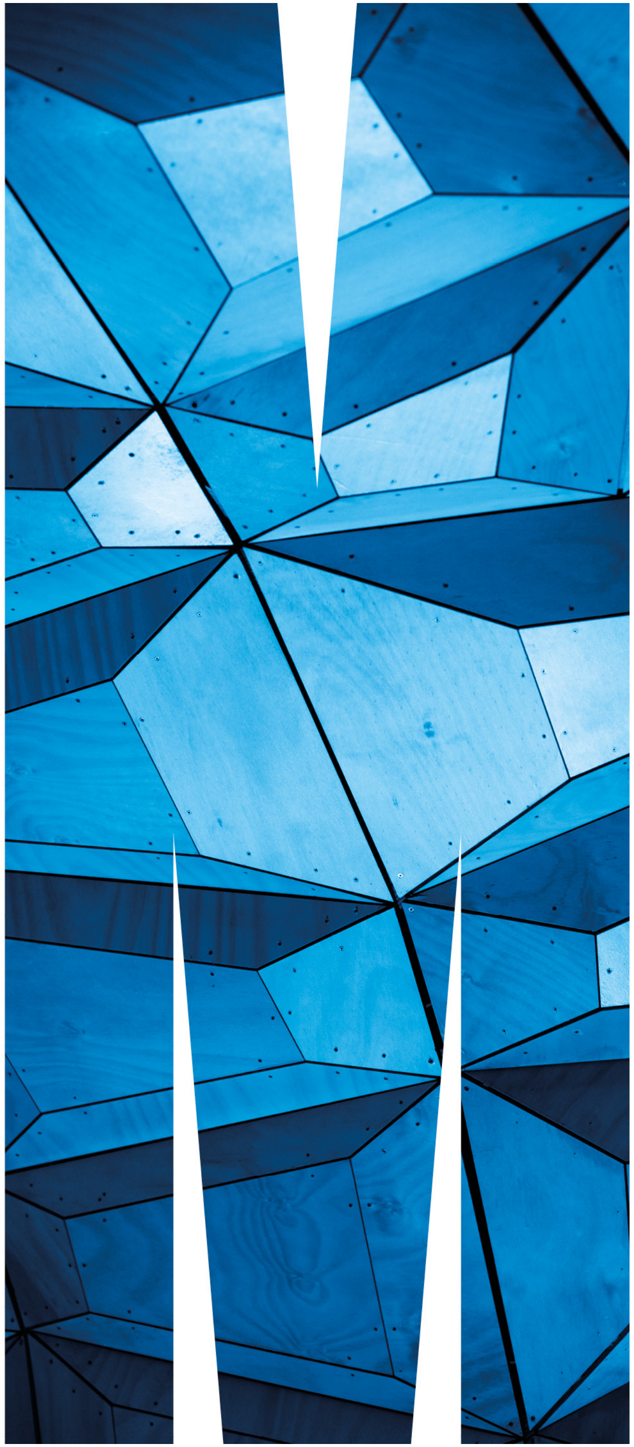
MONASH University

MONASH INFORMATION TECHNOLOGY



Software Specification and Design by Contract 1

FIT2099: Object-Oriented Design and Implementation

GROUP OF EIGHT AUSTRALIA

Lecture Outline

MONASH University

- Object Oriented Programming
  - Classes Revisited
  - Abstract Data Types (ADTs)
  - Client/Supplier Association
- Software Specification
  - Public Interface
- Design By Contract
  - Preconditions, Postconditions, and Invariants
  - Obligations and Benefits

- The basic modular unit in OO programming and design is the class
  – A class describes one implementation of an *abstract data type*
- A class may be *abstract*, in which case it is a *specification* for a set of possible implementations of the abstract data type
  – In Java, a purely abstract class is specified as an *interface*
- Consider the SetDemo example

An Example class: the original `Watch1`

```
public class Watch1 {

  Counter minutes = new Counter();
  Counter hours = new Counter();

  public void tick() {
    minutes.increment();
    if (minutes.getValue() == 60) {
      minutes.reset();
      hours.increment();
      if (hours.getValue() == 24) {
        hours.reset();
      }
    }
  }
```

```
  public void testWatch(int numTicks) {
    for (int i = 0; i < numTicks; i++) {
      System.out.println(
        String.format("%02d”,
                hours.getValue())
        + ":”
        + String.format("%02d”,
                minutes.getValue())
      );
      tick();
    }
  }

}
```

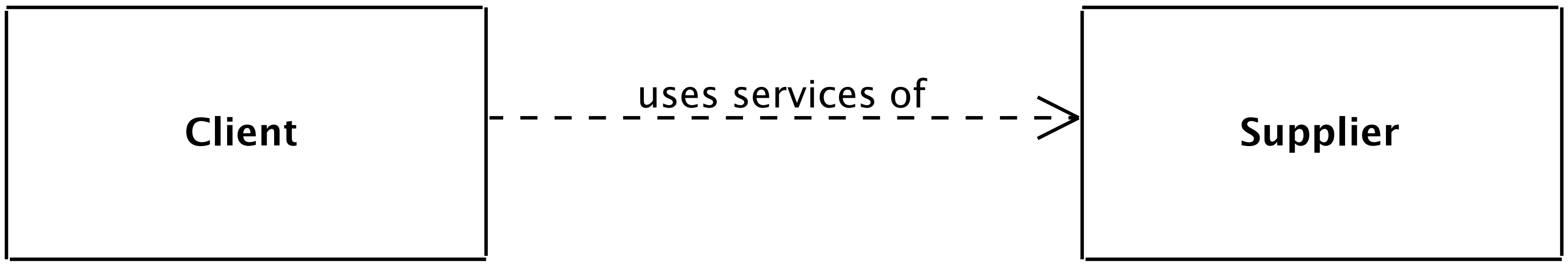
An Example class: the original `Watch1`...

- A class is defined by the text you see in a file such as `Watch1.java`
- A class has a set of attributes and methods
- Class `Watch1` has:
  - two attributes:

    ```
    minutes
    hours
    ```
  - two methods:

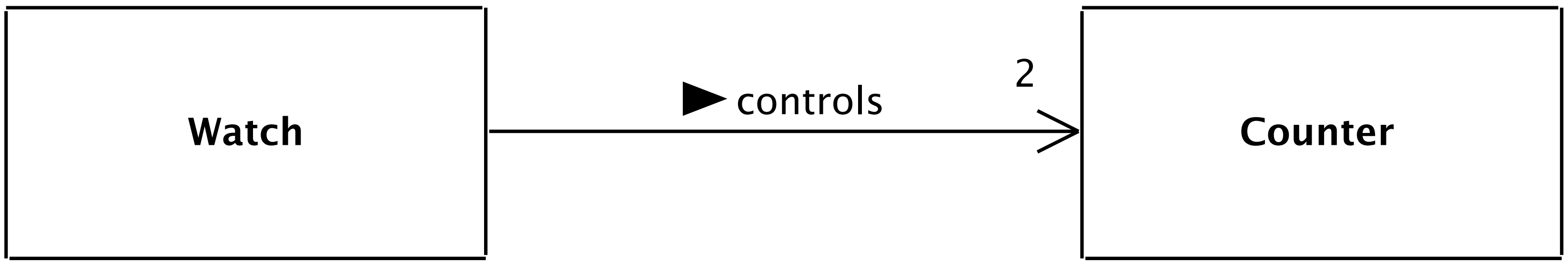
    ```
    tick()
    testWatch(int numTicks)
    ```

Client/Supplier Relationship



- In UML, this is shown as an association or a dependency
  - An association is used if `Client` has an attribute of type `Supplier`

- Class `Watch1` has two attributes of type `Counter`

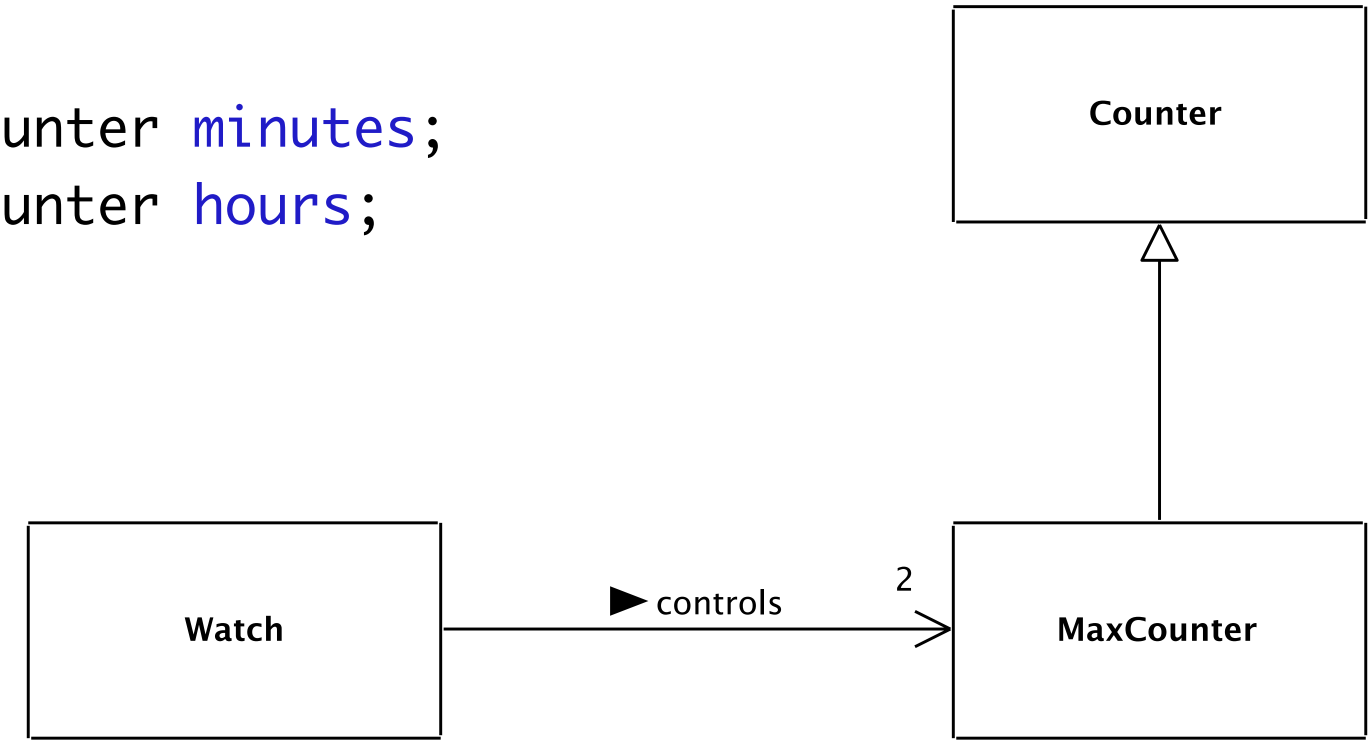


- `Counter` is a supplier of services to `Watch1`
- `Watch1` is a client of `Counter`, and asks it to perform services such as `increment()`, `reset()`, etc.

Client/Supplier and Inheritance: Original `Watch2` Example

- The original `Watch2` has two attributes of type `MaxCounter`:

```
MaxCounter minutes;
MaxCounter hours;
```

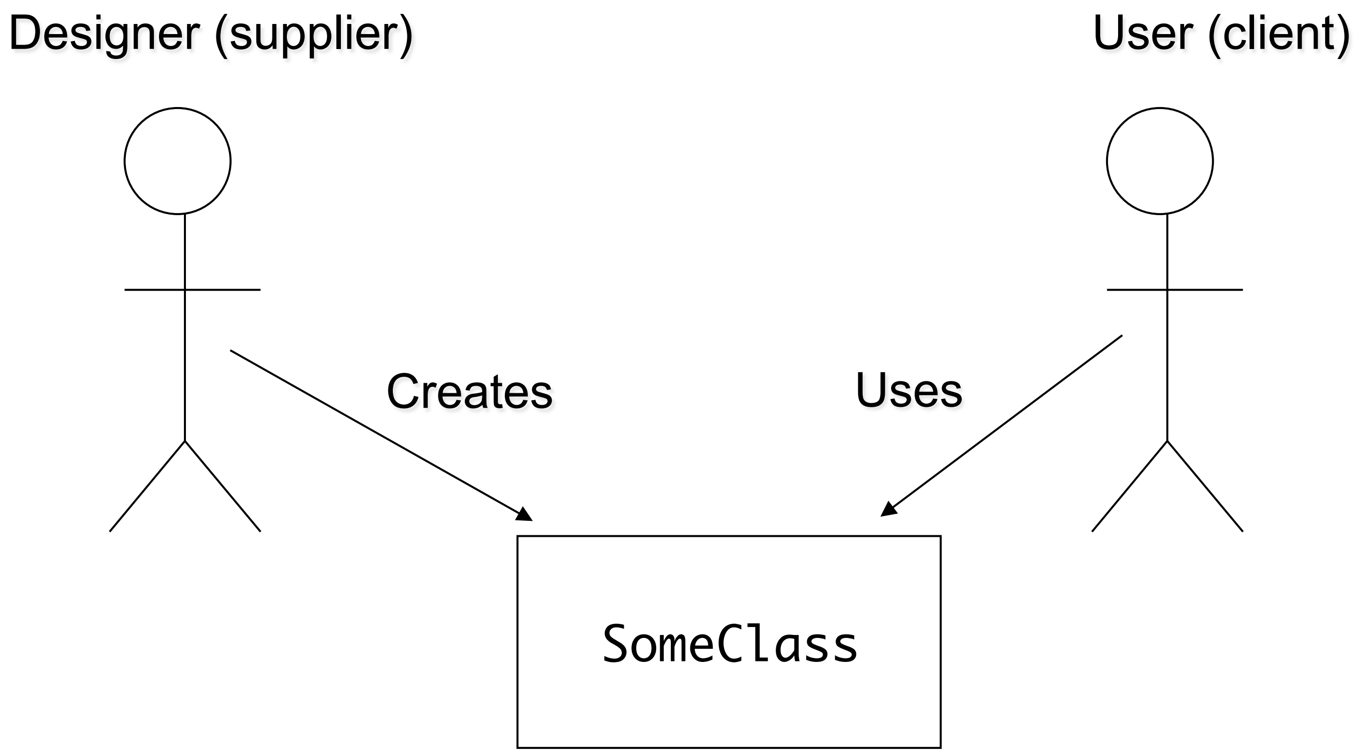


Why isn't software more like hardware? Why must every new development start from scratch? There should be catalogs of software modules, as there are catalogs of VLSI devices: when we build a new system, we should be ordering components from these catalogs and combining them, rather than reinventing the wheel every time. We would write less software, and perhaps do a better job at that which we do get to develop. Wouldn't then some of the high costs, the overruns, the lack of reliability — just go away? Why isn't it so?

Bertrand Meyer, in *Reusability: The Case for Object-Oriented Design*, IEEE Software, 1987.
https://www.computer.org/csdl/mags/so/1987/02/01695711.pdf

- What do hardware components have that software components (usually) lack?
- Hardware Components:
  - Have well-defined *public interfaces* with a hidden (and therefore replaceable) implementation.
  - Have rigorous, unambiguous *specification* of behaviour.
  - Are well-tested, and often *guaranteed*.

- A class designer establishes a *software contract* between *him/herself* and the *user(s)* of the class he/she designs.
- We can make this impersonal, and think of this as a contract between the class that is the *supplier*, and the classes that are *clients* of that class

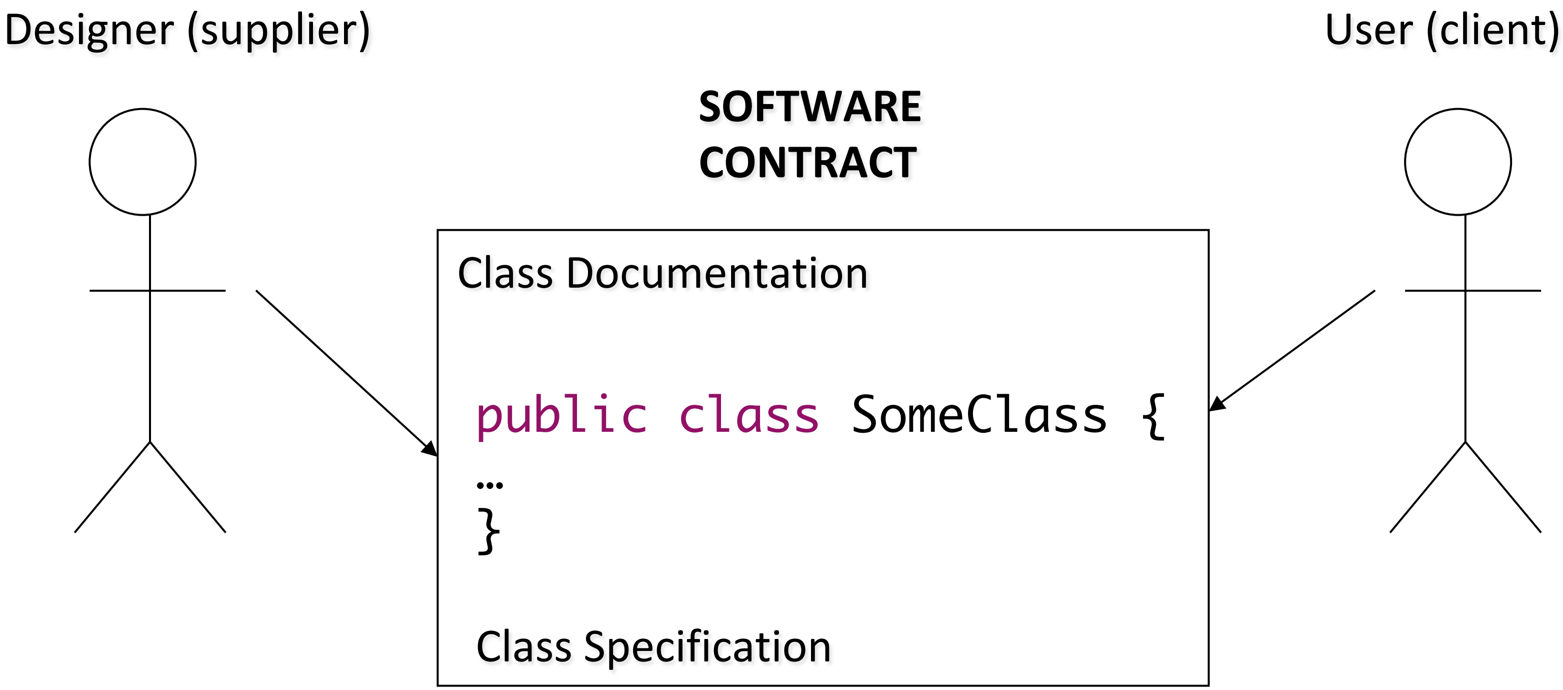


This relationship is governed by the ***contract*** of the class

(NB. This diagram is *not* UML)

- The software contract provides
  - the documentation of the class for the technical user
  - the possibility of enforcing the contract by using exceptions and assertions

Design by Contract...



- The software designer tells the user what the class does by providing a specification for the class
  - What the methods of the class need to operate correctly
  - What the class will guarantee to be true if it is used correctly

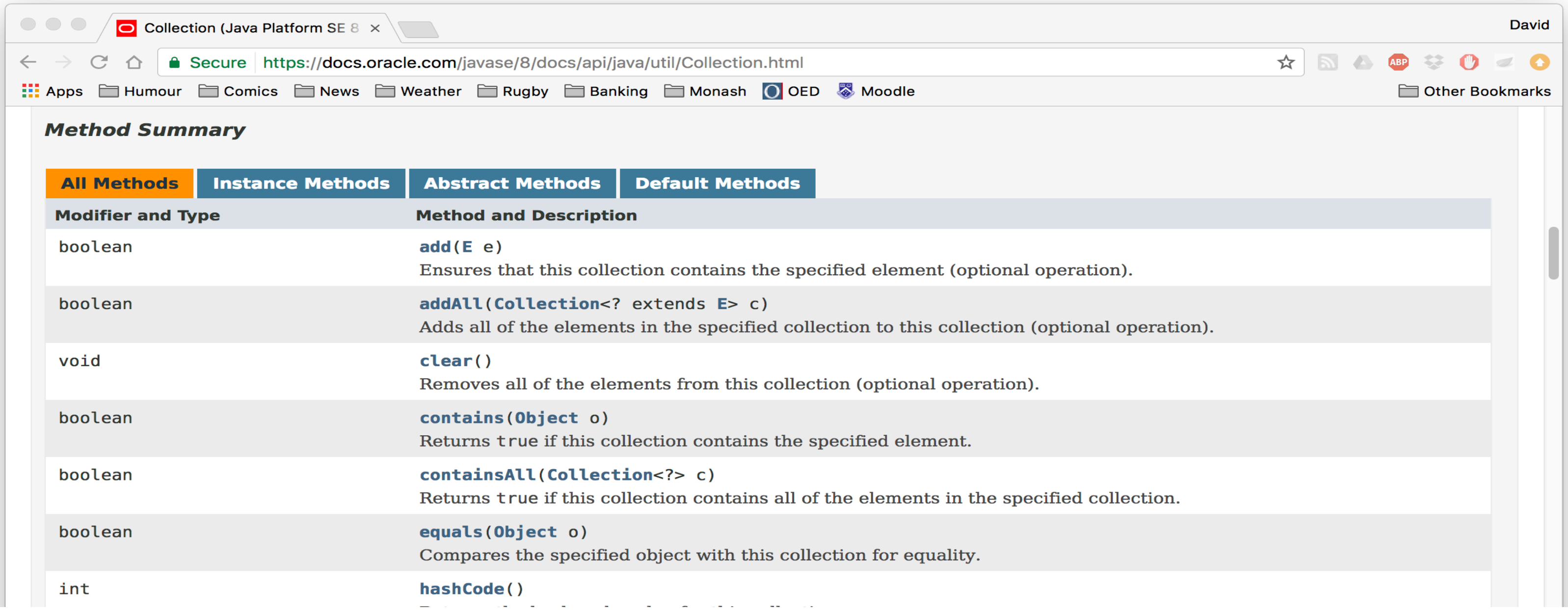
- A specification:
  - Is ideally part of the implementation
    - in some languages, such as Eiffel, this is built in
      - in others it can be done by hand, via the use of assertions and exceptions
    - There are also language extensions available, such as:
      - Cofoja (Contracts for Java):
        http://code.google.com/p/cofoja/
      - Spec# and Code Contracts from Microsoft Research for C# and .Net
        https://www.microsoft.com/en-us/research/project/spec/
        PyContracts for Python
        https://pypi.python.org/pypi/PyContracts

- A specification:
  - should ideally be extractable from the implementation via a tool
    - e.g. by Javadoc when using Cofoja
  - is essential for supporting component reuse, and maintenance
  - Is more than just the API we have gotten used to seeing
    - It includes comments, and crucially, contracts defined by **executable specifications**

- The user
  - should be able to determine how to use the class by reading its specification (only)
  - should not have to look at the implementation details (how the class goes about meeting its specification)
- The specification forms the **public interface** of the class

MONASH University

- Part of the API documentation for the Java `Collection` interface:



Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
`boolean`	`add(E e)` Ensures that this collection contains the specified element (optional operation).
`boolean`	`addAll(Collection<? extends E> c)` Adds all of the elements in the specified collection to this collection (optional operation).
`void`	`clear()` Removes all of the elements from this collection (optional operation).
`boolean`	`contains(Object o)` Returns `true` if this collection contains the specified element.
`boolean`	`containsAll(Collection<?> c)` Returns `true` if this collection contains all of the elements in the specified collection.
`boolean`	`equals(Object o)` Compares the specified object with this collection for equality.
`int`	`hashCode()`

- Method detail - note logical statements specifying behaviour

isEmpty

`boolean isEmpty()`

Returns `true` if this collection contains no elements.

Returns:
`true` if this collection contains no elements

contains

`boolean contains(Object o)`

Returns `true` if this collection contains the specified element. More formally, returns `true` if and only if this collection contains at least one element `e` such that `(o==null ? e==null : o.equals(e))`.

Parameters:
`o` - element whose presence in this collection is to be tested

Returns:
`true` if this collection contains the specified element

Throws:
`ClassCastException` - if the type of the specified element is incompatible with this collection (optional)
`NullPointerException` - if the specified element is null and this collection does not permit null elements (optional)

- Contracts can be defined by using assertions and exceptions to create ***executable specifications***
  - They are then ***verifiable*** by the compiler or (usually) the running code
  - ***Go beyond comments*** that logically describe the behaviour required

- Preconditions: (a.k.a. “requires”)
  - What the *client* must guarantee to do
    - State the requirements for using the method
  - Usually in the form of constraints on the arguments to method calls
  - Violation of a precondition indicates a bug

- Postconditions: (a.k.a. “ensures”)
  - What the *supplier* guarantees to provide
    - State what the method will do
  - Promise that certain conditions will be met after a method has been called
  - Violation of a postcondition is often, but not always, due to a bug. More later.

- Class Invariants:
  - Conditions that must hold at all times for a class to be valid
    - "at all times" actually means before and after a method is called
  - Difficult to implement without true language support for DbC

- The use of supplier methods by a client class should be governed by a precise description of the mutual benefits and obligations

```
public double geometricMean(double a, double b) throws
Exception {
  /*
   * Start Preconditions
   */
  // Precondition: firstArgumentNonNegative
  if (a < 0) {
    throw new Exception("Precondition violated:
firstArgumentNonNegative");
  }
  // Precondition: secondArgumentNonNegative
  if (b < 0) {
    throw new Exception("Precondition violated:
secondArgumentNonNegative");
  }
  /*
   * End Preconditions
   */

  double result = Math.sqrt(a*b);
  //double result = (a + b)/2; // WRONG This is the
arithmetic mean
  /*
   * Start Postconditions
   */
  // Postcondition: invertingGeoMeanAccurate
  /* Note that we should always avoid checking for
   * equality between floating point values, due to
   * finite machine precision
   */
  assert(Math.abs(result*result- a*b) < 1e-10) :
  "Postcondition violated: invertingGeoMeanAccurate";
  /*
   * End Postconditions
   */
   return result;
}
```

`geometricMean(…)`	**Obligations**	**Benefits**
Client	Supply non-negative arguments	Get geometric mean calculated
Supplier	Calculate geometric mean correctly	Simpler processing due to assumption of non-negative arguments

MONASH University

- In some cases the implementation of a routine and its postcondition can look very similar
  - This is not redundant, however
    - The two things are fulfilling very different roles
- Often we can, and should, write the precondition for a routine long before we have decided – or even know – how we are going to implement it
  - It is part of the ***specification***, not the implementation

- Consider the postcondition for a routine to compute the square root of a number:

```
/*
 * Start Postconditions
 */
// Postcondition: squareRootAccurate
/* Note that we should always avoid checking for equality between floating point values, due to
 * finite machine precision
 */
assert(Math.abs(result*result - x) < 1e-10) : "Postcondition violated: squareRootAccurate";
/*
 * End Postconditions
 */
```

  – Note that it is much easier to write this specification than to write the corresponding implementation
    - Do you know how to calculate a square root?

- Specification helps to fill the gap between Analysis and Design
- Exception throwing and assertions can be used to create ***executable specifications***
- Ideally, the ***public Interface*** of a class is the specification:
  - Comments
  - Method signatures (name and typed arguments)
  - Preconditions, postconditions, and invariants
- Design by Contract
  - The use of supplier features by a client class are governed by mutual benefits and obligations

References

MONASH University

- Meyer, B., *Object Oriented Software Construction*, Second Edition, Prentice Hall 1997, Ch. 11.
- Martin, R. *The Liskov Substitution Principle*, The C++ Report, 1996.
  https://drive.google.com/file/d/0BwhCYaYDn8EgNzAzZjA5ZmItNjU3NS00MzQ5LTkwYjMtMDJhNDU5ZTM0MTlh/view
- Oracle, *Java Tutorial on Exceptions*, 2017.
  https://docs.oracle.com/javase/tutorial/essential/exceptions/

Appendix: Contracts for Java (Cofoja)

MONASH University

- The Cofoja version of the `geometricMean(…)` example is much more succinct and clear

```
import com.google.java.contract.Ensures;
import com.google.java.contract.Requires;

public class DBCDemo {

  @Requires({
    "a >= 0”,
    "b >= 0”
  })
  @Ensures({
    "Math.abs(result*result- a*b) < 1e-10"
  })
  public double geometricMean(double a, double b) {

    double result = Math.sqrt(a*b);
    return result;
  }

}
```