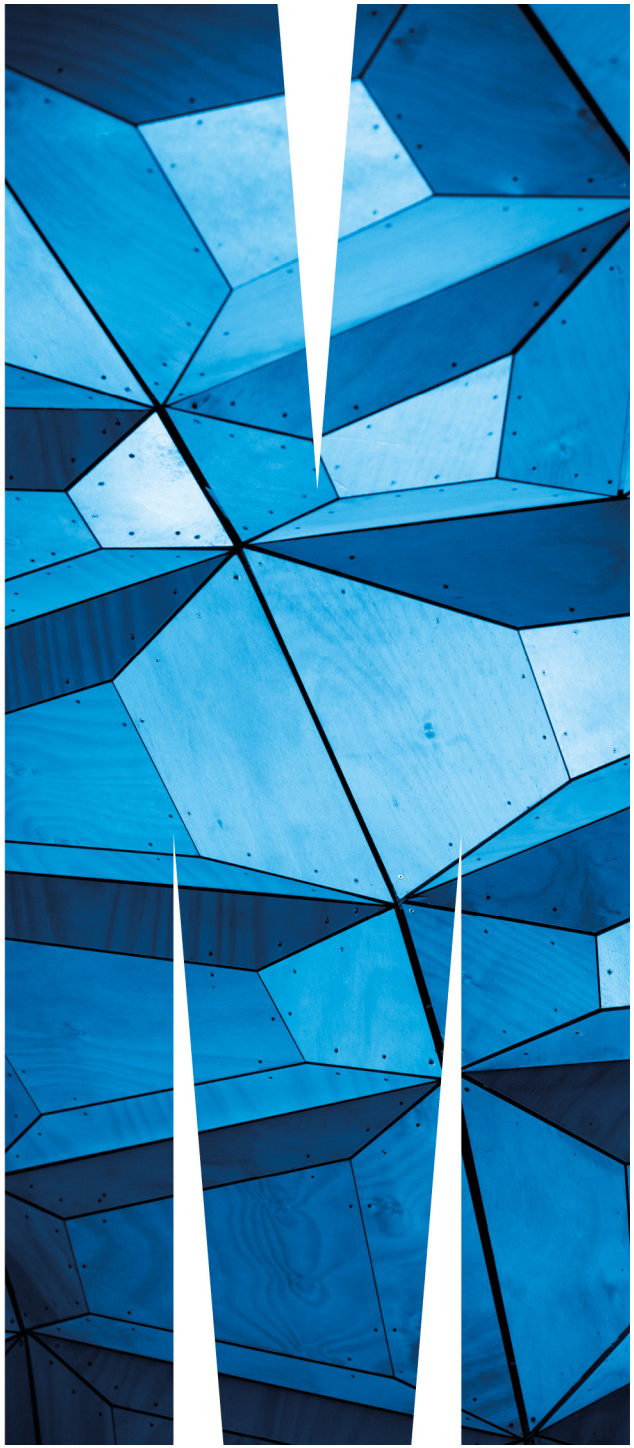
MONASH University

MONASH
INFORMATION
TECHNOLOGY



Java Fundamentals

FIT2099: Object-Oriented Design and Implementation

GROUP OF EIGHT AUSTRALIA

Lecture Outline

MONASH University

- Classes, Methods and Fields
- Java Primitive Types
- The Java Class : `Counter`
- References to an object
- Message-Sending
- The Java Class : `Watch`
- Constructors
- `for`, `if`
- UML representation of system

Java Example Code

MONASH University

- This lecture refers to the example code containing the classes `watch.java`, `counter.java`, etc.
  - The code is available on Moodle
  - Make sure you have a copy, and refer to it throughout the lecture
  - Open it in Eclipse if you can

Classes and Members

MONASH University

- Classes
  - A class is defined by the text you see in a file such as:

    `Counter.java` or `Watch.java`
  - A class has a set of *members*
- Members
  - Members may be:
  - fields (a.k.a. attributes)
    - These store the state of the objects that instantiate the class
  - methods (a.k.a. operations)
    - These specify the Behaviour of the objects − the messages it can receive. They can tell the object to do something (a command), or ask it to answer a question (a query).

A Java Class : `Counter`

MONASH University

- A class is defined by a piece of text typically stored in a single file, e.g. `Counter.java`
- A class consists of:
  - a class declaration, that specifies the name of the class, as well as:
    - its visibility: `public`, `private`, or `protected`
    - any class in inherits from: `extends`
    - any interfaces that it `implements`
  - clauses defining the various members of the class, with their
    - type
    - visibility
    - optionally, initial value for fields
    - definition for methods
      - the code that specifies what the method does

The members of `Counter`

MONASH University

- The member `value` is simply declared as being of type `int`, with no associated algorithm
  - It is a *field*
  - It is initialised to a value of 0
- All other members have a clause of the form

```
public type name() {
      // instructions go here
}
```

  which defines an algorithm
  - This indicates that the feature is a *method*

Type in Java

MONASH University

- Every variable in Java has a type
  - Local variables
  - Instance variables (non-static fields)
  - Class variables (static fields)
- The type of the variable must be specified when the variable is declared
  - This allows compile time type checking
- The type of a variable determines which operations can be applied to it, and what can be assigned to it

```
int num = "frog";
```

will not compile. You can't assign a string to an integer

Benefit of compile-time type checking

MONASH University

- In non-statically typed languages, such as Python, you can do things like this:

```
class Duck:
    def fly(self):
        print("Duck flying")

class Butterfly:
    def fly(self):
        print("Butterfly flying")

class Whale:
    def swim(self):
        print("Whale swimming")

def lift_off(entity):
    entity.fly()

animals = [Duck(), Butterfly(),
Whale()]
```

```
while 1:
    choice = input("Enter 1, 2, or 3: ")
    if (choice >= 1) and (choice <= 3):
        lift_off(animals[choice - 1])
    else:
        print("Invalid choice entered")
```

- This compiles and runs…

  …**until it doesn't**
- You **cannot** compile code like this in Java
- Catching problems early is good!

Java Primitive Types

MONASH University

- Java supports eight *primitive data types*.
  - `byte`: an 8-bit signed two's complement integer
  - `short`: a 16-bit signed two's complement integer
  - `int`: By default, a 32-bit signed two's complement integer
  - `long`: a 64-bit two's complement integer
  - `float`: a single-precision 32-bit IEEE 754 floating point
  - `double`: a double-precision 64-bit IEEE 754 floating point
  - `boolean`: This data type has only two possible values: `true` and `false`
  - `char`: The char data type is a single 16-bit Unicode character
- See https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html for further details
- Non-primitive types can be created by defining *classes* and *interfaces.* Java arrays are also non-primitive.

How to use a class

MONASH University

- One way to use a class is to become a *client* of the class.
  - The simplest and most common way to become a client of a class is to declare an entity (field, local variable, etc.) of the corresponding type
- Formal Definition: client – supplier
  - Let `S` be a class. A class `C` which contains a declaration of the form `S t` (i.e. an entity `t` of type `S`) is said to be a *client* of `S`. `S` is then said to be a *supplier* of `C`
    - In this definition, `t` may be an a field of `C`, a local variable in a method of `C`, or an argument of a method of `C`
- A *client* (or user) of class `Counter` can create an object of type `Counter`

Reference to an object

MONASH University

- At runtime, objects of type `Counter` can be created
- When an object of type `Counter` is created, a *reference* to the object is returned to the client that created it
  - A reference is a run-time value which is either `void` (a.k.a. null) or *attached*
  - If *attached,* a reference identifies a single object
- A reference, if not `void`, is a way to identify an object. It can be thought of as an abstract *name* for the object

Reference and identity of an object

MONASH University

- Every object created during the execution of an OO system has a unique identity, independent of the object's state as defined by the values of its attributes
- In particular:
  - Two objects with different identities may have identical attribute values
  - Conversely, the attribute values of a certain object may change during the execution of a system, but this does not affect the object's identity

Message-Sending

MONASH University

- All work is done by *message-sending*
  - We send a message to an object by invoking one of its methods
- The general form of a method call

```
receiver.someMessage(arg1, arg2,…)
```

- We can read this as

  "Send message `someMessage` to object `receiver` with arguments `arg1`, `arg2`, …"

The methods in Counter

MONASH University

- The methods define the behaviour of the objects of type `Counter`
- Methods `reset()`, `decrement()` and `increment()` do not return a result
  - Methods that do not return a result are often called *procedures* or commands

The methods in Counter (2)

MONASH University

- Other routines can be declared as returning a result:
  - They are often called *functions*
    - If they do not cause any side-effects (i.e. change of any object's state), they represent queries that can be sent to the object
  - They have a type declaration of the form `t f(…)`, for some type `t`, that specifies the type of the value returned by the function `f(…)`
    - The return value of a function is specified by using the `return` keyword, e.g.

```
return value;
```

in the method `getValue()`

The Java Class : `Watch`

MONASH University

- `Watch` has five members:
  - Two fields: `hours`, `minutes`
  - Two methods: `tick()`, `testWatch(int numTicks)`
  - One constructor: `Watch()`

Constructors

MONASH University

- A *constructor* lets us specify what initialization should be done when a new object of the given class is created
  - The constructor `Watch()` creates two new `Counter` objects, and assigns them to attributes `minutes` and `hours`
    - If this were not done, `minutes` and `hours` would still be void references, i.e. not attached to any object
- A constructor is the only sort of method that does not have its type explicitly specified
  - Its return type can be only one thing: an object of the class for which it is a constructor. `Watch()` returns a reference to a `Watch` object
- A class can have multiple constructors, with different signatures
  - i.e. different types and/or numbers of parameters

Creating objects

MONASH University

- The line

```
Counter minutes;
```

Declares a field `minutes` of type `Counter`. It does ***not*** create a `Counter` object

- The line

```
minutes = new Counter();
```

causes a new `Counter` object to be created, and a *reference* to it to be stored in the field `minutes`

- The `Counter` object is initialized by calling the *constructor* method `Counter()` – in this case the Java *default constructor*
  - The default constructor initializes attributes to `0`, `false`, or `void`, depending on the type
- These two things are often combined in a single line:

```
Counter minutes = new Counter();
```

The `for` loop

MONASH University

- The general form of the `for` loop is:

```
for (initializer; entry condition; loop end action) {
    // body
}
```

- The initializer specifies what should be done before entering the loop the first time, e.g.

```
int i = 0 // create a local variable of type int
          // and initialize it to 0
```

- The entry condition is a *boolean* expression that must evaluate to true for the body of the loop to be entered, e.g.

```
i < numTicks // i is less than numTicks
```

- The loop end action is an instruction that is executed after the body has been executed

```
i++ // increment i
```

- After the end action is executed, the entry condition is checked again

The instruction `if`

MONASH University

```
if (boolean_expression) {
  // do something
}
else { // may be empty or omitted
  // do something else
}
```

- This will execute the instructions in the first branch if `boolean_expression` evaluates to `true`, and those in the second branch otherwise, e.g.

```
if (hours.getValue() == 24) { // note the == operator
   hours.reset();             // that checks for equality
}
```

Comment on slides on syntax

MONASH University

- In general, we will not be showing further slides on syntax basics like this in this unit.
  - I hope you already knew all about `for` loops and `if…then…else` from earlier units
- What makes a top programmer?
  - Encyclopaedic knowledge of one language's syntax ✖
  - Excellent knowledge of algorithms and data structures, and when to use which ✔
  - Excellent knowledge of fundamental principles and design patterns ✔
  - Ability to quickly pick up a new language and apply their fundamental knowledge ✔
- Real programmers work with language references available at all times

UML Representation of our system

MONASH University

- Class `Watch` creates two `Counter` objects, `hours` and `minutes`. Each is responsible for its own `value` field, and `reset()` `increment()`, `decrement()`, and `getValue()` methods
  - The user, or client, of these objects need know nothing about how these features are implemented
  - `Watch` is a client of `Counter`