Traits

Learn to Code with Rust / Section Review

Traits

- A **trait** is a contract that requires that a type support a functionality.
- Traits enable similar *behavior* between different types.
- Types opt in to **implementing** a trait.
- A type can implement multiple traits. A trait can be implemented by multiple types.

Trait Definitions

- The **trait definition** defines the required methods and their signatures.
- The **trait definition** can define a default implementation for a method. Rust will use the fallback method if a type does not define the method.
- A trait must be in scope to invoke any of its methods on a type.

Implementing a Trait

- Use the **impl** keyword followed by the trait name, the **for** keyword, and the type.
  - Rust requires one **impl** block per trait.
- Define the trait's required methods within a block. Provide concrete implementations.
- The compiler will raise an error if there are too few methods, too many methods, or any type mismatches.
- Independent methods must be defined in separate **impl** blocks.

Associated Constants

- A trait can define a constant with the **const** keyword. Provide the type and the assignment.
- Trait methods can utilize the constant's value.
- Trait implementations can overwrite the constant for a given type.

Trait Bounds

- A function can accept a generic parameter that implements a given trait.
- The first option is the **impl** keyword followed by the trait.
- The second option is to define a generic and add trait bounds within the angle brackets.
- The third option is to define a generic and pair it with the **where** keyword to add trait constraints.

Trait Objects

- A **trait object** is an instance of a type that implements a trait whose methods will be accessed at runtime using a feature called dynamic dispatch.
- Trait objects enable the code to store different types within collections like vectors.
- Use **&dyn** followed by the trait that all types in the vector will implement.

Getters and Setters

- A **getter** is a trait method whose purpose is to read a piece of data.
- A **setter** is a trait method whose purpose is to write a piece of data.
- Getters and setters bypass the problem that traits can only mandate methods, not data/state.

Supertraits

- A **supertrait** is a trait from which another trait inherits. It is also called the **parent trait**.
- The **subtrait** is the trait that inherits functionality. It is also called the **child** trait.
- Use colon syntax to establish a relationship between subtrait and supertrait.
  - trait **Subtrait: Supertrait**
- Types that implement the subtrait must implement the supertrait. The reverse does not apply.

The **Display** Trait

- The **Display** trait requires that a type can be represented as a human-readable string.
- The **Display** trait requires a **fmt** method. Rust will pass a mutable reference to a **Formatter** struct.
- One option is to use the **write!** macro to write to the **Formatter** struct.
- A second option is to use the **Formatter** struct's methods to build a string for various data types.
- The **{}** interpolation syntax depends on a type implementing the **Display** trait.

The **Debug** Trait

- The **Debug** trait requires that a type can be represented as a technical string for debugging.
- The **Debug** trait requires a **fmt** method. Rust will pass a mutable reference to a **Formatter** struct.
- The same writing options are available to shape the final string.
- The **{:?}** interpolation syntax depends on a type implementing the **Debug** trait.

The **Drop** Trait

- The **Drop** trait and its **drop** method define clean-up execution logic when a heap type is deallocated.
- Rust will invoke the **drop** method when the type is deallocated.
- Our code cannot manually call the **drop** method.

The **Clone** and **Copy** Traits

- The **Clone** trait allows a type to create a duplicate of itself with an explicit call to the **clone** method.
- The **Copy** trait implicitly creates a copy of a type in certain situations (assigning to variable, passing function argument, adding element to array, etc).
- The **Copy** trait is a subtrait of **Clone**. If a type implements **Copy**, it must implement **Clone**.

The **PartialEq** Trait

- The **PartialEq** trait indicates a type can be compared with the equality and inequality signs.
- We can define equality between instances of same types or different types.

The **Eq** Trait

- The **Eq** trait adds 3 more requirements to a type.
- **Reflexive**: a == a;
- **Symmetric**: a == b implies b == a
- **Transitive**: a == b and b == c implies a == c

The **PartialOrd** Trait

- The **PartialOrd** subtrait indicates a type can be ordered/sorted.
- **NaN** (not a number) is a valid floating-point value that is returned in invalid numeric operations.
- Because Rust does not consider **NaNs** equal or sortable, floating-point types implement the **PartialEq** and **PartialOrd** traits, but not implement **Eq** or **Ord.**

Associated Types

- An **associated type** is a placeholder for a type that must be provided in a trait implementation.
- The **Add** trait requires an **Output** associated type that represents the type of the **add** trait method.
- Click into a trait definition and look for the **type** keyword inside the trait definition block.