# M269

## Summaries

## Contents



# 1 Introduction

These notes give some summaries or *cheatsheets* for various parts of M269 and related topics — it is not intended for any particular audience other than the author and includes some topics which may not be directly in M269

# 2 Jupyter Notebook

- **Jupyter** jupyter.org
- **Jupyter Notebook Documentation** jupyter-notebook.readthedocs.io
- **IPython** ipython.org
- **IPython Documentation** ipython.readthedocs.io
- **Anaconda** anaconda.com
- **Anaconda Cloud** anaconda.org
- **Conda Docs** docs.conda.io
- **Conda User Guide** conda.io/projects/conda/en/latest/user-guide

## 2.1 Notebook Terminology

- **Notebook Dashboard** When you launch *jupyter notebook* the first page that you encounter is the Notebook Dashboard.
- **Notebook Editor** Once a Notebook is selected to edit, the Notebook will open in the Notebook Editor
- See User interface components

## 2.2 User Interface: Key Bindings

- The tables below give *key bindings* for some of the Command Mode and Edit Mode commands
- A more complete set of commands for Command Mode can be seen from Help > Edit Keyboard Shortcuts

# Jupyter Notebook Key Bindings — macOS

## Modifier & Other Special Keys — macOS

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Cmd ⌘ | Command | ⇧ | Shift | ␣ | Space | ↑ | Up |
| Ctrl ^ | Control | ⇪ | Caps Lock | ⇥ | Tab | ↓ | Down |
| Alt ⌥ | Option | ⏎ | Return | ⌦ | Del | → | Right |
| | | ⌅ | Enter | ⌫ | Back Del | ← | Left |

## Jupyter Notebook — Command Mode — macOS

| | | | |
|---|---|---|---|
| F | Find & replace | Y | Change cell to code |
| ⏎, ⌅ | Enter edit mode | M | Change cell to markdown |
| ⌘+⇧+F | Open command palette | R | Change cell to raw |
| ⌘+⇧+P | Open command palette | 1 | Change cell to heading 1 |
| P | Open command palette | 2 | Change cell to heading 2 |
| ⇧+⏎, ⇧+⌅ | Run cell, select below | 3 | Change cell to heading 3 |
| ^+⌅ | Run selected cells | 4 | Change cell to heading 4 |
| ⌥+⏎, ⌥+⌅ | Run cell, insert below | 5 | Change cell to heading 5 |
| ⇧+K, ⇧+↑ | Extend selected cells above | 6 | Change cell to heading 6 |
| ⇧+J, ⇧+↓ | Extend selected cells below | K, ↑ | Select cell above |
| ⌘+A | Select all cells | J, ↓ | Select cell below |
| A | Insert cell above | B | Insert cell below |
| X | Cut selected cells | C | Copy selected cells |
| ⇧+V | Paste cells above | V | Paste cells below |
| Z | Undo cell deletion | D , D | Delete selected cells |
| S, ⌘+S | Save and Checkpoint | ⇧+M | Merge selected cells |
| L | Toggle line numbers | ⇧+L | Toggle all cells line numbers |
| O | Toggle selected cells output | ⇧+O | Toggle selected cells output scrolling |
| I , I | Interrupt the kernel | 0 , 0 | Restart the kernel (with dialog) |
| Esc, ⎋, Q | Close the pager | H | Show keyboard shortcuts |
| ⇧+␣ | Scroll notebook up | ␣ | Scroll notebook down |

## Jupyter Notebook — Edit Mode — macOS

| | | | |
|---|---|---|---|
| ⇥ | Code completion or indent | ⌘+⇧+Z | Redo |
| ⇧+⇥ | Tooltip | ⌘+⇧+U | Redo selection |
| ⌘+] | Indent | ^+K | Emacs-style line kill |
| ⌘+[ | Dedent | ⌘+⌫ | Delete line left of cursor |
| ⌘+A | Select all | ⌘+⌦ | Delete line right of cursor |
| ⌘+Z | Undo | ^+M | Enter command mode |
| ⌘+/ | Comment | esc, ⎋ | Enter command mode |
| ⌘+D | Delete whole line | ⌘+⇧+F | Open command palette |
| ⌘+U | Undo selection | ⌘+⇧+P | Open command palette |
| Insert (?) | Toggle overwrite flag (?) | ⇧+⏎, ⇧+⌅ | Run cell, select below |
| ⌘+↑ | Go to cell start | ^+⌅ | Run selected cells |
| ⌘+↓ | Go to cell end | ⌥+⏎, ⌥+⌅ | Run cell, insert below |
| ⌥+← | Go one word left | ^+⇧+Minus | Split cell at cursor |
| ⌥+→ | Go one word right | ⌘+S | Save and Checkpoint |
| ⌥+⌫ | Delete word before | ↓ | Move cursor down |
| ⌥+⌦ | Delete word after | ↑ | Move cursor up |

# Jupyter Notebook Key Bindings

Windows

## Modifier & Other Special Keys

Windows

| | | | |
|---|---|---|---|
| Menu Win Menu | Shift | Space | Up |
| Ctrl Control | Caps Lock | Tab | Down |
| Alt Alt | Return | Del | Right |
| | Enter Enter | Back Del | Left |

## Jupyter Notebook Command Mode

Windows

| | |
|---|---|
| F Find & replace | Y Change cell to code |
| ↵, Enter Enter edit mode | M Change cell to markdown |
| ^ + ⇧ + F Open command palette | R Change cell to raw |
| ^ + ⇧ + P Open command palette | 1 Change cell to heading 1 |
| P Open command palette | 2 Change cell to heading 2 |
| ⇧ + ↵, ⇧ + Enter Run cell, select below | 3 Change cell to heading 3 |
| ^ + Enter Run selected cells | 4 Change cell to heading 4 |
| Alt + ↵, Alt + Enter Run cell, insert below | 5 Change cell to heading 5 |
| ⇧ + K, ⇧ + ↑ Extend selected cells above | 6 Change cell to heading 6 |
| ⇧ + J, ⇧ + ↓ Extend selected cells below | K, ↑ Select cell above |
| ^ + A Select all cells | J, ↓ Select cell below |
| A Insert cell above | B Insert cell below |
| X Cut selected cells | C Copy selected cells |
| ⇧ + V Paste cells above | V Paste cells below |
| Z Undo cell deletion | D , D Delete selected cells |
| S, ^ + S Save and Checkpoint | ⇧ + M Merge selected cells |
| L Toggle line numbers | ⇧ + L Toggle all cells line numbers |
| O Toggle selected cells output | ⇧ + O Toggle selected cells output scrolling |
| I , I Interrupt the kernel | 0 , 0 Restart the kernel (with dialog) |
| Esc, Q Close the pager | H Show keyboard shortcuts |
| ⇧ + ␣ Scroll notebook up | ␣ Scroll notebook down |

## Jupyter Notebook Edit Mode

Windows

| | |
|---|---|
| ⇥ Code completion or indent | ^ + ⇧ + Z Redo |
| ⇧ + ⇥ Tooltip | ^ + ⇧ + U Redo selection |
| ^ + ] Indent | ^ + K Emacs-style line kill |
| ^ + [ Dedent | ^ + ⌫ Delete line left of cursor |
| ^ + A Select all | ^ + ⌦ Delete line right of cursor |
| ^ + Z Undo | ^ + M Enter command mode |
| ^ + / Comment | esc, Esc Enter command mode |
| ^ + D Delete whole line | ^ + ⇧ + F Open command palette |
| ^ + U Undo selection | ^ + ⇧ + P Open command palette |
| Insert (?) Toggle overwrite flag (?) | ⇧ + ↵, ⇧ + Enter Run cell, select below |
| ^ + ↑ Go to cell start | ^ + Enter Run selected cells |
| ^ + ↓ Go to cell end | Alt + ↵, Alt + Enter Run cell, insert below |
| Alt + ← Go one word left | ^ + ⇧ + Minus Split cell at cursor |
| Alt + → Go one word right | ^ + S Save and Checkpoint |
| Alt + ⌫ Delete word before | ↓ Move cursor down |
| Alt + ⌦ Delete word after | ↑ Move cursor up |

### 2.2.1 Jupyter Notebook Key Bindings Table Notes

- The macOS colors are:
  - Keys background: yellow #FFFF00
  - Odd row background: user defined myJupyterMacOSBgClrOdd #FFFF7F — this is a tint of yellow — see color-hex: yellow
  - Even row background: user defined myJupyterMacOSBgClrEven gray!10
- The Windows colors are:
  - Keys background: PaleTurquoise #AFEEEE
  - Odd row background: user defined myJupyterWinBgClrOdd #C4F2F2 — first monochromatic color — see color-hex: #AFEEEE
  - Even row background: user defined myJupyterWinBgClrEven gray!10
- The tables are derived from the *Jupyter Notebook* interface and
  - **Cheatography: Jupyter Notebook Keyboard Shortcuts**
  - **Cheatography: Jupyter Notebook Editor Keyboard Shortcuts**
  - For both of the above you can download the LaTeX source
- **Queries**
  1. Does Q close the pager (as well as Escape) ?
  2. Does S do a Save and Checkpoint (as well as Ctrl+S) ?
  3. Does Ctrl+Return do Run Selected Cells (as well as Ctrl+Enter) ?
- It may be possible to typset the tables in Markdown but not done that yet with the colour banding
- **M269 Colors for Marking**
- from 2021J TMA files
- .answercell{background-color : #FFFFCC;} #FFFFCC
- .feedbackcell{background-color : #C8ECFF;} #C8ECFF
- .guidancecell{background-color : #F2C0D4;} #F2C0D4
- See Converting Colors for display of colors
- FFFFCC is a Websafe version with name conditioner
- Websafe version of C8ECFF is CCFFFF #CCFFFF
- Websafe version of F2C0D4 is FFCCCC #FFCCCC

# 3 Python

## 3.1 Python Expressions & Operator Precedence

The following table is from Python Reference: Section 6.17 Operator precedence in the Python documentation with highest precedence (most binding) at the top of the table. Operators in the same delimited row are left associative except for exponentiation which is right associative.

If in doubt, keep the brackets — you can still be surprised by some expressions

```
(9 // 2) * (9 // 2) == 16
9 // 2 * 9 // 2     == 18
```

The power operator ** binds less tightly than an arithmetic or bitwise unary operator on its right (but I would put the brackets in anyway)

```
2**-1 == 2**(-1) == 0.5
```

M269 uses a subset of the operators given below — any student slides/notes would need a cut-down version

The table below include set operators, which are part of the Python Library

## Python Operator Precedence

| Operator(s) | Description |
|---|---|
| (*expr...*),<br>[*expr...*],<br>{*expr...*},<br>{*key*:*value...*} | Parenthesized form,<br>Displays for lists, sets, dictionaries (comprehensions)<br>Set displays<br>Dictionary displays |
| *xs*[*index*],<br>*xs*[*index*:*index*],<br>*f*(*args...*),<br>*o.attr* | Sequence subscription,<br>Slicing,<br>Function call,<br>Attribute reference |
| *await expr* | Await expression |
| ** | Exponentiation |
| +*x*, -*x*, ~*x* | Unary arithmetic and bitwise operations<br>Unary positive, unary negative, bitwise *not* |
| *, @, /,<br>//, % | Binary arithmetic operations<br>Multiplication, matrix multiplication, division,<br>floor division, remainder or string formatting |
| +, - | Addition, subtraction, set difference |
| <<, >> | Shifts |
| & | Binary bitwise operations Bitwise *and*, set intersection |
| ^ | Bitwise *xor*, set symmetric difference |
| \| | Bitwise *or*, set union |
| <, <=, >, >=,<br>!=, ==,<br>in, not in,<br>is, is not | Comparisons, set subset, superset,<br>value comparisons,<br>Membership tests, set membership<br>Identity tests |
| not *bexpr* | Boolean operations Boolean *not* |
| and | Boolean *and* |
| or | Boolean *or* |
| *exprT* if *bexpr* else *exprF* | Conditional expression |
| lambda | Lambda expression |
| := | Assignment expression |

### 3.1.1 Python Operator Precedence Table Notes

- The table here is derived from Python Reference: Section 6.17 Operator precedence for version 3.8.3 — the table here is reversed compared to the Python documentation so that the highest precedence is at the top of the table.
- The odd row background colour is the HTML color #EEFFCC — the same as the code background colour in the Python documentation
- Lutz (2013, Table 5-2, page 137) also has `yield` *expr* at the lowest precedence — parentheses probably required everywhere except *parentheses may be omitted when the yield expression is the sole expression on the right hand side of an assignment statement.*

## 3.2 Python Data Types: Complexities

- The following tables are based on wiki.python.org/moin/TimeComplexity
- **Health Warning** these notes are work in progress and need to be read with the Python documentation
- Sequence Type — list, tuple, range
- Set Types — set, frozenset
- Mapping Types — dict
- collections.deque
- The complexities here may have some errors or typos or be misleading — **beware**

| Python Data Types: Complexities | | | List |
|---|---|---|---|
| Operation | | Average | Amortized Worst |
| Get item | `x = xs[i]` | O(1) | O(1) |
| Set item | `xs[i] = x` | O(1) | O(1) |
| Append | `xs = ys + zs` | O(1) | O(1) |
| Copy | `xs = ys[:]` | O(n) | O(n) |
| Pop last | `xs.pop()` | O(1) | O(1) |
| Pop other | `xs.pop(i)` | O(k) | O(k) |
| Insert(i,x) | `xs[i:i] = [x]` | O(n) | O(n) |
| Delete item | `del xs[i:i+1]` | O(n) | O(n) |
| Get slice | `xs = ys[i:j]` | O(k) | O(k) |
| Set slice | `xs[i:j] = ys` | O(k + n) | O(k + n) |
| Delete slice | `xs[i:j] = []` | O(n) | O(n) |
| Member | `x in xs` | O(n) | |
| Get length | `n = len(xs)` | O(1) | O(1) |
| Count(x) | `n = xs.count(x)` | O(n) | O(n) |

| Complexities: Set | | Python | | Haskell |
|---|---|---|---|---|
| Operation | | Average | Worst | Worst |
| Member | `x in s` | O(1) | O(n) | $O(\log n)$ |
| Union | `s \| t` | O(m + n) | | $O(m \log(\frac{n}{m} + 1))$ |
| Intersection | `s & t` | O(min(m, n)) | O(m × n) | $O(m \log(\frac{n}{m} + 1))$ |
| Difference | `s - t` | O(m) | | $O(m \log(\frac{n}{m} + 1))$ |
| Insert | `s.add(x)` | O(1) | O(n) | $O(\log n)$ |
| Delete | `s.discard(x)` | O(1) | O(n) | $O(\log n)$ |
| Remove | `s.remove(x)` | O(1) | O(n) | $O(\log n)$ |

| Python Data Types: Complexities | | Dict | |
|---|---|---|---|
| Operation | | Average | Amortized Worst |
| Member | `k in d` | O(1) | O(n) |
| Get item | `d.get(k)` | O(1) | O(n) |
| Set item | `d.get(k)` | O(1) | O(n) |
| Delete | `d.pop(k,default)` | O(1) | O(n) |
| Copy | `d.copy()` | O(n) | O(n) |
| Iterate | `iter(d)` | O(n) | O(n) |

| Python Data Types: Complexities | | collections.deque | |
|---|---|---|---|
| Operation | | Average | Amortized Worst |
| Copy | `dq.copy()` | O(n) | O(n) |
| Append | `dq.append(x)` | O(1) | O(1) |
| Append left | `dq.appendleft(x)` | O(1) | O(1) |
| Pop | `dq.pop()` | O(1) | O(1) |
| Pop left | `dq.popleft()` | O(1) | O(1) |
| Extend | `dq.extend(iterable)` | O(k) | O(k) |
| Extend left | `dq.extendleft(iterable)` | O(k) | O(k) |
| Rotate | `dq.rotate(n=1)` | O(k) | O(k) |
| Remove | `dq.pop(k,default)` | O(n) | O(n) |

# 4 Haskell

## 4.1 Haskell Operator Precedence & Fixity

- In the following table
- **P** Precedence — 9 highest, 0 lowest, default 9
- **A** Fixity, associativity — **L** Left, **N** Non-associative, **R** Right
- The table is based on the Haskell 2010 Language Report (Section 4.4.2 Fixity Declarations) — note that the Haskell libraries have more operators than are in the table

# Haskell Operator Precedence

| Operator(s) | Type | Description | A | P |
|---|---|---|---|---|
| !! | `[a] -> Int -> a` | List index | **L** | 9 |
| . | `(b -> c) -> (a -> b) -> a -> c` | Function composition | **R** | 9 |
| ! | `Ix i => Array i e -> i -> e` | Array index (Data.Array) | **L** | 9 |
| // | `Ix i => Array i e -> [(i, e)] -> Array i e` | Array update (Data.Array) | **L** | 9 |
| ^ | `(Integral b, Num a) => a -> b -> a` | Exponentiation | **R** | 8 |
| ^^ | `(Fractional a, Integral b) => a -> b -> a` | Exponentiation | **R** | 8 |
| ** | `Floating a => a -> a -> a` | Exponentiation | **R** | 8 |
| * | `Num a => a -> a -> a` | Multiply | **L** | 7 |
| / | `Fractional a => a -> a -> a` | Divide | **L** | 7 |
| `div`,`mod`, `quot`,`rem` | `Integral a => a -> a -> a` | Integer division | **L** | 7 |
| % | `Integral a => a -> a -> Ratio a` | Ratio (Data.Ratio) | **L** | 7 |
| +, - | `Num a => a -> a -> a` | Addition, Subtraction | **L** | 6 |
| : | `a -> [a] -> [a]` | List constructor | **R** | 5 |
| ++ | `[a] -> [a] -> [a]` | List append | **R** | 5 |
| \\ | `Eq a => [a] -> [a] -> [a]` | List difference (Data.List) | **N** | 5 |
| ==,/= | `Eq a => a -> a -> Bool` | Equality | **N** | 4 |
| <,<=,>,>= | `Ord a => a -> a -> Bool` | Comparison | **N** | 4 |
| `elem`,`notElem` | `(Foldable t, Eq a) => a -> t a -> Bool` | Membership | **N** | 4 |
| <$> | `Functor f => (a -> b) -> f a -> f b` | Infix fmap (Data.Functor) | **L** | 4 |
| <*> | `Applicative f => f (a -> b) -> f a -> f b` | Sequential application (Control.Applicative) | **L** | 4 |
| && | `Bool -> Bool -> Bool` | Logical *and* | **R** | 3 |
| \|\| | `Bool -> Bool -> Bool` | Logical *or* | **R** | 2 |
| >> | `Monad m => m a -> m b -> m b` | Compose two actions | **L** | 1 |
| >>= | `Monad m => m a -> (a -> m b) -> m b` | Compose two actions | **L** | 1 |
| $,$! | `(a -> b) -> a -> b` | Application (strict) | **R** | 0 |
| `seq` | `a -> b -> b` | Strict eval | **R** | 0 |

# 5 Java

## 5.1 Java Operator Precedence & Associativity

- In the table of expressions and operator precedence and associativity, operators in the same group of rows have higher precedence than those below
- The column **A** represents **operator associativity**
  - **L** left associative

    `e1 + e2 + e3` means `(e1 + e2) + e3`
  - **R** right associative

    `e1 = e2 = e3` means `e1 = (e2 = e3)`
  - **N** not associative

    `1 < x < 4` is not valid
- *integer* means `char`, `byte`, `short`, `int`, `long`, or the boxed forms Character, Byte, Short, Integer, Long
- *numeric* means integer or `float`, `double` or the boxed forms Float, Double
- The table is based on Sestoft (2016, section 11.1, page 37)
- Evans and Flanagan (2018, page 35) gives a similar table
- The information would be derived from Java Language Specification: Chapter 15. Expressions but that is not an easy read

# Java Operator Precedence

| Expression | Meaning | A | P | Arg Types | Result Types |
|---|---|---|---|---|---|
| `a[...]` | array access | | 16 | `t[]`, integer | `t` |
| `o.f` | non-static field access | | | object `o` | type of `f` |
| `C.f` | static field access | | | | type of `f` |
| `this` | current object reference | | | | this class |
| `o.m(...)` | instance method call | | | object `o` | return type |
| `C.m(...)` | static method call | | | | return type |
| `super.m(...)` | superclass method call | | | | return type |
| `C.super.m(...)` | encl. superclass meth. call | | | | return type |
| `t.class` | class object for `t` | | | type `t` | `Class<t>` |
| `t::m` or `e::m` | method reference | | | | |
| `x++` or `x--` | postincrement/decrement | | | numeric | numeric |
| `++x` or `--x` | preincrement/decrement | | 15 | numeric | numeric |
| `-x` | negation | R | | numeric | numeric |
| `~e` | bitwise complement | R | | integer | `int`/`long` |
| `!e` | logical negation | R | | boolean | boolean |
| `new t[...]` | array creation | | 14 | type `t` | `t[]` |
| `new C(...)` | object creation | | | class `C` | `C` |
| `(t) e` | type cast | | | type, any | `t` |
| `*`, `/` | multiplication, division | L | 13 | numeric | numeric |
| `e1 % e2` | remainder | L | | numeric | numeric |
| `+`, `-` | addition, subtraction | L | 12 | numeric | numeric |
| `e1 + e2` | string concatenation | L | | String, any | String |
| `e1 + e2` | string concatenation | L | | any, String | String |
| `e1 << e2` | left shift | L | 11 | integer | `int`/`long` |
| `e1 >> e2` | signed right shift | L | | integer | `int`/`long` |
| `e1 >>> e2` | unsigned right shift | L | | integer | `int`/`long` |
| `<`, `<=`, `>=`, `>` | comparison | N | 10 | numeric | boolean |
| `e instanceof t` | instance test | N | | any, reference type | boolean |
| `e1 == e2` | equal | L | 9 | compatible | boolean |
| `e1 != e2` | not equal | L | | compatible | boolean |
| `e1 & e2` | bitwise and | L | 8 | integer | `int`/`long` |
| `e1 & e2` | logical strict and | L | | boolean | boolean |
| `e1 ^ e2` | bitwise exclusive-or | L | 7 | integer | `int`/`long` |
| `e1 ^ e2` | logical strict exclusive-or | L | | boolean | boolean |
| `e1 \| e2` | bitwise or | L | 6 | integer | `int`/`long` |
| `e1 \| e2` | logical strict or | L | | boolean | boolean |
| `e1 && e2` | logical and | L | 5 | boolean | boolean |
| `e1 \|\| e2` | logical or | L | 4 | boolean | boolean |
| `e1 ? e2 : e3` | conditional | L | 3 | boolean, any, any | any |
| `x = e` | assignment | R | 2 | `e` subtype of `x` | type of `x` |
| `x += e` | compound assignment | R | | compatible | type of `x` |
| `x -> ebs` | lambda expression | R | 1 | | |

## 5.2 Java Statements

- A *statement* is the basic unit of execution in Java
- A statement may change the computer's *state*: the value of variables, fields and array elements; the contents of files; and so on
- The execution of a statement can:
  - terminate normally (meaning execution will continue with the next statement, if any)
  - terminate abruptly by throwing an exception
  - exit by executing a `return` statement (if inside a method or constructor)
  - exit a switch or loop by executing a `break` (if inside a switch or loop)
  - exit the current iteration of a loop and start a new iteration by executing a `continue` statement or
  - does not terminate at all (eg, `while (true) {}`)

### 5.2.1 Expression & Block Statements

- An expression statement is an expression followed by a `;`

```
expression ;
```

- The only forms of expression that may be used here are assignments, increment and decrements, method call, and object creation
- A block statement is a sequence of variable declarations, class declarations and statements

```
{
  variableDeclarations
  classDeclarations
  statements
}
```

- An empty statement consists of `;` only — it is equivalent to the block statement `{ }`

### 5.2.2 Selection Statements

- The `if` statement has the form

```
if (condition)
  trueBranch
```

- The `if-else` statement has the form

```
if (condition)
  trueBranch
else
  falseBranch
```

- The condition must have type `boolean` or `Boolean`

- trueBranch and falseBranch are statements
- What is wrong with the following

```
if (dataAvailable) ;
  processData() ;
```

```
if (dataAvailable)
  processData() ;
  reportResults() ;
```

```
if (dataAvailable)
  processData() ;
  reportResults() ;
else
  reportNoData() ;
```

```
if (dataAvailable) ;
  processData() ;
```

- The trueBranch is an empty statement (`;`)

```
if (dataAvailable)
  processData() ;
  reportResults() ;
```

- `reportResults() ;` will always be executed

```
if (dataAvailable)
  processData() ;
  reportResults() ;
else
  reportNoData() ;
```

- Will not compile
- **Moral** Always use block statements
- A `switch` statement has the form

```
switch (expression) {
  case constant1: branch1
  case constant2: branch2
  ...
  default: branchN
}
```

- expression must be of type `int`, `short`, `char`, `byte` or a boxed version of these or `String` or an enum type
- Each constant must be a compile-time constant expression, consisting only of literals, `final` variables, `final` fields declared with explicit field initialisers or an unqualified enum value
- (not used in M250)

### 5.2.3 Iteration Statements

- A `for` statement has the form

```
for (initialization ; condition ; step)
  body
```

- initialization is a variableDeclaration or an expression
- condition is an expression of type boolean or Boolean
- step is an expression
- body is a statement
- initialization and step may be comma-separated lists of expressions
- initialization, condition and step may be empty. An empty condition is equivalent to true
- The for statement is executed as follows

1. The initialization is executed
2. The condition is evaluated. If it is false, the loop terminates.
3. If it is true then
   (a) the body is executed
   (b) the step is executed
   (c) execution continues at (2.)

- What does the following code do ?

```
for (int i = 1 ; i <= 4 ; i++) {
  for (int j = 1 ; j <= i ; j++) {
    System.out.print("*") ;
  }
  System.out.println() ;
}
```

```
jshell>   for (int i = 1 ; i <= 4 ; i++) {
   ...>     for (int j = 1 ; j <= i ; j++) {
   ...>       System.out.print("*") ;
   ...>     }
   ...>     System.out.println() ;
   ...>   }
   ...>
*
**
***
****
```

- A while statement has the form

```
while (condition)
  body
```

- condition is an expression of type boolean or Boolean and body is a statement
- It is executed as follows:

1. The condition is evaluated. If it is false, the loop terminates
2. If it is true, then
   (a) The body is executed
   (b) Execution continues at (1.)

- Example linear search with while loop

```
String[] wdays =
  {"Monday","Tuesday","Wednesday"
  ,"Thursday","Friday","Saturday","Sunday"} ;

int wdayno(String wday) {
  int i = 0 ;
   while (i < wdays.length
          && ! wday.equals(wdays[i])) {
     i++ ;
   }
   if (i < wdays.length) {
     return i ;
   } else {
     return -1 ;
   }
} ;
```


```
String[] wdays =
  {"Monday","Tuesday","Wednesday"
  ,"Thursday","Friday","Saturday","Sunday"} ;

int wdayno(String wday) {
  int i = 0 ;
   while (i < wdays.length
          && ! wday.equals(wdays[i])) {
     i++ ;
   }
   if (i < wdays.length) {
     return i ;
   } else {
     return -1 ;
   }
} ;
```


```
jshell> int d1 = wdayno("Friday") ;
d1 ==> 4

jshell> int d2 = wdayno("Dimanche") ;
d2 ==> -1
```

- Write code using a `while` statement that is equivalent to a `for` loop statement

```
initialization
while (condition) {
  body
  step
}
```

```
for (initialization ; condition ; step)
  body
```

- Note that this is different behaviour to the `for` statement in Python where assignments to variables in the suite of the loop does not change the assignments made in the target list
- See Python: for statement
- A `foreach` statement has the form

```
for (tx x : expression)
  body
```

- The expression must have type `Iterable<t>` where `t` is a subtype of `tx`

# 6 JavaScript

## 6.1 JavaScript Operator Precedence & Associativity

- Mozilla Developers Network: JavaScript
- Mozilla Developers Network: JavaScript reference
- The following table is based on Mozilla Developer Network: JavaScript Reference: Operator precedence
- The column **A** represents **operator associativity**
  - **R** right associative

    `e1 = e2 = e3` means `e1 = (e2 = e3)`
  - **L** left associative

    `e1 + e2 + e3` means `(e1 + e2) + e3`

    `3 > 2 > 1` is valid and evaluates to `false` — why?
  - **N** not associative
- For information about APIs see
  - Web APIs
  - Document Object Model (DOM)

# JavaScript Operator Precedence

| Operator(s) | Description | A | P |
|---|---|---|---|
| `(expr...)` | Grouping | | 21 |
| `o.prop` | Member Access | L | 20 |
| `o[propStr]` | Computed Member Access | L | |
| `new C(args)` | new (with arg list) | | |
| `f(args)` | Function Call | L | |
| `?.` | Optional chaining | L | |
| `new C` | new (without arg list) | R | 19 |
| `x++,x--` | post-increment/decrement | | 18 |
| `! b` | Logical NOT | R | 17 |
| `~ x` | Bitwise NOT | R | |
| `+x, -x` | Unary plus, negation | R | |
| `++x,--x` | pre-increment/decrement | R | |
| | typeof,void,delete,await | R | |
| `x ** y` | Exponentiation | R | 16 |
| `*,/,%` | Multiplication,Division,Remainder | L | 15 |
| `+,-` | Addition,Subtraction | L | 14 |
| `<<,>>,>>>` | Bitwise Left Shift,Right,Unsigned | L | 13 |
| `<,<=,>,>=` | Comparisons | L | 12 |
| | in,instanceof | L | |
| `==,!=,===,!==` | Equality, Strict Equality | L | 11 |
| `&` | Bitwise AND | L | 10 |
| `^` | Bitwise XOR | L | 9 |
| `\|` | Bitwise OR | L | 8 |
| `&&` | Logical AND | L | 7 |
| `\|\|` | Logical OR | L | 6 |
| `??` | Nullish coalescing operator | L | 5 |
| `b ? x : y` | Conditional | R | 4 |
| `=` | Assignment | R | 3 |
| `op=` | Assignment with operation | R | |
| | yield,yield* | R | 2 |
| `expr1 , expr2` | Comma/Sequence | L | 1 |

# 7 CSS

## 7.1 CSS Links

- CSS Snapshot 2018
- Selectors Level 4 W3C Working Draft 21 November 2018

# 8 macOS

## 8.1 Hidden Files

- Some files and folders in macOS are not displayed by default in Finder or Terminal
- These are files with names starting with a dot (.), Library folders (there are several) and some system folders.
- Here are several ways of making these files visible
- **Finder (1)** with Finder selected, type ⌘+⇧+. — the keystroke command is a toggle so to turn viewing off just re-type the same
- **Finder (2)** to make the change permanent, type the following in Terminal

```
defaults write com.apple.Finder AppleShowAllFiles true
killall Finder
```

- **Finder (3)** to make a permanent change without using Terminal have a look at TinkerTool from https://www.bresink.com/osx/TinkerTool.html — this is free and does lots of other tweaks — the Finder tab first item is Show hidden and system files
- Also make sure you display filename extensions with Finder > Preferences > Advanced and check Show all filename extensions

# References

Beazley, David and Brian K. Jones (2013). *Python Cookbook*. O'Reilly, third edition. ISBN 9781449340377.

Evans, Benjamin J and David Flanagan (2018). *Java In A Nutshell*. O'Reilly, seventh edition. ISBN 1492037257.

Lutz, Mark (2011). *Programming Python*. O'Reilly, fourth edition. ISBN 0596158106. URL http://learning-python.com/books/about-pp4e.html.

Lutz, Mark (2013). *Learning Python*. O'Reilly, fifth edition. ISBN 1449355730. URL http://learning-python.com/books/about-lp5e.html.

Lutz, Mark (2014). *Python Pocket Reference*. O'Reilly. ISBN 9781449357016. URL https://learning-python.com/about-pyref5e.html.

Martelli, Alex; Anna Ravenscroft; and Steve Holden (2017). *Python in a Nutshell: A Desktop Quick Reference*. O'Reilly, third edition. ISBN 144939292X.

Sestoft, Peter (2016). *Java Precisely*. MIT, third edition. ISBN 0262529076. URL http://www.itu.dk/people/sestoft/javaprecisely/.