Programming and Software Development
COMP90041

Lecture 9

Code Quality

Dr. Thomas Christy
thomas.christy@unimelb.edu.au

The `super` Constructor

- Constructors are not inherited, cannot be overriden
- Base class constructor must be run to set up its instance variables (especially if private)
- Constructor chaining: derived class constructor must invoke base class constructor first
- Form: `super(constructor arguments...)`, *e.g.*:

```
public LostPerson(int age, String name,
        String lastLoc, Time lastTime) {
    super(age, name);
    this.lastSeenLocation = lastLoc;
    this.lastSeenTime = lastTime;
}
```

The `this` Constructor

- Sometimes you want to chain to a different overloaded constructor of the same class
- Form: `this(constructor arguments...)`
- Must be first in constructor, in place of `super`
- The constructor chained to will itself chain to super (or another overloaded constructor that will...)

```
public Person(Person orig) {
    this(orig.age, orig.name);
}
```

- Can also chain to constructor that does part of the work, and then do whatever extra is needed

Default Constructor Chaining

- If constructor doesn't begin with `super(...)` or `this(...)`, Java automatically inserts `super();`
- If your class doesn't have a no-argument constructor, this will be an error
- If you write a class without writing any constructor, Java will automatically write a no-argument constructor with body `{super();}`
- But if you write any constructor at all, Java does not give you the free no-argument one

The `Object` Class

- A class that does not declare what class it `extends` automatically extends the `Object` class
- So class hierarchy is a tree (but we don't usually show `Object` class in class diagrams)
- The `Object` class defines a `toString()` method
- . . . and an `equals(Object other)` method
- These are inherited by all classes, but the definitions are not useful
- They should be overriden if they will be used
- `Object` class has no instance variables
- `Object` has a no-argument constructor that does nothing, so default constructor chaining is fine

Quick Poll

bit.ly/tchristy

qp.unimelb.edu.au/tchristy

Overriding

If a base class `S` contains only one method, with this header:

```
public void foo(int i)
```

Which of these definitions would not be allowed in derived classes of `S`:

A. `public void foo(int i){...}`
B. `private void foo(int i){...}`
C. `private void foo(char c){...}`
D. All of A, B, and C would be allowed
E. None of A, B, and C would be allowed

Overriding

If a base class `S` contains only one method, with this header:

```
public void foo(int i)
```

Which of these definitions would not be allowed in derived classes of `S`:

A. `public void foo(int i){...}`
B. `private void foo(int i){...}`
C. `private void foo(char c){...}`
D. All of A, B, and C would be allowed
E. None of A, B, and C would be allowed

The `instanceof` Operator

- Usually you use overriding to arrange each method to behave correctly for every descendant of a class
- So most code does not need to worry about which descendant type of a base class an object is
- Occasionally you want a test to see if an object is a descendant of a class
- Form: *`object instanceof ClassName`*
- *E.g.*:

```
if (p instanceof LostPerson) {
    System.out.println(p + " is missing");
}
```

The `getClass` method

- `Object` class also defines a getClass() method
- `ob.getClass()` returns an object that represents the actual class of `ob`
- You can use `==` and `!=` to compare two of these to see if classes are the same
- *E.g.*, `o1.getClass() == o2.getClass()` is true if `o1` and `o2` are actually instances of the same class, not just descendants of some class
- You cannot override `getClass` for your classes, but you don't need to

Defining the `equals` method

- Must override not just overload the `equals` method
- Must have the signature `equals(Object other)` (class of `other` must be `Object`)
- Must check that `other` is not `null`
- Use `getClass` to check objects are the same class
- Must downcast `other` to correct class so you can access members to compare
- For derived class, use `super.equals(other)` to check base class instance variables

Defining the `equals` method

```
public boolean equals(Object other) {
    if (other == null ||
        this.getClass() != other.getClass() ||
        !super.equals(other)) {
        return false;
    }
    LostPerson lp = (LostPerson) other;
    return (this.lastSeenLocation.equals(
                    lp.lastSeenLocation) &&
            this.lastSeenTime.equals(
                    lp.lastSeenTime));
}
```

Summary

- Methods form a contract between client and implementor
- Use `extends` to define class that inherits members from base class
- Instances of derived class are also instances of the base class
  - Design/implement classes so this makes sense
- Derived class can override base class methods
- Late binding: definition for actual class is used
- Members with default visibility accessible from package; `protected` also accessible from subclasses
- Use `super` constructor to chain to base class constructor; `this` to chain to same class

Correctness is not enough

- Programs need to be maintained
- Requirements change:
  - They need to work with new systems/formats
  - They need to do new things
  - They need to work faster or handle more data
- Most effort is devoted to a program after it is first completed
- A program is read more than it is written
- Prioritise maintainability and readability
- Following are rules of thumb; there are exceptions
  - But you should have a good reason for violating them

What does this code do?

```
public void processItem(Item item, int num) {
    int best = -1;
    // handle everything in the item
    for (int i = 0; i<=num; ++i) {
        if (handleOne(item, i)) best = selectO
ne(best, i);
    }
    // finish it off!
    completeProcessing(item, best, itemCost(it
em, best));
}
```

What does this code do?

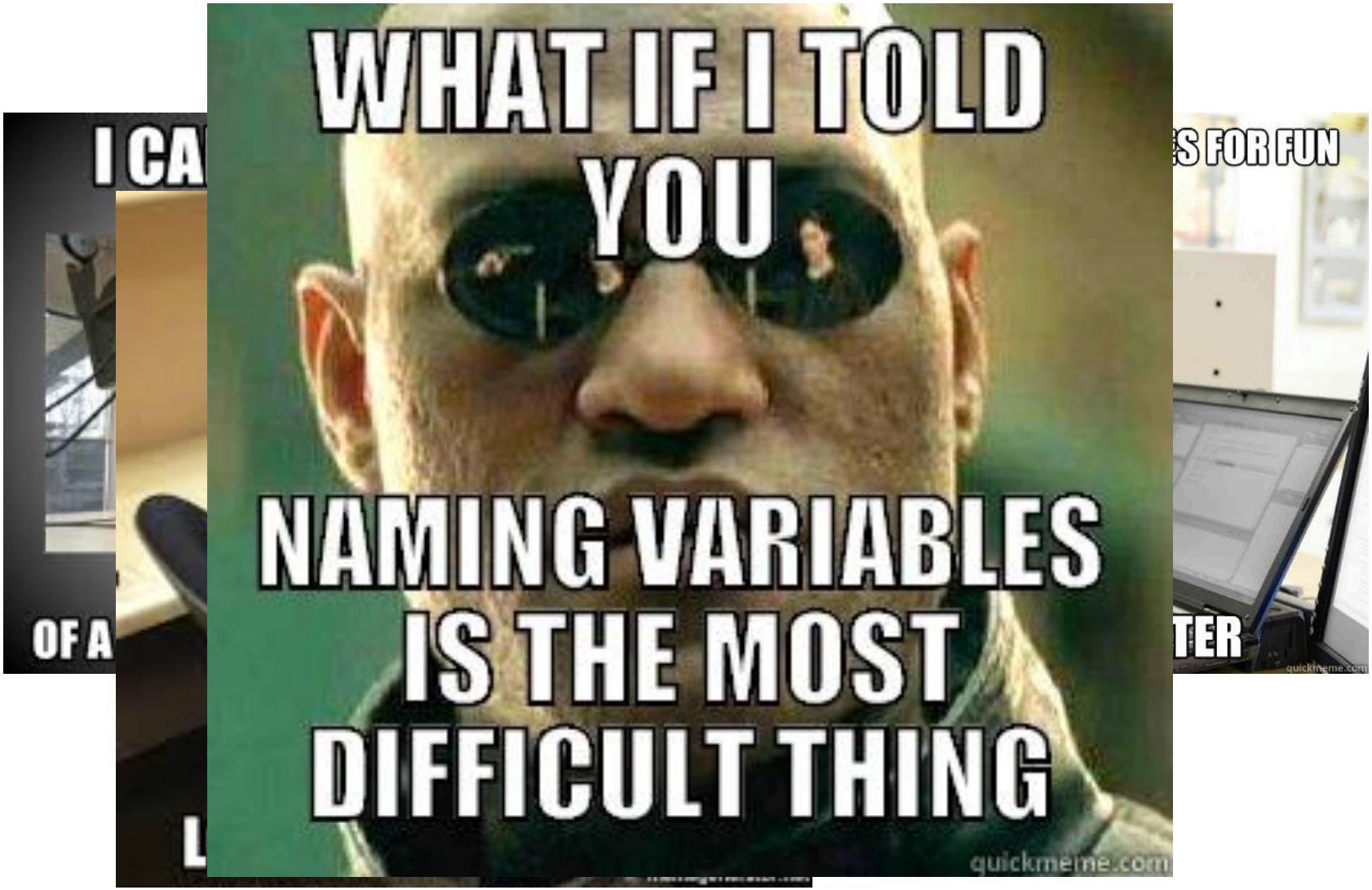
```
public void processItem(Item item, int num)
{
    int best = -1;
    // handle everything in the item
    for (int i = 0; i <= num; i++)
    {
        if(handleOne(item, i)) best = selectOne(best, i);
    }
    // finish it off!
    completeProcessing(item, best, itemCost(item, best));

}
```

Writing understandable programs

- A well-written program is easy to read
- Structure your program like an essay or a newspaper article:
  - "Big picture" stuff comes first
  - Then the few most important pieces
  - Details come later
  - . . . as far as the programming language will allow.
- The best way to say what a class, method, or variable is or does is with its name
  - The name appears wherever it is used
  - Take the trouble to choose good names
- Often the name can't say enough; then augment a good name with good documentation

Variable Naming



Documentation

- Main program (class with `main` method) should begin with documentation:
  - What program as a whole does
  - Broadly how it works
- Every other file should begin with documentation:
  - Broadly what this file is for
  - Who wrote it (who to blame/praise)
- Every non-trivial method should begin with doc:
  - What the method is for, beyond what the name says
  - Code comments are less important
- In code, only need to explain subtleties
  - Don't document what is obvious from the code

JavaDoc

- `javadoc` (JavaDoc)
  - Implemented before any method or Class
  - Use /** **/
  - Generated by JVM (JavaDoc)
  - Integrated into Netbeans
    - (and other IDEs)
  - Creates API style HTML files

Tidiness

- Make your code look neat
  - Consistent layout
  - Avoid long lines (80 columns is standard)
  - Beware tabs (8 column tab stops is standard; better to avoid tabs altogether)
  - Lay out comments and code neatly
  - Use blank lines to separate parts of code
- Divide long files into sections devoted to different aspects
  - Use a comment to explain each section

Good Code

- Avoid "copy and paste" coding
  - Copied code usually needs to be edited every time it's pasted; easy to miss some
  - If you find an error in pasted code, you'll have to fix it everywhere pasted, and you will probably miss some
  - When reading duplicated code, it's hard to see the differences
  - . . . and there's just that much more code to read
- Use abstraction
  - Sometimes you can just reorganise code (loops, if-then-elses) to avoid duplication
  - Otherwise define a method rather than duplicating code

How could you re-write this to simplify?

```
if (a) {
    if (b) {
        X();
    }
    Y();
} else {
    if (b) {
        X();
    }
}
```

How could you re-write this to simplify?

```
if (a) {
    if (b) {
        X();
    }
    Y();
} else {
    if (b) {
        X();
    }
}
```

Consider cases:

a	**b**	action
true	true	`X(); Y();`
true	false	`Y();`
false	true	`X();`
false	false	none

How could you re-write this to simplify?

```
if (a) {
    if (b) {
        X();
    }
    Y();
} else {
    if (b) {
        X();
    }
}
```

Consider cases:

a	**b**	action
true	true	`X(); Y();`
true	false	`Y();`
false	true	`X();`
false	false	none

Simpler equivalent code:

```
if (b) X();
if (a) Y();
```

Principles

- KISS (**K**eep **I**t **S**imple and **S**tupid)
  - First try the simplest solution that could possibly work
  - Only make it more complicated if that doesn't work (*e.g.*, too slow)
- DRY (**D**on't **R**epeat **Y**ourself)
  - Define each piece of logic in only one place
  - Define constants rather than repeating magic numbers throughout your code
  - Less code means fewer chances to make mistakes
  - And less to fix when there's a mistake
- Keep definitions short
  - Methods shouldn't be more than a page; shorter is better
  - Split them into multiple methods if necessary
- Follow standards
  - If there's a preferred way to do something, do it that way

How to simplify your code

- Simpler, shorter code is usually better
- Avoid duplication and complication in your code
- Think of working version of code as a first draft — edit as necessary to simplify, clarify, and organise
- This is called refactoring
- An important part of software development
- The following refactoring example uses odd layout to fit on a slide and make changes clear

What is wrong with this code?

```
public void capture(int rows, int columns) {

    if (Math.abs(rows)==2 && Math.abs(columns)==2) {
        if (king && col+columns<=8 && col+columns>=1
                 && row+rows<=8 && row+rows>=1) {
            row=row+rows;
            col=col+columns;
        } else if (red && rows==2 &&    row<=6
                   && col+columns<=8 && col+columns>=1) {
            row=row+rows;
            col=col+columns;
        } else if (red==false && rows==-2 &&    row>=2
                   && col+columns<=8 && col+columns>=1) {
            row=row+rows;
            col=col+columns;
        }
    }
}
```

Repetition: Factor out common code

```
public void capture(int rows, int columns) {

    if (Math.abs(rows)==2 && Math.abs(columns)==2) {
        if (king && col+columns<=8 && col+columns>=1
                 && row+rows<=8 && row+rows>=1) {
            row=row+rows;
            col=col+columns;
        } else if (red && rows==2 &&    row<=6
                  && col+columns<=8 && col+columns>=1) {
            row=row+rows;
            col=col+columns;
        } else if (red==false && rows==-2 &&    row>=2
                  && col+columns<=8 && col+columns>=1) {
            row=row+rows;
            col=col+columns;
        }
    }
}
```

Repetition: Factor out common code

```
public void capture(int rows, int columns) {
    int newRow = row+rows;
    if (Math.abs(rows)==2 && Math.abs(columns)==2) {
        if (king && col+columns<=8 && col+columns>=1
                 && newRow <=8 && newRow >=1) {
            row=newRow ;
            col=col+columns;
        } else if (red && rows==2 &&    row<=6
                  && col+columns<=8 && col+columns>=1) {
            row=newRow ;
            col=col+columns;
        } else if (red==false && rows==-2 &&    row>=2
                  && col+columns<=8 && col+columns>=1) {
            row=newRow ;
            col=col+columns;
        }
    }
}
```

Factor out MORE common code

```
public void capture(int rows, int columns) {
    int newRow = row+rows;
    if (Math.abs(rows)==2 && Math.abs(columns)==2) {
        if (king && col+columns<=8 && col+columns>=1
                 && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=col+columns;
        } else if (red && rows==2 &&    row<=6
                  && col+columns<=8 && col+columns>=1) {
            row=newRow  ;
            col=col+columns;
        } else if (red==false && rows==-2 &&    row>=2
                  && col+columns<=8 && col+columns>=1) {
            row=newRow  ;
            col=col+columns;
        }
    }
}
```

Factor out MORE common code

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2) {
        if (king && newCol    <=8 && newCol    >=1
                 && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 &&    row<=6
                  && newCol    <=8 && newCol    >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 &&    row>=2
                  && newCol    <=8 && newCol    >=1) {
            row=newRow  ;
            col=newCol    ;
        }
    }
}
```

Every condition has the same Test: Factor it

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2) {
        if (king && newCol     <=8 && newCol     >=1
                 && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 &&    row<=6
                   && newCol     <=8 && newCol     >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 &&    row>=2
                   && newCol     <=8 && newCol     >=1) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

Every condition has the same Test: Factor it

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1) {
        if (king && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 &&    row<=6
                                                ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 &&    row>=2
                                                ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

Rows==2 && row<=6 → newRow<=8

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1) {
        if (king && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 &&    row<=6
                                                ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 &&    row>=2
                                                ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

Rows==2 && row<=6 → newRow<=8

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1) {
        if (king && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 && newRow<=8
                                                ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 &&    row>=2
                                                ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

row>=2 is a bug, should be row>=3

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1) {
        if (king && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 && newRow<=8
                                                ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 &&    row>=2
                                                ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

row>=2 is a bug, should be row>=3

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1) {
        if (king && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 && newRow<=8
                                                ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 &&    row>=3
                                                ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

rows==-2 && row>=3 → newRow>=1

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1) {
        if (king && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 && newRow<=8
                                                ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 &&    row>=3
                                                ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

rows==-2 && row>=3 → newRow>=1

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1) {
        if (king && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 && newRow<=8
                                                ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 && newRow>=1
                                                ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

Combine newRow conditions

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1) {
        if (king && newRow  <=8 && newRow  >=1) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2 && newRow<=8
                                                ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2 && newRow>=1
                                                ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

Combine newRow condition.

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1
                && newRow  <=8 && newRow  >=1) {
        if (king) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2                    ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2
                                                     ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

Same body for all conditions: combine

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                  && newCol     <=8 && newCol     >=1
                  && newRow  <=8 && newRow  >=1) {
        if (king) {
            row=newRow  ;
            col=newCol    ;
        } else if (red && rows==2                    ) {
            row=newRow  ;
            col=newCol    ;
        } else if (red==false && rows==-2
                                                     ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

Same body for all conditions: combine

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1
                && newRow  <=8 && newRow  >=1) {
        if (king


                || red && rows==2


                || red==false && rows==-2
                                                    ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

if inside if is the same as &&

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                 && newCol     <=8 && newCol     >=1
                 && newRow  <=8 && newRow  >=1) {
        if (king


                 || red && rows==2


                 || red==false && rows==-2
                                                    ) {
            row=newRow  ;
            col=newCol     ;
        }
    }
}
```

if inside if is the same as &&

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                 && newCol     <=8 && newCol     >=1
                 && newRow  <=8 && newRow  >=1
        && (king


                || red && rows==2


                || red==false && rows==-2
                                                    )){
            row=newRow  ;
            col=newCol     ;

    }
}
```

==false is the same as negation

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                && newCol     <=8 && newCol     >=1
                && newRow  <=8 && newRow  >=1
        && (king


              || red && rows==2


              || !red        && rows==-2
                                                  )){
           row=newRow  ;
           col=newCol     ;

    }
}
```

Values for rows depend only on red

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                 && newCol     <=8 && newCol     >=1
                 && newRow  <=8 && newRow  >=1
        && (king

                || red && rows==2

                || !red       && rows==-2
                                                 )){
            row=newRow  ;
            col=newCol     ;

    }
}
```

Values for rows depend only on red

```
public void capture(int rows, int columns) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                 && newCol     <=8 && newCol     >=1
                 && newRow  <=8 && newRow  >=1
        && (king


                ||        rows==(red ? 2 : -2)



                                                    )){
            row=newRow  ;
            col=newCol     ;

    }
}
```

Tidy up the definition

```
public void capture(int rows, int columns) {
  int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2 && Math.abs(columns)==2
                  && newCol     <=8 && newCol     >=1
                  && newRow  <=8 && newRow  >=1
        && (king


                ||        rows==(red ? 2 : -2)



                                                    )){
            row=newRow  ;
            col=newCol     ;

    }
}
```

Tidy up the definition

```
public void capture(int rows, int columns          ) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2    && Math.abs(columns)==2
        && newCol <=8 && newCol >=1 && newRow <=8 && newRow >=1
        && (king || rows==(red ? 2 : -2))){
        row=newRow; col=newCol;
    }
}
```

Concept appearing throughout the class

```
public void capture(int rows, int columns          ) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2    && Math.abs(columns)==2
        && newCol <=8 && newCol >=1 && newRow <=8 && newRow >=1
        && (king || rows==(red ? 2 : -2))){
        row=newRow; col=newCol;
    }
}
```

Concept appearing throughout the class

```
public void capture(int rows, int columns          ) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2    && Math.abs(columns)==2
        && onBoard(newRow, newCol)
        && (king || rows==(red ? 2 : -2))){
        row=newRow; col=newCol;
    }
}

public boolean onBoard(int row, int column) {
    return row >= 1 && row <= 8 && column >= 1 && column <= 8;
}
```

Checking magnitude of rows twice

```
public void capture(int rows, int columns          ) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2    && Math.abs(columns)==2
        && onBoard(newRow, newCol)
        && (king || rows==(red ? 2 : -2))){
        row=newRow; col=newCol;
    }
}

public boolean onBoard(int row, int column) {
    return row >= 1 && row <= 8 && column >= 1 && column <= 8;
}
```

Checking magnitude of rows twice

```
public void capture(int rows, int columns          ) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2    && Math.abs(columns)==2
        && onBoard(newRow, newCol)
        && (king || red==(rows>0)      )){
        row=newRow; col=newCol;
    }
}

public boolean onBoard(int row, int column) {
    return row >= 1 && row <= 8 && column >= 1 && column <= 8;
}
```

Code to move almost the same as capture

```
public void capture(int rows, int columns          ) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==2    && Math.abs(columns)==2
        && onBoard(newRow, newCol)
        && (king || red==(rows>0)       )){
        row=newRow; col=newCol;
    }
}

public boolean onBoard(int row, int column) {
    return row >= 1 && row <= 8 && column >= 1 && column <= 8;
}
```

Code to move almost the same as capture

```
public void capture(int rows, int columns) {
    shift(rows, columns, 2);
}
public void move(int rows, int columns) {
    shift(rows, columns, 1);
}
public void   shift(int rows, int columns, int dist) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==dist && Math.abs(columns)==dist
        && onBoard(newRow, newCol)
        && (king || red==(rows>0)       )){
        row=newRow; col=newCol;
    }
}

public boolean onBoard(int row, int column) {
    return row >= 1 && row <= 8 && column >= 1 && column <= 8;
}
```

Final tidied results

```
public void capture(int rows, int columns) {
    shift(rows, columns, 2);
}
public void move(int rows, int columns) {
    shift(rows, columns, 1);
}
public void shift(int rows, int columns, int dist) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==dist && Math.abs(columns)==dist
        && onBoard(newRow, newCol)
        && (king || red==(rows>0))){
        row=newRow; col=newCol;
    }
}

public boolean onBoard(int row, int column) {
    return row >= 1 && row <= 8 && column >= 1 && column <= 8;
}
```

Side by side: readibility

```
public void capture(int rows, int columns) {

    if (Math.abs(rows)==2 && Math.abs(columns)==2) {
        if (king && col+columns<=8 && col+columns>=1
                 && row+rows<=8 && row+rows>=1) {
            row=row+rows;
            col=col+columns;
        } else if (red && rows==2 &&    row<=6
                   && col+columns<=8 && col+columns>=1) {
            row=row+rows;
            col=col+columns;
        } else if (red==false && rows==-2 &&    row>=2
                   && col+columns<=8 && col+columns>=1) {
            row=row+rows;
            col=col+columns;
        }
    }
}
```

```
public void capture(int rows, int columns) {
    shift(rows, columns, 2);
}
public void move(int rows, int columns) {
    shift(rows, columns, 1);
}
public void shift(int rows, int columns, int dist) {
    int newRow = row+rows, newCol = col+columns;
    if (Math.abs(rows)==dist && Math.abs(columns)==dist
        && onBoard(newRow, newCol)
        && (king || red==(rows>0))){
        row=newRow; col=newCol;
    }
}

public boolean onBoard(int row, int column) {
    return row >= 1 && row <= 8 && column >= 1 && column <= 8;
}
```

Why is it better than original code?

- Reorganising found and fixed a bug
- This method is shorter and easier to read
- Logic is more self-evident
- Handles twice as much of the problem in about the same number of lines
- Makes clear how `move` and `capture` are related (they're almost the same)
- Range tests on newRow and newCol are replaced by a method call whose name says what it means
- Can use the `onBoard` method throughout the class

Summary

- Document what program does, what each class is for, and what each method does
- Coding is like essay writing: express yourself clearly, briefly, and simply
- Refactor your code as you would edit an essay
- Organise your documentation and code to help reader understand the problem and your solution
- Chose descriptive names; use comments for subtleties
- Verbose, repetitive code obscures your intentions; keep definitions simple, clear, and short

THE UNIVERSITY OF
MELBOURNE