Designing Classes

How to write classes in a way that they are easily understandable, maintainable and reusable

Main concepts to be covered

- Responsibility-driven design
- Coupling
- Cohesion
- Refactoring

- Enumerations

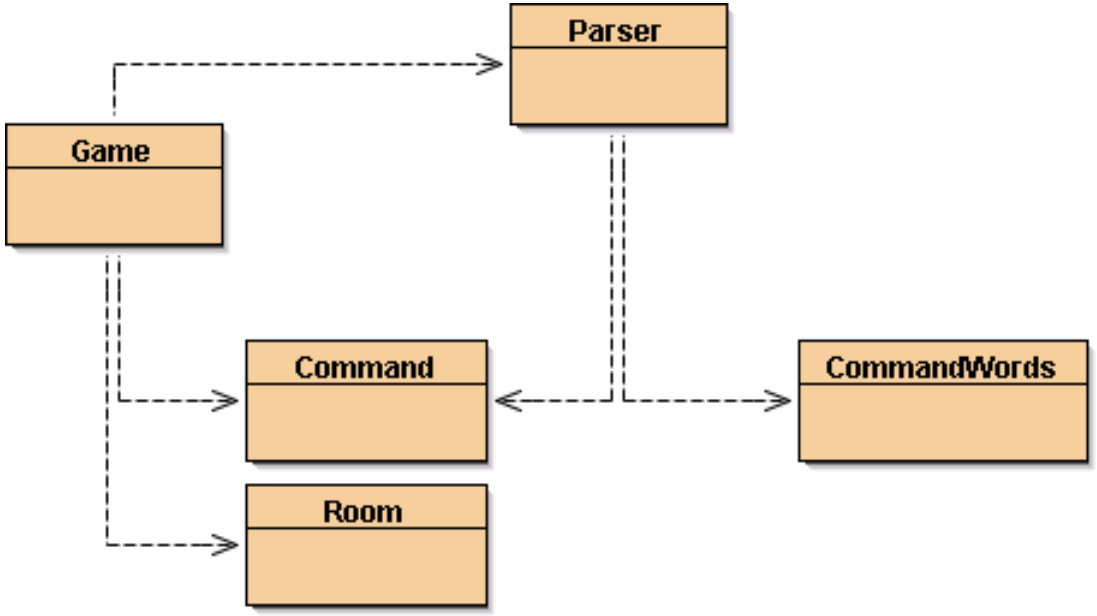
Software changes

- Software is not like a novel that is written once and then remains unchanged.
- Software is extended, corrected, maintained, ported, adapted …
- The work is done by different people over time (often decades).

Change or die

- There are only two options for software:
  - Either it is continuously maintained …
  - Or it dies
- Software that cannot be maintained will be thrown away.

World of Zuul



An Example

- Add two new directions to the *World of Zuul*:
  - up
  - down
- What do you need to change to do this?

Code Quality

- Two important concepts for quality of code:
  - Coupling – *as little as possible please*
  - Cohesion – *as much as possible please*

Coupling

- Coupling refers to links between separate units of a program.
- If two classes depend closely on many details of each other, we say they are *tightly coupled*.
- We aim for *loose coupling*.

Loose Coupling

- *Loose coupling* makes it possible to:
  - understand how to use a class *without having to understand how that class works*;
  - change the implementation of a class *without affecting those classes that use it*.

This improves maintainability … makes change easier.

Cohesion

- *Cohesion* relates to: the *the number and diversity of tasks* for which a single unit is responsible.
- If a programming unit is responsible for one logical task, we say it has *high cohesion*.
- Cohesion applies both at the level of *classes* and of *methods*.
- We aim for *high cohesion*.

High Cohesion

- *High cohesion* makes it easier to:
  - understand what a class or method does *(because it only does one thing)*;
  - Choose and use *descriptive names*;
  - *reuse* classes or methods.

This improves usability … and makes change easier.

Cohesion of Classes

Classes should represent just one, and only one, well defined entity.

Cohesion of Methods

A method should be responsible for one, and only one, well defined task.

Code Duplication

This is an indicator of bad design.

It makes maintenance harder.

It can lead to introduction of errors, especially during maintenance.

Responsibility-Driven Design

- Where should we add new fields and new methods *(and to which class)*?
- The class that holds the data *(fields)* processes *(methods)* the data.
- *Responsibility-driven design* leads to low coupling.

Localising Change

- One aim of reducing coupling and responsibility-driven design is to localise change.
- When a change is needed, as few classes as possible should be affected.

Improve maintainability … make change easier.

Think Ahead

- When designing a class, think what changes are likely to be made in the future.
- Aim to make those changes easy.

Improve maintainability … make change easier.

Refactoring

- When classes are maintained, code usually needs to be added.
- Lists of fields, lists of methods and the code inside methods tend to become longer.
- Every now and then, classes and methods should be *refactored* to maintain *cohesion* and *low coupling*.

Refactoring and Testing

- When *maintaining* code, separate the *refactoring* from making other changes.

e.g. `zuul-bad` to `zuul-better`.

- *Do the refactoring first*, without adding to the functionality.

- *Test before and after refactoring* to ensure that nothing gets broken.

Design Questions

- Common questions:
  - How long should a class be?
  - How long should a method be?

Answered with regard to ***cohesion*** and ***coupling***.

Design Guidelines

- A method is *too long* if it does *more then one logical task*.
- A class is *too complex* if it represents *more than one logical entity*.

Note: these are ***guidelines*** …
everything is still open to the designer.

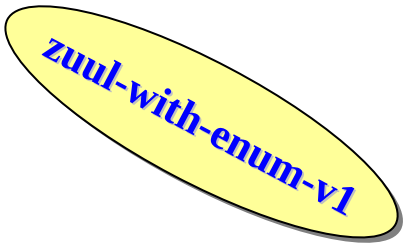
Enumerated Types

- A new language feature.
- Uses **`enum`** instead of **`class`** to introduce a new type name and a set of named constants of that type.
- This is a better (safer) alternative to a set of named static **`int`** constants.
- For the *World of Zuul*, we shall use names for command words, rather than string literals (which will appear only in one place).

`java.sun.com/docs/books/tutorial/java/javaOO/enum.html`

A basic enumerated type

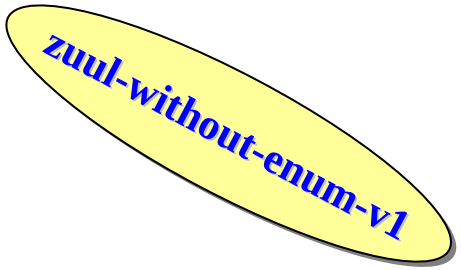
```
public enum CommandWord
{

    // A value for each command word,
    // plus one for unrecognised commands.
    GO, QUIT, HELP, UNKNOWN

}
```



Enumerated objects are constructed implicitly.

Each name represents an *object* of the enumerated type, e.g. `CommandWord.HELP`.

Equivalent class type

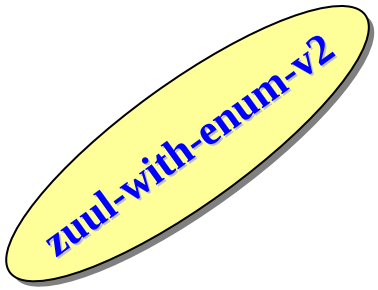



```
public class CommandWord
{

  // A object for each command word,
  // plus one for unrecognised commands.
  public final static Commandword GO = new CommandWord ();
  public final static Commandword QUIT = new CommandWord ();
  public final static Commandword HELP = new CommandWord ();
  public final static Commandword UNKNOWN = new CommandWord ();

  private CommandWord ()
  {
    // nothing to do
  }

}
```

The constructor is declared **`private`** so that no **`CommandWord`** *objects*, other than the ones declared here, can be constructed.

Another enumerated type

```
public enum CommandWord
{

  // A value for each command word,
  // plus one for unrecognised commands.
  GO("go"), QUIT("quit"), HELP("help"), UNKNOWN("unknown");

  private String commandString;

  private CommandWord (String commandString)
  {
    this.commandString = commandString;
  }

  public String toString ()
  {
    return commandString;
  }

}
```



Equivalent class type

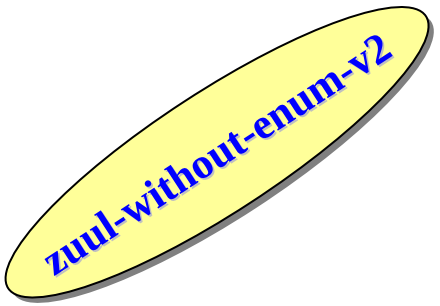
```
public class CommandWord
{

  // A object for each command word,
  // plus one for unrecognised commands.
  public final static Commandword GO =
    new CommandWord ("go");
  public final static Commandword QUIT =
    new CommandWord ("quit");
  public final static Commandword HELP =
    new CommandWord ("help");
  public final static Commandword UNKNOWN =
    new CommandWord ("?");

  ...  private final field (String)
  ...  private constructor (to construct above constants)
  ...  public toString (returns String field)
  ...  public static values (returns array of above constants)

}
```

zuul-without-enum-v2

Automatically provided by an **enum**

Equivalent class type

```
public class CommandWord
{

  // A object for each command word,
  // plus one for unrecognised commands.
  public final static Commandword GO =
    new CommandWord ("go");
  public final static Commandword QUIT =
    new CommandWord ("quit");
  public final static Commandword HELP =
    new CommandWord ("help");
  public final static Commandword UNKNOWN =
    new CommandWord ("?");

  private final String commandString;

  private CommandWord (String commandString) {
    this.commandString = commandString;
  }
```



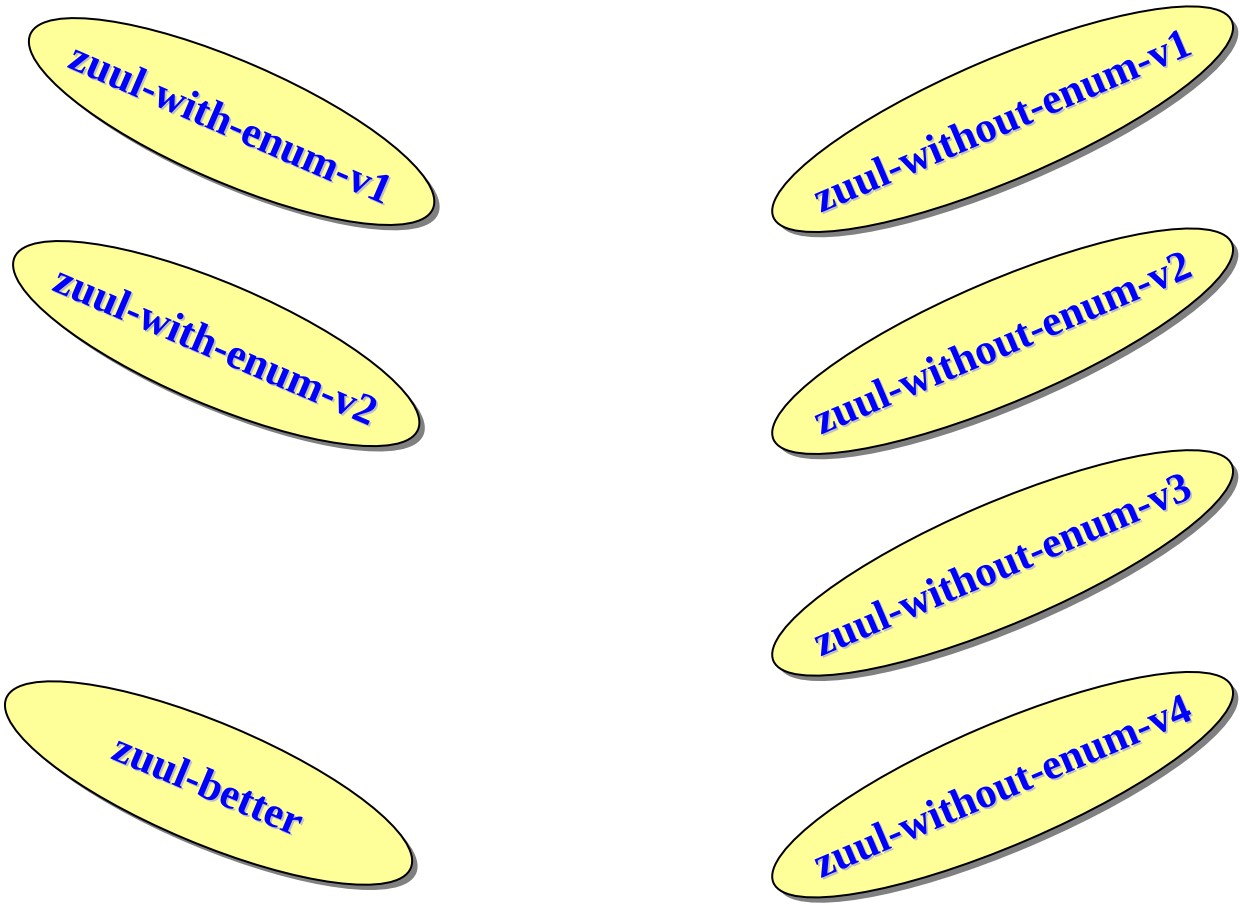
Equivalent class type

```
public class CommandWord
{

  ...  public final statics (GO, QUIT, HELP, UNKNOWN)

  private final String commandString;

  private CommandWord (String commandString) {
    this.commandString = commandString;
  }

  public String toString () {
    return commandString;
  }

  public static CommandWord[] values () {
    return new CommandWord[] {GO, QUIT, HELP, UNKNOWN};
  }

}
```

zuul-without-enum-v2

Automatically provided by an **enum**

See also ...



`projects\chapter07\`

`courses\co882\projects-phw`

Review

- Programs are continuously changed.
- It is important to anticipate change and make it as easy as possible.
- *Quality of code requires much more than just performing correctly!*
- Code must be *understandable* and *maintainable*.

Review

- Good quality code *avoids duplication*, displays *high cohesion*, *low coupling*.
- Coding style (*commenting*, *naming*, *layout*, etc.) is also very important.
- There is a big difference in the amount of work required to *maintain* poorly structured and well structured code.
- In fact, unless it is well structured, *the code is doomed* ... it will not be used for long.