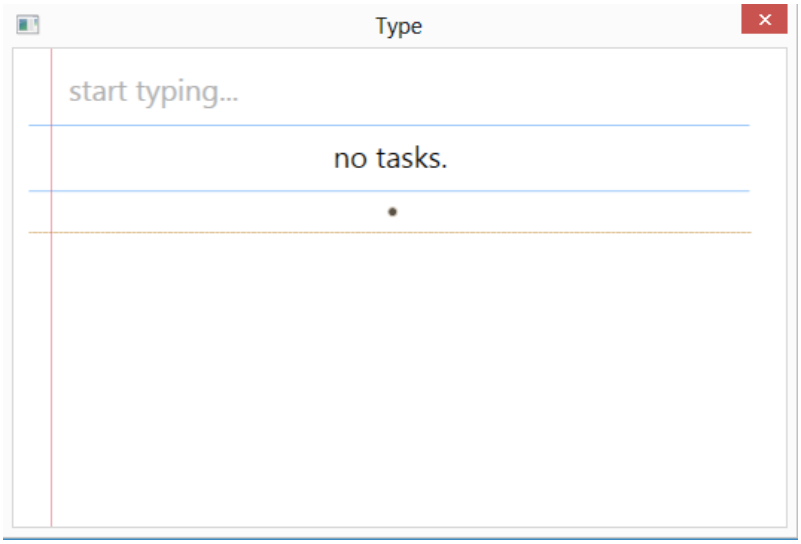
type

User Guide



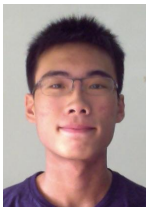
			
Camillus Gerard Cai	**Michael Yong**	**Ang Civics**	**Yan Rong**
Controller, System and Integration Testing	*Model, Presentation Logic and Language Processing*	*Data Model and Storage*	*Presenter, User Experience and Design*

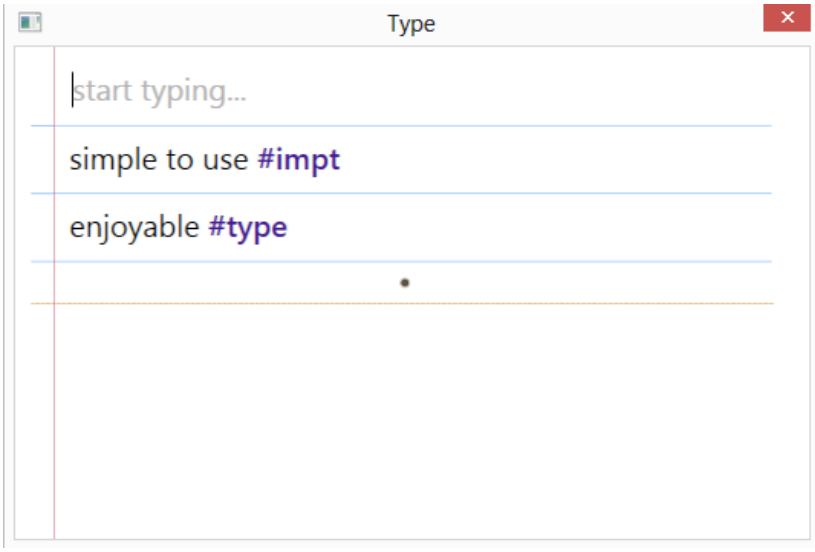
Table Of Contents

Introduction

Type is a desktop task manager that aims to help users prioritise what to do and when to do things.

Tasks Re-imagined

Type rethinks every single aspect of the typical task manager. We wanted to make something that users would truly love and enjoy using.



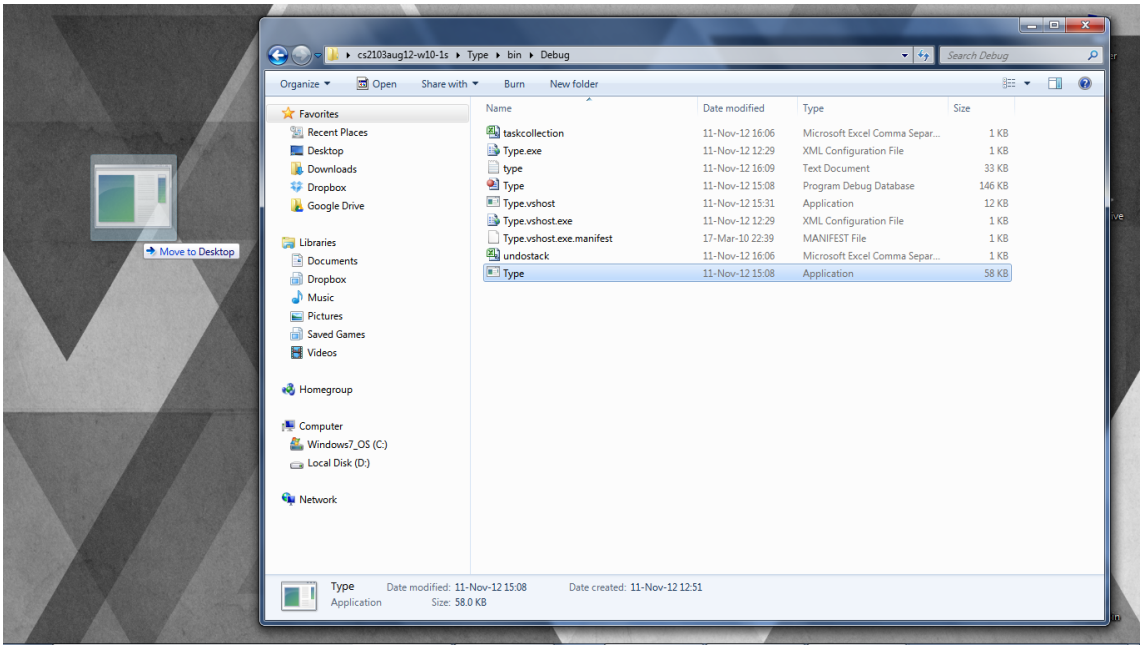
At the same time, we wanted to make using Type as unobtrusive as possible. It means that user interactions had to be thoroughly re-thought to reduce any unnecessary steps and time spent using the application.

This basic idea of simplicity has permeated the design of Type and we hope you like it as much as we do.

Getting Started

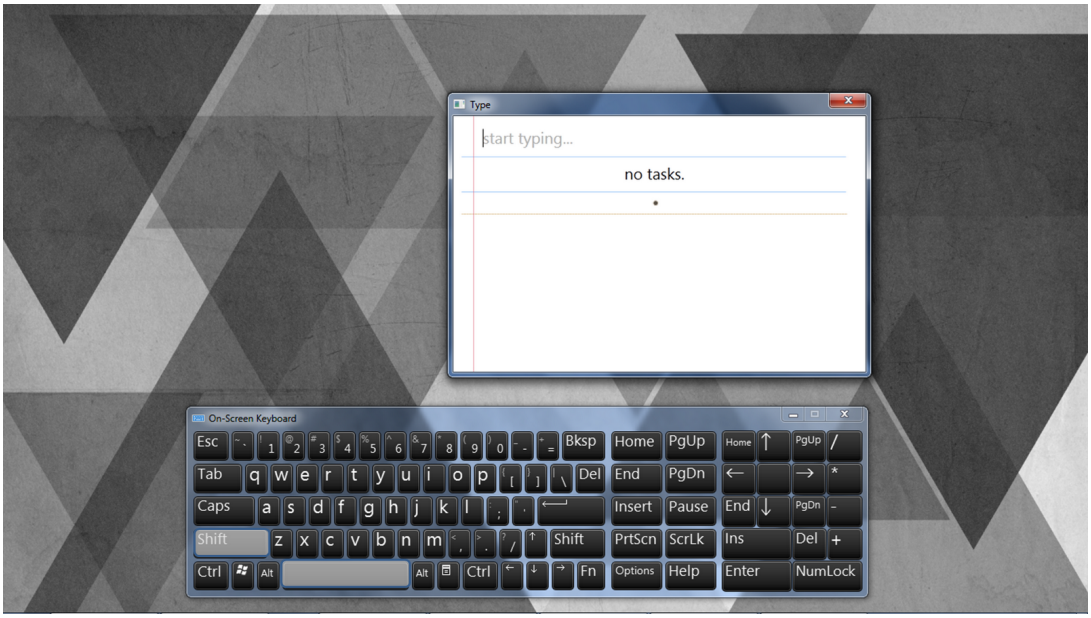
Installation

Drag and drop Type.exe to anywhere on the computer. Double-clicking on the file will install Type.



Launching Type

Pressing `Shift + Space` will show Type onscreen.



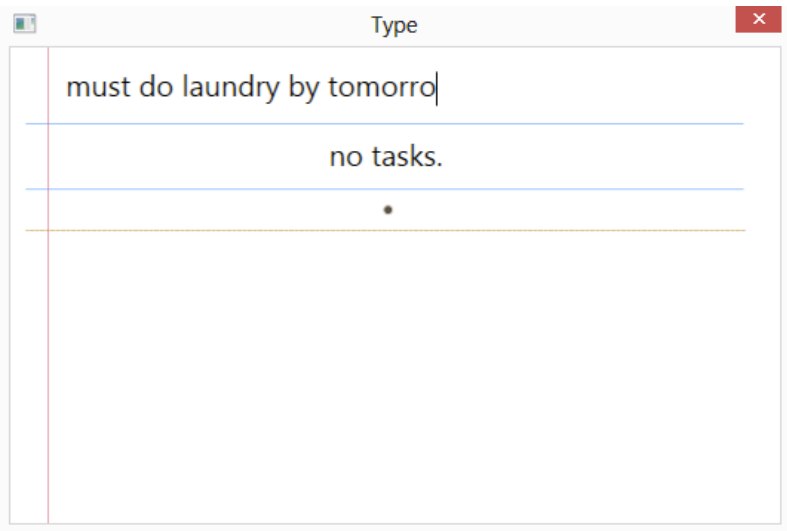
Dismissing Type

Pressing `Escape` will dismiss the main window of Type. Type, however, will continue to run in the background.

The main reason for this is the latency of launching our application. Launching and waiting for a task management application to start every time a task needs to be added is unacceptable. In order to truly replace physical alternatives, we needed to be instant. Hence our application is always waiting. Waiting for you to call it.

Adding a Task

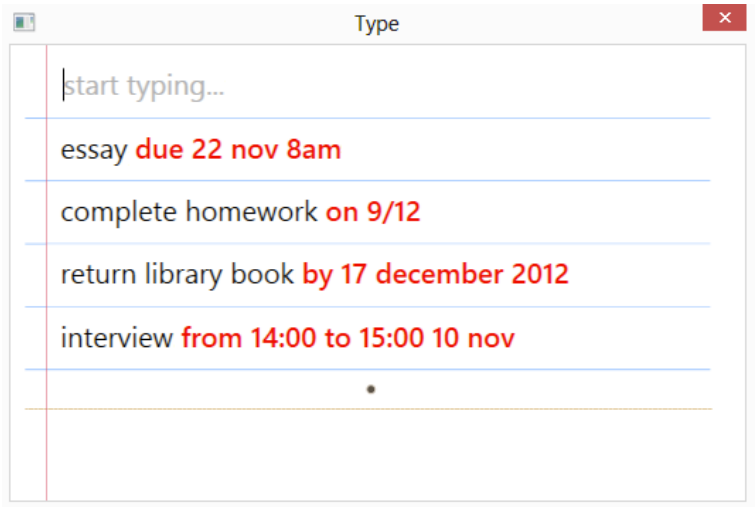
Simply start typing and hit `Enter/Return` to create a new task.



In addition to simple text, Type allows you to augment your task descriptions with more information by following several simple rules.

Date and Time

Type allows tasks to be augmented with date and time information. It supports a wide variety of time and date formats *(the full list of which is available at the end of this user guide)*.



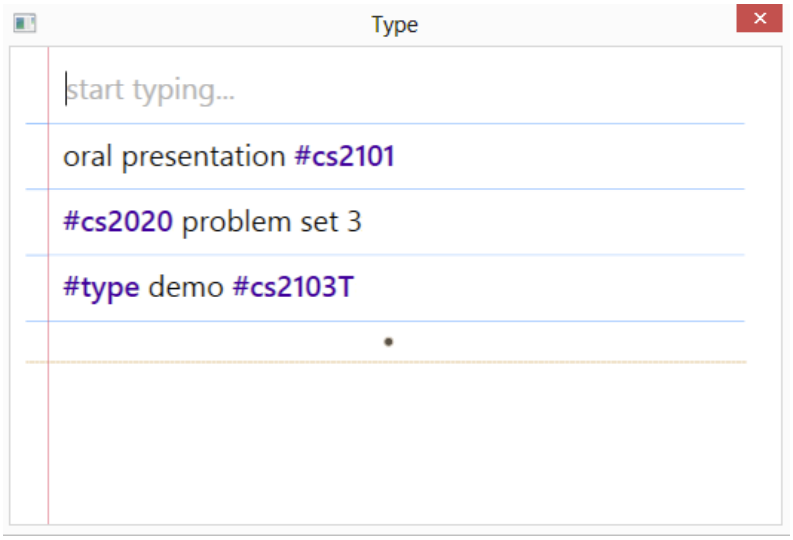
In addition, Date and Time information can be added with various keywords.

Tasks with a deadline can be marked with keywords like **by**, **due**, and **on <datetime>**.

Tasks that span a duration of time can be defined using the keywords **from <start datetime> to <end datetime>**.

Grouping Tasks

Type facilitates grouping of tasks by tagging them.



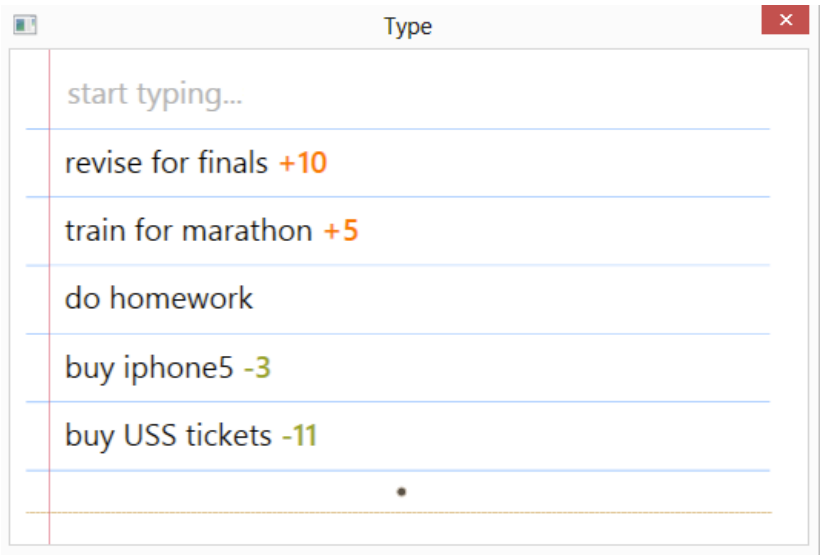
Use **#<tag>** to group tasks.

Tags can appear anywhere in the task description and you can use more than a single tag per task.

Tags may later be used for retrieving and searching groups of tasks as well as performing bulk actions on groups of tasks.

Relative Priorities

Besides Date and Time, Type allows you to assign relative priorities to tasks.



Adding **+<value>** or **-<value>** to the end of a task's description gives it increased or decreased priority.

Priorities can only appear at the end of a task.

The default priority for task with no priority stated is 0.

Basic User Actions

The Command Character (:)

Commands are prefixed with a **:**

The idea behind this was that we wanted to make it as trivial as possible for users to add tasks. We wanted users to be able to *just start typing*.

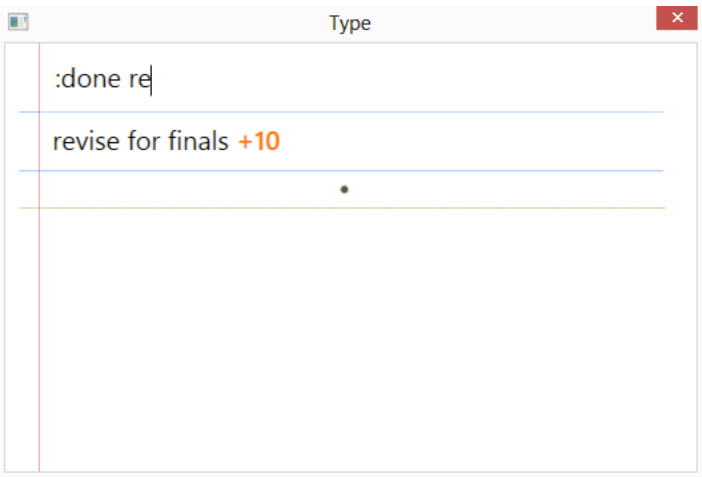
In addition, the command prefix allows us to unambiguously determine what your intentions are within the first two keystrokes. The allows us to subsequently guide you better, via autocomplete.

Auto-complete (tab)

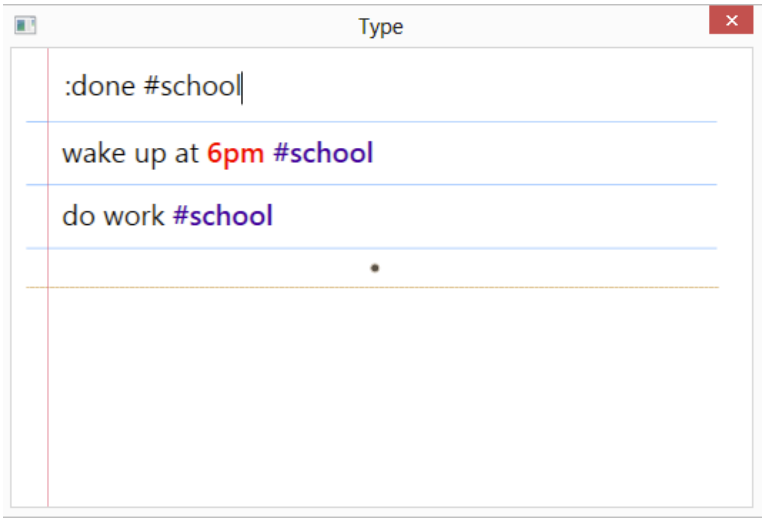
Where applicable, pressing `Tab` will auto-complete either a command or the first task description.

Done:

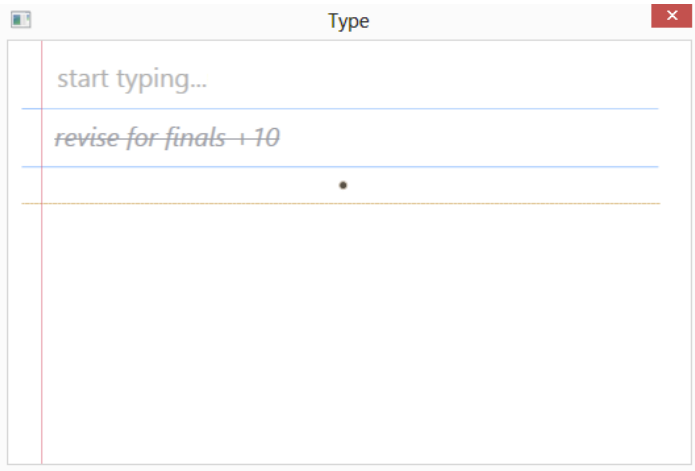
:done <text> completes the first task that start with that text.



:done #<hash tag> completes all tasks with a particular tag.

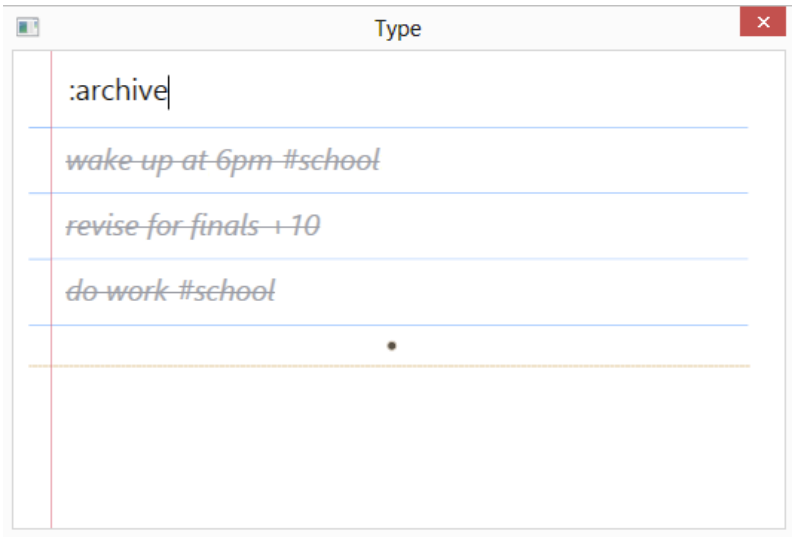


Completed tasks are displayed with their text struck-through.

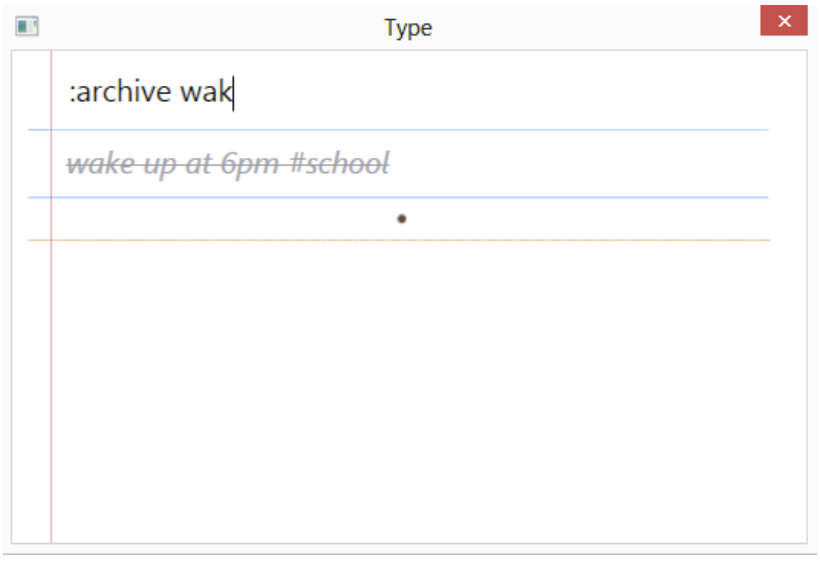


Archive

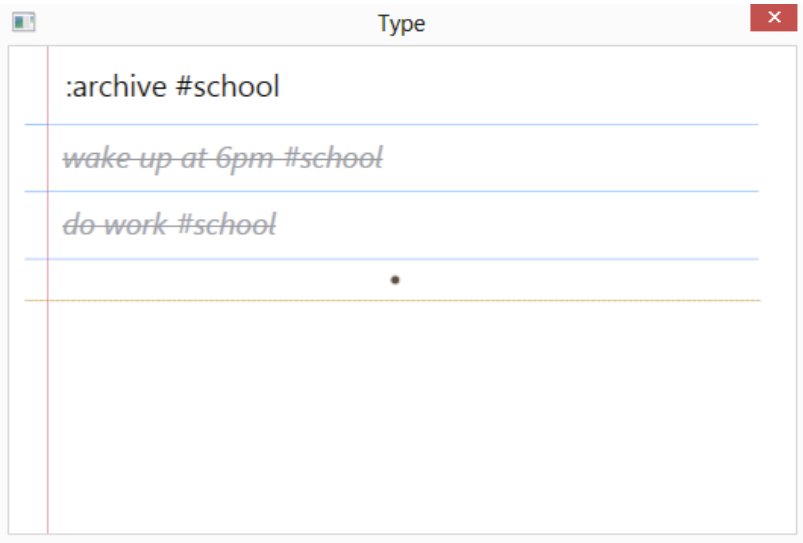
:archive will only archive completed tasks.



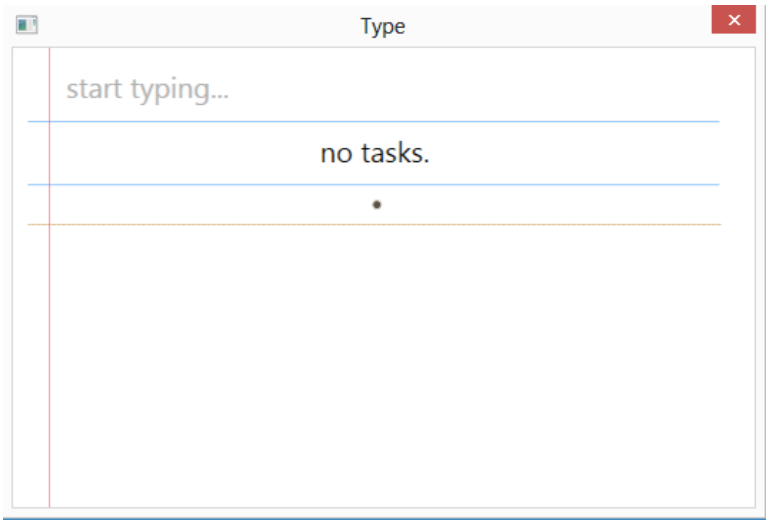
:archive <text> archives the first task or the selected task that start with that text.



:archive #<hash tag> archives all tasks marked with that hash tag.



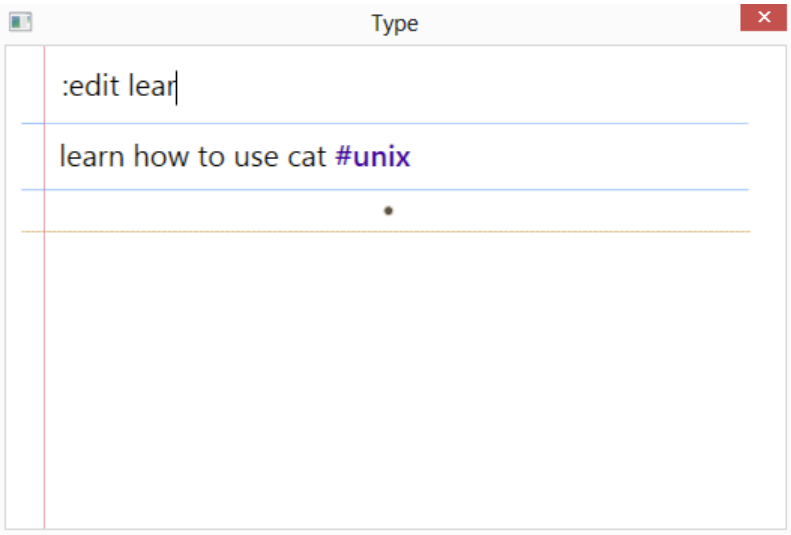
Archived tasks are not displayed unless the user searches for them explicitly.



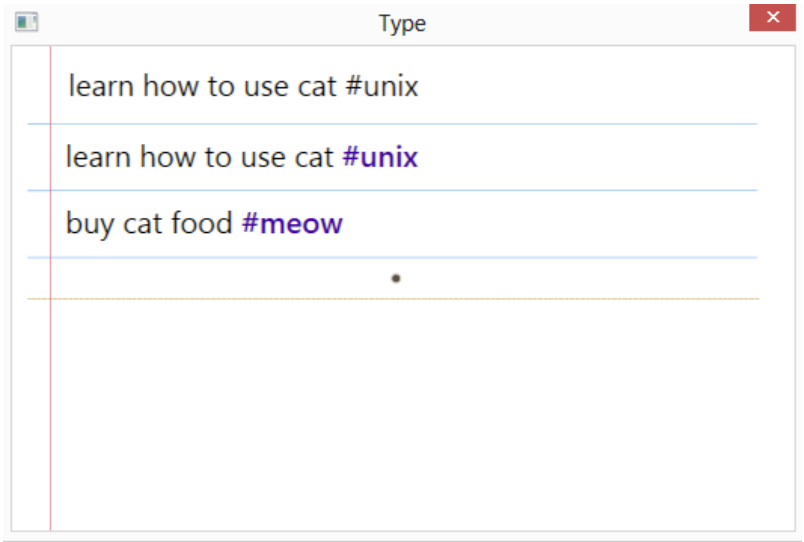
Advance User Actions

Edit

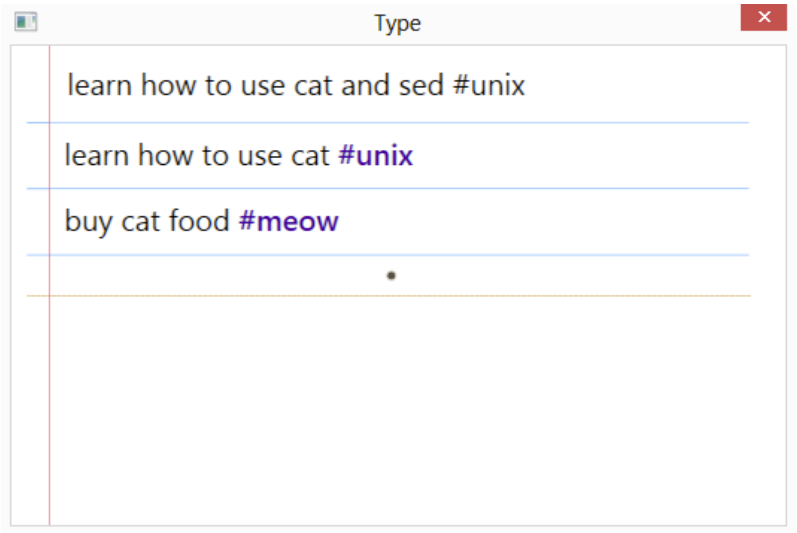
:edit <task> allows you to edit a task.



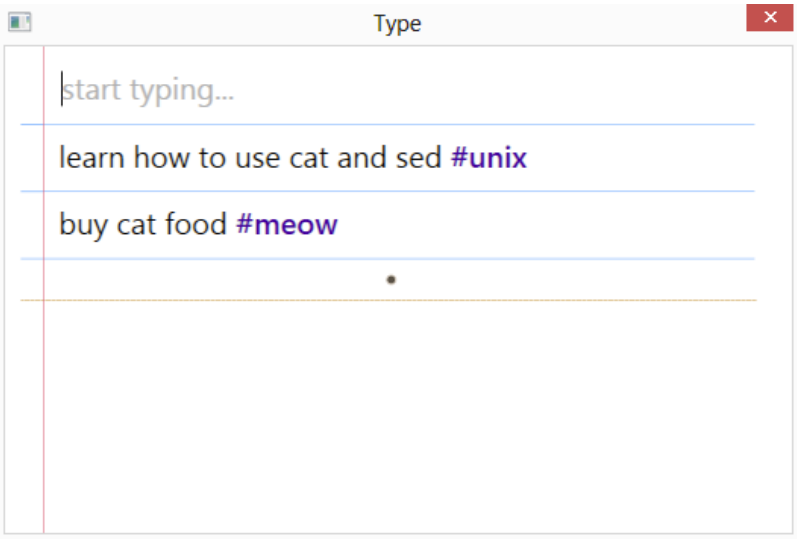
Hitting `Enter/Return` selects the first task and allows you to edit its task description.



The selected task description then placed in the input box for editing.

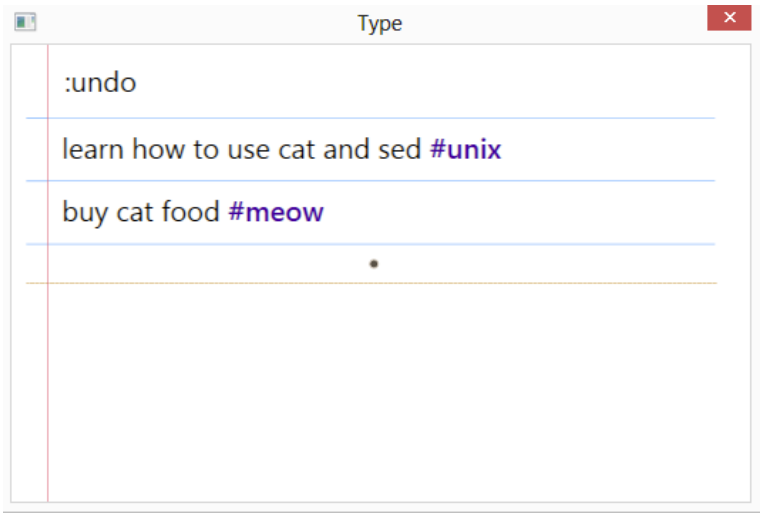


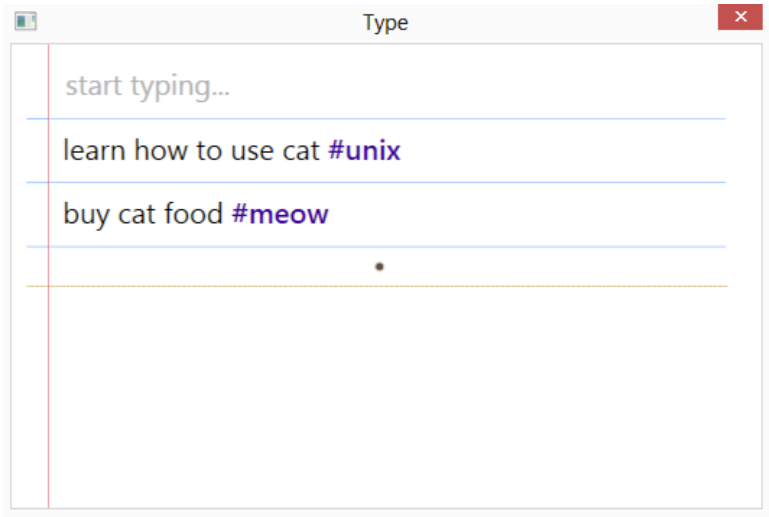
After editing the task description, hitting `Enter/Return` again saves your changes.



Undo

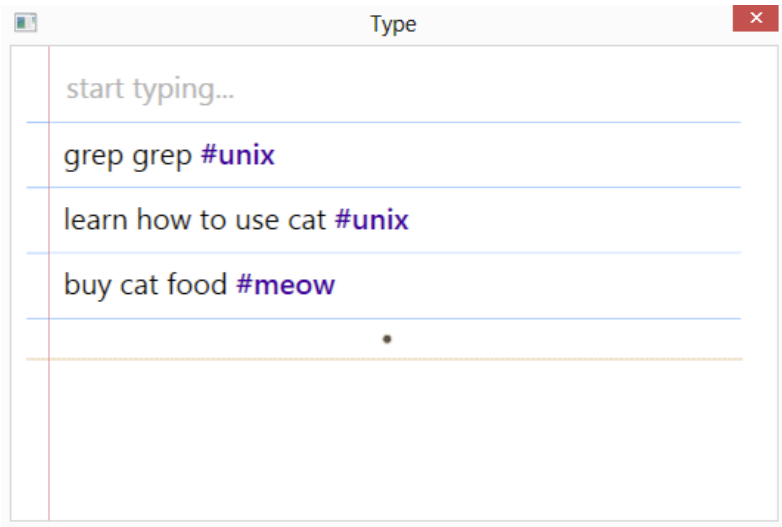
:undo rolls back last action. Type supports unlimited levels of undo.

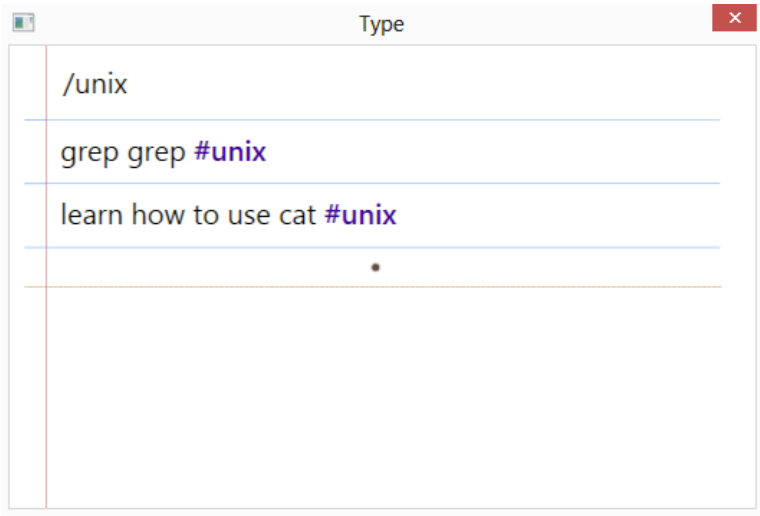




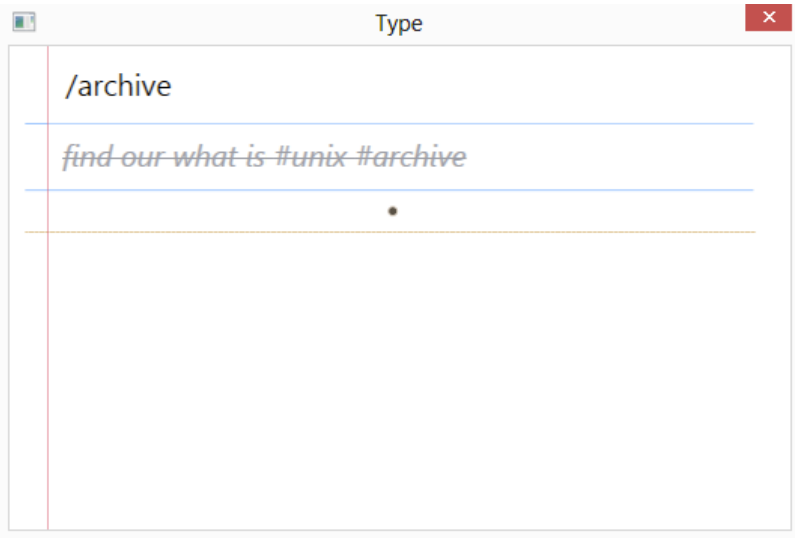
Search

/<tag name> filters tasks displayed and only displays tasks with the selected hashtag.



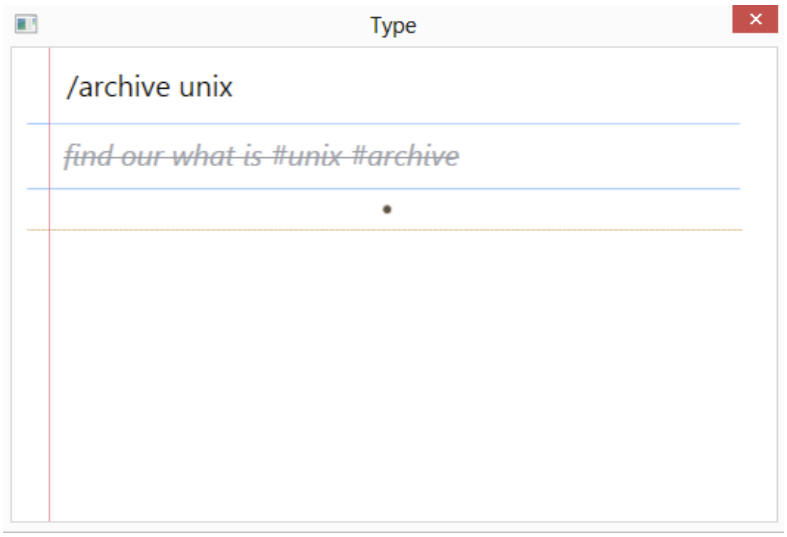


/archive shows all archived tasks.



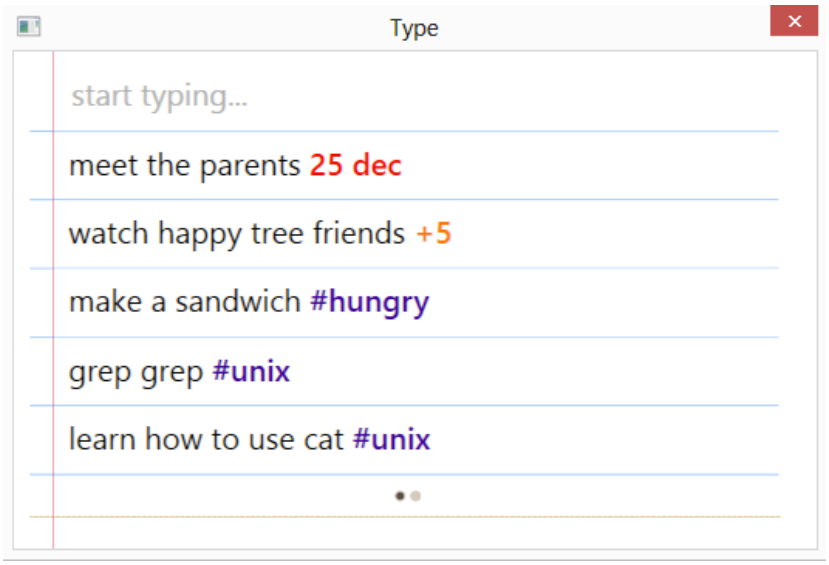
Search terms can be stacked.

/<tag 1> <tag 2> will return tasks strictly matching both tags.

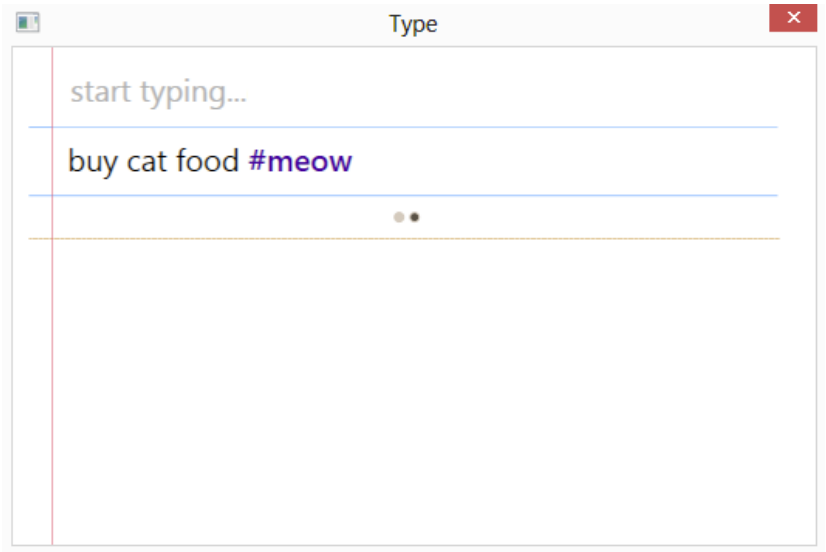


Pagination/Arrow Keys

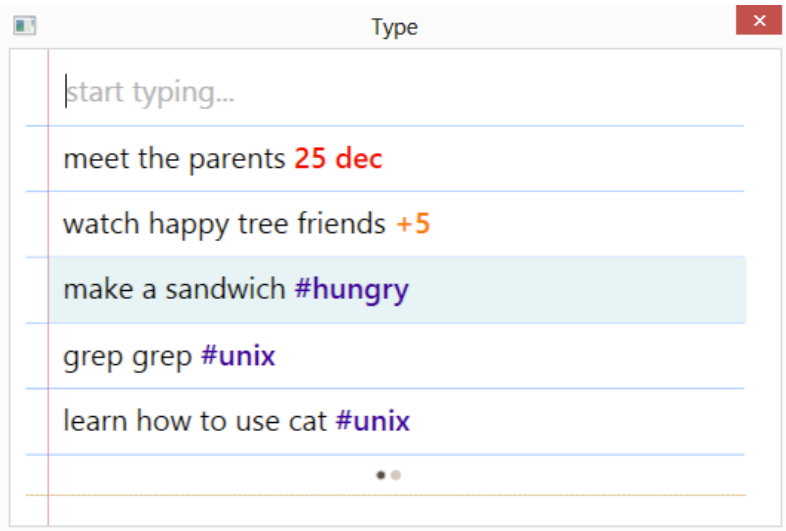
The `left` and `right` arrow keys allow horizontal page navigation.



The right arrow reveals more tasks. The left arrow toggles back to the first page.



The up and down arrow keys allow vertical page navigation.

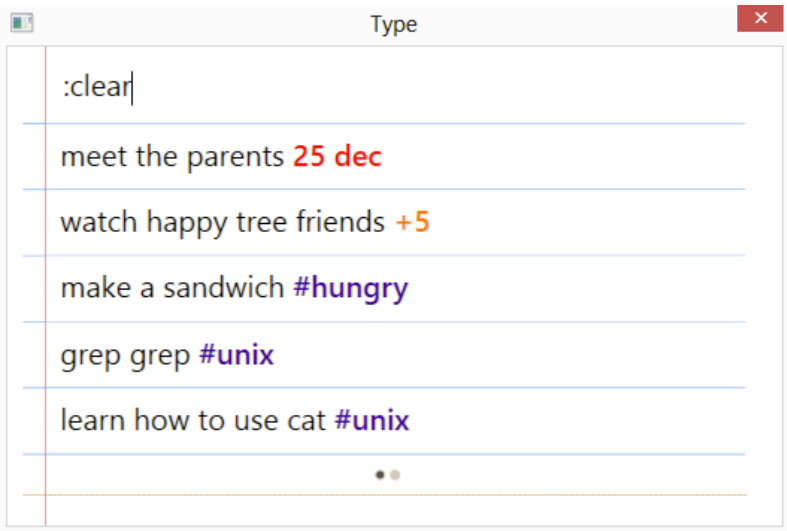


The “current” task is highlighted in blue.

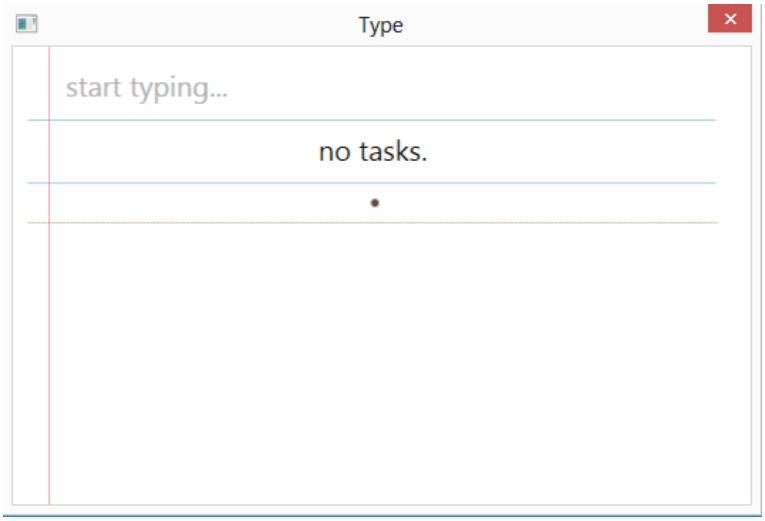
Pressing the `Enter/Return` key will select the highlighted task only when a command is entered in the input box.

Clear

:clear deletes the entire list of tasks.



Once deleted, it is not possible to recover the original list.



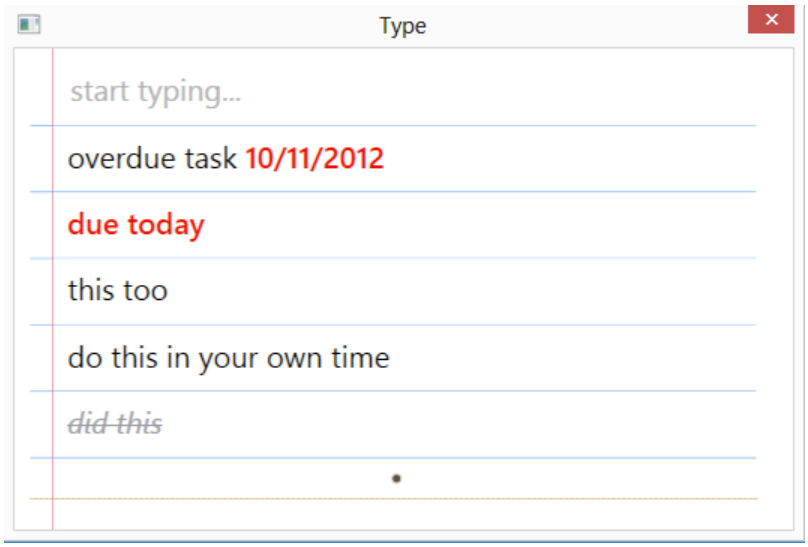
Help

? or **:help** brings up a summary table of the commands of Type.

Additional Information

Display Order of Tasks

Tasks are displayed in a natural manner, just like how you might write them down on a piece of paper.



- Overdue tasks appear right at the top, and the longer a task is overdue, the higher it appears.
- Tasks that are due today appear just below overdue tasks.
- Tasks that are only due in the future appear below tasks due today, but in a reverse order - the longer you have to complete it, the lower in the list it appears.
- Tasks without specified deadlines appear in the order they were entered, with later entries coming before earlier ones. These tasks are always displayed below overdue, due today, and future tasks.
- A task that is not yet active will appear towards the bottom of the list, and completed tasks appear right at the bottom with a strikethrough.

Reference

Commands

Create new task	<task>
Complete a Task	:done <task>
Complete all tasks with the hash tag	:done #<tag name>
Archive all completed tasks	:archive
Archive a single task	:archive <task>
Archive all tasks with the hash tag	:archive #<tag name>
Edit a Task	:edit <task>
Undo last action	:undo
Filter by hash tags	/<tag name> [<tag name>] ...
Show archived tasks	/archive <tag name> [<tag name>] ...
Help (Displays this list)	:help or ?

Task Attributes

Tagging	#<tag name>
Priority	+/-<value>

Date Formats

12 November 2012	12 nov 2012, 12/11/12, 12/11/2012
12 nov	12/11, 12 nov, 12 November
Today	Today, tdy
Tomorrow	tomorrow, tmr

Time Formats

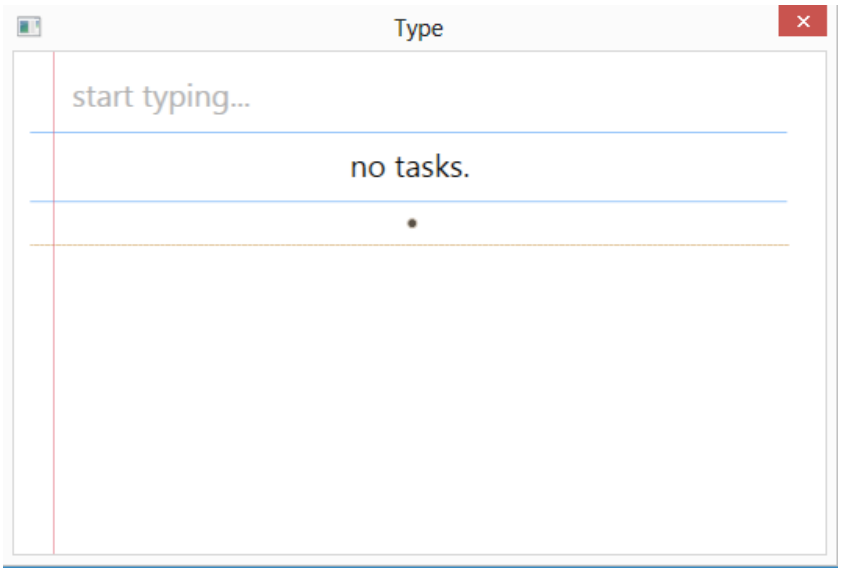
4pm today	4pm, 16:00
5:45am today	5:45, 5:45am

Keywords

... **from** <date/time> **to** <date/time>
... **by** <date/time>
... **due** <date/time>
... **on** <date/time>

type

Developer Guide



type 1

Table of Contents

type 2

Introduction

This is the developer guide for Type. If you are looking for the user guide, please visit this page instead: Type's user guide

Type is a desktop task manager that aims to help users prioritise what to do and when to do things.

This guide is meant as a primer for developers new to Type's codebase. It starts with a broad overview of the application architecture followed by increasingly specific sections of the various components.

Who this document is for

Knowledge of Programming is assumed.

Knowledge of Client-side Development, Microsoft Windows, C#, and the .NET framework is highly recommended but not necessary. Where possible, we have included links to references and other materials that we have found useful in understanding the content.

The Application Stack

Type targets .NET Framework 4, and is built on Windows Presentation Foundation (WPF). Type will run on any Windows computer with at least Windows Vista.

At the time of writing, Type will not run on Mono[1] on other platforms due to its dependency on WPF[2].

Type also makes Windows system calls to register hotkeys and to automatically launch at startup.

Getting Started

Development Tools

Visual Studio 2010

Type is developed using Visual Studio 2010[3].

[1] http://www.mono-project.com/
[2] http://www.mono-project.com/WPF
[3] http://msdn.microsoft.com/en-us/library/dd831853(v=vs.100).aspx

Mercurial

We use Mercurial[4] for version control.

Operating System

You need to be running a Windows machine with at least Windows Vista installed.

Downloading the Source

A copy of the source code may be obtained from:
http://code.google.com/p/cs2103aug12-w10-1s/

The are two branches that we maintain, the default 'branch' as well as the 'dev' branch.

The 'default' branch contains a stable version of our application.

Development occurs on the 'dev' branch, which gets merged into the 'default' branch when various milestones are achieved.

Building the Application

Here are instructions for building and running the application.

1. Open the solution file (*.sln) in Visual Studio.
2. Click on Build > Clean Solution in the menu bar.
3. Click on Build > Build Solution in the menu bar[5].

Testing and Development

Run the application in either Debug or Release configuration.

In Debug configuration you will need to press the shortcut `Shift + Space` to show the UI every time.

If run in Release configuration, the UI will show itself the first time Type is launched. Subsequently, you will need to press the shortcut `Shift + Space` to show the UI.

> **Note:**
> Type detects first launch when run in Release configuration by setting an `auto-start` registry key in `HKCU\Software\Microsoft\Windows\CurrentVersion\Run`. This key has the name of "`type-bin`". Developers are advised to clear these keys after testing in Release configuration.

[4] http://mercurial.selenic.com/

[5] Alternatively, press F6 in Visual Studio.

Overview

Architecture

Type follows the Model-View-Presenter (MVP) paradigm (Fig 1.1).

MVP is conceptually similar to the more commonly known Model-View-Controller (MVC), but different in that the View does not interact directly with the Model.

In Type, the Presenter is the logical entry point to the application, and mediates the flow of data between the View and the Model.

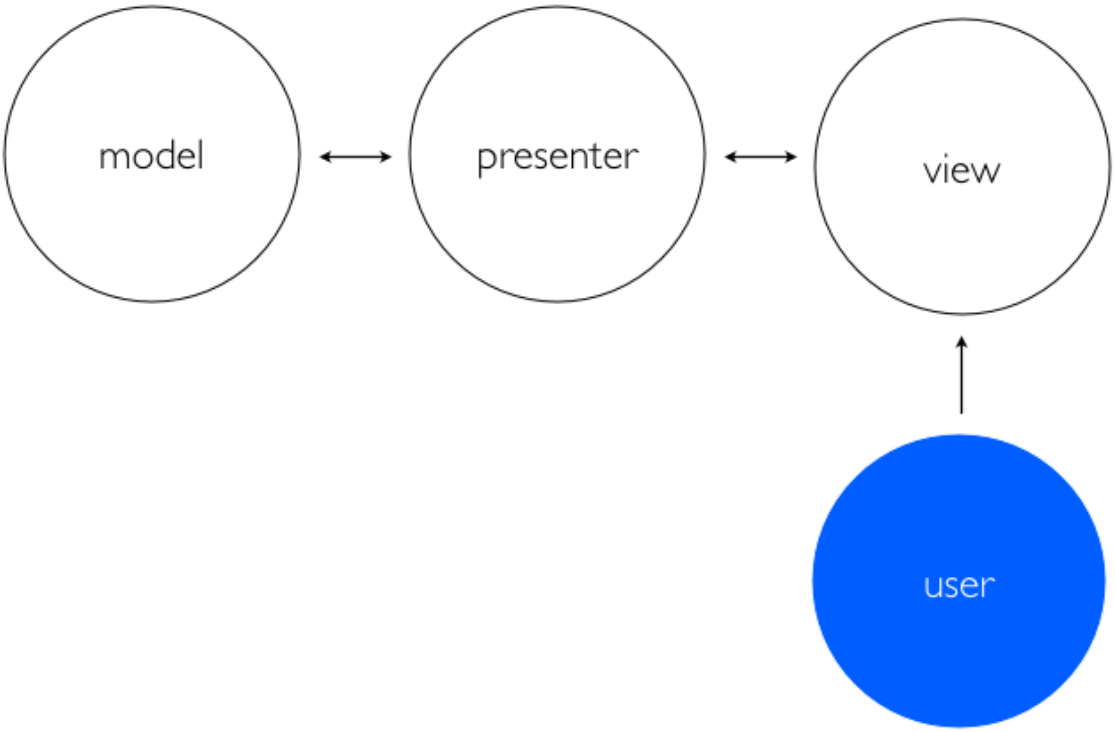


Fig 1.1 MVP Paradigm

MVP Components

Model

The model is mainly responsible for the processing and storing of the application's data.

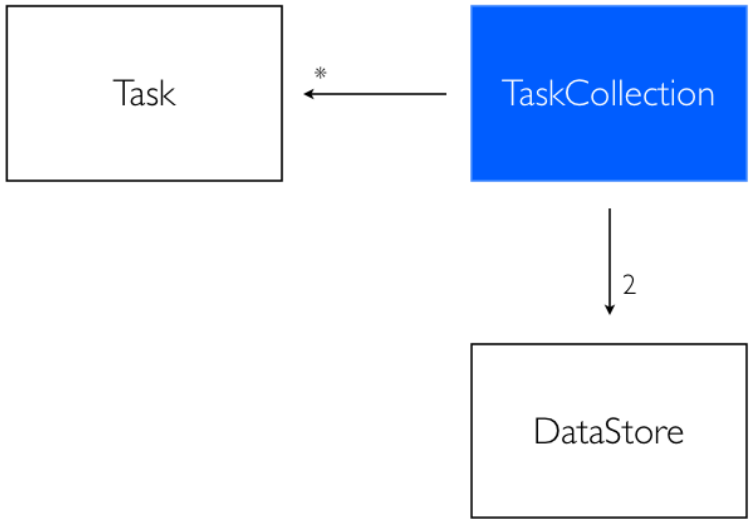


Fig 2.1 Model

It comprises of three main parts:

TaskCollection

The `TaskCollection` is responsible for coordinating the parts that comprise the Model.
It maintains a collection of `Task` Objects that can then be queried against by the presenter.

It interacts with the `DataStore` to persist information to disk as well as retrieving information from disk. This information is then passed along to the `Task` class to generate the subsequent list of `Task` Objects.

The `TaskCollection` uses two DataStores. One is used to store Task data while the other is used to persist an undo stack to disk.

Implementation Details

`TaskCollection` is backed by a List of `Task` Objects modelled as a Collection. It uses .NET's

Language-Integrated Query (LINQ) framework[6] to manage this collection. The underlying code supporting the API is a series of Lambda expressions[7] that query the backing Collection over LINQ.

Important APIs

- `public Task Create(string input)`
  Creates a new Task Object, and returns a reference to it.

- `public List<Task> GetNotArchiveTask()`
  Retrieves a list of all unarchived tasks.

- `public List<Task> FilterAll(string input)`
  Retrieves a list of all unarchived tasks that match a prefix specified by input.

- `public List<Task> GetTasksByHashTags(IList<string> hashTags)`
  Retrieves a list of all tasks that contain all of the hash tags supplied in hashTags.

Note:
Some of the following operations are undo-able. Where the parameter addToUndoStack is set to 'true', calling the method will store the current state of the target in the undo stack. A further call to Undo() will revert to that initially stored state.

- `public Task UpdateRawText(int id, string str, bool addToUndoStack = true)`
  Re-parses the Task Object specified by id with the new text in str.

- `public Task UpdateDone(int id, bool done, bool addToUndoStack = true)`
  Marks the Task Object specified by id as Done.

- `public Task UpdateArchive(int id, bool archiveStatus, bool addToUndoStack = true)`
  Marks the Task Object specified by id as Archived.

- `public void ArchiveAll()`
  Archives all Task Objects that are marked as Done. This method is always undo-able.

- `public void ArchiveAllByHashTags(IList<string> hashTags)`
  Archives all Task Objects that contain all of the hash tags supplied in hashTags. This method is always undo-able.

[6] http://en.wikipedia.org/wiki/Language_Integrated_Query
[7] http://msdn.microsoft.com/en-us/library/bb397687.aspx

- `public void UpdateDoneByHashTags(IList<string> hashTags)`
  Marks all Task Objects that contain all of the hash tags supplied in hashTags as Done. This method is always undo-able.

- `public void Clear()`
  Removes all Task Objects from the Task Collection. Also resets the undo stack.

- `public void Undo()`
  Reverts the state of the Task Collection to the top state in the Undo stack.

- `public void ReparseAll()`
  Re-parses all Task Objects in the Task Collection.

Note:
We re-parse the Task Collection at midnight (when the date changes) to update Task Objects whose start and/or end dates were stored as relative markers (such as Today, and Tomorrow).

DataStore

The DataStore is used as our application's persistence layer. It provides a convenient way to store rows of data and subsequently to retrieve, update, or delete them.

Implementation Details

Data is stored in the comma-separated values (CSV) format[8]. Each line in the CSV file represents a row. Fields in a row are delimited by a comma. Delimiters are escaped with a backslash. The parameter is passed by references, but data written to the file will not be affected.

Know Limitation:
During regular operation, pressing Enter/Return will execute a command rather than append a newline character sequence. However, line breaks ('\n', '\r', '\r\n') are not escaped. If input contains line breaks, the application's behavior will become non-deterministic.

Important APIs

- `public DataStore(string fileName)`
  Creates a DataStore object backed by the CSV file specified by fileName. If the file does not exist, it is created. Otherwise, it is opened and parsed.

[8] CSV is human-readable and lightweight, making it ideal for an application like ours that aims to consume as few resources on the host system as possible. http://en.wikipedia.org/wiki/Comma-separated_values

- `public int InsertRow(List<string> row)`
  Inserts a row represented by a list of fields into the DataStore.

- `public void ChangeRow(int index, List<string> row)`
  Changes a row specified by `index` by updating its fields with a new list represented by `row`.

- `public List<string> Get(int index)`
  Returns the list of fields associated with a particular row in DataStore specified by index.

- `public Dictionary<int, List<string>> Get()`
  Returns a representation of the state of the DataStore as a Dictionary that maps a row's index to its corresponding list of fields. The return value is not synchronized with the Task Collection. Changes made in either object will not be reflected to the other.

Note:
The returned Dictionary is not synchronized with the DataStore. Changes on either object do not propagate to the other object.

- `public void DeleteRow(int index)`
  Permanently removes a row specified by index from the DataStore. Its associated fields are also deleted.

- `public void ClearFile(String fileName)`
  Deletes the file specified by fileName, and creates an empty file with the same name.

Note:
The method `ClearFile()` is destructive and non-undoable. It should be used when a reset of the application's state is desired. `ClearFile()` does not clear the log file.

Task

The `Task` Class is responsible for parsing the user's task description into useful meta information. This information is used by the `TaskCollection` to organise and sort the Tasks.

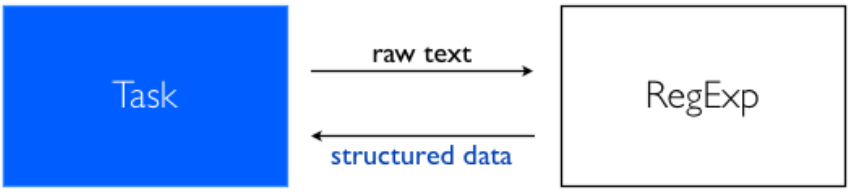


Fig 2.2 Task

Implementation Details

The Task Class delegates to the `RegExp` Class for parsing of the user's raw text.

The `RegExp` class returns structured data that is then used to set the various properties within the task object.

Important APIs

- `public void Parse()`
  Bootstraps the various properties of a task by calling various other functions and methods.

- `public List<string> ToRow()`
  Returns a standardised representation of the task, for use in persisting tasks to disk.

Properties

Tasks Objects expose several important properties that are expected and used by other classes[9].

> **Note:**
> "Properties" in C# are syntactic sugar for accessors and mutators in other object oriented languages. Each of the following properties is backed by a private field in the Task Object itself, and is modified through implicit accessors and mutators. The access specifier of the property refers to the access level of its implicit accessors and mutators, rather than the backing field.

- `public int Id`

[9] In rendering, filtering and sorting etc.

Uniquely identifies tasks, both in memory as well as on disk.

- `public string RawText`
  The raw input from the user.

- `public DateTime LastMod`
  Stores the last-Modified DateTime of the task. Used to unambiguously determine relative DateTimes[10].

- `public int Priority`
  The arbitrary priority of the task.

Note:
The priority of a Task must be within `-256` and `255` (inclusive).

- `public DateTime Start`
  The Start DateTime of the task (if exists).

- `public DateTime End`
  The End DateTime of the task (if exists).

- `public bool Done`
  Indicates if a particular task is done.

- `public bool Archive`
  Indicates if a particular task has been archived.

- `public IList<Tuple<string, ParsedType>> Tokens`
  An augmented representation of RawText with the various tokens tagged with a ParseType to indicate their semantic meaning. This is used in the rendering of the tasks description.

[10] Eg. Today, Tomorrow etc.

RegExp

The `RegExp` class is responsible for extracting useful information from plain text.

Fig 2.3 RegExp

Implementation Details

It relies heavily on Regular Expression[11] and does pattern matching for Dates, Times, Priorities, and Hashtags.

Important APIs

- `public static Tuple<string, int> Priority(string input)`
  Returns a tuple containing the matching string and the arbitrary priority that the user assigned to the task, returns null otherwise.

- `public static List<string> HashTags(string input)`
  Returns a list of matching hashtags within the input string.

- `public static Tuple<string, DateTime?, DateTime?> GetDateTime(string input, DateTime today)`
  Returns a tuple of the matching string, the start/end datetime where appropriate, returns `null` otherwise.

[11] http://en.wikipedia.org/wiki/Regular_expression

View (MainWindow)

The View comprises mainly of the User Interface (UI). It is responsible for displaying tasks to the user as well as delegating user interactions to the Presenter.

It consists of a single class, `MainWindow`, split logically into three partial classes - `Drawing`, `Navigation`, and `UILogic`.

Fig 3.1 View

Implementation Details

The `MainWindow` is implemented as a single Windows Presentation Foundation[12] (WPF) window.

Important APIs

- `public MainWindow(FilterSuggestionsCallback GetFilterSuggestions, GetTasksCallback GetTasks, GetTasksByHashTagCallback GetTasksByHashTag)`
  Creates an instance of MainWindow with the specified callbacks.

Note:
Type's Presenter and View interact through a system of delegates to reduce bidirectional dependencies.

- `public void ForceRedraw()`
  Forces the UI to synchronize itself with the Presenter, and re-render all Task Objects.
- `public event RequestExecuteEventHandler RequestExecute;`
  Event thrown when the UI wants a command to be executed. In Type, the Presenter listens for this event and takes over execution when it is thrown.

[12] http://en.wikipedia.org/wiki/Windows_Presentation_Foundation

Presenter

The Presenter acts as a bridge between Type's User Interface and Model. It comprises of three main Classes.

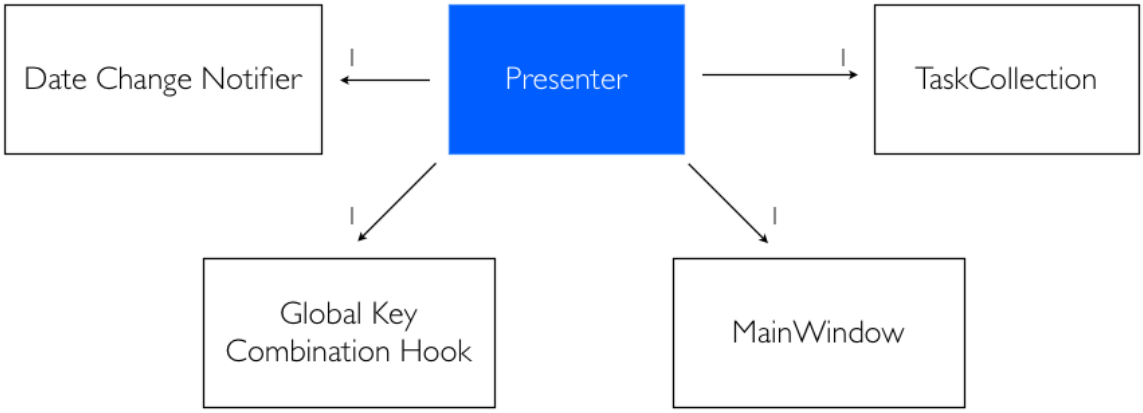


Fig 4.1 Presenter

Presenter (Class)

The `Presenter` *Class* is the logical entry point of the application and is responsible for creating resources and setting them to their initial states. It listens to Command `RequestExecute` events from the UI, and makes execution decisions based on the context and content of a command.

`Presenter` listens to the `ShortcutPressed` event emitted by the Global Key Combination Hook. When the event occurs, it displays Type's UI.

`Presenter` also listens to the `DateChange` event emitted by the Date Change Notifier. When the date changes, it instructs the Task Collection to re-parse its contents. The purpose of this action is to update Task Objects that may have start and/or end dates stored as relative markers like "today", or "tomorrow".

Implementation Details

When the `Presenter` *Class* is initialized, the order in which resources are created is important:

1. The Model, abstracted in the Task Collection must be the first Object created.
2. The Main Window can only be initialized after the Model.
3. The Global Key Combination Hook should only be initialized after the Main Window.
4. Finally, the Date Change Notifier must be initialized last.

On initialization, the Class also assigns methods that allow the User Interface to retrieve Task

Objects to common delegates.

The `Presenter` *Class* runs every command request received through a decision structure to determine what actions to follow up with. It is the responsibility of the User Interface to retrieve an updated list of Task Objects after a command has been executed.

Important APIs

The `Presenter` *Class* does not expose an API.

Date Change Notifier

The Date Change Notifier emits a `DateChange` event when the system date changes.

Implementation Details

On initialization, Date Change Notifier calculates the number of seconds to one second past midnight. It then spawns a system timer on a separate thread that sleeps for the calculated number of seconds. When the timer elapses, Date Change Notifier emits the `DateChange` event. The number of seconds till the next date is recalculated, and the timer is put to sleep again.

Note:
We calculate to one second past midnight to introduce a tolerance for processing delays on the user's computer.

Known Limitation:
Date Change Notifier is not able to detect the case where a user manually changes his/her system date via the Windows Control Panel, or when Windows automatically updates the system date when a user travels between time zones.

Important APIs

- `public DateChangeNotifier()`
  Instantiates a `DateChangeNotifier` Object.

- `public event DateChangeEventHandler DateChange;`
  Event thrown when the system date changes. In Type, the Presenter listens for this event and takes over execution when it is thrown.

Global Key Combination Hook

The Global Key Combination Hook listens to the `Shift + Space` combination globally, and emits a `ShortcutPressed` event when the combination is detected.

Implementation Details

`GlobalKeyCombinationHook` sets a callback with the Windows API. Windows executes this

callback when the combination is pressed, and `GlobalKeyCombinationHook` emits the `ShortcutPressed` event.

Note:
The Windows API requires a `hWnd` handle[13] on an active window in order to set a callback for a key combination. On initialization, `GlobalKeyCombinationHook` temporarily an invisible UI window to create a hWnd.

Known Limitation:
In this version, if another application has previously reserved the `Shift + Space` combination, Type's shortcut key will not work. They may be no alternative way to force Type to show its User Interface.

Important APIs

- `public GlobalKeyCombinationHook(Window targetWindow, uint modifier, uint vkey)`
  Creates a global key combination hook on the combination described by `modifier` and `vkey`. `targetWindow` is a reference to the window for which a `hWnd` is created.

- `public GlobalKeyCombinationHook StartListening()`
  Begins listening to key combinations.

- `public event ShortcutPressedEventHandler ShortcutPressed;`
  Event thrown when the Shift + Space shortcut key is pressed. In Type, the Presenter listens for this event and takes over execution when it is thrown.

- `public GlobalKeyCombinationHook StopListening()`
  Stops listening to key combinations.

[13] http://msdn.microsoft.com/en-us/library/windows/desktop/aa383751.aspx#hwnd

Helpers

Command

The Command *Class* contains general methods to parse text into Type Commands, as well as string enumerations for various accepted commands and tokens. It provides autocomplete functionality on commands.

Important APIs

- `public static string TryComplete(string partial)`
  Tries to complete a partial command. If there are no possible completions, the method returns an empty string. Otherwise, the method returns the remaining text to complete the partial command.

- `public static Command Parse(string input)`
  Parses input and converts it to a Type Command. If there are no valid matches, the returned Command is of the type "`Invalid`". Otherwise, the returned Command will contain the type of the command, and the context of the input.

Installer

The Installer *Class* checks if the program is launched for the first time. It also sets Type to launch at startup.

Important APIs

- `public static bool IsInstalled()`
  Returns '`true`' if Type has already been set to auto-start on the local machine, '`false`' otherwise.

- `public static void EmbedOnFirstRun()`
  If it is already not installed, writes the registry key that allows Type to auto-start.

Note:
Type maintains its install state by setting a registry key in `HKCU\Software\Microsoft\Windows\CurrentVersion\Run.`

Known Limitation:
In this version, if Type is initially launched from removable media, the auto-start pointer will refer to a non-persistent location. When the computer is rebooted without the same removable media mounted to the same drive letter, Type will fail to start.

Notable Algorithms

Sorting

The motivation behind our sorting algorithm is to avoid the following alternatives that we consider to be sub-optimal:

- Lengthy list of `If-Else` statements when comparing multiple data fields in two tasks
- Using a more complex stable sort and sorting multiple times per query
- Simplifying the sort order

Tasks are sorted in the following order:

1. Overdue Tasks; Tasks that are overdue for a longer period appear higher.
2. Tasks Due Today,
3. Deadlined Tasks that are due in the future,
4. Non-deadlined Tasks,
5. Tasks that have not 'Started',
6. Completed Tasks.

Within each primary sort criteria, Tasks that have a higher priority appear higher on the list. If two tasks are otherwise identical, they are displayed in the order in which they were entered, with the later Task appearing higher.

Implementation Details

We represent each Task Object as a 64-bit Two's Complement[14] integer. Each sort criteria is represented by an *n*-bit field within these 64-bits. In general, higher priority sort criteria occupy more significant bits in the resultant hash. In our implementation, the Most Significant Bit (MSB) is functionally equivalent to a sign bit.

We then sort our Task Collection in Descending Order based on the natural ordering of integers using C#'s built-in sorting functionality. Due to the way we represent each Task Object, this action automatically orders Tasks according to the order specified above.

You should read the documentation for the method `public long DefaultOrderHash()` in `Task.cs` for exact implementation details.

[14] http://en.wikipedia.org/wiki/Two's_complement

Note:
As we are sorting Tasks based on a positional notation, our algorithm has the advantage of being able to utilize linear time non-comparison sorts with some modifications.

Known Limitation:
Packing all the information of a Task Object into 64-bits is necessarily lossy. The allocated length of each field limits its range:

- We only support Tasks that are at most about 127.68 years overdue or in the future.
- The priority of a Task must be within -256 and 255 (inclusive).
- We only support a maximum of about 33.5 million Tasks.

Additional Information

Unit Tests

Unit Tests for various components of Type may be found in individual Test Projects in the Solution. New tests should be placed in the appropriate Test Projects, or in a new Test Project if they cover components that are not already unit tested.

The following list enumerates current Test Projects in Type.

- Command Object Tests
- Data Store Tests
- Language Tests
- Default Sort Order Tests
- Model Tests

Developers should ensure that all unit tests pass before pushing code to the "`default`" branch of the repository.

Logging

Type logs interactions with the Presenter in a file "`type.log`". Each log entry is prefixed with a timestamp with an accuracy in the order of seconds. Developers may review the log to aid in debugging, or to analyse performance.

Known Limitation:
Type does not currently log application shutdown information. You will not be able to determine how and/or when the process hosting Type is terminated.