DESIGN BY CONTRACT

What is a contract?

In the real world, a contract is a binding agreement between two parties, the *supplier* and the *client*: the client buys the products or services of the supplier.

Contracts specify the obligations and benefits of the two parties.

Example of contract between airline (supplier) and passenger (client):

> Provided that the client appears at the airport 2 hours early, pays the full fare in advance, and brings acceptable luggage, the airline will ensure that the client and their luggage will be taken to West Palm Beach.

	Obligations	**Benefits**
Client		
Supplier		

Contracts in Programming

The contract idea can be applied to programming.

Design by Contract: introduced by Bertrand Meyer in the late 1980's, and used first in the Eiffel programming language. Built around the concept of assertions.

Assert: To state or express positively.

Assertion: a boolean-valued statement that describes the program state (the variable values) at a point in a program's execution. The code that follows an assertion can be written relying on that assertion.

```
a print method for CircularQueue:
    loop i from head to tail (wrapping around) {
        Assertion: 0 <= i < A.length
        print contents[i]
    }
```


Method contracts

Every method provides services, and thus is a supplier. These services are described in the method comment.

- A legal contract describes the services provided by a supplier.
- A method comment describes the services provided by that method.

The Method Contract:

> If the method precondition holds (evaluates to true) before a method call, then (1) the method will halt without crashing, and (2) after the method is executed the postcondition will hold.

Just like legal contracts, method comments almost always just state *what* is done, not *how* it gets done.

Writing method preconditions and postconditions

In these slides, method comments are written as follows:

```
/**
 * Method description
 * Requires: ...
 * Ensures: ...
 */
```

- Preconditions are marked with `Requires`
- Postconditions are marked with `Ensures`

Postconditions are sometimes described in the method description, rather than as an explicit separate statement.

Why all the legal mumbo-jumbo?

Design by Contract means viewing software as a set of contracts. This is very practical and useful. The larger the program, the more important it is.

- It helps you **write** the program and, especially, track down bugs.
- It helps other programmers (clients of your code) **use** the classes and methods you write.
- It helps you and your team **maintain** the code you own.
- It helps you **convince** yourself and others that your code is correct.

General contracts

Preconditions and postconditions can be applied to any section of code.

```
// Precondition
section of code
// Postcondition
```

The Contract:

> If the precondition holds (evaluates to true) before the code is executed, then (1) the code will halt without crashing, and (2) after the code is executed the postcondition will hold.

Example:

```
// 0 <= i < b.length
b[i] = 12;
// b[i] == 12
```

Audiences

There are two audiences for your comments:

- Programmers who use your code
  - **External** to the project
  - Can't see your implementation (method bodies, private stuff)
  - Need to know **what** classes and methods are for
- Programmers who maintain your code
  - **Internal** to the project
  - Including yourself
  - Need to know **why** code was written and **how** it works

External comments

Think of the Java APIs. You can't see the implementation:

- No local variables revealed
- No private variables or methods revealed

Implications:

- The API simply **can't** mention that stuff in external comments.
- It also can't mention particular algorithms. (Unless it's necessary information, eg., information relevant to performance.)

Exercise: Read code and comments that you wrote several months ago. Is it as clear how to use your classes and methods as it is when reading the online Java APIs, or do you have to look at the implementation?

Internal comments

Internal comments explain *how* the code works and *why* it was designed that way. They:

- explain design decisions, algorithm choices, tricky bits of code, and the purpose of member and local variables.
- state assertions about the class data.

Internal comments should appear on member variables and *inside* method bodies.

Exercise: Read code and comments that you wrote several months ago. If you had a bug, are your comments useful for understanding how the code should work?

Types of comments

Class comments (external)

Class comments describe the purpose of the class:

- What the class represents (e.g., student, network connection, queue)
- General properties of the class (e.g., a Set contains unique entries)

A brief summary comment at the top of the class is usually enough for small classes, but with confusing or large ones it is reasonable to provide examples of use.

Example:

```
/**
 * A skeletal implementation of the Set
 * interface to minimize the effort required
 * to implement that interface.
 */
public abstract class AbstractSet
    extends AbstractCollection implements Set
```


Class comments (internal)

Internal class comments consist of comments on the instance variables and "representation invariants".

Invariant: Something that never changes. In programming, something that is always true, such as a restriction on the range of values an `int` variable will have.

Representation invariant: describes the properties that instance variables of a class must have at all times, except while its methods are executing:

- how the instance variables represent the ADT.
- constraints and restrictions on the values of *individual* instance variables.
- relationships *between* the values of the instance variable.

For example, in the `CircularQueue` implementation our representation invariant specified

- that the queue data is stored in array `contents`,
- what order the objects are in,
- where the front and back of the queue are,
- restrictions on the possible values of `size`, `head`, `tail`, and
- the relationship between `size`, `head`, `tail`, and the contents of array `contents`.

Without this, how would you know what it means when `head` and `tail` are equal – is the queue empty or full?

A representation invariant is very helpful because it tells the person writing the methods:

- exactly what properties they must maintain in order for the instance variables to make sense, and for subsequent method calls to work. Collectively, this is called *internal consistency*.
- to watch out for particular borderline cases and so avoid errors.

Method comments (external)

Method comments specify the method's contract. They should include:

- **Method Summary:** A concise and complete summary of the method's purpose.
  - *must* mention every parameter
  - must describe the method's effects.
- **Precondition:** Properties that the method depends on.
- **Important performance facts (if any):** For example, how fast is your code and how much memory does it use?

In Java, they are usually written using JavaDoc. Sun's Java APIs are all produced from JavaDoc comments.

External method comments should not mention any private or local information. Instead, external comments describe everything using only:

- the method parameters,
- the public methods,
- the abstract contents of the class.

Exercises

Read the Java API specification for method `String.regionMatches()` and find the components listed above.

Identify flaws in the following method specification comments:

```
From ArrayQueue:
    /** Return the first object in contents. */
    public Object head();


From CircularQueue:
    /** Append o to me, wrapping tail around to the
        front of the array if necessary. */
    public void enqueue(Object o);


From java.lang.StringBuffer:  (not as published)
    /** Replaces part of me with a new String. */
    public StringBuffer replace(
        int start, int end, String str);
```


Example: The Queue contract

Here are two method comments from our `Queue` interface:

```
/** Append o to me. */
public void enqueue(Object o);

/** Remove and return my front Object.
    Requires: size() != 0. */
public Object dequeue();
```

Note the precondition on `dequeue`. Here is how to read the specification as a contract:

> "If the queue is not empty then a call to `dequeue()` will halt without crashing, and will remove and return the front object in the queue."

There is no precondition on `enqueue()`. This means the programmer is claiming there are $\boxed{\text{no}}$ conditions that cause it not to fulfill its contract.

Is this claim accurate?

Comments inside methods (internal)

Internal method comments should describe:

- Local variable descriptions, including their relationships.
- Descriptions of tricky bits of code. (Every loop should have a comment saying in general what it does.)
- Reasons for choice of algorithms. (Performance vs. ease of programming, for example.)

Summary of Comment Types

External comments

- class: class summary
- method: method specifications
- class or method: performance facts

Internal comments

- class: representation invariants
- method: assertions
- method: other comments explaining how the code works and why it was designed that way.