

# JuMP Syntax Cheatsheet

## Applied Optimization with Julia

Dr. Tobias Vlček

### Basic Setup

```
using JuMP, HiGHS

# Create a model
model = Model(HiGHS.Optimizer)

# Set optimizer attribute (optional)
set_optimizer_attribute(model, "time_limit", 60.0)
```

### Variables

#### Declaration

```
# Continuous Variables
@variable(model, x)                # Unbounded continuous
@variable(model, x >= 0)            # Non-negative continuous
@variable(model, 0 <= x <= 10)      # Bounded continuous

# Integer Variables
@variable(model, x, Int)            # Unbounded integer
@variable(model, x >= 0, Int)       # Non-negative integer
@variable(model, 0 <= x <= 10, Int) # Bounded integer

# Binary Variables
@variable(model, x, Bin)            # Binary (0 or 1)
```

### Containers

```
# Arrays
@variable(model, x[1:5])            # Array of 5 variables
@variable(model, x[1:5] >= 0)       # Non-negative array
@variable(model, x[1:5], Bin)       # Binary array

# Matrices
@variable(model, x[1:3, 1:4])       # 3x4 matrix of variables
@variable(model, x[1:3, 1:4], Int)  # Integer matrix

# Custom Indexing
indices = ["A", "B", "C"]
@variable(model, x[i in indices])   # Custom indexed array
```

---

## Constraints

### Declaration

```
# Basic constraints
@constraint(model, con1, 2x + y <= 10)
@constraint(model, con2, x + 2y >= 5)
```

### Containers

```
# Array of variables
@variable(model, x[1:5] >= 0)

# Constraint for each variable
@constraint(model, capacity[i=1:5],
    x[i] <= 100
)

# Sum constraint
@constraint(model, total_sum,
    sum(x[i] for i in 1:5) <= 500
)

# Matrix constraints
@variable(model, y[1:3, 1:4])
@constraint(model, matrix_con[i=1:3, j=1:4],
    y[i,j] <= i + j
)
```

### Conditional

```
# Basic conditional constraint
@constraint(model, cond[i=1:5; i > 2],
    x[i] <= 10
) # Only applies when i > 2

# Multiple conditions
@constraint(model, cond2[i=1:10, j=1:10; i != j && i + j <= 15],
    x[i,j] + x[j,i] <= 1
)
```

### Conditions while Summing

```
@constraint(model, total_sum,
    sum(x[i] for i in 1:5 if i > 2) <= 500
)
# Only sums over i > 2

@constraint(model, total_sum2,
    sum(x[i,j] for i in 1:5, j in 1:5 if i != j && i + j <= 7) <= 1
)
```

```
)
# Only sums over i != j and i + j <= 7
```

## Key Points for Constraints

- Use semicolon (;) to separate indices from conditions
  - Conditions can use any valid Julia boolean expression
  - Multiple conditions can be combined with && (and) or || (or)
- 

## Objective Function

### Declaration

```
# Maximize objective
@objective(model, Max, 5x + 3y)

# Minimize objective
@objective(model, Min, 2x + 4y)
```

## Containers

```
# Container objective
@variable(model, z[1:10])
@objective(model, Min, sum(z[i] for i in 1:10))

# Weighted objective
weights = [1, 2, 3, 4, 5]
@objective(model, Max,
    sum(weights[i] * z[i] for i in 1:5)
)
```

## Key Points

- Objective functions can be linear or nonlinear
  - Containers are useful for weighted objectives
  - Can reference external data (parameters)
- 

## Additional Features

### Checking Bounds

# Checking bounds	
has_lower_bound(x)	# Check if lower bound exists
has_upper_bound(x)	# Check if upper bound exists
lower_bound(x)	# Get lower bound value
upper_bound(x)	# Get upper bound value

### Checking Properties

```

# Check variable type
is_binary(x)           # Is variable binary?
is_integer(x)          # Is variable integer?
is_continuous(x)       # Is variable continuous?

# Get variable info
name(x)                # Get variable name
num_variables(model)   # Count variables in model
all_variables(model)   # Get all variables

```

## Solver Options

```

# Create model with solver
model = Model(HiGHS.Optimizer)

# Time limits
set_time_limit_sec(model, 60)      # 60 second limit
time_limit_sec(model)              # Get current time limit

# Tolerance settings
set_optimizer_attribute(model, "mip_rel_gap", 0.01) # 1% gap tolerance
set_optimizer_attribute(model, "mip_abs_gap", 0.1)  # Absolute gap

# Presolve options
set_optimizer_attribute(model, "presolve", "on")    # Enable presolve
set_optimizer_attribute(model, "presolve", "off")   # Disable presolve

```

## Solution Status Checks

```

# Check solution status
status = termination_status(model) # Get solution status
is_optimal = status == MOI.OPTIMAL # Check if optimal

# Get detailed status
primal_status(model) # Primal solution status
dual_status(model)   # Dual solution status
solve_time(model)    # Solution time

# Common status checks
if termination_status(model) == OPTIMAL
    println("Solution is optimal")
elseif termination_status(model) == TIME_LIMIT && has_values(model)
    println("Time limit reached with feasible solution")
else
    println("Problem could not be solved")
end

```

## Key Points

- Always check solution status before using results
- Set appropriate time limits for large problems
- Use gap tolerances to balance precision and speed

- Monitor solve time for performance optimization
- Consider presolve for complex problems

## Bibliography