CSSE2002

Alistair Michael

# Contents

---

pdf version
The markdown source for this page is available here.
Contributions are welcome.

---

# Language

## Variables

- Primitives vs Classes
- References
- Stack vs Heap

All local variables (references and primitives) are stored on the stack.

Heap memory exists independently of the callstack, and all objects are stored on the heap. Hava has a garbage collector which deletes objects when they are no longer being held.

### References

This means *all objects in java are held by reference.*

So for objects equality has this meaning:

`a = b` : `a` now refers to the same thing as `b`

`a == b` : Does `a` refer to the same piece of memory as `b`?

While for primitive types `a = b` assigns a the value of b, and `a == b` compares the value of `a` and `b`.

If you want to compare the value of objects you use `a.equals(b)`.

So objects are always passed by reference and primitives are always passed by value.

Note: To compare floating point values always use `Math.abs(a - b) < threshold`.

### Mutability

Strings are immutable, for efficiency, the code

```
String a = "AAA";
String b = "AAA";
```

Will produce two variables which both reference the same string.
In general, string-modification methods return a new reference to a new string.
Meanwhile, arrays and Lists are mutable, their contents can be changed.

## Classes

Inheritance represents a specialisation relationship between parent and child, while interfaces separate implementations of a similar abstraction.

Interfaces define how a to interact with a class or object.

- when and why would you define an interface instead of a superclass and vice versa
- What is the purpose of `@override`-ing a method.
- What is the benefit of polymorphism and dynamic binding for design.

### Overloading

Methods are resolved at compile-time so for the following code:

```
public class A {
    public int foo(Object o) {
        return 1;
    }
}

public class B extends A {
    public int foo(Object o) {
        return 2;
    }

    public int foo(int i) {
        return 3;
    }
}
```

Running the following code will return `2`, not `3`.

```
Object o = new A();
A a1 = new A();
A a2 = new B();  // B is implicitly upcast to A
a2.foo(20);       // resolves to B.foo(Object)
```

Because `a2` is declared as type `A`, the compiler only knows about the method that takes an object.

Java will only let you access part of an object if it is sure it exists, at **compile time**. However an objects type is computed at compile time will determine what you can do with it.

When executing a method, java finds the closest up the class heirachy, according to the type it thinks that object is, (according to cast or declaration) that matches the method signature (name + argument types and order). Then when calling the method,

it calls whatever function reference is contained in the object actually stored by that variable.

### Interfaces

Defines which methods a class must implement.

### Casting

Casting changes how the compiler interprets a specific object, ie what type it thinks it is.

Upcasts can be implicit, but downcasts cannot. This means you can pass a `String` to a method that expects `Object` without explicitly casting, for example.

You can implicitly cast from `float` to `double`, since a float has less precision and they have a common super-class.

## Exceptions

Exceptions are used to recover from unexpected problems or bad state.

*Checked Exceptions* (`IOException`) are checked for at compile time so are mandatory to handle if it is possible for a method to cause them. They can be handled either by adding them to the method signature to be handled elsewhere, or catching them in a try-catch block.

| Exception Kind | Parent | Cause |
|---|---|---|
| Error | Throwable | An error in the program execution, unrecoverable |
| IOException | Exception | Input-output Error |
| FileNotFoundException | IOException | Cannot find a file |
| RuntimeException | Exception | An exception at runtime |
| ArithmeticException | RuntimeException | Divide by 0, and FP errors |
| NullPointerException | RuntimeException | Doing something with an unintialised reference |
| IllegalArgumentException | RuntimeException | A method is invoked with wrong arguments |
| IllegalStateException | RuntimeException | A method invoked at the wrong time |
| IndexOutOfBoundsException | RuntimeException | Accesing an array element with an invalid index. |
| ClassCastException | RuntimeException | Performing an impossible cast |

See also: Try With resources

# Java Collections

- can not store primitive types, there are wrapper classes to use
  - `Integer`, `Boolean`, `Byte`, `Double`, `Character`, `Short`, `Long`, `Double`
  - automatically converted on construction

### Stack

- Last in first out
- empty()
  - empty stack
- peek()
  - return reference to top of stack
- pop()
  - return top of stack and remove from stack
- push(obj)

### Generic Types

- collections contain generic types. The specific type of a generic is defined when the object is declared.

### Lists

- grow and shrink automatically
- walk along
- insert anywhere
- remove an item
- check if an item is in the list

### Different Implementatiosn

- LinkedList
  - adding and removing items in the middle
- ArrayList
  - good for random access
- Vector
  - for concurrency

### Methods

- size(); the number of elements in the list

// literally just look up the documentation

### Iterators

- more flexible way of moving through the list than using `for each` loops
- Call .iterator() method on another collection to get the iterator.
- starts before the 1st element in the list
- `iterator.next()` returns the next element in the list
- `it.hasNext()` return true if there is another element in the list.
- they can manipulate the contents of the collection
- if you modify one iterator, other iterators on the same object will fail fast

#### Kinds

- ListIterator; knows more about how to iterate lists `lists list.listIterator()`
  - .add() method (at iterators current position)

### Lists

- ArrayList LinkedList et al
- Can store duplicates
- Automatically grows and shrinks depending on what you do to it
- Ordered
- Iterable

### Sets

- An orderless collection of objects
- Every value must be unique

#### TreeSet

E needs to implement Comparable

#### HashSet

E neeeds to implement hashCode() and equals(), and have the label.

### Map

- stores unordered key-value pairs
- `Map<Integer, String>`; set type of key and value
- not good for iterating
- should not use mutable objects as keys although it works unless you change the object after setting it as the key
- All *keys* must be unique, but not the values.

## Automated Testing

### Terms

- unit testing

- regression testing
- black box
- white box
- test driven development

### Procedures

- unit testing
  - each unit (class) works)
- integration testing
  - components work together
- system testing
  - does the whole program work
- acceptance testing
  - do the users agree that the system does what it is supposed to
- regression testing
  - does new stuff work
  - has it broken old things
- blackbox testing
  - test the interface does what it is supposed to without knowledge of the implementation
  - does not test whether the implementation has like bad programming
- glass box testing
  - knowledge of internal
  - code coverage
  - tests whether the complex parts of the internal is complex

### Test Driven Design

- write the tests before the code
- requirements drive the code more directly
- If you find a bug that is not caught by the tests, you can write a unit test, and add it to the regression test suite

### Junit4 Framework

- Write a test class for each object
  - all test methods have the `@Test` annotation.
  - method names tell you what they do.
  - use a piece of code, and use assert methods, for example `Assert.assertEquals(val, val)`

  - Each test method should only test one thing (conceptually / logically)
  - `@Before` and `@After` labels put on each test, are used to setup and tear-down the environment before and after each test. To ensure they don't interfere with eachother.

### Assert

- assertEquals
- assertArrayEquals
- assertFalse / assertTrue
- assertSame / assertNotSame
- fail
- can import all these as static methods so they can be called without the junit.etst.sjhda.Assert garbage.

### Setup

- need junit and hamcrest libraries

### What to test

- Input possibilities and features
- Should identify potential problem areas
- Should not be too big

### Things to test

- boundary cases
  - 0, negative numbers
  - NULL, empty containers, sets lists
- Floating point extremities
- large datasets
- resource access denied, failed
- non-existent resources

### Code Coverage

- How much of the code is tested
- statement coverage
  - the code is tested by being executed once
- branch coverege
  - all possible branching paths that the logic creates are executed
- path coverege
  - all paths thru the loop are tested
  - for for loops

        * init fails
        * init test fails
        * init test body fails
        * init test body iterate fails
    - eg recursion is also painful

## Procedural Abstraction

- interested in what the methods **do**
    - javadoc clearly explain functionality
- methods need to have one clear function

Be suspicious of

- control flow on parameter types
- long and complex methods
- repeated code
- methods that have more than one function
- can change the implementation without changing the specification

### Specifications

- javadoc
- allow the implementation to be changed without changing methods that use it
- draw attention to possible consequences of implementation details
- be sufficiently restrictive if implementation is limited.
- the information needed to use the method
- be precise to avoid incorrect implementations
- have **generality** to allow acceptable alternative versions
- have **clarity**: utilise formal languages,

## Contract-driven design

Meaning if the caller meets the precondition, the method guarantees that postcondition will be true.

```
/**
@require precondition
@ensure postcondition
*/
```

Can use java boolean expressions to formally and precisely specify conditions as well as using the following mathematical signifiers.

- `\result` return value
- `a ==> b` implication
- `a <==> b` if and only if
- `\old(x)` the value of `x` before the method

- `\forall C c` for all objects c in class `C`
- `\exists C c` there exists an object of class `C`

Can use `assert` to ensure this at compile time, and can also use it as a test system at runtime with special conditions. These can be checked at runtime using `java -ea ClassName`.

### Defensive Programming

Assume that input is bad.

- Explicitly check for invalid inputs
  - Ensure that no dangerous behaviour results
- Always apply it to data coming from outside of the program, and treat things
- coming from inside the program as valid.
- Always reject `null`, because it can easily appear inside the program.

### Substitution Principle

> An object of a subclass type can be used at any point where a super-class's method is expected

- WRT to Contracts, the child is bound by the contracts with the parent.
  - Parents contract must be sufficient for child's contract.
  - The child contract can only be a *weakening* of the parent's precondition, that is, the pre-condition may be weaker and the post-condition stronger, but the precondition cannot be stronger, nor the post-condition weaker.
    - the child must not reject any states the parent would allow (precondition)
    - the child must not have a result that the parent does not ensure

Contracts are 'inherited': if you are overriding a sub-class's method, it must still follow the contracts of the parent, and the original version it is overriding.

## Miscellaneous Java

### Instanceof

expression `instanceof` type

Returns true if the value of *expression* is an instance of type *type*.

### Questionable Uses

- Use it in conditionals to determine what methods should be used on that thing.
  - Otherwise: use encapsulation
  - Put the code in the classes themselves so you do not need to use a conditional to determine which class is being used
  - use a generic interface
  - use helper methods in the classes

## Newlines

- `println` will always use the correct newline character for the operating system (at runtine)
  - Unix uses "\n"
  - Windows uses "\r\n"
- `System.lineSeparator()` and `String.format("%n")` will return the correct line separator.

## Pre / post increment

- The same as C

Starting with x = 0:

```
expression      x =        returns
++x              1          1
x++              2          1
x--              1          2
--x              0          0
```

## Ternary

```
int a = (conditional) ? do if true : do if false;
```

## Final keyword

`final` has two meanings depending on its context.

### Variables

```
public void stuff() {
    final int x = 5;
    x = 4; // compile error
}
```

In the case of member variables, they must be set *once* in the constructor, and nowhere else.

```
public class Chem {
    public static final double AVAGADRO = 3.023e23;
}
```

### References

It means that the reference value cannot be changed, *not* that the object it refers to cannot be changed.

```
final Test x = new Test(10);
x.q = 15;
```

Will compile, you can change the state of the object without changing it's identity.
// What about the objects hash value, for maps etc. That would probably be weird.

### Methods

A method that is labelled final cannot be overwritten in a subclass.

### Classes

- Cannot create new classes that inherit from final classes
- Member variables cannot be changed once initialised

### Abstract

An abstract class is a *class* that it is not meaningful to create an object of, but is useful in the structure of the class heirachy. An abstract class does not have to have abstract methods.

An abstract method has no implementation, it exists to be defined later.
`public abstract void doStuff();`

If a class contains abstract methods, the class must also be declared abstract. And abstract classes cannot be instantiated, although they can be extended.

This is used for defining interfaces.

### Short Circuit evaluation

The entirety of a conditional statement is not computed if its value can be determined earlier.

`True || f(x) || g(x)` will not execute f or g.
`g(x) && False && g(x)`: `g(x)` will never get executed.

## StringBuilder / StringBuffer

`StringBuffer` is older and slower, but threadsafe.

Strings are immutable so `+=` on strings requires re-allocating memory, doing it in a loop is very inefficient.

`StringBuilder` is **not** threadsafe.

```
StringBuilder sb = new StringBuilder("starting text");
for (int i = 0; i ++; i < 10) {
    sb.append(i);
}
System.out.println(sb.toString()); // need to use toString()
                                   //to get the string value
```

## Copying (`clone`)

Objects are always manipulated by reference, by default.

```
Objext x = y // makes x refer to the same thing as y
```

Object class has a protected `.clone()` method, since it is protected some work is required to be able to use it. This is to ensure it is implemented properly for the classes you are truing to use it for, since reference members may also need to be copied.

```
public class MessageHolder implements Cloneable { // need to implement
                                                   // Cloneable interface
    private MessageBuilder msg;
    //...
    @Override
    public Object clone() {
        try { // have to catch CloneNotSupportedException
            MessageHolder nm = super.clone();
            // we can call clone() here because this class has the
            // Cloneable marker interface set, indicating it is
            // allowed
            nm.msg = new StringBuilder(msg.toString);
            return nm;
        } catch (CloneNotSupportedException e) {
        }
        return null;
    }
}
```

- Typically you can just call `Object.clone()`
- Deep-copy is not a concern for immutable objects.
- Deep-copy is general expensive

Note that:

- `x.clone != x`
- `x.clone.getClass() == x.getClass()`
- `x.clone().equals(x)` should usually be true, but there are practical exceptions

## Properties of `.equals()`

- reflexive
- symmetric (in terms of the same class type)
- transitive
- deterministic, should always give the same result, unless the objects' state changes
- `x.equals(null)` is false

Be mindful of the comparability of the classes whose `.equals()` method you are calling.

- If you override `.equals()` you also need to override `.hashCode()`
- `x.equals(y)` $\implies$ `x.hashCode() == y.hashCode()`
- `.equals()` can refer to object **state** or object **identity**
  - you need to make a decision on whether to include the mutable parts of the class in computing the hashcode accordingly.
  - including mutable state in the hashcode means changing the state changes whether `.equals()` is true for a copy with different state.

It is probably better to use `.equals()` for state, since the reference provides the object's identity.

### Hash Codes

Returns a numeric value for some given values.

- has to be deterministic
- may not neccessarily uniquely identify an object
  - collisions should be rare and able to be managed
  - not as good for determining object identity
- `x.equals(y)` $\implies$ `x.hashCode() == y.hashCode()`
- Must be efficient, for fast lookup
  - used for looking up items quickly
  - comparing passwords without comparing the actual values
- if it is being used as a lookup key
  - does changing the state make it a different object?

## Input-Output

### Scanners

There is a collection of scanner object methods that can be used to get specific items from these filetypes.

See Javadoc

```java
import java.util.Scanner;
// ...
Scanner input = new Scanner(System.in);
Scanner file = new Scanner(new File("filename.txt"));

int total = 0;
while (true) {
    total += input.nextInt();
}
```

### Encoding

Java uses unicode, chars are not neccessarily single bytes and there is no single automatic way to translate between bytes and UTF-X chars.

So I'm assuming their environment system is less of a tranwreck than C's.

1. Binary
    - More compact
    - Sensitive to system differences
    - Otherwise it is more direct
2. Text
    - Human readable
    - Requires parseing and markup delimiters etc.
    - Not really efficient for machine-interpretation as a middle abstraction

### Streams

Abstractions for input and output

- Works with multiple origins of input; keyboard, disk, files, network
- Buffering
- Unified interface for different types of input/output data

`java.io.InputStream`

- `FileInputStream`; for files
- `ByteArrayInputStream`; get bytes from an array in memory as an input stream

Methods that use InputStreams should ask for an object of the `InputStream` class (interface style calss), not its children classes.

Low-level `read()`

### Buffered input

`BufferedInputStream` is a class that wraps `InputStream` to provide buffering to improve performance. It can read a single character, a specific number of characters, or a line. For other methods see javadoc

```
try {
    InputStream is = new BufferedInputStream(new FileInputStream("dat"));
    String s = is.readLine();
} catch (FileNotFoundException e) {
    // fail
} finally {
    is.close();
}
```

You have to then parse the returned string with string-parsing functions.

### Readers

`java.io.Reader` is a base class for objects which read `InputStreams`.
`new FileReader("filename")`
≈ `new InputStreamReader(new FileInputStream("filename"))`

### Try with Resources

Readers and streams need to be closed. Can use `try catch` and `finally` to close files at the end, however `close()` can also throw exceptions hence Java provides `try catch` syntax that automatically handles file resources.

```
try (BufferedReader r = new BufferedReader(new FileReader("file"))) {
    r.readLine();
} catch (IOException e) {
    // handle
}
```

### Parsers

Each primitive type (`int, char...`) has a wrapper class and these have string parsing methods.

```
int i = Interger.parseInt("1");
```

`parseInt()` will throw a number format exception if the format is invalid.
Now $i = 1$.

## Output

For the most part just replace `Input` with `Output` and `Reader` with `Writer`.

The standard output is `System.out`, standard error is `System.err`, it is a print stream that provides print methods.

- `print()`, `println()`
- `flush()` flushes output buffer
- `printf(String format, Object args)` Use C-style format strings.
- `write(byte[] buf, int off, int len)` write `len` bytes from a byte array.

### PrintWriter

`PrintWriter` is a better tool for writing characters. It can write to any type of Stream or file. Look at the constructors in the javadoc.

You often have to flush the output because otherwise it only gets sent once the output is full, or if it is closed.

This is important for

- interactive programs where you prompt for input
- debugging where you need outputs to be in order and up to date.

### Serialisation

Converting a java object to bytes. For a class to be serialisable it must:

- Must implement `Serializable` interface
- Any objects referenced must also be `Serializable`
- Object streams can read and write to any type of `Stream`—to files, networks etc.
- Read using `ObjectInputStream`
- Write using `ObjectOutputStream`

```
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("file"));
oos.write(new Integer(5));
oos.close();
ObjectInputStream ois = new ObjectInputStream(new FileInputStream("file"));
Integer five = ois.readObject();
```

### Limitations

- If you change the class after saving a serialisable object and try to read it again it will produce an version error.
- Deserialising untrusted data is very unsafe / insecure

## Parsing Text Files

Often need to

- find delimiters
- split strings on delimiters
- convert strings to primitive types
- construct new objects based on parameters from a file

### Split Strings

```
String[] splitstrings = "a b c".split("delimiter");
String[] splitstrings = "a b c".split("delimiter");

String s = "lalala".substring(0, 2);
// s = la
```

### Regular Expressions

Exist in java (`pattern`)

### Converting Strings

```
Integer.parseInt()
Float.parseFloat()
Double.parseDouble()
Boolean.parseBoolean() // converts to false for anything other than true
```

All number types return `NumberFormatException` if the number format is wrong.

### Objects

Usually putting the parsing code in an initialiser is not appropriate, it doesnt separate IO classes from logic classes, and it requires files to be organised as one class to a file or other complexity.

It is better to have another IO class that parses the file, then calls the objec initialisers.

### File Objects

Creating, renaming, moving etc; filsystem manipulations, not manipulating file objects (that is through the stream abstraction).

`java.nio.File` package contains

- `java.nio.file.Path`
- `java.nio.file.Files`

### Exit

```
System.exit(1); // exit with an error code
```

Default exit status is 0, which means success.

## JavaFX

Good example of Object Oriented, and event-driven programming.

It has newer features than older GUI libraries, it automatically manages threads, etc.

Need to add a java module through `VM options` in IntelliJ (to add jvm arguments).

Create a class - Extends `javafx.application.Application` - Override the `start()` method

### Stage

A **stage** is the main window which displays a single **Scene** which holds all the widgets.

The scene holds a heirachial scene-graph that holds GUI elements.

### Layout Panes

- groups nodes (GUI Elements)
- can use column, row ordering to place nodes within a grid
  - `grid.add(new Label("hello"), col,row)`

### Controls

Buttons, labels, areas, textfields etc

### Panes

Boxes that hold groups of nodes and can be laid out as nodes themselves.
`BorderPane()`: a pane with borders

### Canvas

Draw shapes and things in a space.
Look at examples its not very complex.

### EventHandlers

Interactions with the gui generate events, and program functions also generate events.
We only consider `ActionEvent` in `javafx.event` for handling gui events.
For something to happen as a result of an Event, there needs to be an `EventHandler` associated with that event.
`EventHandler` is an interface which we use to implement our event handlers.
They can be connected to buttons with `setOnAction`

```
// package private class, can be in the same file as ButtonDemo
// note that package private classes still generate .class files so
// you have to be careful of name conflicts in the package scope still
class ButtonDoer implements EventHandler<ActionEvent> {
    public void handle(ActionEvent e) {
        System.out.println("Send to console");
    }
}

public class ButtonDemo extends javafx.application.Application {
    ...
    public void start() {
        ...
        Button button = new Button("useless button");
        button.setOnAction(new ButtonDoer());
        ...
    }
}
```

### Better Design

- link the events and what they do more loosely,
  - to avoid putting application logic in event handlers
  - use inner classes instead of package private classes for event handlers for better encapsulation

```
public class ButtonDemo extends javafx.application.Application {
    private Stage stage;

    public static void main(String[] args) {
        Launch(args);
    }

    public void start(Stage stage) {
        this.stage = stage;
        Button button = new Button("useless button");
        button.setOnAction(new ButtonDoer());

        GridPane grid = new GridPane();
        grid.add(button, 0,0);
        Scene scene = new Scene(grid);

        stage.setScene(scene);
        stage.show();
    }

    public void respondToButton() {
        // do stuff
    }

    // no one else can do actions that aren't tied to events
    // logic is encapsulated with the gui interactions
    private class ButtonDoer implements EventHandler<ActionEvent> {
        public void handle (ActionEvent e) {
            repondToButton();
        }
    }
}
```

- can use a single handler for many buttons, make buttons private memebers and switch over `event.getSource() == button1`.
- nested classes are non-examinable

### Anonymous Classes

Allows you to create an event handler immediately to do the things that you want the button to do.

```
button.setOnAction(new EventHandler<ActionEvent>() {
                        public void handle(ActionEvent e) {
                            respondToButton();
                        }
                    }// end of class
button2.setOnAction(new EventHandler<ActionEvent>() {
                        public void handle(ActionEvent e) {
                            respondToButton2();
                        }
                    }// end of class
);
```

A similar effect can be achieved with lambdas.

```
button1.setOnAction((ActionEvent event) -> respondToButton());
```

### TextFields

For entering text

```
tf = new TextField();
String text = tf.getText();
tf.setText("Hello World")
```

### Dialogs

For confirmation prompts etc, there are pre-designed dialogue types.

```
public void doButton() {
    TextInputDialog inputDialog = new TextInputDialog();
    inputDialog.initStyle(StageStyle.DECORATED);
    inputDialog.setHeaderText("Hello World");

    ImageView iv = new ImageView(new Image("photo.png"));
    iv.setFitHeight(40);
    iv.setPreserveRatio(true);
    inputDialog.setGraphic(iv);

    Optional<String> result = inputDialog.showAndWait();
    if (result.isPresent()) {
        System.out.println(result.get())
    }
}
```

### FileChooserDialog

```
public class FileChooserDemo extends javafx.application.Application {
    private Stage stage
    private FileChooser fileChooser = new FileChooser();
```

```
    private void respondToButton() {
        File file = fileChooser.showOpenDialog(stage);
        if (file != null) {
            openFile(file);
        }
    }

    private void openFile (File file) {
        // do stuff
    }
}
```

## Design Quality

The most fundamental design guidelines are

1. Every class and method has a single clearly defined purpose and reason for existing.
2. Classes encapsulate all their *own* state and actions.

### Cohesion

- Does a class/object make sense as a single entity
- Do all the data and methods fit together for a single purpose or abstract concept
  - minimises extraneous ideas to understand
  - simpler unit to test
  - modification is easier

### Coupling

- How strongly a class depends on another class
  - How much of the internal state is passed to another class through methods?
  - How many methods of other classes are called?
  - Can another object influence the flow of control in this object?
- Low coupling is preferable
  - highly coupled classes are harder to write and test in isolation
  - high coupling can indicate that a class has been split when it shouldn't have been

#### Law of Demeter

The target of a message can only be one of the following objects:

- The methods object (`this`)
- An object passed as a parameter
- An object referred to by an attribute of the object

  - Weak form of Demeter: and anything in that collection o

- An object created by the method
- Object referred to by a global variable

Avoid chained messages `a.getB().getC().doSomething()`, since this increases coupling

### Mindless Classes

- A class should manage its own flow of control
  - restrict other classes from accessing its state
  - data members are private
  - minimise accessor methods
- The logic that is applied to the classes data should be within the class, not in other classes that access the data through getters
- These tend to have low cohesion and high coupling

### God Classes

- Classes that do everything within their context and contain all the data
- High coupling and low cohesion

### Mitigation

A class should only depend on the public interface of another class.

- Attributes should only belong to one class.
  - This is often violated when classes have many accessors
- A class should represent a single abstract concept
  - Unrelated data and functionality should be factored out to other classes
- system logic should arise from the classes working to gether to implement behaviour, it should be shared between classes uniformly

### Fragile Super Class

- Inheritance creates strong coupling between the superclass and the subclass
- Does the design rely on the knowledge of the private methods of the superclas; changing the privates in the superclass should not change behaviour or cause problems for the subclasses.
- Public or protected methods should only change behaviour if the specification of their functionality changes
  - should usually be overridden in the subclasses

#### Downcall

- Calling a method from a childs class

### Further Reading

- Abstraction and Encapsulation
- SOLID OO
- Papers by Bob Martin

## GUI Design

- Model
  - Conceptual things: entities, in the system
  - State
  - invariants
  - methods that enforce the invariants
- View
  - A presentation of the state, and a way to interact with it

### Why MVC?

- Decompose the task
- Separate interface from model
  - Can change the UI independently of the model
  - Might want to support multiple interfaces
  - GUIs, web, mobile screenreaders
- Responsibility for enforcing invariants should be in only one place

### How MVC: Challenges

View and Model need to communicate - find current state - may need to notify the interface for when state changes

- The user has to get information and send commands
- The interface needs to know if the model has changed

We want them to be loosely coupled, so the model shouldn't know about the view.

- one way access from interface to model is satisfactory only for a small model and interface. It is generally not safe to assume that the model is synchronous with the view.
- Dont want to make the user wait for the model to sort itself out
- Multiple interfaces being connected could easily make a deadlock
- Model generally is updated independently to the actions happening in the interface, from things like external input, network input, or just calculating results of requested functions

### Callbacks and Observers

- User Interfaces implement an `interface` that the model knows about, and which it uses to tell the interface that the model has changed, and details about what has changed
  - The interface can then ask the model for further information only when there is new information available, without having to poll constantly to ask
  - So UI updates are driven by events sent by the model

### Input Processing

1. Getting Values from UI components to assemble a method call
   - maybe the processing to generate the call to the model, needs to be in a separate class if it is very complex
2. Making changes to the model based on that call
   - belongs in the model

### Model View Controller

- View:
  - sends messages to the controller based on User interaction
  - recieves callbacks from the model and queries details about state
- Controller:
  - recieves requests from the view and figures out what to do to the model in response
- Model:
  - Stores state, does program logic, and implements functionality
  - Tells the view when it has changed

#### Model View Adapter

Isolate the view from the model using an adapter

- View
  - interacts with user
  - sends events to an adapter
  - recieves updates from adapter
- Adapter
  - reads state from model
  - manipulates model
  - recieves callbacks from model about state
  - sends updates to the view
- Model:

  - Stores state, does program logic, and implements functionality
  - Tells the adapter when it has changed

An example of an Adapter might be a REST API to a server-side functionality

### Model View Presenter

Same as MVA except the view and presenter is more tightly coupled. Every view class has its own presenter class.

Presenter manages the display, not just bridging between view requests and the model, so it can do more complex things with the view to allow it to be more responsive and intelligent, without having a lot of that complexity in the actual UI code.

### Model View ViewModel

ViewModel:

- encapsulates the state (of the model) that is displayed by the view
- tight coupling with the view: two way binding
- user changing state immediately updates viewmodel,
- Is provided by libraries
  - less boilerplate code required for implementing event-sending between the view and controller and model
- the most restricted version here
  - Cannot easily have different views

## Generic Programming

- Using `Object` is bad because it has no type safety.
- Generic types solve this, since you can use parameterised types.
  - still has compile-time type checking
  - don't need to cast in and out

```
public class X<T> Boo {
    private T myFirstVariable;
    // T is the type of myFirstVariable
    // ... constructor goes here ...
    ...
}
```

You can now instantiate the class and give it a specific type.

```
Boo<Integer> = new Boo<>();
```

You can an arbitrary number of type parameters in clases.
There is a naming convention.

- E: Element (in collections)
- K: Key
- N: number
- T: Type
- V: Value
- S,U,V... Additional types

### Generic Methods

Do not have to be in a generic class

```
public static <T> int count(T[] array, T value) {
    for (T item : array) {
        ;

    }
}
```

### Bounded Type Parameters

- Types can be restricted to being a subset of classes.

```
public class<T extends Number> {...}
```

This allows anything that is a sub-class of Number.

### Generic Inheritance

```
class X <T> extends class Y <T> {}
class X <T> extends class Y <String> {}
```

- note that using a subclass as a generic parameter, does not imply that the classes themselves have an inheritance relationship.

### Wildcards

`?` Represents an unknown type, but not a specific unknown type. They are useful when generic types are needed but they do not need to be named and referenced.

`?` - any type `? extends Type` - any subclass of `Type` `? super Type` - any superclass of `Type`

### Implementation

Type Erasure:

Generics are handled at compile time by replacing the generic types with `Object`, replacing bounded generics with the bounding `Type` and adding casts and bridging methods.

Java only knows that types are at runtime, not at compile time.

### Restrictions on Generics

- Cannot be primitives
- Cannot instantiate generics: `new T()`
- Cannot be static
- Cannot have arrays of generic types
- No generic exceptions
- There are restrictions on overloading

## Object Oriented Design

### Textual Analysis

Considering the description of the system

Identify elements to be modelled. - nouns -> data, categories -> attributes, classes - verbs -> processes -> methods

Be mindful of relevance, relatedness, and relationships between the nouns in the model.

### Common Class Patterns

Find candidate classes using classification theory:

- Concepts
- Events
- Organisation
- People
- Place

are all class-candidates.

### Class-Responsibilities-Collaborators

After identifying candidate-classes, consider the behaviour and interactions between classes.

Using humans, roleplay to model how the classes should deliver the system behavoir.

For pinning down the responsibilities of a class, and defining the collaborators that facilitate their interaction.

This helps develop a shared understanding of the system design.

Reading

#### OOPSLA89 Paper Summary

Responsibilities are problems to be solved. The responsibilities of an object are active verb phrases.

All objects exist in relationships to other objects. Collaborators are objects that send messages, or are sent messages, in order to satisfy their responsibilities.

Make cards like this

The first line is the class name, followed by a list of responsibilties.

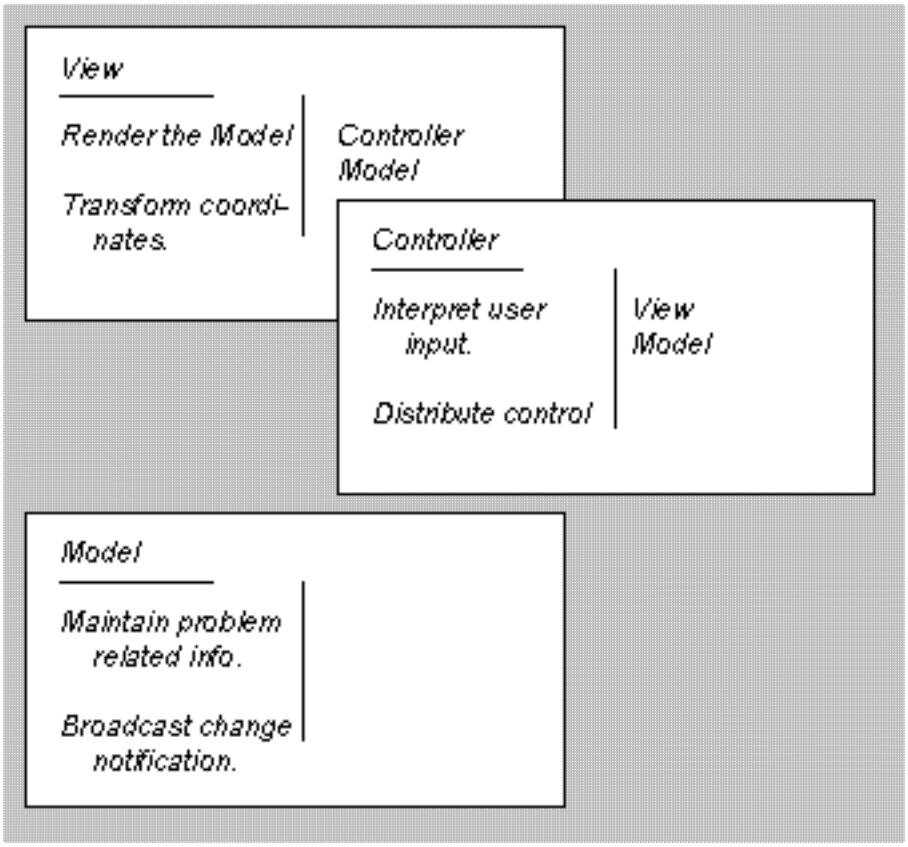


Figure 0.1: 3 index cards showing a model, view and controller. The top of the card is the Class name and below it a list of responsibilities

Design the model by role playing the execution of the model from some starting point in the model:

> start with only one or two obvious cards and start playing "what-if". If the situation calls for a responsibility not already covered by one of the objects we either add the responsibility to one of the objects, or create a new object to address that responsibility

Only create objects to address the immediate need, not a hypothetical future need. If they are needed in the future, then they will be created in the future.

## Functional Abstractions in Java

### Lambdas

- Anonymous methods
- Do not have a name
- Do not belong to a class, analagous to a c function
- Do not have return type

(`MathOperation` is an interface defining one method.)

```
MathOperation addition = (int a, int b) -> a + b;
MathOperation subtraction = (a, b) -> {return a - b};
```

It is also a more clear expression for attaching event handlers.

```
button.setOnAction((ActionEvent event) -> respondToButton());
```

### Functional Interfaces

The idea is they work like a single function, outside of a class. This allows passing functions and logic to methods as an attribute.

- They contain one (1) interface with a single abstract method and potentially default methods, static methods or overridden methods inherited, to support the core function.
- need the `@FunctionalInterface` label.

```
@FunctionalInterface
interface MathOperation {
    int operation(int a, int b);
}
```

### For Each loop

For each loops can be simplified to use an iterator to apply a passed lambda function to each element of a collection.

```
list.forEach(thing -> System.out.println(thing))
list.forEach(thing -> thing.toUpperCase())
```

Be careful, strings are immutable so the second line does not actually modify the string in the list, it only returns a reference to a new string.

### Method References

A reference to a member of a `FunctionalInterface`, static method, instance method, constructor, an arbitrary instance method, by using the syntax `ClassName::methodName` to refer to a specific method.

```
@FunctionalInterface
interface Doable {
    void do();
}

class MethodReference {

    public void method(String message) {
        System.out.println(message);
    }
```

```
    public static void staticMethod(String message) {
        System.out.println(message);
    }

    public static void main(String[] argv) {
        Doable memberExample = MethodReference::method;
        Doable staticMemberExample = MethodReference::staticMethod;
        memberExample.do("hello"); // prints "hello\n"
        staticMemberExample.do("world"); // prints "world\n"
    }
}
```

Another example:

```
list.forEach(System.out::println);
```

It is possible to reference the constructor with `Classname::new`.

### Standard Functional Interfaces

```
Consumer<T> ::  void accept(T t)

Function<T, R> :: R apply(T t)

Predicate<T> :: boolean test(T t)

Supplier<T> :: T get()

UnaryOperator<T> T apply(T t)
```

See javadoc on `java.util.function`.

## Streams

(monads)

A stream of data (different to IO streams), that aggregate functions can work on in a chain from source to an output.

- doesn't hold data
- doesn't modify the data source

Aggregate Operations

- functions that use the stream contents

Pipelining

- Operations can be daisy-chained together

Automatic Iteration

- Iteration is performed in the stream, over the data source
- Can process data that doesn't fit in memory
- Enables lazy invocation (compiler only calling a function when neccessary)

See `java.util.streams`

### Intermediate Streams

- Processes elements in a stream
- Returns another stream so they can be pipelined
- `map`, `filter`, `sorted` are examples

### Terminal Streams

The end of a stream that returns a result.

For example, `forEach`, `collect`, `reduce`

```
List<Student> students = new ArrayList<>(); // and add students to a list

// print failing students
students.stream()
    .filter(student -> student.getGpa() < 4.0)
    .forEach(System.out::println)
```

## Recursion and Sort Algorithms

The basic case of recursion is a function calls itself, reducing the size of the problem at each subsequent call, to progress towards a base case.

```
function {
    if (base case) {
        return;
    }
    else (reduction cases) {
        return function();
    }
}
```

This has the risk of stack overflows, because each call adds to the callstack.

Recursion is more elegant for some types of problems, which are naturally expressed in recurrent logic.

*Recursion is 'easier'*

To read, write, and for who. More understandable code that is harder to write is often valueable.

### Java Sorting

Java by default uses Timsort algorithm. `List.sort`.

### Merge Sort

Sorts bottom-up, by recursively halving an array and sorting the halves until the base case of 1 element is reached, and then merging the sub-arrays together by repeatedly choosing the smallest and adding it to the result array.

### Quick Sort

First find a `partition`, the position of one element that is already in the correct position for the sorted result, such that everything to the left of it is $<$ and everything to the right is $>$ it.

Then, repeatedly, a value in the left partition that is greater than the middle value is found, and a value to the right partition that is greater than the central value is found, and they are swapped, until everything to the right is greater than the middle value, and everything to the left is less than the central value.

This process is then recursively applied to the left and right sub-arrays.

- The partition scheme is central to the performance, the array needs to be divided evenly while maintaining the requirements.

https://en.wikipedia.org/wiki/Quicksort#Algorithm

## Which is Better?

- merge is easier to understand
- merge has better worst-case performance
- quicksort has better average performance
- quicksort has lower average memory requirements
- quicksort requires the data to fit in memory

Visualisation Sites:

- visualgo.net
- usfca
- sorting.at