impl Person + Debug)) {
    println!("{:?}",person);
    println!("Age: {}",person.get_age());
}
```

In [32]:
```
multiple_1(&zlatan);
```

```
multiple_1(&zlatan);
           ^^^^^^^ `SoccerPlayer` cannot be formatted using `{:?}`
multiple_1(&zlatan);
^^^^^^^^^^ required by a bound introduced by this call
`SoccerPlayer` doesn't implement `Debug`
help: the trait `Debug` is not implemented for `SoccerPlayer`
```

In [33]:
```
multiple_1(&mlk);
```

```
RegularPerson { year_born: 1929, first_name: "Martin", middle_name: "Luther", last_name: "King" }
Age: 93
```

USING TRAITS IN FUNCTIONS: MULTIPLE TRAITS

In [34]:
```rust
// three options, useful for different settings

fn multiple_1(person: &(impl Person + Debug)) {
    println!("{:?}",person);
    println!("Age: {}",person.get_age());
}

fn multiple_2<T: Person + Debug>(person: &T) {
    println!("{:?}",person);
    println!("Age: {}",person.get_age());
}

fn multiple_3<T>(person: &T)
    where T: Person + Debug
{
    println!("{:?}",person);
    println!("Age: {}",person.get_age());
}
```

USING TRAITS IN FUNCTIONS: MULTIPLE TRAITS

In [34]:
```
// three options, useful for different settings

fn multiple_1(person: &(impl Person + Debug)) {
    println!("{:?}",person);
    println!("Age: {}",person.get_age());
}

fn multiple_2<T: Person + Debug>(person: &T) {
    println!("{:?}",person);
    println!("Age: {}",person.get_age());
}

fn multiple_3<T>(person: &T)
    where T: Person + Debug
{
    println!("{:?}",person);
    println!("Age: {}",person.get_age());
}
```

In [35]:
```
multiple_1(&mlk);
multiple_2(&mlk);
multiple_3(&mlk);
```

```
RegularPerson { year_born: 1929, first_name: "Martin", middle_name: "Luther", last_name: "King" }
Age: 93
RegularPerson { year_born: 1929, first_name: "Martin", middle_name: "Luther", last_name: "King" }
Age: 93
RegularPerson { year_born: 1929, first_name: "Martin", middle_name: "Luther", last_name: "King" }
Age: 93
```

RETURNING TYPES IMPLEMENTING A TRAIT

In [36]:
```
fn get_zlatan() -> impl Person {
    SoccerPlayer::create(String::from("Zlatan Ibrahimovic"), 40, String::from("AC Milan"))
}
```

RETURNING TYPES IMPLEMENTING A TRAIT

In [36]:
```
fn get_zlatan() -> impl Person {
    SoccerPlayer::create(String::from("Zlatan Ibrahimovic"), 40, String::from("AC Milan"))
}
```

In [37]:
```
{
    let zlatan_2 = get_zlatan();
    long_description(&zlatan_2);
};
```

```
Zlatan Ibrahimovic, who is 40 old
```