Rust Cheatsheet

Basic Structure and Execution

Rust is a systems programming language that focuses on performance, memory safety, and concurrency.

```
// Program entry point
fn main() {
    println!("Hello, World!");
}
```

- Cargo (Package Manager):
  - `cargo new my_project`: Create a new project.
  - `cargo build`: Compile the project.
  - `cargo run`: Compile and run.
  - `cargo test`: Run tests.
  - `cargo check`: Check code for errors without compiling to binary.

Variables and Data Types

- Variables:
  - `let x = 5;`: Immutable by default.
  - `let mut y = 10;`: Declared mutable with `mut`.
  - `const MAX_POINTS: u32 = 100_000;`: Constant (type must be explicit).
- Scalar Types: `i32`, `u32` (integers), `f64` (float), `bool`, `char`.
- Compound Types:
  - Tuple: `let tup: (i32, f64, u8) = (500, 6.4, 1);`
  - Array (fixed size): `let a = [1, 2, 3, 4, 5];`
- Enums:

```
enum IpAddrKind { V4, V6 }
let four = IpAddrKind::V4;
```

Ownership and References

Rust's most unique feature for memory safety without a garbage collector.

- Ownership Rules:
  1. Each value in Rust has a variable called its **owner**.
  2. There can only be one owner at a time.
  3. When the owner goes out of scope, the value is dropped.
- References and Borrowing:
  - `&T`: Immutable reference (multiple allowed).
  - `&mut T`: Mutable reference (only one allowed at a time).

Control Flow

- `if-else`:

```
let number = 3;
if number < 5 { /* ... */ } else { /
* ... */ }
```

- `match` (Powerful pattern matching):

```
match value {
    1 => println!("one"),
    2 => println!("two"),
    _ => println!("anything"),
}
```

- Loops:
  - `loop { ... }`: Infinite loop.
  - `while condition { ... }`
  - `for item in elements { ... }`

Functions and Methods

- Function:

```
fn add(x: i32, y: i32) -> i32 {
    x + y // Expression (no semicolon)
returns the value
}
```

- Struct and Method:

```
struct Rectangle { width: u32, height:
u32 }
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

Collections

- `Vec<T>`: Dynamic array. `let mut v = vec![1, 2, 3];`
- `String`: UTF-8 encoded, heap-allocated string.
- `HashMap<K, V>`: Key-value pair collection.
- `Slice`: `&[T]`. References a contiguous sequence of elements.

Type Conversions

- `as` **Keyword**: `x as u64`. Used for numeric casting and pointer conversion.
- `From` & `Into`: `impl From<A> for B`. Use `B::from(a)` or `a.into()` for lossless conversion.
- `TryFrom` & `TryInto`: Fallible conversions. Returns a `Result`.
- `AsRef` & `AsMut`: Cheap reference-to-reference conversion (`&A` to `&B`).
- `Deref`: Define behavior for `*x`. Allows smart pointers to behave like the underlying data.
- `String` **conversion**: `s.parse::<i32>()` (str to int), `x.to_string()` (int to String).

Iterators

- **Creation**:
  - `iter()`: Iterate over immutable references (`&T`).
  - `iter_mut()`: Iterate over mutable references (`&mut T`).
  - `into_iter()`: Moves ownership or iterates over appropriate types.
- **Adapters (Lazy)**:
  - `map(|x| ...)`: Transform each element.
  - `filter(|x| ...)`: Keep elements matching a predicate.
  - `enumerate()`: Returns `(index, value)`.
  - `zip(other)`: Combine two iterators.
- **Consumers**:
  - `collect()`: Convert iterator into a collection (like `Vec`).
  - `fold(init, |acc, x| ...)`: Reduce to a single value.
  - `find(|x| ...)`, `any(|x| ...)`: Search or check conditions.

Error Handling

- `Result<T, E>`: For recoverable errors (`Ok`, `Err`).
- `Option<T>`: For potentially absent values (`Some`, `None`).
- `panic!`: For unrecoverable errors (stops program).
- `?` operator: Propagation of errors/options.

Traits and Generics

- Generics: `struct Point<T> { x: T, y: T }`
- Common Traits:
  - `Clone` / `Copy`: Define duplication and bitwise copy behavior.
  - `Debug` / `Display`: Formatting for output (`{:?}` vs `{}`).
  - `Default`: Create default values (`Default::default()`).
  - `Drop`: Custom behavior on memory deallocation (destructor).
  - `PartialEq` / `Eq`: Define equality comparison.

Concurrency

- `thread::spawn`: Create a new thread.
- `mpsc`: Message passing between threads (Multiple Producer, Single Consumer).
- `Arc<T>` and `Mutex<T>`: Shared state concurrency.

Macros

- `println!`, `vec!`, `format!`: Metaprogramming that generates code.

Naming & Style

- Types/Traits: `UpperCamelCase`
- Functions/Variables/Modules: `snake_case`
- Constants/Macros: `SCREAMING_SNAKE_CASE`
- Formatting: Use `rustfmt` (4-space indent, 100 char limit).