Traits

Aaron Turon

Memory safety without GC

Memory safety without GC

\+ Abstraction without overhead

Memory safety without GC
+ Abstraction without overhead
+ Concurrency without data races

Memory safety without GC
\+ Abstraction without overhead
\+ Concurrency without data races
= ***Hack without fear***

By QuentinUK (Own work), via Wikimedia Commons

Shop till you drop!

Quick review: structs

```
struct Store {
    name: String,
    items: Vec<Item>,
}

struct Item {
    name: &'static str,
    price: f32,
}
```

Quick review: construction

```
impl Store {
    fn new(name: String) -> Store {
        Store {
            name: name,
            items: Vec::new(),
        }
    }
}
```

Quick review: methods

```
impl Store {
    fn price(&self, name: &str) -> Option<f32> {
        for item in &self.items {
            if item.name == item_name {
                return Some(item.price);
            }
        }
        None
    }
}
```

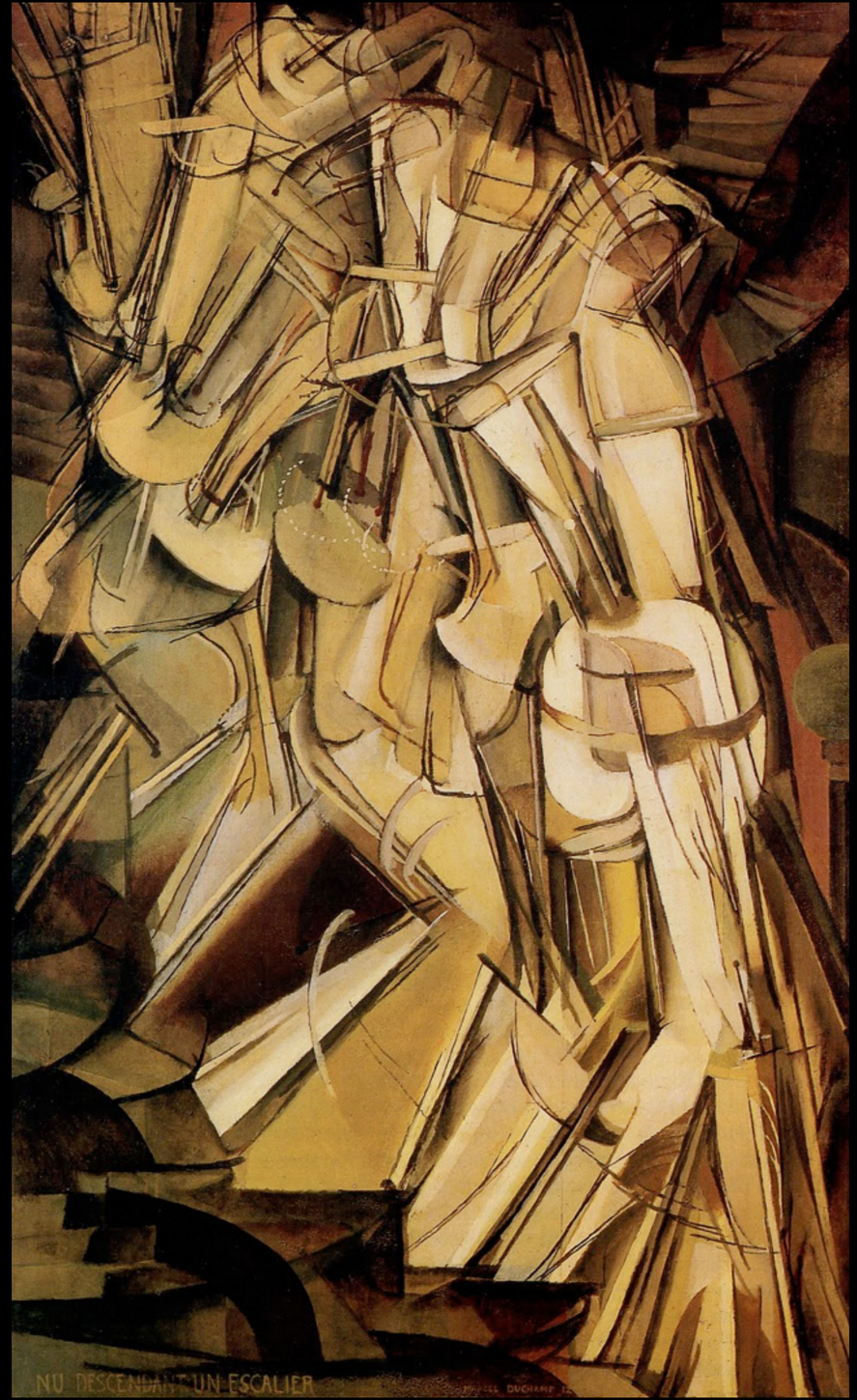
Quick review: mutation

```
impl Store {
    fn add_item(&mut self, item: Item) {
        self.items.push(item);
    }
}
```

Quick review: matching

```
impl Store {
    fn total(&self, list: &[&str]) -> Option<f32> {
        let mut sum = 0.0;
        for name in shopping_list {
            match self.price(name) {
                Some(v) => sum += v,
                None => return None
            }
        }
        Some(sum)
    }
}
```

Abstraction



Abstraction: the plan

* Generics
* Traits
  * as interfaces
  * for code reuse
  * for operator overloading
* Trait objects

Generics

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

fn unwrap_or<T>(opt: Option<T>, default: T) -> T {
  match opt {
    Some(t) => t,
    None => default,
  }
}
```

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

fn unwrap_or<T>(opt: Option<T>, default: T) -> T {
  match opt {
    Some(t) => t,
    None => default,
  }
}
```

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

fn unwrap_or<T>(opt: Option<T>, default: T) -> T {
  match opt {
    Some(t) => t,
    None => default,
  }
}
```

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

fn unwrap_or<T>(opt: Option<T>, default: T) -> T {
  match opt {
    Some(t) => t,
    None => default,
  }
}
```

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

fn unwrap_or<T>(opt: Option<T>, default: T) -> T {
  match opt {
    Some(t) => t,
    None => default,
  }
}
```

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

fn unwrap_or<T>(opt: Option<T>, default: T) -> T {
  match opt {
    Some(t) => t,
    None => default,
  }
}
```

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

fn unwrap_or<T>(opt: Option<T>, default: T) -> T {
  match opt {
    Some(t) => t,
    None => default,
  }
}
```

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

fn unwrap_or<T>(opt: Option<T>, default: T) -> T {
  match opt {
    Some(t) => t,
    None => default,
  }
}
```

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

fn unwrap_or<T>(opt: Option<T>, default: T) -> T {
  match opt {
    Some(t) => t,
    None => default,
  }
}
```

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

fn unwrap_or<T>(opt: Option<T>, default: T) -> T {
  match opt {
    Some(t) => t,
    None => default,
  }
}
```

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

fn unwrap_or<T>(opt: Option<T>, default: T) -> T {
  match opt {
    Some(t) => t,
    None => default,
  }
}
```

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

fn unwrap_or<T>(opt: Option<T>, default: T) -> T {
  match opt {
    Some(t) => t,
    None => default,
  }
}
```

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

fn unwrap_or<T>(opt: Option<T>, default: T) -> T {
  match opt {
    Some(t) => t,
    None => default,
  }
}
```

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

fn unwrap_or<T>(opt: Option<T>, default: T) -> T {
  match opt {
    Some(t) => t,
    None => default,
  }
}
```

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

fn unwrap_or<T>(opt: Option<T>, default: T) -> T {
  match opt {
    Some(t) => t,
    None => default,
  }
}
```

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

impl<T> Option<T> {
  fn unwrap_or(self, def: T) -> T {
    match self {
      Some(t) => t,
      None => def,
    }
  }
}
```

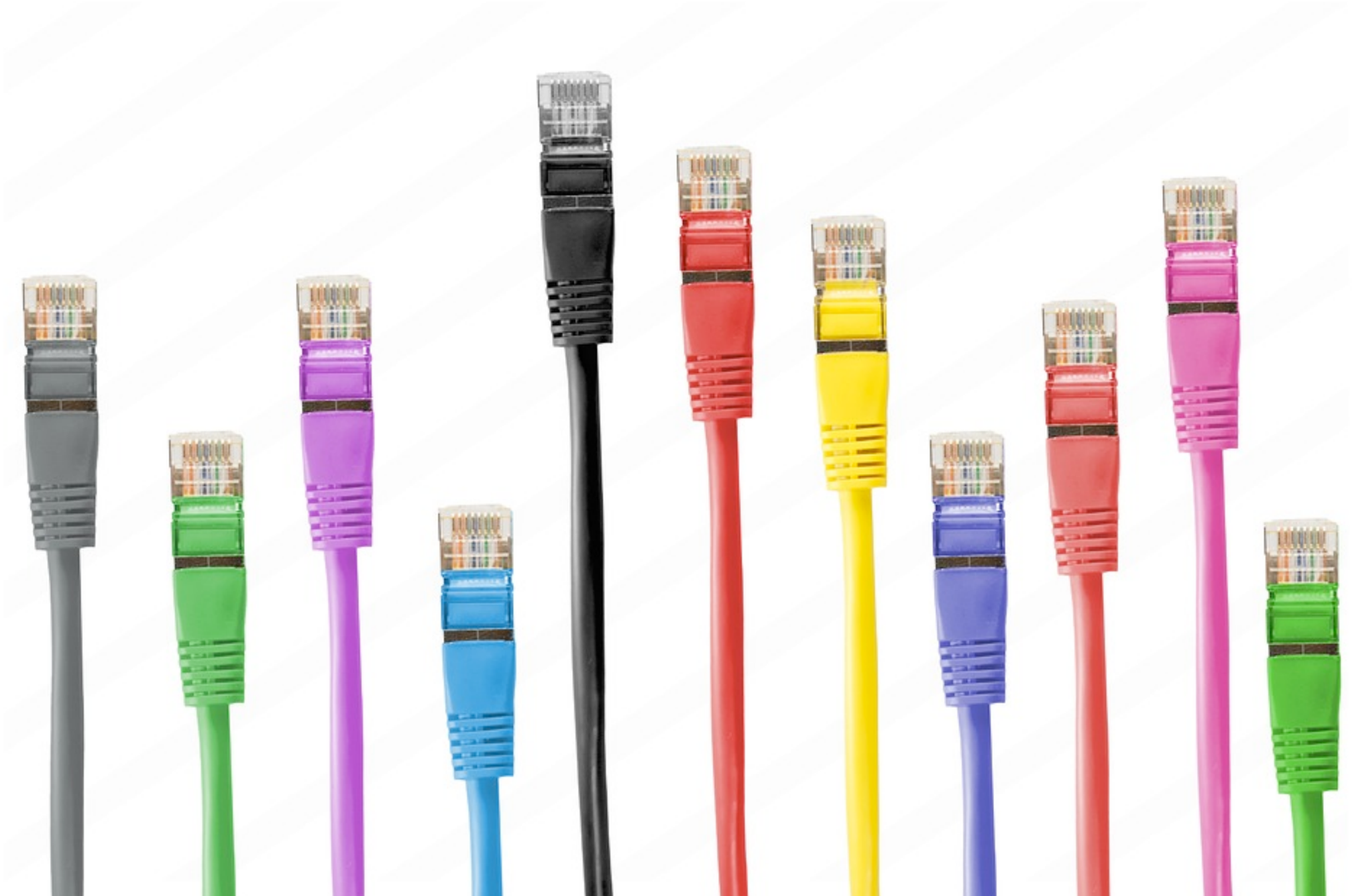
Basic generics

```
enum Option<T> {
  Some(T),
  None
}

impl<T> Option<T> {
  fn unwrap_or(self, def: T) -> T {
    match self {
      Some(t) => t,
      None => def,
    }
  }
}
```

Basic generics

```
enum Option<T> {
  Some(T),
  None
}

impl<T> Option<T> {
  fn unwrap_or(self, def: T) -> T {
    match self {
      Some(t) => t,
      None => def,
    }
  }
}
```

Interfaces



Traits are interfaces

```
trait Print {
  fn print(&self);
}

impl Print for u64 {
  fn print(&self) { println!(“{}”, self) }
}

impl Print for char {
  fn print(&self) { println!(“‘{}’”, self) }
}
```

Traits are interfaces

```
trait Print {
  fn print(&self);
}

impl Print for u64 {
  fn print(&self) { println!(“{}”, self) }
}

impl Print for char {
  fn print(&self) { println!(“‘{}’”, self) }
}
```

Traits are interfaces

```
trait Print {
  fn print(&self);
}

impl Print for u64 {
  fn print(&self) { println!(“{}”, self) }
}

impl Print for char {
  fn print(&self) { println!(“‘{}’”, self) }
}
```

Traits are interfaces

```
trait Print {
  fn print(&self);
}

impl Print for u64 {
  fn print(&self) { println!(“{}”, self) }
}

impl Print for char {
  fn print(&self) { println!(“‘{}’”, self) }
}
```

Traits are interfaces

```
trait Print {
  fn print(&self);
}

impl Print for u64 {
  fn print(&self) { println!(“{}”, self) }
}

impl Print for char {
  fn print(&self) { println!(“‘{}’”, self) }
}
```

Traits are interfaces

```
trait Print {
  fn print(&self);
}

impl Print for u64 {
  fn print(&self) { println!(“{}”, self) }
}

impl Print for char {
  fn print(&self) { println!(“‘{}’”, self) }
}
```

Traits are interfaces

```
trait Print {
  fn print(&self);
}

impl Print for u64 {
  fn print(&self) { println!(“{}”, self) }
}

impl Print for char {
  fn print(&self) { println!(“‘{}’”, self) }
}
```

Traits are interfaces

```
trait Print {
  fn print(&self);
}

impl Print for u64 {
  fn print(&self) { println!(“{}”, self) }
}

impl Print for char {
  fn print(&self) { println!(“‘{}’”, self) }
}
```

Traits are interfaces

```
trait Print {
  fn print(&self);
}

impl Print for u64 {
  fn print(&self) { println!(“{}”, self) }
}

impl Print for char {
  fn print(&self) { println!(“‘{}’”, self) }
}
```

Traits are interfaces

```
trait Print {
  fn print(&self);
}

impl Print for u64 {
  fn print(&self) { println!(“{}”, self) }
}

impl Print for char {
  fn print(&self) { println!(“‘{}’”, self) }
}
```

Traits are interfaces

```
trait Print {
  fn print(&self);
}

impl Print for u64 {
  fn print(&self) { println!(“{}”, self) }
}

impl Print for char {
  fn print(&self) { println!(“‘{}’”, self) }
}
```

Traits are interfaces

```
trait Print {
  fn print(&self);
}

impl Print for u64 {
  fn print(&self) { println!(“{}”, self) }
}

impl Print for char {
  fn print(&self) { println!(“‘{}’”, self) }
}
```

Traits are interfaces

```
trait Print {
  fn print(&self);
}

impl Print for u64 {
  fn print(&self) { println!(“{}”, self) }
}

impl Print for char {
  fn print(&self) { println!(“‘{}’”, self) }
}
```

```rust
// client of the interface
fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice {
    elem.print();
  }
}

fn main() {
    let s1: &[u64] = &[0, 1, 2];
    print_slice(s1);

    let s2: &[char] = &['h', 'i'];
    print_slice(s2);
}
```

```
// client of the interface
fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice {
    elem.print();
  }
}

fn main() {
    let s1: &[u64] = &[0, 1, 2];
    print_slice(s1);

    let s2: &[char] = &['h', 'i'];
    print_slice(s2);
}
```

```rust
// client of the interface
fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice {
    elem.print();
  }
}

fn main() {
    let s1: &[u64] = &[0, 1, 2];
    print_slice(s1);

    let s2: &[char] = &['h', 'i'];
    print_slice(s2);
}
```

```
// client of the interface
fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice {
    elem.print();
  }
}

fn main() {
    let s1: &[u64] = &[0, 1, 2];
    print_slice(s1);

    let s2: &[char] = &['h', 'i'];
    print_slice(s2);
}
```

```rust
// client of the interface
fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice {
    elem.print();
  }
}

fn main() {
    let s1: &[u64] = &[0, 1, 2];
    print_slice(s1);

    let s2: &[char] = &['h', 'i'];
    print_slice(s2);
}
```

```
// client of the interface
fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice {
    elem.print();
  }
}

fn main() {
    let s1: &[u64] = &[0, 1, 2];
    print_slice(s1);

    let s2: &[char] = &['h', 'i'];
    print_slice(s2);
}
```

```rust
// client of the interface
fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice {
    elem.print();
  }
}

fn main() {
    let s1: &[u64] = &[0, 1, 2];
    print_slice(s1);

    let s2: &[char] = &['h', 'i'];
    print_slice(s2);
}
```

```rust
// client of the interface
fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice {
    elem.print();
  }
}

fn main() {
    let s1: &[u64] = &[0, 1, 2];
    print_slice(s1);

    let s2: &[char] = &['h', 'i'];
    print_slice(s2);
}
```

```rust
// client of the interface
fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice {
    elem.print();
  }
}

fn main() {
    let s1: &[u64] = &[0, 1, 2];
    print_slice(s1);

    let s2: &[char] = &['h', 'i'];
    print_slice(s2);
}
```

```
// client of the interface
fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice {
    elem.print();
  }
}

fn main() {
    let s1: &[u64] = &[0, 1, 2];
    print_slice(s1);

    let s2: &[char] = &['h', 'i'];
    print_slice(s2);
}
```

```rust
// client of the interface
fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice {
    elem.print();
  }
}

fn main() {
    let s1: &[u64] = &[0, 1, 2];
    print_slice(s1);

    let s2: &[char] = &['h', 'i'];
    print_slice(s2);
}
```

Exercise: **interfaces**

http://www.rust-tutorials.com/RustConf17/

Implement

```
fn price(..)
```

```
fn total_price(..)
```

Cheat sheet:

```
let mut some_var = 0.0;

some_var += x;

for s in &v { … }
```

```
match … {
  Some(x) => { … }
  None => { … }
}
```

http://doc.rust-lang.org/std

Reuse: default methods

```
trait Read {
  fn read(&mut self, buf: &mut [u8])
          -> Result<usize>;

  fn read_to_end(&mut self, buf: &mut Vec<u8>)
                 -> Result<usize> {
    // generic implementation
  }

  // additional default methods:
  // read_to_string, read_exact, by_ref, bytes,
  // chars, chain, take
}
```

Reuse: default methods

```
trait Read {
    fn read(&mut self, buf: &mut [u8])
            -> Result<usize>;

    fn read_to_end(&mut self, buf: &mut Vec<u8>)
                   -> Result<usize> {
      // generic implementation
    }

    // additional default methods:
    // read_to_string, read_exact, by_ref, bytes,
    // chars, chain, take
}
```

Reuse: default methods

```
trait Read {
  fn read(&mut self, buf: &mut [u8])
          -> Result<usize>;

  fn read_to_end(&mut self, buf: &mut Vec<u8>)
                 -> Result<usize> {
    // generic implementation
  }

  // additional default methods:
  // read_to_string, read_exact, by_ref, bytes,
  // chars, chain, take
}
```

Reuse: default methods

```
trait Read {
  fn read(&mut self, buf: &mut [u8])
          -> Result<usize>;

  fn read_to_end(&mut self, buf: &mut Vec<u8>)
                 -> Result<usize> {
    // generic implementation
  }

  // additional default methods:
  // read_to_string, read_exact, by_ref, bytes,
  // chars, chain, take
}
```

Reuse: default methods

```
trait Read {
  fn read(&mut self, buf: &mut [u8])
          -> Result<usize>;

  fn read_to_end(&mut self, buf: &mut Vec<u8>)
                 -> Result<usize> {
    // generic implementation
  }

  // additional default methods:
  // read_to_string, read_exact, by_ref, bytes,
  // chars, chain, take
}
```

Reuse: default methods

```
impl Read for File {
  fn read(&mut self, buf: &mut [u8])
          -> Result<usize> { ... }
}

fn read_file(f: &File) -> String {
  let mut contents = String::new();
  f.read_to_string(&mut contents).unwrap();
  contents
}
```

Reuse: default methods

```
impl Read for File {
  fn read(&mut self, buf: &mut [u8])
          -> Result<usize> { ... }
}
```

```
fn read_file(f: &File) -> String {
  let mut contents = String::new();
  f.read_to_string(&mut contents).unwrap();
  contents
}
```

Reuse: default methods

```
impl Read for File {
  fn read(&mut self, buf: &mut [u8])
          -> Result<usize> { ... }
}

fn read_file(f: &File) -> String {
  let mut contents = String::new();
  f.read_to_string(&mut contents).unwrap();
  contents
}
```

Reuse: default methods

```
impl Read for File {
  fn read(&mut self, buf: &mut [u8])
          -> Result<usize> { ... }
}

fn read_file(f: &File) -> String {
  let mut contents = String::new();
  f.read_to_string(&mut contents).unwrap();
  contents
}
```

Reuse: default methods

```
impl Read for File {
  fn read(&mut self, buf: &mut [u8])
          -> Result<usize> { ... }
}

fn read_file(f: &File) -> String {
  let mut contents = String::new();
  f.read_to_string(&mut contents).unwrap();
  contents
}
```

Exercise: **defaults**

http://www.rust-tutorials.com/RustConf17/

Implement

```
fn total_price(..)
```

Cheat sheet:

```
let mut some_var = 0.0;

some_var += x;

for s in &v { … }
```

```
match … {
   Some(x) => { … }
   None => { … }
}
```

http://doc.rust-lang.org/std

Layering

Reuse: layering implementations

```
trait Clone {
  fn clone(&self) -> Self;
}

impl<T: Clone, U: Clone> Clone for (T, U) {
  fn clone(&self) -> (T, U) {
    (t.clone(), u.clone())
  }
}

impl<T: Clone> Clone for Vec<T> { ... }
```

Reuse: layering implementations

```
trait Clone {
  fn clone(&self) -> Self;
}

impl<T: Clone, U: Clone> Clone for (T, U) {
  fn clone(&self) -> (T, U) {
    (t.clone(), u.clone())
  }
}

impl<T: Clone> Clone for Vec<T> { ... }
```

Reuse: layering implementations

```
trait Clone {
  fn clone(&self) -> Self;
}

impl<T: Clone, U: Clone> Clone for (T, U) {
  fn clone(&self) -> (T, U) {
    (t.clone(), u.clone())
  }
}

impl<T: Clone> Clone for Vec<T> { ... }
```

Reuse: layering implementations

```
trait Clone {
  fn clone(&self) -> Self;
}

impl<T: Clone, U: Clone> Clone for (T, U) {
  fn clone(&self) -> (T, U) {
    (t.clone(), u.clone())
  }
}

impl<T: Clone> Clone for Vec<T> { ... }
```

Reuse: layering implementations

```
trait Clone {
  fn clone(&self) -> Self;
}

impl<T: Clone, U: Clone> Clone for (T, U) {
  fn clone(&self) -> (T, U) {
    (t.clone(), u.clone())
  }
}

impl<T: Clone> Clone for Vec<T> { ... }
```

Reuse: layering implementations

```
trait Clone {
  fn clone(&self) -> Self;
}

impl<T: Clone, U: Clone> Clone for (T, U) {
  fn clone(&self) -> (T, U) {
    (t.clone(), u.clone())
  }
}

impl<T: Clone> Clone for Vec<T> { ... }
```

Reuse: layering implementations

```
trait Clone {
  fn clone(&self) -> Self;
}

impl<T: Clone, U: Clone> Clone for (T, U) {
  fn clone(&self) -> (T, U) {
    (t.clone(), u.clone())
  }
}

impl<T: Clone> Clone for Vec<T> { ... }
```

Reuse: layering implementations

```
trait Clone {
  fn clone(&self) -> Self;
}

impl<T: Clone, U: Clone> Clone for (T, U) {
  fn clone(&self) -> (T, U) {
    (t.clone(), u.clone())
  }
}

impl<T: Clone> Clone for Vec<T> { ... }
```

Reuse: layering implementations

```
trait Clone {
  fn clone(&self) -> Self;
}

impl<T: Clone, U: Clone> Clone for (T, U) {
  fn clone(&self) -> (T, U) {
    (t.clone(), u.clone())
  }
}
```

```
impl<T: Clone> Clone for Vec<T> { ... }
```

Reuse: layering implementations

```
trait Clone {
  fn clone(&self) -> Self;
}

impl<T: Clone, U: Clone> Clone for (T, U) {
  fn clone(&self) -> (T, U) {
    (t.clone(), u.clone())
  }
}

impl<T: Clone> Clone for Vec<T> { ... }
```

```
impl<T: Clone, U: Clone> Clone for (T, U) { ... }
impl<T: Clone> Clone for Vec<T> { ... }
impl Clone for u64 { ... }
impl Clone for String { ... }
```

```
impl<T: Clone, U: Clone> Clone for (T, U) { ... }
impl<T: Clone> Clone for Vec<T> { ... }
impl Clone for u64 { ... }
impl Clone for String { ... }


u64: Clone
```

```
impl<T: Clone, U: Clone> Clone for (T, U) { ... }
impl<T: Clone> Clone for Vec<T> { ... }
impl Clone for u64 { ... }
impl Clone for String { ... }
```

u64: Clone
String: Clone

```
impl<T: Clone, U: Clone> Clone for (T, U) { ... }
impl<T: Clone> Clone for Vec<T> { ... }
impl Clone for u64 { ... }
impl Clone for String { ... }
```

```
u64: Clone
String: Clone
Vec<u64>: Clone
```

```
impl<T: Clone, U: Clone> Clone for (T, U) { ... }
impl<T: Clone> Clone for Vec<T> { ... }
impl Clone for u64 { ... }
impl Clone for String { ... }
```

```
u64: Clone
String: Clone
Vec<u64>: Clone
(u64, String): Clone
```

```
impl<T: Clone, U: Clone> Clone for (T, U) { ... }
impl<T: Clone> Clone for Vec<T> { ... }
impl Clone for u64 { ... }
impl Clone for String { ... }
```

```
u64: Clone
String: Clone
Vec<u64>: Clone
(u64, String): Clone
(Vec<u64>, String): Clone
```

```
impl<T: Clone, U: Clone> Clone for (T, U) { ... }
impl<T: Clone> Clone for Vec<T> { ... }
impl Clone for u64 { ... }
impl Clone for String { ... }
```

```
u64: Clone
String: Clone
Vec<u64>: Clone
(u64, String): Clone
(Vec<u64>, String): Clone
Vec<(Vec<u64>, String)>: Clone
```

Exercise: **layering**

http://www.rust-tutorials.com/RustConf17/

Implement

`Print for char`

`Print for Vec`

Cheat sheet:

http://doc.rust-lang.org/std

Standard traits

`Clone`	create explicit copies by writing `foo.clone()`
`Copy`	create implicit copies (requires Clone)
`Debug`	debug printing with `println!("{:?}", foo)`
`PartialEq`	equality comparisons (`foo == bar`)
`PartialOrd`	inequality comparisons (`foo > bar` etc)
`Hash`	hashing for a hashmap

…

Operator overloading

```
struct Item {
  name: &’static str,
  price: f32,
}

impl PartialEq for Item {
  fn eq(&self, other: &Item) -> bool {
    self.name == other.name &&
      self.price == other.price
  }
}

// Now item1 == item2 works
```

Derive

```
#[derive(PartialEq)]
struct Item {
    name: &’static str,
    price: f32,
}
```

Derive

```
#[derive(PartialEq)]
struct Item {
    name: &’static str,
    price: f32,
}
```

Objects

Trait objects

```
trait Print {
  fn print(&self);
}

fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice { elem.print(); }
}

fn main() {
  print_slice(&[0, ‘x’]);
}
```

Trait objects

```
trait Print {
  fn print(&self);
}

fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice { elem.print(); }
}

fn main() {
  let slice: &[???] = &[0, ‘x’];
  print_slice(slice);
}
```

Trait objects

```
trait Print {
  fn print(&self);
}

fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice { elem.print(); }
}

fn main() {
  let slice: &[???] = &[0, ‘x’];
  print_slice(slice);
}
```

Trait objects

```
trait Print {
  fn print(&self);
}

fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice { elem.print(); }
}

fn main() {
  let slice: &[???] = &[0, ‘x’];
  print_slice(slice);
}
```

Trait objects

```
trait Print {
  fn print(&self);
}

fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice { elem.print(); }
}

fn main() {
  let slice: &[???] = &[0, ‘x’];
  print_slice(slice);
}
```

Trait objects

```
trait Print {
  fn print(&self);
}

fn print_slice<T: Print>(slice: &[T]) {
  for elem in slice { elem.print(); }
}

fn main() {
  let slice: &[???] = &[0, ‘x’];
  print_slice(slice);
}
```

Trait objects

```
trait Print {
  fn print(&self);
}

fn print_slice(slice: &[&Print]) {
  for elem in slice { elem.print(); }
}

fn main() {
  let slice: &[&Print] = &[&0, &'x'];
  print_slice(slice);
}
```

Trait objects

```
trait Print {
  fn print(&self);
}

fn print_slice(slice: &[&Print]) {
  for elem in slice { elem.print(); }
}

fn main() {
  let slice: &[&Print] = &[&0, &'x'];
  print_slice(slice);
}
```

Trait objects

```
trait Print {
  fn print(&self);
}

fn print_slice(slice: &[&Print]) {
  for elem in slice { elem.print(); }
}

fn main() {
  let slice: &[&Print] = &[&0, &'x'];
  print_slice(slice);
}
```

Trait objects

```
trait Print {
  fn print(&self);
}

fn print_slice(slice: &[&Print]) {
  for elem in slice { elem.print(); }
}

fn main() {
  let slice: &[&Print] = &[&0, &'x'];
  print_slice(slice);
}
```

Trait objects

```
trait Print {
  fn print(&self);
}

fn print_slice(slice: &[&Print]) {
  for elem in slice { elem.print(); }
}

fn main() {
  let slice: &[&Print] = &[&0, &'x'];
  print_slice(slice);
}
```

Trait objects

```
trait Print {
  fn print(&self);
}

fn print_slice(slice: &[&Print]) {
  for elem in slice { elem.print(); }
}

fn main() {
  let slice: &[&Print] = &[&0, &'x'];
  print_slice(slice);
}
```

Trait objects

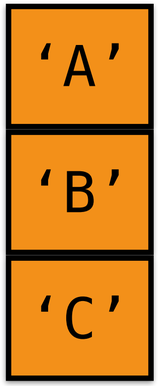
```
trait Print {
  fn print(&self);
}

fn print_slice(slice: &[&Print]) {
  for elem in slice { elem.print(); }
}

fn main() {
  let slice: &[&Print] = &[&0, &'x'];
  print_slice(slice);
}
```

[char]

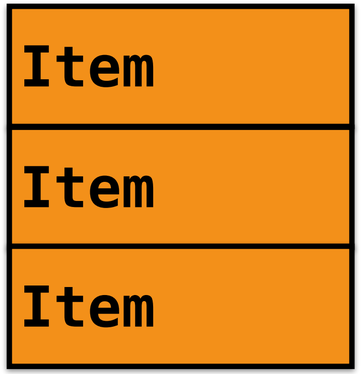
[Item]

[&Print]

[char]

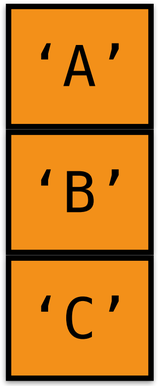


[Item]

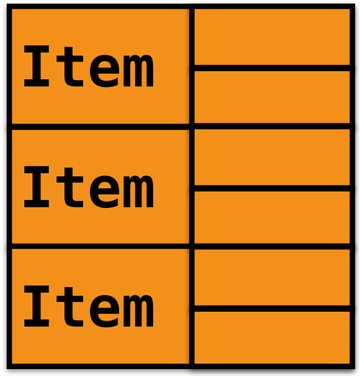


[&Print]

[char]

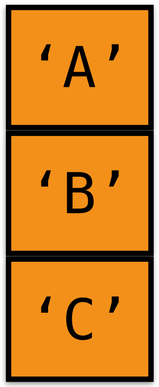


[Item]

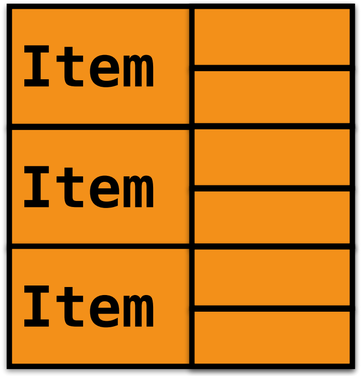


[&Print]

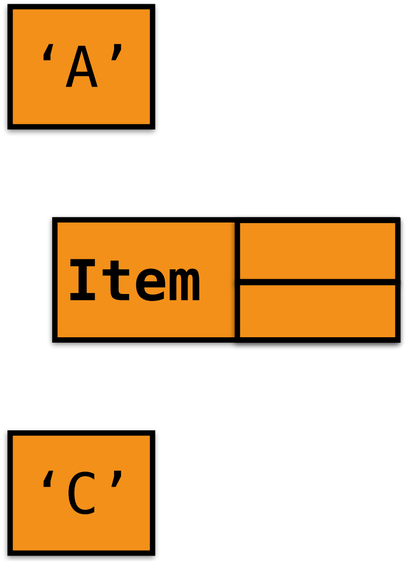
[char]



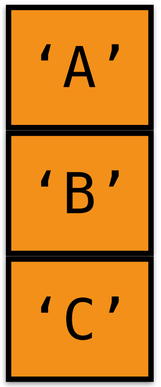
[Item]



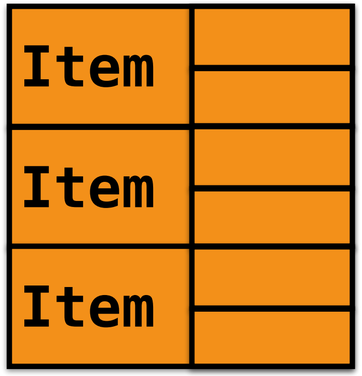
[&Print]



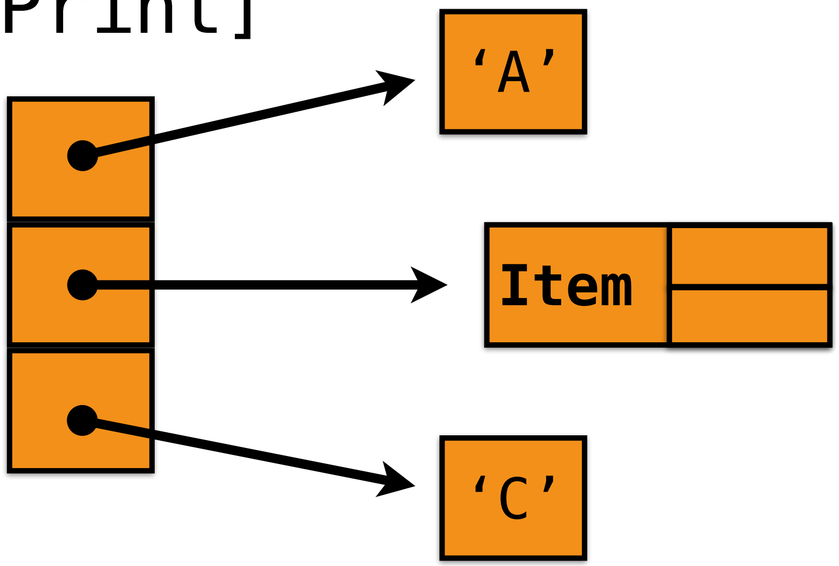
[char]



[Item]



[&Print]



Generics vs objects

* **Generics** (aka “zero cost abstraction”)
  * Work for singletons (`Option<T>`)
  * Work for *uniform* collections (`Vec<Item>`)
  * Provide *static* dispatch — every use specialized
    * Good for performance, bad for code size

* **Objects**
  * Must live behind a pointer (`&`, `Box`)
  * Work for *heterogenous* collections (`Vec<&Price>`)
  * Provide *dynamic* dispatch — a “vtable”
    * Bad for performance, good for code size

Abstraction: what we saw

* Generics
* Traits
  * as interfaces
  * for code reuse
  * for operator overloading
* Trait objects

Thanks for listening!