
Australian
National
University
Review
R1
Structured Programming 1110/1140/6710

Java

J1	Imperative programming, standard library, types	All
J2	Types, objects, classes, inheritance, interfaces	All
J3	Naming, literals, primitives	All
J4	Arrays, operators, expressions, statements, blocks	All
J5	if-then-else, switch	All
J6	while, do-while, for	All
J7	parameters, return values	All
J8	Nested classes	Q1,Q3
J10	Integer, autoboxing, Math, Random	Q1
J11	Character and String	Q1,Q3
J12	Generics	Q4
J13	Type Inference	Q4
J14	Collections and sorting	Q3, Q4
J15	Java exceptions, catch or specify, Java syntax	Q4
J16	Threads	Q6

Object Oriented Programming

O1	Class declaration, object creation	All
O2	Initializers, access control, nested classes, enum types	Q1,Q3, Q4
O3	Interfaces	Q3,Q4
O4	Inheritance, overriding, polymorphism, super	Q3,Q4
O5	java.lang.Object, final, abstract	Q5

Java FX

"I want to know [what] we need to grasp about JavaFX. "

JavaFX **is examinable** in main exam, but isn't in the sample exam.

You won't be expected to memorize details, but understand concepts.

Abstract Data Types (ADTs)

A1	List implementation 1	Q1,Q3,Q4
A2	List implementation 2	Q1,Q3,Q4
A3	The set ADT and its implementation	Q1,Q3,Q4
A4	Hash tables	Q4
A5	Trees	Q1,Q4
A6	Map ADT implementation, ADT recap	Q1,Q3,Q4

Core Computer Science

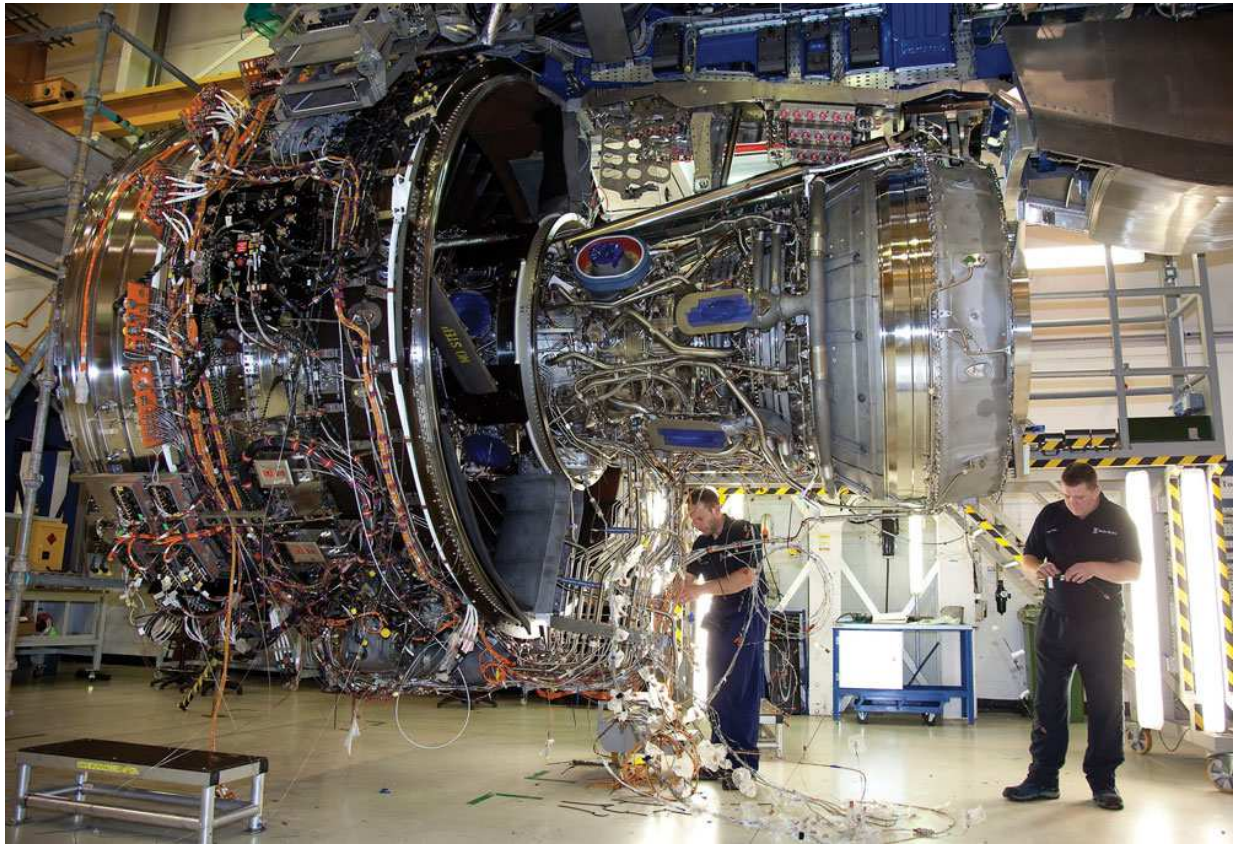
C1	Recursion	Q1
C2	Hash functions, choosing a good hash function	Q5
C3	Hashing applications, Java's hashCode()	Q5
C4	Files	Q2
C5	Computational Complexity	Q6
C6	Grammars	Q6
C7	Threads	Q6

Software Engineering

S1	IDEs, revision control, Git	All
S2	Git	All
S3	TDD, JUnit	Q3


Australian
National
University
Review
R2
Introduction to Software Systems 1110/1140/1510/6710

Introduction



Rolls Royce Trent XWB for the A350.

Photo: AINonline

Australian National University

Call by value and call by reference

- Parameters are values in Java
- Java cannot pass objects, just **references** to objects

Methods

Methods

Parameters

Return values

Parameters (method arguments)

Parameters are the mechanism for passing information to a method or constructor.

- Primitive types passed by *value*
  - Changes to parameter **are not seen** by caller
- Reference types passed by *value*
  - Changes to the *reference* **are not seen** by caller
  - Changes to *object referred to* **are seen** by caller
- Your last parameter may in fact be more than one parameter (*varargs*), and treated as an array

Australian National University

Collections & ADTs

- Collections: ‘Containers for objects’
  - set: mathematical set, unordered, can add, remove, test for membership
  - list: ordered list of objects, can add, can remove, can traverse
  - map: key, value pairs, keys used to add and retrieve values
- Implemented using the following fundamental ADTs (abstract data types):
  - Trees
  - Linked lists
  - Hashmaps


Australian
National
University
Collections
J13
Collections

The Collection Framework

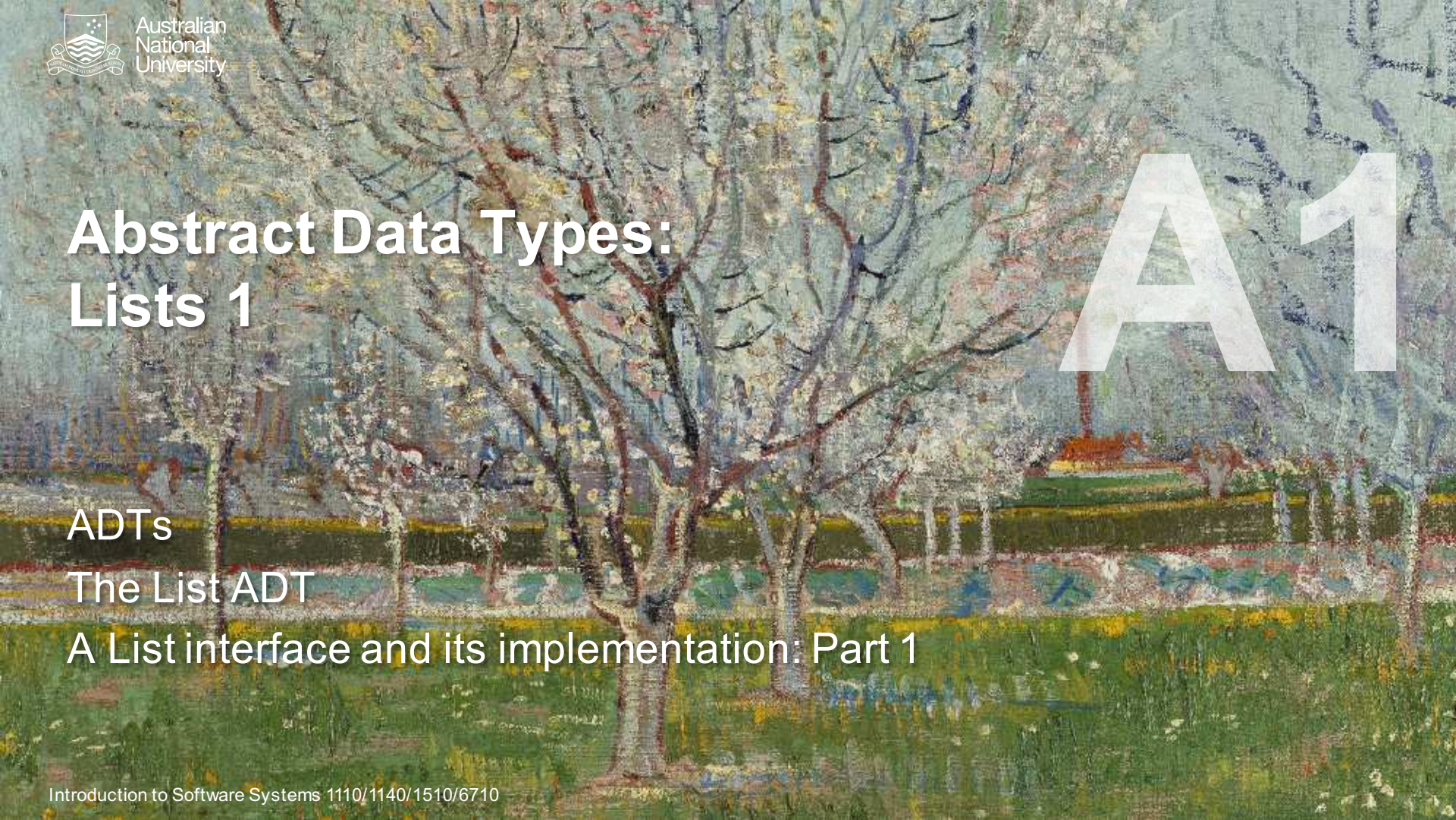
- Interfaces
  - Implementation-agnostic interfaces for collections
- Implementations
  - Concrete implementations
- Algorithms
  - Searching, sorting, etc

Using the framework saves writing your own: better performance, fewer bugs, less work, etc.

Concrete Collection Types

	Implemented Using				
Interfaces	**Hash table**	**Resizable array**	**Tree**	**Linked list**	**Hash table + linked list**
Set	`HashSet`		`TreeSet`		`LinkedHashSet`
List		`ArrayList`		`LinkedList`	
Queue				`LinkedList`	`LinkedHashMap`
Map	`HashMap`		`TreeMap`		

Based on table from http://docs.oracle.com/javase/tutorial/collections/implementations/index.html



Abstract Data Types: Lists 1

ADTs

The List ADT

A List interface and its implementation: Part 1

Abstract Data Types (ADTs)

Abstract data types describe containers for storing data elements. An ADT is abstract, not concrete.

A **container** is a very general ADT, serving as a holder of objects. A **list** is an example of a specific container ADT.

An ADT can be described in terms of the semantics of the operations that may be performed over it.

The List ADT

The **list** ADT is a container known mathematically as a *finite sequence* of elements. A list has these fundamental properties:

- duplicates *are* allowed
- order is preserved

A list may support operations such as these:

- *create*: construct an empty list
- *add*: add an element to the list
- *is empty*: test whether the list is empty

Australian National University

Hashing

- Hash functions
- Hashing applications
- Java's hashcode()

Hash Functions

C2

Hash functions

Choosing a good hash function

Hash Functions

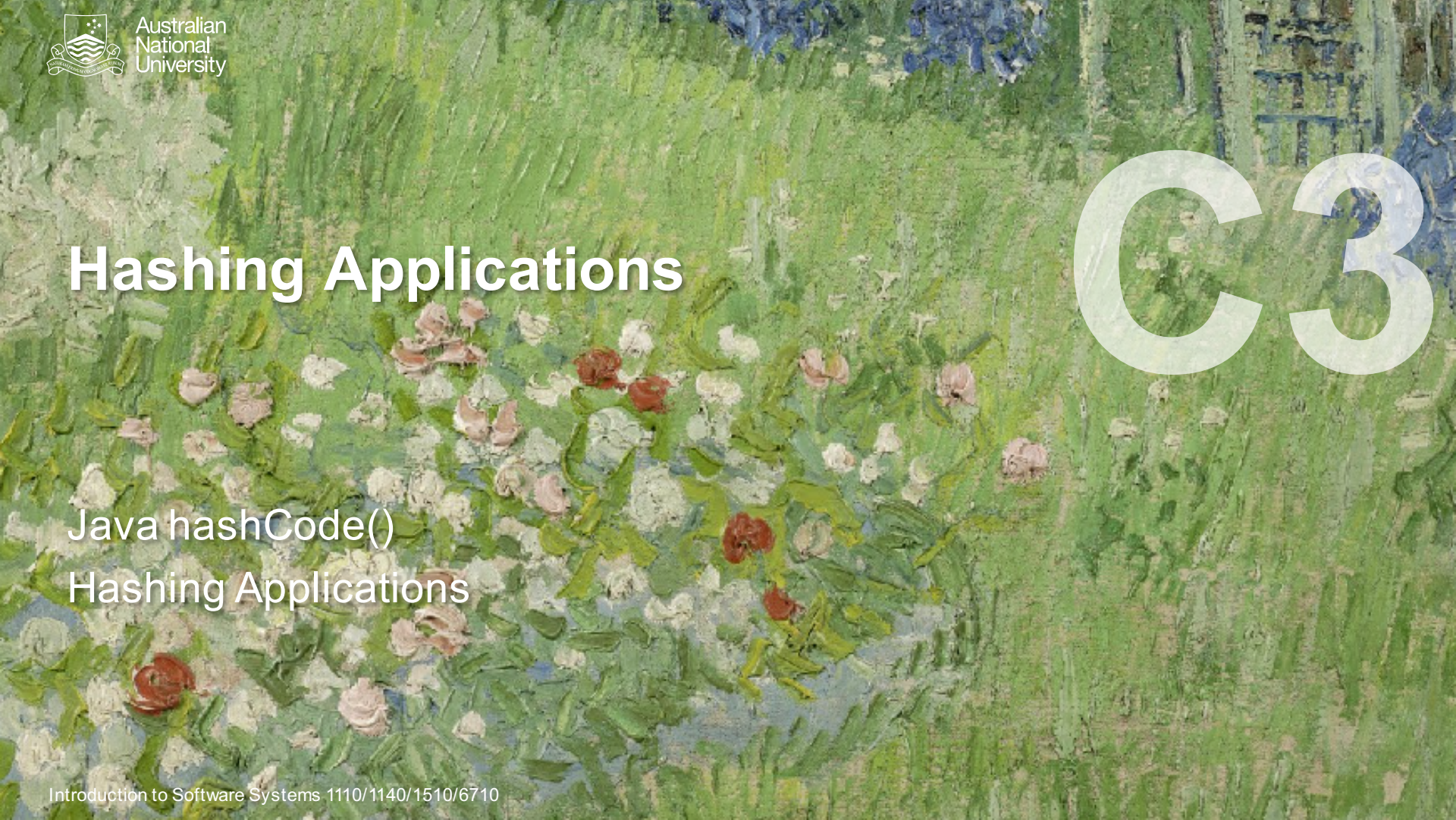
A hash function is a function $f(k)$ that maps a key, k, to a value, $f(k)$, within a prescribed range.

A hash is deterministic. (For a given key, k, $f(k)$ will always be the same).

Choosing a Good Hash Function

A good hash for a given population, P, of keys, $k \in P$, will distribute $f(k)$ evenly within the prescribed range for the hash.

A *perfect hash* will give a unique $f(k)$ for each $k \in P$



Hashing Applications

C3

Java hashCode()

Hashing Applications

Java hashCode()

Java provides a hash code for *every* object

- 32-bit signed integer
- Inherited from `Object`, but may be overwritten
- Objects for which `equals()` is `true` must also have the same `hashCode()`.
- The hash need not be perfect (i.e. two different objects may share the same hash).

Uses of Hashing

- Hash table (a map from key to value)
- Pruning a search
  - Looking for duplicates
  - Looking for similar values
- Compression
  - A hash is typically much more compact that the key
- Correctness
  - Checksums can confirm inequality

Practical Examples…

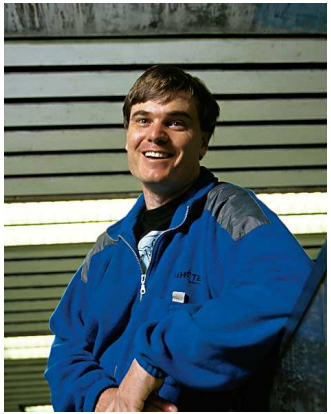
Luhn Algorithm

Used to check for transcription errors in credit cards (last digit checksum).

Hamming Codes

Error correcting codes (as used in EEC memory)

Practical Examples…



rsync (Tridgell)

Synchronize files by (almost) only moving the parts that are different.

MD5 (Rivest)

Used to encode passwords for a long time (but no longer).

Australian National University

Computational Complexity

- How will the execution time of a problem change as the size of the problem changes?
  - Need to define ‘size of problem’
  - Need to understand how problem time changes as that variable changes

Computational Complexity

C5

Time and Space Complexity

Big O Notation

Examples

Practical Study: Sets

Context

Key computational resources:

- Time
- Space
- Energy

Computational complexity is the study of how problem size affects resource consumption for a given implementation.

- Worst case
- Average case

Broad Approach

1. Identify *N*, the number that characterizes the problem size.
   - Number of pixels on screen
   - Number of elements to be sorted
   - etc
2. Study the algorithm to determine how resource consumption changes as a function of *N.*

Concrete Examples

```
public int mindist(ArrayList<Integer> values) {
  int min = Integer.MAX_VALUE;
  for (int i = 0; i < values.size(); i++) {
    for (int j = i + 1; j < values.size(); j++) {
      int diff = values.get(i)-values.get(j);
      if (Math.abs(diff) < min)
        min = Math.abs(diff);
    }
  }
}
```

1

N

(N – 1)N/2

(N – 1)N/2

(N – 1)N/2

(N – 1)N/2

$$S(N) = 1 + N + 4((N-1)N/2) = 1 + N + 2N^2 - 2N = 2N^2 - N + 1 \in O(N^2)$$

Note: N -1 + N – 2 + … 2 + 1 = (N – 1)N/2

Australian National University

Formal Grammars (EBNF)

- Not about semantics, just about rules that define relationship among symbols


Australian
National
University
C6
Formal Grammars
Grammars
EBNF
Introduction to Software Systems 1110/1140/1510/6710

Formal Grammars

Formal languages are distinguished from natural languages by their artificial construction (rather than natural emergence).

Noam Chomsky is often credited with opening the field of formal grammars while studying natural languages.

Duncan Rawlinson (Creative Commons)

Noam Chomsky

Extended Backus-Naur Form

EBNF is a standard way of representing the syntax of a formal language (but *not* the semantics!)

- Terminal symbols
  - e.g. characters or strings
- Production rules
  - combinations of terminal symbols

Robert McClure

Niklaus Wirth

Extended Backus-Naur Form

Very basic syntax of EBNF production rules:

- ‘=’ defines a production rule
- ‘|’ identifies alternates (e.g. `‘1’ | ‘2’ | ‘3’`)
- ‘{’, ‘}’ identify expressions that may occur zero or more times (e.g. `‘1’, { ‘0’ }`)
- ‘[’, ‘]’ identify expressions that may occur zero or one time (e.g. `‘1’, [ ‘0’ ]`)
- ‘,’ identifies concatenation
- ‘-’ identifies exceptions
- ‘(’, ‘)’ identify groups
- ‘;’ terminates a production rule

Example

```
(* a simple program syntax in EBNF – Wikipedia *)
program = 'PROGRAM', white space, identifier, white space,
          'BEGIN', white space,
          { assignment, ";", white space },
          'END.' ;
identifier = alphabetic character, { alphabetic character | digit } ;
number = [ "-" ], digit, { digit } ;
string = '"' , { all characters - '"' }, '"' ;
assignment = identifier , ":=" , ( number | identifier | string ) ;
alphabetic character = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                     | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                     | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                     | "V" | "W" | "X" | "Y" | "Z" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
white space = ? white space characters ? ;
all characters = ? all visible characters ? ;
```

Example

```
‘0’ | ‘1’ | ’00’ | ‘11’ | ‘000’ | ‘101’ | ‘111’ | ’010’
```

```
pal = ‘0’ | ‘1’ | (‘0’ , [pal], ‘0’) | (‘1’, [pal], ‘1’);
```

Australian National University

Exam Q1

- Need to understand basic Java collections
  - How do you add, get and remove elements
- Need to understand recursion
  - Stopping conditions
  - Tracing execution

Exam Q2

- Only answer questions you're confident about
- Can get 10/10 marks by only answering 10/15 questions
  - Don't stress if there are some you don't know
- Ensure you mark your answer clearly

Exam Q3

- Read all parts of the question very carefully
- Ensure you include **all relevant code**
- May want to revisit design after other parts of Question
- 3i) About clearly explaining a good OO design
  - Does your design make good use of OO?
  - Does it make sense to use inheritance?
  - Does it make sense to use interfaces?
  - What relationship should there be among classes?
  - Should you use collection types?

Exam Q3

- 3ii) Know how to declare a class and its fields
- 3iii), iv), & vi) ensure you write all relevant code
- 3v) know how to write a unit test

Exam Q4

- Very close to example in lecture
- Ensure you include all relevant code
- Don't implement `add(V value)` as `{ secretadd(value); }`
- Notice differences with lecture code
- Answer this question yourself and then compare to lecture code

Exam Q5

- i) Be clear and specific. Need to understand what a race is (J16)
- ii) Need to understand sets, linked lists and complexity
- iii) Not too hard, only four digits, each can be `1' or `0'. Try to do it.
- Iv) Harder; see revision lecture

Exam Q6

- Provide five clearly identified major points
- Write in simple, plain clear English
- Clarity is essential
- Less is more

Exam, Overall

- Budget your time
- State your assumptions
- Try to communicate your understanding clearly