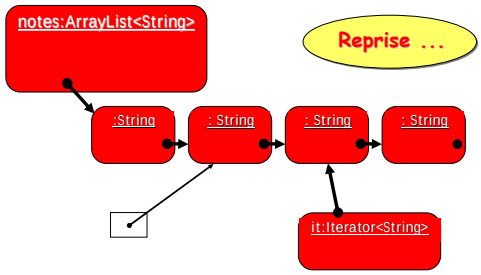
Make Sure You Know All This!

☺☺☺




```
Iterator<String> it = notes.iterator();
while (it.hasNext()) {
    System.out.println (it.next());
}
```

✔

object diagram

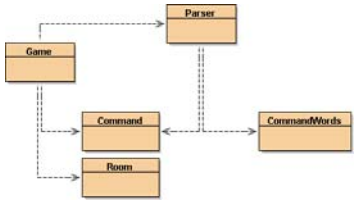
- Objects and Classes
  - State
    - fields
  - Constructors
    - parameters
    - *Initialise the fields*
  - Methods
    - parameters
    - `return` Values
    - *Operate on the fields*

Class diagrams show if one class refers to another (anywhere in its implementation).

Object diagrams show field values (which may refer to specific other objects).

- A *class* is a template from which *objects* can be constructed.
- A *class* defines the *data fields* held by its objects, together with logic for *constructors* of its objects and *methods* for accessing and operating on their data.
- Methods (and constructors) may declare and use their own *local variables* – these exist only for the duration of their method (or constructor). On the other hand, *data fields* last for the duration of their object.
- Methods and constructors may be passed *parameters* – these exist only for the duration of the method or constructor.

World of Zuul



class diagram

- Methods may be classified as:
  - ***accessors*** : these return some value computed from the data fields (often just the value of a field – e.g. `getYear()`). They do not change any field values.
  - ***mutators*** : these return nothing (i.e. `void`) but they do operate on and change (some) field values.
  - some are both ***accessor*** and ***mutator***.
- Data fields, parameters and local variables either have *class* types *(i.e. they are references to other objects)* or *primitive* types (e.g. `int`, `float`, `boolean`, `char`, …).
- Data fields, constructors and methods may be declared `public` or `private`:
  - fields should (almost) always be `private`.
  - some methods and (most) constructors will be `public`.
  - Supporting methods and (some) constructors will be `private`.

- Execution Structures
  - Assignment
    - `col = something.getColour ();`
  - Conditional
    - `if (a <= b) { ... }`
    - `switch (ch)*` `{ ... }`

    * not taught in this module
  - Looping
    - `for (Thing t : allMyThings) { ... }`
    - `for (int i = 0; i < A.length; i++) { ... }`
    - `while (searching && (i < n)) { ... }`

- Method Invocation
  - within the same class:
    - `checkDayOverflow ();`
    - `if (!legalDate ()) { ... }`
  - on objects other than `this` one:
    - `responder.generateResponse ();`
  - on static methods of another class:
    - `if (DateStuff.legalDate (day, month, year)) { ... }`
- Object Construction
  - from the `Notebook` project:
    - `notes = new ArrayList<String> ();`

- Commonly Used Classes
  - Collections
    - Lists (e.g. `ArrayList<String>`)
    - Sets (e.g. `HashSet<Car>`)
    - Iterators (obtained from the above)
    - Arrays (e.g. `Date[366]`)
    - Maps (e.g. `HashMap<String, Address>`)
  - Others
    - `String` (e.g. `equals()` method, `+` operator)
    - `Random` (e.g. `nextInt(n)` method)
    - `System` (e.g. `static` field: `System.out`)

- Coding standards
  - Layout
    - indentation
    - position of `{`…`}`s
    - Spacing (e.g. after commas, around operators, …)
  - Documentation
    - class comments (formal javaDoc syntax)
    - method comments (formal javaDoc syntax)
    - local comments (e.g. `// single line comments`)
  - Naming
    - class names are *nouns* (and start with Upper case)
    - method names are *verbs* (and start with Lower case)
    - fields and variables can be *abstract* (and start with Lower case)

- Other Things
  - Package Libraries
    - standard JDK API (Google: "Java API")
    - `import` statements (e.g. `java.util.*`)
    - write our own packages (e.g. `package co520.stuff;`)*
  - Declaration Qualifiers
    - `public` / `private` (*visibility* of classes, fields, constructors and methods)
    - `static` (*class ownership* of fields and methods)
    - `final` (fields *initialised once – no re-assignment allowed*)

    * not taught in this module

- Other Things
  - Design Issues
    - *coupling* between classes (bad)
      - where the code in one class depends on implementation decisions (e.g. field structure, magic strings) in another.
    - *cohesion* (good)
      - a class should be responsible for *one* conceptual item.
      - a method should be responsible for *implementing one* action.
      - a method may action several things … but only by *invoking* other methods that *implement* them separately.
      - the set of methods in a class should be *complete*.

– Design Issues *(Continued)*

- *STREAM*
  - write *stub* methods, with parameters and documentation, and check they compile.
  - think about *testing* as you design (may need more methods).
  - consider alternative *representations* to hold class information.
  - *evaluate* consequences for implementation from each representation.
  - declare the *attributes (fields)* for chosen representation.
  - complete the stub bodies for all the *methods:*
    - » *Mañana* principle: *defer* special cases, hard logic, heavy functionality, nested loops, duplicated code to `private` methods.
    - » write stubs for these deferred methods, invoke where needed, compile and test (as far as the stub code allows).
    - » declare extra fields (as necessary) and complete the stub bodies for the deferred methods.

- Other Things
- – Testing
  - *unit testing*
    - test each class, and each method of each class, separately. BlueJ helps!
  - *regression testing*
    - build a test suite *(more classes)* for classes.
    - as a class is changed *(refactored, respecified, extended)*, re-run the test suite to check nothing has broken. Of course, test suites themselves need to kept up to date.
    - BlueJ helps automate the building of testing classes. See Chapter 6.4.
  - *manual walkthrough*
    - manual walkthrough *(read through the code away from any computer, explain your code to others, understand others' code)*. See Chapter 6.8.
  - *debuggers*
    - set breakpoints, add print statements, step execution of code *(from breakpoint to breakpoint)*. See Chapter 3.12-13, 6.11.

– Design Issues *(Continued)*

- *STREAM*
  - write *stub* methods, with parameters and documentation, and check they compile.
  - think about *testing* as you design (may need more methods).
  - consider alternative *representations* to hold class information.
  - *evaluate* consequences for implementation from each representation.
  - declare the *attributes (fields)* for chosen representation.
  - complete the stub bodies for all the *methods:*
    - » *Mañana* principle: *defer* special cases, hard logic, heavy functionality, nested loops, duplicated code to `private` methods.
    - » write stubs for these deferred methods, invoke where needed, compile and test (as far as the stub code allows).
    - » declare extra fields (as necessary) and complete the stub bodies for the deferred methods.

REFACTORING: be prepared to change your mind about design decisions if working with them (as user or implementer) proves difficult. Expect to make wrong decisions. As projects develop, they will show up. For commercial systems, development (and refactoring) never ends …

– Design Issues *(Continued)*

REFACTORING: when changing requirements (e.g. from a single to a multi player game) force a refactoring of your class structures, do that refactoring – but initially make no changes to functionality. Does the system still perform the same way ***after*** this refactoring as ***before***? Only when the answer is ***yes***, introduce the new functionality for which the refactoring was designed.

REFACTORING: be prepared to change your mind about design decisions if working with them (as user or implementer) proves difficult. Expect to make wrong decisions. As projects develop, they will show up. For commercial systems, development (and refactoring) never ends …