
Australian National University
Structured Programming
COMP1110/6710
Needs ANU Account!
pollev.com/fabianm
Register for Engagement
Image Courtesy NASA/JPL-Caltech.

Fifty50

FIRST YEAR MENTORING PROGRAM

Are you a **first year STEM student** learning how to navigate uni life?

Or a **2nd+ year STEM student** keen to share your hard-earned wisdom and

Sign up as a **mentor** or a **mentee** below!

Admin

- Class representatives (COMP6710) – apply by today COB

- Reminder: Follow the Design Recipe!
- gitlab-CI file available for P1

- We’re working on increasing drop-in capacity
- P1 Submission
  - We’re hoping to get CWAC available soon (check Ed for announcements)
  - Otherwise, use Office form, and we’ll release some way of using assignment variables if necessary
  - By default, latest pushed commit before deadline will be tested and used for code walks

The Testing Interface

Two kinds of tests: yours and ours

You can imagine that our test assertions look like:

```
testEqual(new Nothing<Date>(),
          nextSalaryIncrease(
            makeStudent(“Jen”, GetDate(2005, 4, 4))));
```


Finishing up: World Program Start

[Live Demo]

Practice

Fork and clone the comp1110-2025s1-workshops project.
Create a folder “ws2b”, and work in “World.java” in there.
Commit and push when you are done.

Design a World Program in stages.

Stage 1: the player controls a rectangle that starts in the middle of the world, and can be moved up, down, left, and right with the arrow keys.

Stage 2: on a click, the square turns into a circle. On another click, the circle turns back into a rectangle. The circle can move just like the rectangle.

Stage 3: if there is currently a rectangle, pressing the “w” key increases its width, pressing “e” reduces the width, “h” increases the height, and “j” decreases the height. When changing from a circle to a rectangle, the rectangle’s width and height are reset to their original values.

Stage 4: if there is currently a circle, pressing “r” increases its radius, and pressing “e” decreases the radius. When changing to a circle, the circle’s radius is the smaller of the rectangle’s width and height.

Input/Output

Part 2 – Reading from the Standard Input

Australian National University

Input/Output

```
/** Prints given value to standard output
 *  (usually: the console) */
void println(Object)
```

has a sibling:

```
/** Prints a prompt to the standard output and waits for
 *  input on standard input (usually: the console). Returns
 * next line of input. */
String readln(String)
```


Abstraction

Value Abstraction

Type Abstraction

Function Abstraction

Australian National University

DRY - Don’t Repeat Yourself (in code)

Copying code is bad!

- It duplicates bugs
- Changes might not be applied everywhere
- Larger code base ➔ harder to understand

Abstraction is about finding and making use of commonalities.

Value Abstraction

```
String dogInfo(String name, int age) {
    if(age < 3) { return name+” is a young dog”; }
    else { if(age < 10) { return name + “is a dog”; }
    else { return name+” is an old dog”; }
}
String catInfo(String name, int age) {
    if(age < 4) { return name+” is a young cat”; }
    else { if(age < 12) { return name + “is a cat”; }
    else { return name+” is an old cat”; }
}
```


Value Abstraction

```
String petInfo(String name, int age, int young,
                            int old, String type) {
   if(age < young) { return name+" is a young "+type; }
   else { if(age < old) { return name + "is a "+type; }
   else { return name+" is an old "+type; }
}
String catInfo(String name, int age) {
   return petInfo(name, age, 4, 12, "cat");
}
String dogInfo(String name, int age) {
   return petInfo(name, age, 3, 10, "dog");
}
```


Java Generics

Type Abstraction

A Brief User's Guide

Australian National University

Maybe

`Maybe<T>` is for cases where there may or may not be a result.

For example:

```
/** Tries to turn a String into an int if possible
  * … */
Maybe<Integer> tryStringToInt(String str) {
  if(isIntString(str)) {
    return new Something<>(StringToInt(str));
  } else {
    return new Nothing<>();
  }
}
```


Maybe

`Maybe<T>` is for cases where there may or may not be a result.

This actually comes up a lot, and the patterns are always the same.
Except for the type of the result.

But for the pattern, that does not matter.

Don’t Repeat Yourself (in code)

```
sealed interface MaybeInt permits NoInt, SomeInt {}
record NoInt() implements MaybeInt {}
record SomeInt(int i) implements MaybeInt {}

sealed interface MaybeString permits NoString, SomeString {}
record NoString() implements MaybeString {}
record SomeString(String s) implements MaybeString {}
```

…

Don’t Repeat Yourself (in code)

```
sealed interface Maybe<T> permits Nothing, Something {}
record Nothing<T>() implements Maybe<T> {}
record Something<T>(T elem) implements Maybe<T> {}
```

Much better! T can stand for (almost) anything!

➔ This is just a general itemization, albeit with lots of specializations.

Almost anything

The primitive types (those whose name starts with a lowercase letter) can't be used with generics ☹.

That's why it's Maybe<Integer> and not Maybe<int>

```
   int ⇔ Integer
  long ⇔ Long
 float ⇔ Float
double ⇔ Double
boolean ⇔ Boolean
  char ⇔ Character
```

You can freely convert between the two.
Just don't use == near the capitalized versions.
Use `Equals` instead.

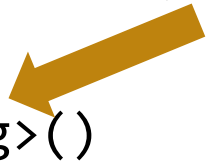
How to use Maybe<T>

Step 1: replace the T with something concrete.

```
Maybe<String>, Maybe<Image>, Maybe<Maybe<String>>, …
```

Step 2A: create new Maybe values

```
new Something<>(“Hello”)
new Nothing<>()
new Nothing<String>()
```



You can leave out the type argument when it is clear from context (according to Java). It doesn’t hurt to have it, though.

Step 2B: apply `Maybe`’s template to distinguish between values

Pair<S,T>

```
/**
 * Represents a pair of two values of given types.
 * Examples:
 * - Pair(5, "Hello")
 * - Pair(16.0, 32.5)
 * - Pair(Pair("", true), 0)
 * @param first The first value in the pair
 * @param second The second value in the pair
 */
record Pair<S, T>(
    S first,
    T second) {}
```


Calling Generic Functions

This just says that this is a generic function with one type parameter T.
You have to pick exactly one T when you use it.
Java usually does this for you based on the arguments.

```
<T> T Default(Maybe<T> maybe, T else);
```

Essentially, if you give this a `Maybe<String>`, the second argument also has to be a `String`, and the return type will also be `String`

Java Generics

Creating Generic Data Types and Functions

> DISTINCTION-LEVEL CONTENT <

Australian National University

Generic Types

```
/** …    New!
 * @param <T> the type of things that should be named
 * … */
record NamedSomething<T>(T thing, String name) {}

sealed interface NTuple<T> permits Single, Tuple, Triple {}
record Single<T>(T t) implements NTuple<T> {}
record Tuple<T>(T t1, T t2) implements NTuple<T> {}
record Triple<T>(T t1, T t2, T t3) implements NTuple<T> {}
```

Generic Functions

<T, ...> [returnType] [functionName]([argType1] [argName1], ...) {

    ...

}

Can use Type Arguments in:

- ✓ return type
- ✓ argument types
- ✓ local variable types
- ✓ type arguments in code

- ✗ Constructors
  e.g. new T(...)
- ✗ casts, instanceof,
  typecase, static member access
  (not available right now anyway)

Lambdas

λ

Function Abstraction in Java

Australian National University

Recall: Function References

```
runAsTest(this::testSumExample1);
```

These are references to functions you wrote

```
BigBang("test", 0, this::draw);
```

In turn, runAsTest and BigBang abstract over many possible functions.

Function Types

Java allows arbitrary ones, but in Functional Java, we have:

`Supplier<T>` - zero-argument functions that return values of type T

`Function<S, T>` - one-argument functions that take an argument of type S and return something of type T

`BiFunction<S, T, R>` - two-argument functions that take arguments of types S and T, respectively, and return an R

`Predicate<T>` - one-argument functions that take a T, return a boolean

`BiPredicate<S, T>` - two-argument functions that take arguments of types S and T, respectively, and return a boolean

`runAsTest` accepts a `Runnable`, the type of void functions with no arguments.

Currying

Need more arguments?

Use `Function<S, Function<T, R>>`, etc.

See Standard Library for some helper functions around this.

Lambdas

Create an anonymous function

Can use values from context

```
Function<Integer, Integer> makeAdder(int amount) {
    return x -> x + amount;
}
```


Lambdas

Need context

var fun = x -> x;  Don’t know type of function.

Function<Integer, Integer> fun = x -> x;  Context tells us x is an Integer, so result is an Integer

Using Function Values

General form:

`[funVal].[call]([arg1],…)`

[call] depends on function type

`Supplier`	⇔	`get`
`Function`	⇔	`apply`
`BiFunction`	⇔	`apply`
`Predicate`	⇔	`test`
`BiPredicate`	⇔	`test`

```
Function<Integer, Integer> fun = …;
fun.apply(5);
```


Debugging

Part 1:

Reading error messages

Part 2:
Using `println` to figure out where things go wrong

Australian National University

Recap: Evaluation Order

This time for real!

[Live Demo]

I’m going to use IntelliJ here. Don’t use it on your own until Week 6!

Finding Bugs

Questions to ask:

- What should actually happen?
- In order for that to happen, what path should the program take?
- Does that match the path that is taken?
- Where on the path do things go wrong?

  Is the reason there that
  - We are taking the wrong path?
  - We are getting a wrong value?

Practice

Fork and clone the comp1110-2025s1-workshops project.
Create a folder “ws2b”, and work in “World.java” in there.
Commit and push when you are done.

Design a World Program in stages.

Stage 1: the player controls a rectangle that starts in the middle of the world, and can be moved up, down, left, and right with the arrow keys.

Stage 2: on a click, the square turns into a circle. On another click, the circle turns back into a rectangle. The circle can move just like the rectangle.

Stage 3: if there is currently a rectangle, pressing the “w” key increases its width, pressing “e” reduces the width, “h” increases the height, and “j” decreases the height. When changing from a circle to a rectangle, the rectangle’s width and height are reset to their original values.

Stage 4: if there is currently a circle, pressing “r” increases its radius, and pressing “e” decreases the radius. When changing to a circle, the circle’s radius is the smaller of the rectangle’s width and height.