Functional Java

On this page

- Version History

Download version 2025S1-7 of the Standard Library here: zip (classic), jar for IntelliJ. See Installation Instructions

Functional Java

This page will contain information about the way we use Java in the first weeks of the course, and will evolve as the course progresses.

In the first few weeks, we will use Java in a very different way from how one would normally use Java. This is because Java as a language is best suited for professionally engineering large software, and thus many of its standard parts make a lot less sense when used in an introductory programming context. Luckily, the language designers themselves are working on bridging this gap. The (currently) newest release of Java - Java 23 - contains experimental features that drastically simplify early programming in Java, which we will use, in addition to a number of additional rules and a different standard library. We may update the standard library for every assignment, so please make sure to check if you are on the right version.

```
import static comp1110.lib.Functions.*;

void main() {
    CheckVersion("insert-required-version-here");
}
```

For example, to check if your current copy of the standard library is compatible with, say, version 2025S1-7, you would replace the string `"insert-required-version-here"` with `"2025S1-7"` in the program above. You can check the Minimum Library Version required for each assignment on the assignment specification.

Language Definition

A Functional Java program consists of six top-level parts:

- Zero or more `import` statements
- Zero or more *data definitions*
- Zero or more *constant definitions*
- Zero or more *function definitions*
- A `main` *function definition*
- A `test` *function definition*

Each of those parts may contain or be flanked by comments.

Constant definitions further contain *Expressions*. Function definitions contain *Statements*, which in turn may also contain *Expressions*.

Comments

There are several ways to write comments:

```
// two slashes make any text on the rest of the line a comment.
println("Hello World"); // this means they can follow some code

/* A slash / followed by a star * starts a comment block.
This block continues until a * is immediately followed by a slash,
across lines */
println("Hello World!" /* this means that such a comment
can also appear in the middle of some code */);
println("/* however, no comments can appear within strings*/");

/**
 * Finally, there is a special kind of block comment meant
 * to document definitions. Editors often insert the leading
 * stars * for you when you switch to a new line.
 * This block comment simply starts with a slash / and two
 * starts * instead of just one.
 * We use this kind of comment *a lot*!
 */
```

The Design Recipe page has a lot more information about which comments you should use where.

Imports

Functional Java is essentially a special way of using Java - mostly in a restricted way. You may not import any packages that are not part of our special standard library below. Do not use any `import` statements that do not appear *exactly* on this page. Copy the import statements as they appear in the library documentation below for the packages that you are using in your particular program.

Data Types and Data Definitions

Following the Design Recipe, data definitions may either consist solely of comments, or are limited versions of Java data definitions. The following standard Java data types are available:

```
boolean
int
```

```
long
float
double
char
String
Supplier<R>
Function<A, R>
BiFunction<A1, A2, R>
Predicate<A>
BiPredicate<A1, A2>
```

Importantly, you should not be using the Java `Object` type. Additionally, the data types provided by the Functional Java Standard Library below are available, as are user-defined data types as explained next. To see what the angle brackets `<>` here and in the standard library means, see the section on Generics.

The following Java definitions are allowed in the following ways:

- `enum` definitions. Java enumeration definitions have the following form:

  ```
  enum [Name] { [CASE1], [CASE2], ... }
  ```

  Enumeration names start with a capital letter, and insert more capital letters where multiple words are concatenated, as in `TrafficLightState`. Case names use only capital letters, and insert underscores '_' in between concatenated words, as in `LEFT_CLICKED`. Aside from the case names separated by commas `,`, nothing else may appear between the curly braces `{}`.
- `record` definitions. Java record definitions have one of the following two forms:

  ```
  record [Name]([Type1] [fieldName1], [Type2] [fieldName2], ...) {}
  record [Name]([Type1] [fieldName1], [Type2] [fieldName2], ...) im
  ```

  Record names start with a capital letter, and insert more capital letters where multiple words are concatenated, as in `StudentRecord`. Types refer to available types, and field names start with a lower case letter, but use capital letters where multiple words are concatenated, as in `previousStudent`. Nothing may appear between the curly braces `{}`. The `implements` clause is restricted to be used with a single `sealed interface` defined for the same program.
- `sealed interface` definitions. Java sealed interface definitions have the following form:

```
sealed interface [Name] permits [RecordName1], [RecordName1], ...
```

Sealed interface names start with a capital letter, and insert more capital letters where multiple words are concatenated, as in `LandVehicle`. Record names refer to the names of records as defined above. Each of those records in turn must have an implements clause referring to this sealed interface. Nothing may appear between the curly braces `{}`.

Constant Definitions

A constant definition has the following form:

```
[Type] [NAME] = [Expression];
```

Types refer to data types. Constant names use only capital letters, and insert underscores '_' in between concatenated words, as in `WORLD_HEIGHT`. Expressions are defined below.

Function Definitions

A function definition has the following form:

```
[Type] [name]([ArgType1] [argName1], [argType2] [varName2], ...) {
  [Statement1]
  [Statement2]
  ...
}
```

Types and Arg(ument) Types refer to data types. The first type at the start of the function definition is called the return type. For certain kinds of functions, in addition to regular data types, it may also be specified as `void`, indicating that the function does not return any value.

Function names start with a lower case letter, but use capital letters where multiple words are concatenated, as in `computeArea`. Arg(ument) names start with a lower case letter, but use capital letters where multiple words are concatenated, as in `firstShape`. Statements are defined below.

The `main` Function

The `main` function is a required part of every Functional Java program. It has one of the two following forms:

```
void main() {
  [Statement1]
```

```
  [Statement2]
  ...
}
```

or

```
void main(String... args) {
  [Statement1]
  [Statement2]
  ...
}
```

In the second form, a special kind of expression is available in the main function only: array-accessors `args[N]`. The brackets are actually part of Java syntax here, while the capital `N` stands for an integer indicating the 0-based index of a command-line argument. That is, if the program is called with `java --enable-preview Main.java foo bar 15` then `args[0]` stands for `"foo"`, `args[1]` stands for `"bar"`, and `args[2]` stands for `"15"`. Note that command-line arguments are always Strings. If you use an index that is either negative or for a later index for which there is no command-line argument, your program will crash. You can use the expression `Length(args)` to determine how many command-line arguments were given to the program. In the above example, this function would return the integer `3`.

The `test` Function

The `test` function is a required part of every Functional Java program. It has the followin form:

```
void test() {
  [Statement1]
  [Statement2]
  ...
}
```

The use of the `test` function is further explained in the testing package documentation below, and on the Design Recipe page.

Expressions

Expressions are parts of a program that evaluate to values. They can be nested within each other or be siblings of each other. Siblings are evaluated left-to-right, whereas nested expressions are evaluated inside-out, except for the control-flow expressions `if` and `switch`, and short-circuited logical expressions `&&` and `||` You can write the following kinds of expressions:

- Base type literals: these are the booleans `true` and `false`; `String`s (e.g. `"Hello World!"`, `""`, and `"COMP1110/6710"`); `char`s (e.g. `'a'`, `'0'`, `'!'`); and literals of the various number types:
  - (possibly negative) `int`s: `5`, `-2`, `0`, `1337`
  - (possibly negative) `long`s: `5l`, `-2l`, `0l`, `1337l`
  - (possibly negative) `float`s: `5.0f`, `-2.0f`, `0.0f`, `1337.6f`
  - (possibly negative) `double`s: `5.0`, `-2.0`, `0.0`, `1337.6`
- Enumeration literals: these refer to a particular case of an enumeration, and have the general form:

  ```
  [EnumerationName].[CASE_NAME]
  ```

- Global constant names, as defined above
- Local variable names: these may be arguments to the current function or variables defined in a variable definition statement
- Primitive unary operations: `!b` negates a boolean value `b`, `-n` inverses the sign of a numeric value `n`.
- Primitive binary operations:
  - `+` adds two numbers or concatenates a String with a string representation of any kind of value
  - `-`, `*`, `/`, `%` are standard numeric operations (`%` is the remainder operation, which coincides with modulo on positive numbers)
  - `&&`, `||` are the logical operators *and* and *or*, respectively, with the added twist that if the left-hand side of an *and* operation evaluates to `false`, the right-hand side is not evaluated and the result of the operation is `false`, and conversely, the left-hand side of an *or* operation evaluates to `true`, the right-hand side is not evaluated and the result of the operation is `true`.
  - For **numbers and booleans only**, you can also use the operators `==`, `>`, `>=`, `<`, and `<=`. For all other types, in particular Strings, use the comparison functions from the standard library (`Equals`, `GreaterThan`, `LessThan`, `Compare`), where applicable.
- Parentheses: you can always put any expression in between two parentheses `()`. This is particularly useful to control operator associativity and precendence, as in `(5 + x) * 3`.
- Record constructor calls: for any record definition that starts with

  ```
  record [Name]([Type1] [fieldName1], [Type2] [fieldName2], ...)
  ```

  regardless of whether the definition contains an `implements` clause, you can create a new instance of this record by writing

  ```
  new [Name]([Expression1], [Expression2], ...)
  ```

The number of expressions needs to match the number of fields in the record definitions, and the types of the expressions need to match the types of the respective fields.

- Lambda expressions: A lambda expression creates a new anonymous function of zero, one, or two arguments. It has one of the following forms:

  ```
  () -> [Expression]
  ```

  or

  ```
  [varName] -> [Expression]
  ```

  or

  ```
  ([Type] [varName]) -> [Expression]
  ```

  or

  ```
  ([varName1], [varName2]) -> [Expression]
  ```

  or

  ```
  ([Type1] [varName1], [Type2] [varName2]) -> [Expression]
  ```

  These usually have the types `Supplier<R>`, `Function<A,R>`, `Function<[Type], R>`, `BiFunction<A1, A2, R>`, and `BiFunction<[Type1], [Type2], R>`, respectively, where `R` is the type of the expression on the right-hand side of the arrow `->`, and (where applicable) `A`, `A1`, and `A2` are the types of the arguments as derived from the context of the lambda, where such a context exists. A special exception is when `R` is `boolean`, in which case the respective types are `BooleanSupplier` (unused), `Predicate<A>`, `Predicate<[Type]>`, `BiPredicate<A1, A2>`, and `BiPredicate<[Type1], [Type2]>`. **Distinction-level content:** If you need to define an anonymous function that accepts more arguments, produce more functions. For example, the type `Function<Integer, Function<Integer, Function<String, String>>>` represents a function that takes two integers and a string, and produces a string, but does so via several steps.
- Record field accesses: if you have an expression `r` that has a record type which is defined as:

  ```
  record [Name]([Type1] [fieldName1], [Type2] [fieldName2], ...)
  ```

you can access a field of the record value by writing `r.[fieldName]()`, for any of its field names. The general form of this is:

```
[Expression].[fieldName]()
```

- References to functions you wrote: these are of the form

```
this::[functionName]
```

- Function calls: for any function that is either defined in your program, in the below standard libraries, or the special Java function `readln` (see below) you can write:

```
[functionName]([Expression1], [Expression2], ...)
```

  Where the number of expressions matches the number of parameters specified in the function's signature, and the types of the expressions match the types of the respective parameters. The function `readln` takes one parameter of type `String` and returns a `String`. The parameter is printed as a prompt to the user in the console, and the returned value is whatever the user entered until pressing the ENTER key, as a String.
- Lambda calls: If you have an expression of a function type, you can call that function. Depending on the function type, this has slightly different forms.
  - For `Supplier<R>`:

    ```
    [Expression].get()
    ```

  - For `Function<A, R>`:

    ```
    [Expression].apply([ArgExpression])
    ```

  - For `BiFunction<A1, A2, R>`:

    ```
    [Expression].apply([Expression1], [Expression2])
    ```

  - For `Predicate<A>`:

    ```
    [Expression].test([ArgExpression])
    ```

  - For `BiPredicate<A1, A2>`:

    ```
    [Expression].test([Expression1], [Expression2])
    ```

- if-expressions: these have the form

```
[Expression] ? [ThenExpression] : [ElseExpression]
```

The first expression needs to evaluate to a boolean. If that boolean is true, then the `ThenExpression` is evaluated and produces the result of the expression as a whole. Otherwise, the `ElseExpression` is evaluated and produces the result of the expression as a whole.

- switch-expressions: these have the form

```
switch([Expression]) {
case [Pattern1] -> [Expression1];
case [Pattern2] -> [Expression2];
...
[default -> [DefaultExpression]];
}
```

This first evaluates the parenthesized expression, and then moves through the case list from top to bottom, evaluating the first expression whose pattern matches the result of the parenthesized expression, and returning the result of the former expression as a result of the overall `switch` -expression. If no pattern matches, a `default` case provides the final way of producing a value. In many situations, your case patterns will be exhaustive, so no `default` case is necessary. Avoid it where possible. Patterns are one of:

  - A Base type literal, as above
  - An enumeration literal, as above, though in this case you can also omit the enumeration name and the dot '.', and just write the case name
  - A record pattern match, as follows:

```
[RecordName](var [varName1], var [varName2], ...)
```

There must be as many variable names as the named record type has fields. This pattern matches if the given value is an instance of the record, and it will bind its fields to the given variable names, which are available in the expression on the right-hand side of the arrow `->`.

Statements

Statements are higher-level program parts that are evaluated top-down, only modified by the control-flow statements `if` and `switch`. Statements may be grouped in blocks, which are enclosed by curly braces `{}`. Blocks control the scope of local variable definitions. Once you leave a block in which a local variable was defined, the local variable is not available anymore. Statements are always terminated by a semicolon `;`. You can write the following kinds of statements:

- `return` statments. These must be the last statement in a block, and may only not be the last statement in a function if nested within a control-flow statement. They are of the form:

  ```
  return [Expression];
  ```

  The return value of a function is whatever the expression in the return statement evaluates to.
- `println` statements. These are essentially function calls to a print function, with a single String argument specifying what to print on a line in the program's console output. For example:

  ```
  println("Hello World!");
  ```

- *Variable definition statements*: these define intermediate variables as the result of some expression. These variables are accessible in later statements in the same scope. The are of the form:

  ```
  [Type] [varName] = [Expression];
  ```

  If the expression is not a lambda-expression, you may also use the special type `var` to indicate that the variable should simply have whatever type the expression has.
- `if` statements: these have the form

  ```
  if([Expression]) {
  [ThenStatement1]
  [ThenStatement2]
  ...
  } [else if([Expression2]) {
  [Then2Statement1]
  [Then2Statement2]
  ...
  }] ... else {
  [ElseStatement1]
  [ElseStatement2]
  ...
  }
  ```

  The statement first evaluates the expression, which must return a `boolean` value. If that value is `true`, evaluation continues with the `ThenStatement`s, ignoring the statements in all other branches. Otherwise, we evaluate any following `else-if`'s expressions, and for the first one that is `true`, we evaluate the corresponding `ThenNStatement`s, again ignoring the statements in all other branches. If none of

the expressions evaluate to `true`, evaluation continues with the `ElseStatement`s, ignoring the `Then[N]Statement`s.

- `switch` statements: these have the form

```
switch([Expression]) {
case [Pattern1]: {
  [Case1Statement1]
  [Case1Statement2]
  ...
}
case [Pattern2]: {
  [Case2Statement1]
  [Case2Statement2]
  ...
}
[default: {
  [DefaultStatement1]
  [DefaultStatement2]
  ...
}]
}
```

  This works largely like the `switch`-expression, including the possible patterns. The main difference is that instead of expressions, we have full blocks of statements for the various cases. Each block needs to end in a `return` statement. The `default` case may not be necessary in many situations, avoid it where possible.

Generics

Several types and functions in the standard libraries have *generic* type arguments between angle brackets `<>`. These indicate that there are lots of versions of this type or function, one for each potential type for each type argument variable between the angle brackets. For the first half of the semester, creating such types and functions is **distinction-level content**, but you have to be able to use them.

For functions with generic type arguments, you have to do nothing special, except maybe adhere to any restrictions on which types you can use as described in the documentation.

For types with generic type arguments, you have to include the particular argument in the type. For example, the `Maybe` type is generic, and it expresses an optional value of some particular type. A `Maybe<Integer>` may either be `Nothing` or `Something<Integer>`, from which you can retrieve a particular integer. The same with `Maybe<String>`. A key wrinkle is that you cannot use any of the basic Java types that use lower-case letters in

such positions, i.e. there is no `Maybe<int>`. The following mapping defines which types you need to use instead:

- `int` -> `Integer`
- `long` -> `Long`
- `float` -> `Float`
- `double` -> `Double`
- `boolean` -> `Boolean`

You can arbitrarily convert back and forth between these two versions of writing the types, but you should never use the equality comparison operator `==` on values that are typed with a type that starts with a capital letter. Use the `Equals` function instead.

Specifically, you need to explicitly mention type arguments whenever you declare a variable of a generic type (e.g. `Maybe`) or create a new instance of it (e.g. `new Something<String>("hello")`, `new Nothing<Integer>()`). You may optionally specify the generic type in record patterns, e.g.:

```
switch(maybe) {
  case Nothing<String>() -> ...;
  case Something<String>() -> ...;
}
```

This is also part of the pattern of the type, but you need to replace the type variable with the concrete type argument that is applicable in your function.

Comparison to Regular Java

Control Flow

Yes

The only allowed control flow constructs are function calls, `switch`-expressions, the ternary condition operator `?:`, `switch`-statements, `if`-statements, and `return`-statements.

No

This in particular excludes all kinds of loop constructors (`for` / `while` / `do-while`) and `try`-blocks, as well as `break` statements in `switch` statements.

Assignment

Yes

The only valid assignment operator is `=`, and only in definition statements, i.e. statements of the form:

```
T x = ...;
```

No

Similarly, increment and decrement operators are not allowed. To reiterate, there may not be any statements of the form:

```
x = ...;
```

Without a type or `var` keyword in front of the variable.

Member Access

Yes

The dot operator '.' may only appear in `import` statements, in selecting a case of an enumeration, for accessing a field of a `record`, or for calling a lambda function.

No

For anyone who knows Java, this means that there are to be no qualified method calls of any kind, except for record field access.

Type Declarations

Yes

Valid type declarations are for `enum`s, `record`s, and `sealed interface`s. Only `enum`s may have anything between their curly braces `{}`, and even `enum`s may only simply list their cases.

No

In particular, this means that there may be no `class` declarations or non-`sealed` interfaces.

Other Modifiers, Keywords

Yes

The `this` keyword may only be used to obtain function references for the `BigBang` and `Start` library functions, as in

```
...

void main() {
    BigBang("MyProgram", new State(), this::draw, this::step);
}
```

The `final` keyword, while often unnecessary, is okay.

No

No Attributes, access modifiers, `static`, `volatile`, or `synchronized` keywords.

General Guidelines

- Do not declare packages. All files should be in the root folder.
- Avoid `null` wherever possible
- Avoid `default` cases in `switch` -statements and -expressions wherever possible.
- Prefer expressions over statements.
- Include a `void main()` function in every file
- Include a `void test()` function in every file
- Follow the Design Recipe

The Standard Library

The naming scheme of the libraries below is to distinguish it from Java's standard libraries (where function names start with a lower-case letter). This is to help you distinguish what you can use now from what you will use later. In your own code, you should follow Java's naming conventions.

Adding the Standard Library to Your Project

The most recent version of the Functional Java Standard Libraries is: `2025S1-7`

You need to add the standard library files to every one of your projects in which you use it separately.

Download the zip-file here. This zip file contains a folder named `comp1110`. To use the standard libraries in your project, copy that folder into the root folder of your project. Make sure to not check it into your git repository!

To do this in terminal, run the following commands in your project's root folder (where your java files are):

```
curl https://comp.anu.edu.au/courses/comp1110/assets/code/stdlib.zip
unzip stdlib.zip
```

```
rm stdlib.zip
```

On MacOS, the `unzip` command is installed by default. On WSL and Linux installations, it may not be there by default. In this case, you need to run the following commands once:

```
sudo apt update
sudo apt install unzip
```

IntelliJ

When using IntelliJ, if you want access to the Functional Java Standard Libraries, download the jar-file here.

- In IntelliJ, create a new folder “libraries”, at the same level as the “src” and “out” folders. Put the jar-file in there.
- Click “File” -> “Project Structure...”.
- Then, in the Project Settings screen, select the “Libraries” menu
- Click the “+” button in the middle towards the top, to add a new library, and select “Java”
- In the file search dialog, find the folder for your current project, and within it, the “libraries” folder, and within that, the COMP1110 libraries jar file.

Running a Program

We’ll assume that you are in a terminal, in the root folder of your project. That root folder contains:

- the `comp1110` folder from the standard library zip file
- your code, in a file `XYZ.java`

To run your program, run the following command:

```
java --enable-preview XYZ.java
```

To run the tests for your program (i.e. to run the function `void test()` instead of `main`), run the following command:

```
java --enable-preview comp1110.testing.Test XYZ.java
```

The latter command may create a number of files with the `.class` extension. You can remove them by running:

```
rm *.class
```

But they also don't hurt. Just make sure to not check them into your repository!

You can view details for all Standard Library Packages in one page, or browse the packages and links to individual detail pages below.

The lib Package

View Full Package Documentation

This is the core standard library for Functional Java. It contains a selection of data types and functions that are useful in many programming scenarios. To import this package, use the following import statements:

```
import comp1110.lib.*;
import comp1110.lib.Date;
import static comp1110.lib.Functions.*;
```

General

Records

Pair

Itemizations

Maybe

Functions

GetVersion ; CheckVersion ; DefaultCaseError ; Equals ; Compare ; Min ; Max ; GreaterThan ; LessThan ; ToString ; IsIntString ; StringToInt ; IsLongString ; StringToLong ; IsFloatString ; StringToFloat ; IsDoubleString ; StringToDouble ; StringToBoolean ; Length ; RandomNumber ; Default ; Join

DateTime

Opaque Data Types

Date ; DateTime

Functions

CurrentDate ; CurrentTime ; IsDateValid ; GetDate ; GetDateTime ; GetYear ; GetMonth ; GetDay ; GetDayOfWeek ; GetHour ; GetMinute ; GetSecond ; GetMillisecond ; GetTimeZone ; AddYears ; AddMonths ; AddDays ; AddHours ; AddMinutes ; AddSeconds ; AddMilliseconds ; WithTimeZone ; InTimeZone ; ToDate ; YearsBetween ; MonthsBetween ;

DaysBetween ; HoursBetween ; MinutesBetween ; SecondsBetween ; MillisecondsBetween ; FormatDate ; ParseDate ; ParseDateTime

HigherOrder

Functions

Bind ; Map ; BuildList ; Sort ; Fold ; FoldLeft ; Map ; Map ; Filter

Lambdas

Functions

Curry ; UnCurryFunction ; CurryPredicate ; UnCurryPredicate ; PairFunction ; UnpairFunction ; Compose ; ComposeAnd ; ComposeOr ; BiComposeAnd ; BiComposeOr

Lists

Itemizations

ConsList

Functions

MakeList ; BuildList ; BuildList ; IsEmpty ; First ; Rest ; Length ; Nth ; Append ; Sort ; Zip ; Unzip

Maps

Functions

MakeConsMap ; MakeHashMap ; Put ; Put ; Remove ; Remove ; Get ; Get ; ContainsKey ; ContainsKey ; GetKeys ; GetKeys

Math

Constants

PI ; E

Functions

Abs ; Exp ; Pow ; Sqrt ; Round ; RoundInt ; Ceil ; CeilInt ; Floor ; FloorInt ; Log ; Log10 ; Sin ; Cos ; Tan ; ASin ; ACos ; ATan

Strings

Functions

Format ; Concatenate ; SubString ; Contains ; StartsWith ; EndsWith ; Replace ; Length ; IndexOf ; LastIndexOf ; Trim ; UpperCase ; LowerCase ; GetCharAt

The Testing Package

View Full Package Documentation

This package provides basic testing functionality.

Each of your tests should be in its own function, named `test[FunctionName][testName]`. These test functions should use the assertion functions `testEqual`, `testNotEqual`, `testTrue`, and `testFalse` in this package to assert things about the results of the test. For example:

```
/** tests some simple examples of sums */
void testSumOfNumbersSimple() {
    testEqual(10, sumOfNumbers(3,7));
    testEqual(-5, sumOfNumbers(-10,5));
}
```

You then run these testing functions from your main functions, using the `runAsTest` function from this package as a wrapper, as in

```
void test() {
    runAsTest(this::testSumOfNumbersSimple);
    ...
}
```

To run your tests, you can either call the test function from `main`, or you can run

```
java --enable-preview comp1110.testing.Test yourfilename.java
```

This will just run the `test` function.

To import this package, use the following import statements:

```
import static comp1110.testing.Comp1110Unit.*;
```

Testing

Functions

runAsTest ; testEqual ; testNotEqual ; testTrue ; testFalse

The Universe Package

View Full Package Documentation

To import this package, use the following import statements:

```
import comp1110.universe.*;
import static comp1110.universe.Colour.*;
import static comp1110.universe.Image.*;
import static comp1110.universe.Universe.*;
```

Images

Opaque Data Types

Colour ; Image

Enumerations

FontStyle ; Mode ; TextAlign

Functions

GetImageWidth ; GetImageHeight ; GetImagePinX ; GetImagePinY ; Intersects ; MovePinBy ; MovePinTo ; Rectangle ; Circle ; Ellipse ; Polygon ; Text ; DrawLine ; FromFile ; SavePNG ; SaveJPEG ; Overlay ; OverlayXY ; Place ; PlaceXY ; Rotate ; Scale ; ScaleX ; ScaleY ; Mask ; ShowImage

World

Enumerations

KeyEventKind ; MouseEventKind

Records

KeyEvent

; MouseEvent

Functions

BigBang

Version History

Version	Date	Comments

2025S1-7	10/03/2025 09:00	Released definition of Maps and associated functions Used for U2. P2 may use older versions of the libraries, but can upgrade
2025S1-6	03/03/2025 08:00	Released definition of Cons-lists and associated functions Used for P2. U1 Part 4 may use older versions of the libraries, except for higher-level list functions - in that case, please upgrade to this version
2025S1-5	27/02/2025 20:30	Fixed a bug in the implementation of OverlayXY If you are not using this function, you do not need to update
2025S1-4	26/02/2025 21:30	Fixed a bug in the implementation of EndsWith If you are not using this function, you do not need to update
2025S1-3	24/02/2025 17:00	Addressed a mismatch in key names between MacOS, Linux, and Windows This is an important update, please replace any older versions of the library
2025S1-2	23/02/2025 21:00	Added key missing testing functions, in particular runAsTest This is an important update, please replace any older versions of the library

Acknowledgement of Country

The Australian National University acknowledges, celebrates and pays our respects to the Ngunnawal and Ngambri people of the Canberra region and to all First Nations Australians on whose traditional lands we meet and work, and whose cultures are among the oldest continuing cultures in human history.

Contact ANU | Copyright | Disclaimer | Privacy | Freedom of Information

+61 2 6125 5111 |
The Australian National University, Canberra

TEQSA Provider ID: PRV12002 (Australian University) |
CRICOS Provider Code: 00120C | ABN: 52 234 063 906